Open meta-modelling frameworks via meta-object protocols

Jesús Sánchez Cuadrado

Universidad de Murcia (Spain)

Juan de Lara

Universidad Autónoma de Madrid (Spain)

Abstract

Meta-modelling is central to Model-Driven Engineering. Many meta-modelling notations, approaches and tools have been proposed along the years, which widely vary regarding their supported modelling features. However, current approaches tend to be *closed* and rigid with respect to the supported concepts and semantics. Moreover, extending the environment with features beyond those natively supported requires highly technical knowledge. This situation hampers flexibility and interoperability of meta-modelling environments.

In order to alleviate this situation, we propose *open* meta-modelling frameworks, which can be extended and configured via meta-object protocols (MOPs). Such environments offer extension points on events like element instantiation, model loading or property access, and enable selecting particular model elements over which the extensions are to be executed. We show how MOP-based mechanisms permit extending meta-modelling frameworks in a flexible way, and allow describing a wide range of meta-modelling concepts. As a proof of concept, we show and compare an implementation in the MetaDepth tool and an aspect-based implementation atop the Eclipse Modelling Framework (EMF). We have evaluated our approach by extending EMF and MetaDepth with modelling services not foreseen initially when they were created. The evaluation shows that MOP-based mechanisms permit extending meta-modelling frameworks in a flexible way, and are powerful enough to support the specification of a broad variety of meta-modelling features.

Keywords: Model-Driven Engineering; Flexible Meta-Modelling; Meta-Object Protocol; Aspect Orientation; Multi-level Modelling; Extensibility

1. Introduction

Model-Driven Engineering (MDE) is based on models to capture the essential properties of systems, and on their automated processing to support all kinds of development activities, including model analysis, execution and code generation [1, 2]. The abstract syntax of models is described through a meta-model, and hence meta-modelling becomes recurrent in MDE.

Meta-modelling systems should offer appropriate meta-modelling primitives and services for the problem at hand. Many meta-modelling languages – like the Meta-Object Facility (MOF) [3], ConML [4], the Unified Modelling Language (UML) [5] or GOPRR [6] – frameworks – like the Eclipse Modelling Framework (EMF) [7] – and tools [6, 8, 9, 10] have been proposed along the years. While most of them are based on object-oriented concepts, their features and supported meta-modelling facilities widely vary. For example, some are based on two-level modelling [3, 5, 7] while others [9, 10] permit arbitrary meta-levels. Some support class cardinalities, or prototype-based modelling [9, 11], while others lack these features, but users would benefit from them in different scenarios.

The meta-modelling facilities that each approach offers are typically fixed, with few possibilities for extension or adaptation. This means that providing support for functionalities beyond those offered natively becomes a costly activity, which only experts in the meta-modelling framework can accomplish. This is typically performed by either explicitly modelling those facilities (i.e., by creating a customized meta-model and supporting operations), or by some problem-specific hack.

For example, should we like to support prototype-based modelling within the EMF[1], a possibility would be to create a specific meta-model with a number of operations to emulate cloning and slot access (so that, if a value is not provided in a clone, the slot value is retrieved from the prototype). This solution is not optimal because one cannot use prototype-based modelling in combination with standard domain meta-modelling. This is so as it would not be possible to use a domain meta-model, but only the prototype-based meta-model. Another typical issue is that some design decisions of the meta-modelling framework lead to good performance for many cases, but are not optimal for others. For example, in EMF it is efficient to retrieve the superclasses of a given class, but it is less efficient to retrieve the subclasses. In scenarios with frequent downwards navigation through inheritance hierarchies, this may be problematic. In this case, a specific hack, like the creation and maintenance of an index every time a class is connected via inheritance, would be needed.

In order to improve this situation, we propose using the concept of *meta-object protocol* [12, 13] in meta-modelling environments. A meta-object protocol (MOP) is a technique developed in the Object Oriented Programming (OOP) community (originally for CLOS [14]) to *open* a language infrastructure so that

[1]In prototype-based languages, like Javascript, objects are created by cloning other objects (called prototypes).

it can be extended and customized. The key point is to enable language users to add behaviour at certain points (e.g., when instantiating an object) and exposing the essential structure of the language so that it can be queried and manipulated. We call *open* to meta-modelling frameworks that enable their extension through a MOP. In contrast *closed* frameworks do not permit extending the semantics or implementations of their abstractions, or the services they offer.

In this paper, we argue that open meta-modelling frameworks present benefits due to their enhanced flexibility. Ideally, such frameworks could be constructed by a small core of meta-modelling abstractions and operations, which can then be extended via libraries accounting for different semantics, like prototype-based modelling, or multi-level modelling. Moreover, those libraries can provide support for facilities like visualization, a high-level constraint catalogue, or model/meta-model co-evolution, among many others.

As a proof of concept, we present an implementation of MOPs over MetaDepth [9] and another over EMF. The purpose is to show how our general architecture can be realized for two different frameworks: one built by us, and another developed by a third party. We compare both approaches by means of examples, and evaluate them through case studies against a set of requirements, demonstrating the benefits of MOPs for meta-modelling.

The rest of the paper is organized as follows. Section 2 motivates the need for open meta-modelling frameworks and introduces some background. Section 3 describes the main elements of an open meta-modelling framework. The next three sections detail its main parts. Section 4 describes the requirements for open meta-modelling APIs, and typical event types they may expose. Section 5 explains approaches to identify the model elements subject to extensible semantics and Section 6 shows how these semantics can be described. Section 7 describes two implementations: one over MetaDepth and another over EMF. Section 8 evaluates both approaches through case studies. We relate our work with the existing literature in Section 9 and conclude the paper in Section 10.

2. Motivation and background

This section motivates the need for adaptation and extensibility mechanisms in meta-modelling frameworks, formulating some requirements to cover such needs (Section 2.1). Then, we review some basic notions of MOPs and aspect orientation that we have adapted to work within meta-modelling frameworks in order to satisfy the stated requirements (Section 2.2).

2.1. Motivation

Meta-modelling frameworks are built around a fixed set of meta-modelling constructs (e.g., classes, attributes) and facilities (instantiation, validation of integrity constraints, feature inheritance, field access, model load) with fixed semantics, which are typically hard to extend. However, in many cases, one would like to add specific behaviour to be executed on certain events to change the default semantics or contribute with additional behaviour. For example, in a meta-model to create expressions, we would like to automatically instantiate the supported data types (like Integer or Boolean) and some special values (like True and False, which would become singleton objects) every time the meta-model is instantiated. Without such automation, the user needs to manually instantiate these elements every time s/he instantiates the meta-model.

Other times, one would like to add special semantics to some existing meta-modelling construct. For example, in MetaDepth, there is no notion of composition reference. Instead, the composition semantics needs to be emulated by adding OCL constraints on specific references. Similarly, EMF does not support cardinalities for classes, which should be emulated via constraints. Having the possibility to extend the framework with support for these concepts would save effort to the developers since they can simply reuse the extensions as libraries.

More advanced scenarios like adding new concepts to the meta-modelling language or changing between modelling paradigms can also be implemented if the meta-modelling framework is externally configurable. For example, while EMF supports a two meta-level approach, in some cases, it might be more appropriate to use (potency-based) multi-level modelling [15], prototype-based modelling [16] or powertype-based modelling [17]. Of course, one could use different tools depending on the problem at hand, but being able to configure the environment in different ways promotes flexibility by enabling mixing paradigms and facilitates experimentation. For example, in [18] a language design was evolved from an initial prototype-based concept to powertype modelling until the final multi-level design. A configurable meta-modelling environment could have facilitated this experimentation task before building the actual language.

Another setting in which configurable meta-modelling environments would be beneficial is within multi-level modelling [15]. In this context, several concepts, like potency [19, 20] have been proposed to control the instantiability of classes, and the features of their instances beyond the immediate meta-level below. These concepts have evolved and have been refined along the years, for example, introducing the notions of mutability and durability for fields [21], dual potencies for connectors [22], connector cardinalities [23] or different semantics of potencies for constraints. This means that different tools currently implement slight variations of core multi-level modelling concepts, with the result of being incompatible to each other [24]. The availability of meta-modelling extension mechanisms would have enabled to experiment with variations of these notions without changing the internals of the meta-modelling framework. This way, these concepts could have been extracted into a separate library, whose elements could be selected and combined by the final users. Hence, the same tool could be customized with different multi-level semantics.

Altogether, we call "open" to meta-modelling frameworks that permit the external configuration of meta-modelling facilities with user-defined behaviour. From the described scenarios, we can enumerate a number of requirements for open meta-modelling frameworks:

R1 Exposing a well-defined meta-modelling API that can be accessed from user extensions.

R2 Opening up relevant meta-modelling actions, like class instantiation or model load. The user should be able to extend the standard behaviours of these actions, override them, or forbid their ocurrence.

R3 Offering a mechanism to load user extensions on demand.

R4 Supporting mechanisms to select the model elements (e.g., a particular class, or a set of features) that are affected by the user extension.

R5 Provide suitable high-level languages to describe the user extensions. Ideally, those languages could be model management languages the user knows about.

Once we have analysed the required features, next we provide background on some techniques from programming that we propose to adapt to meta-modelling in order to tackle these requirements.

2.2. Meta-Object Protocols and Aspect Orientation

MOPs were devised by the OOP community in response to similar problems as we have described in the previous subsection [12, 13]. In particular, for the need to support different variants of a language (e.g., different strategies to order the priority of superclasses in case of multiple inheritance), and performance issues (e.g., different ways to represent slots in objects depending on the expected number of objects and slot access). MOPs offer the possibility to provide user code overriding the default behaviour of the framework when certain actions are performed, like instantiating a class or accessing an attribute. Typically, the code can be executed before, after or around the infrastructure action. Not only such behaviour can be overridden, but (if executed before or around) the user code may prevent the execution of the infrastructure code. Hence a MOP reifies language notions (e.g., classes, inheritance, methods) in meta-objects [25]. These are regular objects with an API that is exposed to the language user. The user can then define subclasses of the meta-object, or provide code to override its default behaviours. This results in an adaptation of the behaviour of the base language itself.

Related to MOPs, aspect-oriented programming (AOP) [26] is a programming technique that aims at increasing program modularity, by modularizing cross-cutting concerns. These are concerns that are scattered throughout the code (e.g., like a logging functionality that is needed in every public method). This way, these scattered concerns are modularized in *advices* that modify the behaviour of the base code. Advices are applied to the base code at particular *join points*. The selection of joint points is typically performed by specifiying a *pointcut*, which is a predicate of some kind that matches join points. The concepts of AOP have been lifted to models and modelling giving rise to aspect-oriented modelling (AOM) [27]. Hence, AOM approaches enhance modularity of crosscutting concerns by using aspect-orientation at the level of models.

While we take terminology and ideas from AOM, our goals differ, as our aim is to adapt the semantics of the meta-modelling facilities offered by the meta-modelling framework. Instead, in AOM the goal is to construct models using advanced modularity mechanisms based on concerns. This way, commonly, pointcuts in AOM refer to specific points in the model, over which advices (typically model fragments) are to be applied. This application is generally done using model weaving [28] or model transformation techniques [29], resulting in a model that contains the base model enriched with the advice models. In contrast, in our approach pointcuts should also consider meta-modelling events, like creating or deleting an object. In our setting, advices will typically contain behaviour specification which modifies the standard execution of the meta-modelling framework over the selected modelling elements. Hence, a key difference between AOM and our approach, is that, while AOM focusses on model construction by composing concerns, we focus on extending or adapting the meta-modelling facilities which are used in the modelling process.

The OOP community [30, 31] has identified several useful dimensions of control for MOPs, like *temporal control*, which refers to the possibility to activate and deactivate user extensions (which we call *advices* using AOP terminology) and *level control* to allow knowing whether certain code is executed due to a regular program or due to an advice, which is sometimes needed to avoid the same advice to be executed again (e.g., an advice that creates objects and is activated upon the creation of objects), leading to an infinite recursion. *Placement control* refers to the need to activate advices for a particular class, object or method. This is similar to pointcut selection in AOP. Therefore, a MOP needs a mechanism to indicate for which elements an advice is applicable.

Next, we show how MOPs can be integrated into meta-modelling environments to enhance their extensibility.

3. The ingredients of open meta-modelling frameworks

Open meta-modelling frameworks must provide mechanisms to extend their behaviour in unforeseen ways. Figure 1 shows a conceptual model with their main ingredients, and an indication on the sections of this paper where we address them in detail.

In the first place, an open meta-modelling framework should make available an API that can be used externally. Such an API should expose a set of EventTypes (join points in AOP terminology) which occur during its operation, like *validating* a model or *instantiating* a class. These events can be captured and processed in an Extension. An extension is triggered by one or more event types (e.g., if the same behaviour is to be executed on different events like object creation or update).

The extension behaviour is implemented by an Advice, which may perform any kind of model manipulation by accessing the meta-modelling API. We propose two alternative means to access the meta-modelling API: by means of model management languages, and by using a general purpose programming language (GPL). The first alternative may be more familiar to the engineer, as s/he can use model management languages to express advices, in the form of a model-to-model transformation, code generation, an in-place transformation,

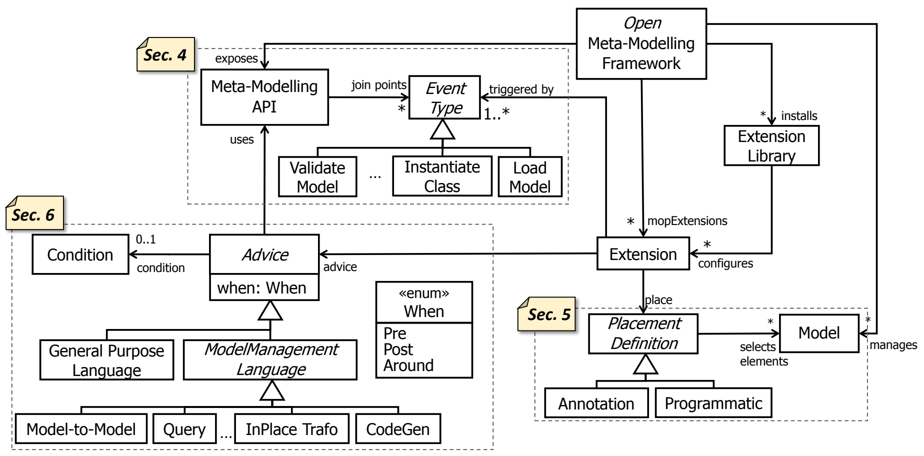


Figure 1: Conceptual model with the elements of an open meta-modelling framework. Labels on the dashed rectangles indicate the section where the contained elements are discussed

or a constraint execution. However, this requires the model management languages to have mechanisms to interact with the meta-modelling API (e.g., to create a field reflectively, or to delete an object). In the second case, advices are specified using a general purpose programming language, typically the one used to create the meta-modelling framework. This solution is lower level and may be more verbose to perform common actions.

Similar to AOP [26], the advice associated to an extension can be processed before the joint point operation is executed (Pre in the When enumeration), afterwards (Post) or can be intercepted so that the handling code decides when to forward its execution (Around). Moreover, the triggering of the advice may contain additional conditions.

In addition to join points in the meta-modelling API, we need to identify the model elements over which the advice will be executed. Therefore, similar to AOM techniques, we also have structural join points like model, entity (i.e., a class or an object), field and reference. Hence, a mechanism is needed to indicate the placement of the extension, which is called placement control in MOP approaches [30], and are similar to the pointcuts of AOM approaches.

Placement control can be done programmatically e.g., by a piece of code that searches in the model the elements to be considered in the advice. Alternatively, we also propose the use of annotations. These are typically a set of related tags to be placed in model elements, which help in directing the execution of their associated advice when the given EventType(s) occur. For example, to enable the automatic instantiation of specific elements in a meta-model, we may introduce an annotation @autoInstantiate applicable over meta-models, and an annotation @auto applicable over classes. An associated advice will be executed when the annotated meta-model is instantiated, and will create an instance of every class annotated with @auto.

Finally, related extensions can be packaged in libraries, containing both the advices and the configuration of these via placement control mechanisms.

The next sections describe typical requirements and styles of meta-modelling APIs and the event types they should expose (Section 4), the placement control mechanisms we propose (Section 5), and the advice mechanisms (Section 6).

4. Designing open meta-modelling APIs

Figure 2 depicts a feature model which describes the design space and the main features of open meta-modelling APIs.

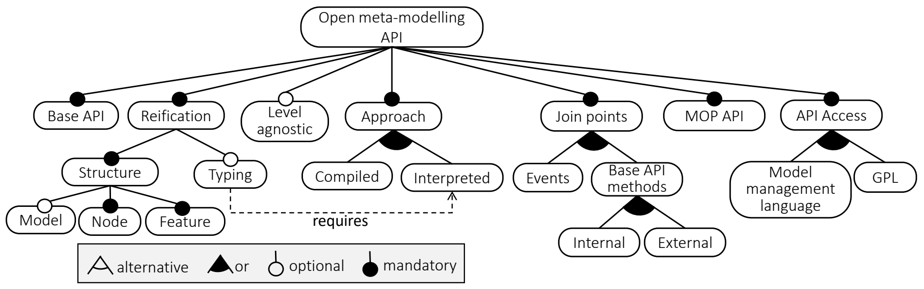


Figure 2: Feature model of an open meta-modelling API

In the first place, an open meta-modelling framework should make available a base API that permits its use programmatically, e.g., to instantiate classes or assign field values. The programmatic use of a base API is needed to define external advices that can adapt or extend the meta-modelling features (see Section 6).

MOP techniques require reifying the language concepts in terms of meta-objects, so that they can be queried and manipulated from advices. In meta-modelling frameworks this is normally granted, as models and meta-models are explicitly represented using the meta-modelling facilities of the framework. Typically these include structural notions like model/meta-model, class/object, attribute/slot and reference/link and also the typing relationship between them. Moreover, the base API usually consists of methods attached to these meta-objects. Some frameworks (like MetaDepth) are level-agnostic, so that they represent uniformly these concepts at any meta-level. While this feature is not essential for open frameworks, it simplifies creating extensions, as the number of join points to consider is lower.

The support and flexibility of meta-modelling frameworks for reflection varies, but at the bare minimum we require the ability to reify nodes (classes/objects) and features (attributes/slots/references/links). For example, the EMF does not explicitly provide a notion of model or meta-model (but of Package, which exists at the meta-model level only, and is a container gathering related classes), but offers elements to represent classes and features. EMF is not level agnostic, and so it represents differently elements at the meta-model and model levels.

Meta-modelling frameworks can follow a compiled or an interpreted approach. A compiled approach produces code in an implementation language (Java in case of EMF). The generated code typically extends runtime infrastructure of the meta-modelling framework. For example in EMF, the generated interfaces extend EObject, which provides reflective capabilities to navigate to the meta-class. Compiled approaches limit and complicate user extensions, as the reflection possibilities are reduced. For example, advices modifying the meta-model transparently would become more challenging.

Some meta-modelling frameworks work in an interpreted mode, where the meta-model is not compiled. This provides more flexibility as e.g., the meta-model can be modified programmatically, and then instantiated. EMF supports both a compiled and an interpreted mode. Frameworks like MetaDepth support interpreted mode only. This allows richer reifications and reflection capabilities. For example, MetaDepth explicitly represents the typing relation between elements and their type (instead of relying on the instantiation provided by a programming language), which then can be queried and modified at run-time [32].

Open frameworks should be extensible. One lightweight extension mechanism is through event notifications (feature JoinPoints.Events in Figure 2). Hence, facilities like class instantiation generate notifications, which can then be captured by callback functions to execute custom code. Both EMF and MetaDepth support event-based notifications. Nonetheless, we consider this a *weak* extension mechanism, because the code cannot for example forbid the action associated to the occurrence of the event, and there may be actions (e.g., getting the superclasses of a class) for which no events may be produced.

A stronger extension mechanism is based on instrumenting the meta-modelling API. This can be supported by design in the API, for example by adding appropriate hooks on selected methods of the base meta-modelling API. These would become join points to which advices can be attached. This approach is illustrated in our MetaDepth implementation. Another possibility for enabling internal instrumentation is to profit from extension points of component-based technologies, like Eclipse [33]. An internal approach requires having control of the meta-modelling framework source code. This may not be possible if reusing a meta-modelling framework built by third parties. Hence, alternatively, one can use an external mechanism, based on AOP. This approach will be illustrated in our EMF implementation. While external mechanisms are unobtrusive, they may be limited by the aspect-oriented technology used.

Regardless of the join point mechanism used, a helper MOP API is needed to define the extensions. This MOP API is responsible for selecting pointcuts and execute advices on them; and should offer facilities e.g., to track whether base API code, or advice code is being executed. If join points are internal API methods, the MOP API will typically be integrated within the base API. Instead, with an external approach to join points, the MOP API will be separated from the base API, in a different library. The base and MOP API may be accessible using GPLs, or through model management languages.

Table 1 shows the set of meta-modelling join points that we consider, to-

Table 1: Meta-modelling join points, and example advices

	Name	**Example Advice**
Model	Create model	Automatic model population with predefined objects
	Load model	Automatic model visualization using external diagramming libraries
	Save model	Serialize additional artefacts (e.g., creating HTML documentation)
	Validate model	Extend basic validation to provide semantics of e.g., composition
Object	Instantiate object	Automatically create inheritance to powertype object
	Delete object	Create execution trace object
	Validate object	Mark object as draft, so that its validation is deferred
	Set inheritance rel.	Disallow multiple inheritance, or inheritance at the object level
Feature	Instantiate feature	Support for field/reference redefinition
	Set value	Unit conversion (e.g., yards to meters)
	Get value	Default values different from standard ones (e.g., get value from prototype object)

gether with some example advices. Please note that in our approach meta-modelling joint points are fixed, and thus there is no pointcut language but the events are exposed through an API.

Meta-modelling join points include those at the feature level (set, get, instantiation), object/class level (instantiation, deletion, validation), and model level (create, load, save, validate). Depending on the reification capabilities of the framework, some join points may not be directly available. For example, in EMF there is no first-class notion of model, so detecting meta-model instantiation is not possible unless EMF is instrumented using e.g., AspectJ, as we do in our implementation. In MetaDepth inheritance is a built-in concept (available at any meta-level), so that one can capture the creation of such relations, e.g., to forbid multiple inheritance.

5. Placement control

In addition to join points offered by the meta-modelling API, our approach needs join points in modelling elements, and pointcuts to select them. Following MOP terminology, we call this placement control [30], and propose two mechanisms. The first one is based on a programmatic API (Section 5.1), while the second one is based on annotating the model elements (Section 5.2).

5.1. Programmatic MOP configuration

Extension libraries need to configure the meta-modelling framework with their particular behaviour. One way to perform this task is by providing programmatic access to the MOP, in the style of MOPs for OOP [12].

A meta-modelling MOP API must provide a way to install callbacks on meta-modelling events as well as a filtering mechanism to emit only the concrete events a particular extension wants to process. There are many ways to define such an API. Figure 3 shows a class diagram describing the fluent API that we have designed for our EMF implementation. The EMOP class defines one "on" method for each kind of supported meta-modelling event. These methods return an object providing access to a simple API (e.g., an object of type OnSet) to install callbacks before, after or around the selected event and take the callback as a Java 8 lambda. The around method allows us to cancel the execution of the event by throwing an InterruptAround exception or to return a different value (e.g., overriding onGet semantics). In addition, it is possible to install filters to rule out certain events we are not interested in. These methods are a fluent API, which internally packages the configured callbacks into an advice object which is installed into the AdviceRepository. We have instrumented EMF (see Section 7) to notify about meta-modelling events to the advice repository, so that the callbacks are invoked when certain points of the EMF framework are executed. Finally, we have considered three variants of the EMOP class. The global EMOP is intended to notify about any event. The resource EMOP restricts events to those happening in a certain resource, whereas the thread EMOP restricts events to those happening within the execution of a certain thread. The intention of this variant to a allow the MOP configuration to be active only during certain execution scope (e.g., during the execution of an specific model transformation).

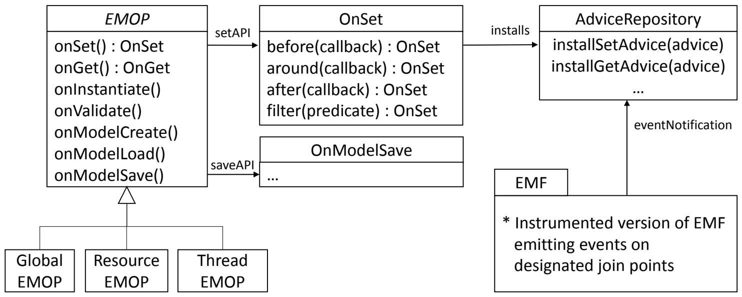


Figure 3: Class diagram describing the EMOP API

Example. ResourceType – Detecting conformance relationships. In EMF there is no explicit notion of model or meta-model[2], hence we may be interested in building a facility to make explicit the conformance relationship between an EMF resource and the EPackages containing the types of the resource's objects. A simple implementation could place an advice on model load to compute the conformance relationship for each loaded model.

Listing 1 shows how to configure the MOP using our implementation on EMF. First, we obtain a handle to the global MOP (line 1) through the call to the global static method. Then, we use the fluent API to configure the onModelLoad event, so that our callback (defined using a Java 8 lambda; lines 3–6) is invoked each time a resource is loaded and has been populated with elements. The behaviour in this case is to compute the set of EPackages which the EClasses used by the resource's objects belong to (method getAllPackages; line 12). Then, we use a model index (implemented as a singleton) to register the conformance relationship. This index can be queried by other extensions

[2]In EMF a resource holds any kind of object. A resource's meta-model can be seen as the set of packages defining the types of the resource objects.

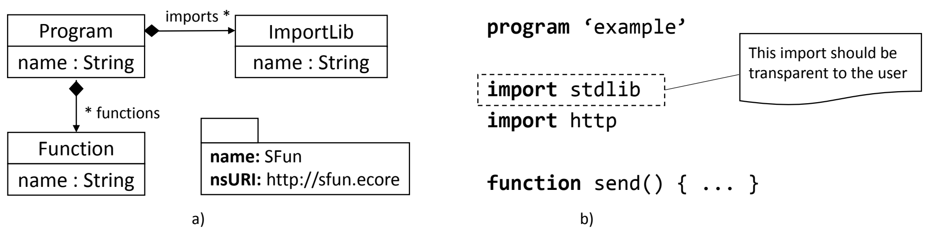
or by some user interface component to determine the type of a resource, or conversely, which loaded resources are instances of a given EPackage.

```
EMOP.global().
   onModelLoad().
   after(resource -> {
      Set<EPackage> pkgs = getAllPackages(resource);
      ModelIndex.instance().register(resource, pkgs);
   });

/**
 * Helper method to determine the "meta-model" of a resource as a set of
 * EPackages which hold the EClasses of the resource's objects (code omitted)
 */
Set<EPackage> getAllPackages(Resource r) { /*...*/}
```

Listing 1: Programmatic configuration to determine the conformance relationship of a model

Example. AutoInst – Automatic instantiation of model elements. In this example we aim at automatically instantiate certain model elements when a model or certain model element is created. This need arises in scenarios in which there are default objects which must be present in any model conforming to some meta-model. As a concrete example Figure 4(a) shows the meta-model of a simple programming language that supports the definition of functions, and also allows functions from other programs to be imported via an import statement. Figure 4(b) shows an example of this language using a textual concrete syntax. We want the stdlib module to be automatically added to any program, without requiring the user to write an explicit import statement.



```
program 'example'

import stdlib
import http

function send() { ... }
```

Figure 4: (a) Meta-model of a simple language supporting functions and import statements. (b) Example program in which the standard library is automatically imported

We can use our MOP facility to describe an extension which detects when a Program object is created in order to automatically instantiate an ImportLib object, setting its name to "stdlib". Listing 2 shows how the extension could be configured programmatically with EMOP. We extend the regular EMF instantiation using the onInstantiate method, selecting only those objects belonging to the SFun package and when the object in question represents a program (filter method, lines 3–6). Then, we use the regular EMF dynamic API to create the corresponding ImportLib object and add it to the program object.

```
1 EMOP.global().
2    onInstantiate().
3    filter(obj ->
4         "http://sfun.ecore".equals(obj.eClass().getEPackage().getNsURI())
5         &&
6         "Program".equals(obj.eClass().getName())).
```

```
  after(progObj -> {
    // Access the meta-model to get relevant meta-elements
    EPackage sfunPkg = progObj.eClass().getEPackage();
    EClass importClass = (EClass) sfunPkg.getEClassifier("ImportLib");
    EStructuralFeature nameAtt = importClass.getEStructuralFeature("name");
    EStructuralFeature importRef = progObj.eClass().getEStructuralFeature("imports");

    EList<EObject> allImports = (EList<EObject>) progObj.eGet(importRef);
    for(EObject o : allImports)
      if ( "stdlib".equals(o.eGet(nameAtt)) ) return; // The program already has an stdlib import

    // Create and configure the import object
    EObject importObj = EcoreUtil.create(importClass);
    importObj.eSet(nameAtt, "stdlib");
    allImports.add(importObj);
  });
```

Listing 2: Programmatic configuration to automatically instantiate import statements

The programmatic configuration approach has two main issues. Firstly, facilities like this one (i.e., automatic instantiation of model elements) are not reusable across meta-models since the placement of the extension (i.e., the Program class) is hardcoded in the extension. Secondly, the code to establish the placement of the extension can be cumbersome, as it requires programming at the meta-level[3]. To address these issues we propose an alternative placement control mechanism based on annotations.

5.2. Annotation-based MOP configuration

As an alternative mechanism to programmatic placement, we propose using annotations to identify the concrete model elements over which a particular advice needs to be executed. For illustration, we use the MetaDepth textual language [9], but annotation-based configuration is applicable to other meta-modelling frameworks supporting some kind of annotation mechanism, like EMF. In fact, to demonstrate the generality of the approach we have also implemented it for EMF by using promotion transformations. To avoid repetition, it is available as supplemental material available at `http://miso.es/tools/mop`.

Following the example of the automatic instantiation used before, Listing 3 shows a meta-model described in MetaDepth to represent a language for expressions.

```
@autoInstantiate
Model Expressions {
  abstract Node BooleanValue{}
  @auto(name="trueValue")
  Node TrueVal[1] : BooleanValue{}
  @auto(name="falseValue")
  Node FalseVal[1] : BooleanValue{}
...
}
```

Listing 3: Annotation-based configuration in MetaDepth (automatic-instantiation)

[3]We have used dynamic EMF in the example because it is more general, since it works both with generated and not generated meta-models

The Expressions meta-model is tagged as autoInstantiate, while the TrueVal and FalseVal classes are tagged as auto. The objective is that, by such annotations, every time we instantiate the Expressions meta-model, one instance of TrueVal and another of FalseVal are automatically created, with identifiers trueValue and falseValue respectively. MetaDepth supports class cardinality (indicated by the [1] intervals in lines 5 and 7), so trueValue and falseValue become singleton objects. In both lines, ":" is used to indicate inheritance, therefore both TrueValue and FalseValue inherit from BooleanValue.

Many meta-modelling frameworks (like the EMF) have built-in support for annotations but they are just informal notes that can be attached to arbitrary modelling elements. For example, while in EMF, tools for textual modelling like Xcore[4] facilitate adding annotations to model elements, they are not subject to a predefined syntax, and cannot be applied at the model level. Instead, we need annotations that can be applied at any meta-level and have a formal syntax, so that they can be processed in a type-safe way. Hence, our solution involves defining a meta-model for annotations.

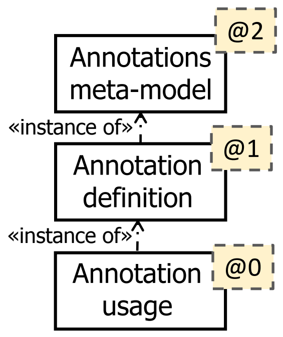


Figure 5: Annotation meta-levels

Conceptually, defining and instantiating annotations spans three meta-levels, as seen in Figure 5. The top-level contains the meta-model for annotations, which is used to define particular annotations at the middle level, while these are used to annotate particular model elements at the bottom level. In the Figure, we decorate each model with its potency [34] (shown after the "@" symbol). The potency (which in MetaDepth is applicable to models, classes and fields) is an integer number that indicates how many meta-levels the element can be instantiated. At every instantiation, the potency decreases, and when it reaches 0, the element cannot be instantiated anymore, becoming a pure instance. The elements at intermediate levels retain both a type and an instance facet, hence they are called clabjects (contracting the words *class* and *object*) [19]. This way, a clabject is an element that both has a type at an upper meta-level (just like objects in two-level modelling) and can be instantiated at a lower meta-level (just like classes in two-level modelling).

An excerpt of our meta-model for annotations is shown in Figure 6. It defines a hierarchy of annotation types, depending on the target element to be annotated: the Model, a Node (corresponding to *clabjects*), an Edge, or a Field. While we take as targets the meta-modelling elements of MetaDepth, this is easily adaptable to other meta-modelling environments. As previously mentioned, MetaDepth is a meta-level agnostic language, so that modelling concepts are uniform at every level. Hence, Model refers both to meta-models and models, Node refers to classes and objects (i.e., to clabjects), PrimField refers to attributes and slots, while References refers to both references and links. Finally,

[4]https://wiki.eclipse.org/Xcore

Edge refers to associations (which may hold fields).

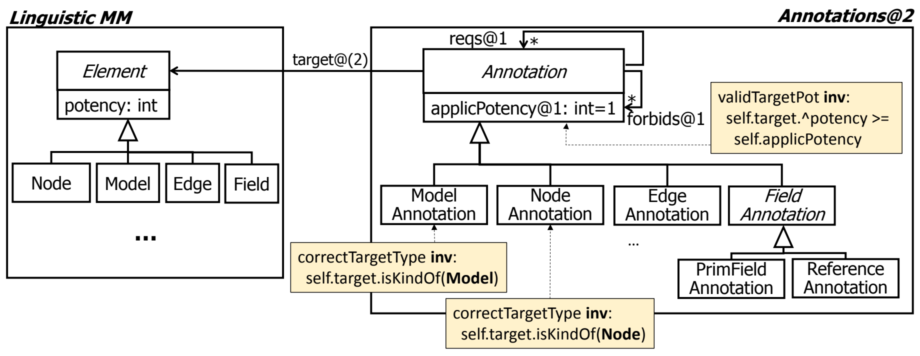


Figure 6: Annotation meta-model

We model incompatibility of annotations (i.e., annotations that cannot be placed on the same element) with the forbids relation. Additionally, annotations may require each other (reqs relation). In our example, we require an autoInstantiate annotation over a Model in order to allow an auto annotation over a Node. This means that auto requires autoInstantiate.

Annotations may declare expectations on the potency of the target element they annotate. This is done with field applicPotency. As this field has potency 1, it needs to be given a value at the next meta-level (when specific annotations are declared). A value of 1 for applicPotency means that the target element should be instantiable at least once. A value of 0 means that there is no need for the target element to be instantiable. Checking that the potency of the target is bigger or equal than the value of applicPotency is performed with the constraint validTargetPot. This constraint has potency 2, and is evaluated two levels below. Constraints in MetaDepth are expressed using the Epsilon Object Language (EOL) [35]. MetaDepth exposes a meta-modelling API (the linguistic meta-model), which can be transparently accessed through EOL (and in fact through any Epsilon model management language [36]). This way, the expression self.target.^potency traverses reference target and accesses property potency, which is defined in the linguistic meta-model. We use the escape character ^ to signal access to the linguistic layer, but its use is optional. However, it needs to be used if there are collisions with equally named properties in the ontological (domain) meta-model.

As seen in Figure 5, the Annotations meta-model is expected to be instantiated at the two next meta-levels. For this purpose, we use the multi-level modelling capabilities of MetaDepth [9], and label the model as having potency 2 (i.e., can be instantiated twice). All clabjects inside the Annotations meta-model have potency 2 as well.

Figure 7 shows a simplified excerpt of the definition of the annotations for automatic instantiation. These definitions are included in model AutoInstantiate, which is an instance of the Annotations meta-model of Figure 6. The AutoInstantiate model has potency 1, so it can be instantiated once more, at the lower meta-level. The model contains two concrete annotations: an annotation (autoInstantiate), applicable to models (an instance of ModelAnnotation), and another one (auto) applicable to nodes (an instance of NodeAnnotation). As previously mentioned, auto requires autoInstantiate.

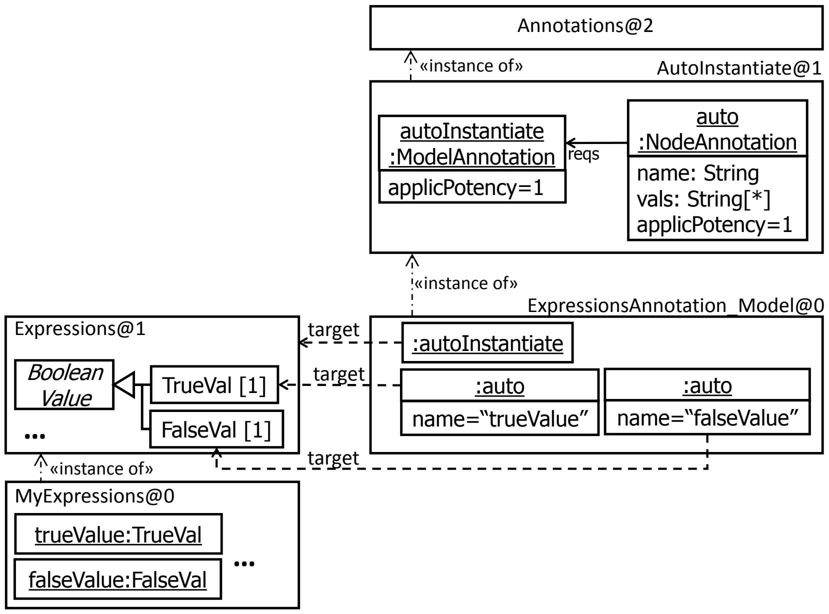


Figure 7: Annotations for auto-instantiation (excerpt)

Annotation definitions, like auto, may define fields. For this purpose, we rely on MetaDepth's support for linguistic extensions [9]. These are elements with no ontological type (but with linguistic type only). This permits extending nodes with new fields, as shown in Figure 7. In the example, auto declares two new fields: name and vals. The former specifies the identifier of the instantiated element, while the latter holds initialization values for the attributes of the instantiated element.

An annotation meta-model like AutoInstantiate can be instantiated in two ways. The first one is using the concrete syntax of the meta-modelling framework, as shown in Listing 3. MetaDepth has a textual syntax, which allows placing annotations on arbitrary elements. These are type-checked with respect to the corresponding annotation meta-model. When parsed, a model is produced, instance of the annotation meta-model (model ExpressionsAnnotation_Model in Figure 7). Alternatively, this model can be input directly, without resorting to the concrete syntax. The elements inside this model contain references, called target, to the annotated elements in model Expressions. These references are automatically set when parsing the concrete syntax representation. Reference target was declared in the Annotation meta-model of Figure 6, as having potency 2 of type *leap* [37]. Leap potency is indicated between parenthesis (target(2) in Figure 6) and means that it has to be instantiated two levels below, but not at intermediate meta-levels. Please note that in the meta-model of Figure 6, we ensure that the target has correct type by the use of constraints named correctTargetType, placed on the subclasses of Annotation (some of them have been omitted for clarity).

Listing 4 shows the annotations meta-model and the complete definition of the annotations for auto-instantiation using MetaDepth syntax.

```
 1 Model Annotations@2 {
 2    abstract Node Annotation {
 3       reqs@1 : Annotation[*];
 4       forbids@1: Annotation[*];
 5       applicPotency@1 : int = 1;
 6       target@(2): Node[0..1];
 7
 8       validTarget : $self.target.^potency >= self.applicPotency$
 9    }
10
11    Node ModelAnnotation : Annotation {
12       correctTargetType: $self.target.isKindOf(Model)$
13    }
14
15    Node NodeAnnotation : Annotation {
16       correctTargetType: $self.target.isKindOf(Node)$
17    }
18    ...
19 }
20
21 Annotations AutoInstantiate {
22    ModelAnnotation autoInstantiate {}
23
24    abstract Node AutoAnnotation {
25        name : String[0..1]; // The name of the created instance
26        vals : String[*]; // The initialization values
27        duplicate : boolean = false; // Whether we allow object duplication
28    }
29
30    NodeAnnotation auto : AutoAnnotation {
31        reqs = [autoInstantiate]; // using auto requires the model using autoInstantiate
32    }
33
34    NodeAnnotation populated {} // Marks nodes owning a reference that needs to be populated
35
36    ReferenceAnnotation populate : AutoAnnotation {
37        obj : Node[0..1]; // The node that needs to be instantiated
38        reqs = [populated];
39    }
40 }
```

Listing 4: Annotations meta-model in MetaDepth (excerpt) and annotations for automatic-instantiation

Lines 21–40 shows a more complete version of the annotations for auto instantiation. In addition to auto, we define an annotation populate applicable to references. They indicate objects that need to be created and added to such reference. Both auto and populate share three parameters (name, vals and duplicate) and so they inherit from the abstract node AutoAnnotation. The duplicate flag signals whether existing objects may be reused or they need to be newly created. Annotation populate needs to indicate the node that is to be instantiated (field obj in line 37), and the container class needs to be annotated with populated. This annotation simply marks that some owned reference needs to be populated upon instantiation of the class.

Looking back at the example in Listing 2 we rewrite it in MetaDepth using annotations (Listing 5). The populate annotation would allow creating an ImportLib object with name "stdlib", in the imports reference of a newly created Program object. This would be done by annotating the Program.imports with @populate(obj=ImportLib, vals=["stdlib"]), and the class Program as @populated. In this case, the duplicate flags avoids re-creating unnecessary stdlib ImportLib objects when e.g., loading an existing model where the stdlib ImportLib has already been created.

```
@autoInstantiate
Model SFun {
  @populated
  Node Program{
     name : String;
     functions : Function[*]
     @populate(obj=ImportLib, vals=["stdlib"])
     imports : ImportLib[*]
  }

  Node ImportLib {}
  Node Function {}
...
}
```

Listing 5: Example of automatic instantiation for import statements MetaDepth

This approach to define annotations greatly benefits from a multi-level framework, but it could also be emulated in a two-level framework like EMF. In this case a mechanism, like promotion transformations, would be needed [15] to emulate the three meta-levels that Annotations span. A promotion transformation is a model-to-model transformation that takes a model and outputs a meta-model (so that it can be instantiated again).

5.3. Comparison of approaches

As a summary, next we briefly discuss on strengths and limitations of both approaches to placement control, in order to clarify for which scenarios each one works better.

The programmatic API is the right choice when the extension needs to act over all model elements affected by a given event, independently of their type. For example the ResourceType extension in Listing 1 cannot be configured using annotations, since it acts over all elements in a model.

The programmatic access is also useful when the meta-model to be annotated is not available for modification. For example, we may want to apply the automatic instantiation scenario to UML, in order to automatically import the primitive types library. This cannot be done using annotations because the UML meta-model is standard, and is frequently made available to tools in a read-only way. The programmatic API is the most general mechanism, as one can always hard-code the placement control, although it is likely that this kind of usages are only accessible to advanced programmers, while it may result in meta-model specific, non-reusable code.

The placement control based on annotations is more appropriate when the model elements that need to be exposed to the MOP are known at design time. By annotating these elements we can configure the extension to make it reusable, because the annotated elements act as parameters for the advice. In comparison, in a programmatic approach like the one in Listing 2, the advice code is dependent on the specific elements to be instantiated, like ImportLib. This means, the advice cannot be reused for other meta-models.

Overall, while annotations lead to an *extensional* placement control approach (each element must be manually annotated), the programmatic access is *intensional* (a query is written to select them). Both approaches are useful and serve complementary purposes. In general, the programmatic API approach should be used when the configuration is global (i.e., for any model or meta-model), whereas annotations should be used to define reusable extensions for different meta-models by defining the model joint points as annotations.

6. Advice mechanisms

Advices can be attached to the meta-modelling join points explained in Table 1, and be executed on selected modelling elements, using the placement control mechanisms described in Section 5. As seen in Figure 1, we consider two styles of advices: using general purpose languages and using model management languages. They are explained in Sections 6.1 and 6.2.

6.1. Programmatic advices

A programmatic advice is written in the programming language in which the meta-modelling framework is implemented. For instance, in our EMF implementation we write programmatic advices in Java. The advice written in Listing 1 (lines 4–5) makes use of an auxiliary Java method (getAllPackages) and also uses the ModelIndex class to make the result available to other MOP extensions or Eclipse plug-ins. In this case, a programmatic advice is adequate because the extension does not perform a "pure" model management task (i.e., like a model-to-model transformation, or an in-place model modification).

In contrast, the advice presented in Listing 2 (lines 8–21) is a model management operation (an in-place model modification) written in Java using the EMF dynamic API. For this kind of model manipulation it is more adequate to use a dedicated model management language. Hence, our approach considers the integration of this type of languages in the MOP.

6.2. Model management advices

Depending on the action to be performed by the advice, using a model management language may be more appropriate than an advice written using a GPL. Figure 8 shows a meta-model for advices, which complements the annotation based-placement mechanism described in Figure 6. The figure also shows how advices are instantiated, using the auto instantiation example as illustration.

The meta-model (named Advices) is aimed to be instantiated once, hence it has potency 1. Advices can be implemented using different model management languages, shown by the Advice subclasses. An Advice has a reference to the associated annotation, some triggering EventType, an indication of its execution point (when), and the file where the model management operation is located.

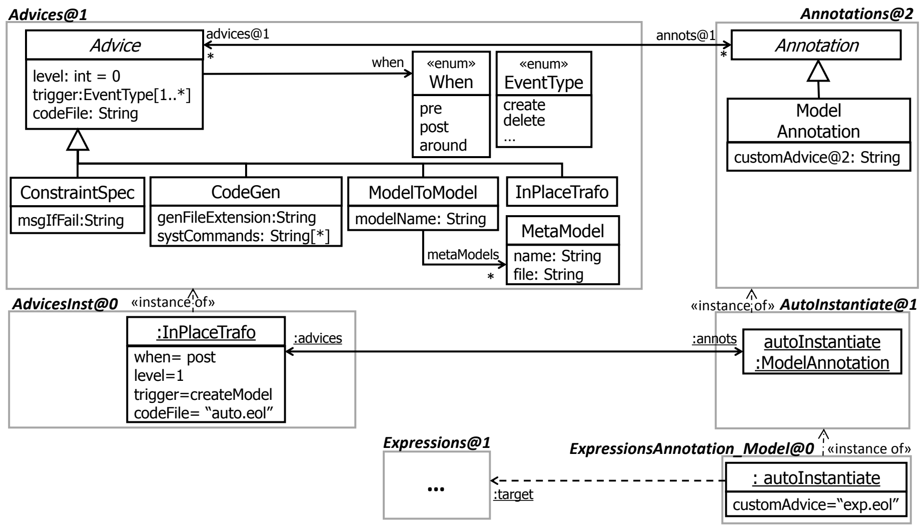


Figure 8: Meta-model for advices, interaction with Annotations, and overriding a default advice

Advices implemented with different model management operations need different properties. For example a ConstraintSpec advice is specified using a constraint language (e.g., OCL, EOL) and provides a message if the constraint fails. A code generation advice (CodeGen) is specified using a template language (e.g. Acceleo, EGL) and holds the extension of the file to be generated, and further system commands (e.g., for compiling the generated code). A model-to-model transformation advice is specified using a model-to-model transformation language (e.g., ATL, ETL) and needs the name of the model to be created, and information of any further meta-model required by the transformation (so that they can be loaded if not in memory). In MetaDepth, all Epsilon languages [36] for model management are available, and can be used to specify advices. This way, we use the Epsilon Object Language (EOL) [35] for specifying constraints and in-place transformations; the Epsilon Generation Language (EGL) [38] for code generation advices, and the Epsilon Transformation Language (ETL) [39] for model-to-model transformation advices.

Additionally, advices have a level. This number indicates at how many meta-levels below the advice has to be executed. If 0 (the default) it will be executed over the elements the annotation is attached to. In the figure, it means the advice would be executed over the Expressions model, because it has been annotated with autoInstantiate. However in our case we need to execute the advice on instances of Expression, and so the advice has level 1.

Listing 6 shows the advice attached to the auto annotation. The advice is expressed in EOL and performs the automatic instantiation of the annotated node. The main operation is to be executed on instances of Expression. In line 5, the advice iterates over all nodes that need to be instantiated. Such nodes are obtained through operation annotatedNodes (lines 12-14). This operation selects the nodes that are annotated with auto, and which need to be instantiated (operation needsCreating, whose code is not shown). Model represents the current model, while type accesses the model type. For each node that requires instantiation, an instance is created (line 6), then its identifier (line 7) and the initial attribute values (line 8) are set.

```
 1 // auto.eol (default advice for the auto annotation)
 2 import 'InstCommon.eol'
 3
 4 operation main() : Boolean {
 5    for (n in annotatedNodes()) {
 6          var node = Model.createInstance(n.^name);
 7          node.^name = n.getAnnotation('auto').get('name');
 8          node.setValues(n.getAnnotation('auto').get('vals'));
 9    }
10 }
11
12 operation annotatedNodes() : Sequence(Node) {
13     return Model.^type.children−>select( c | c.hasAnnotation('auto') and needsCreating(c, 'auto'));
14 }
```

Listing 6: Default advice for automated instantiation (auto annotation)

Please note that EOL allows mixing both imperative constructs (e.g., the for loop in line 5) with declarative iteration (e.g., the select construct in line 13). EOL can be used as an advice language as it can reflectively access the MetaDepth API to e.g., obtain the type of a model (line 13), create a node instance using a String containing its type name (line 6), change the identifier of a node (line 7), and obtain the annotations of an element (lines 7 and 8). Moreover, EOL permits the user to enrich the API offered by MetaDepth by adding new methods on classes of its API. Both operations main and annotatedNodes are declared on a global scope, but they could also be declared on classes of the domain meta-models involved, or on the classes of MetaDepth's base API. For example the method setValues (line 8, code not shown) has been declared on the context of the Node class, part of the MetaDepth API. This reflective capability is present in all Epsilon languages through Java reflection. Please note that common operations for the auto and the populate annotations have been extracted to the InstCommon.eol library, imported in line 1.

If we need more sophisticated instantiation mechanisms or we require some kind of interconnection between objects, the @auto annotation and the default advice of Listing 6 may not be enough. Therefore, the developer of a certain meta-model has the option to provide its own in-place transformation code as an advice, so that it overrides the default advice. Hence, every model annotation can include the parameter customAdvice=..., pointing to a model management program. In Figure 8, this is realized with field customAdvice in ModelAnnotation. The field has potency 2, so that a value can be given on particular usages of the model annotation.

As an example, let us assume we need to instantiate both TrueVal and FalseVal and connect them to the instance of BooleanType. Then, the meta-model designer would specify the custom advice shown in Listing 7.

```
// exp.eol (custom advice), overrides auto.eol (default advice)
operation main() {
   var tv = new TrueVal;
   var fv = new FalseVal;
   var bt = new BooleanType;
   tv.dataType := bt;
   fv.dataType := bt;
   ...
}
```

Listing 7: Custom advice for automated instantiation of Expressions

7. Implementation approaches

This section describes implementations of our proposed MOP concept using MetaDepth (Section 7.1) and EMF (Section 7.2). We use the feature model in Figure 2 as a guide to present both approaches. Implementation artefacts for MetaDepth and EMF, as well as screencasts and installation instructions are available at `http://miso.es/tools/mop`.

7.1. Implementation in MetaDepth

For the implementation of MOPs in metaDepth, we have relied on the possibility to modify its source code (feature JoinPoints.BaseAPIMethods.Internal in the feature model of Figure 2). MetaDepth already produced meta-modelling events, using the *command* design pattern, but no custom code could be attached on them. Therefore, we enhanced its extensibility by identifying join points in its codebase, and taking the Advices meta-model of Figure 8 as a basis for its identification, which relies on the Annotations meta-model of Figure 6. As previously mentioned, MetaDepth is integrated with the Epsilon model-management languages [36], which we use to specify advices.

Figure 9 shows a scheme of how we incorporated MOPs within MetaDepth. MetaDepth follows an interpreted approach and is level agnostic, reifying uniformly the meta-modelling concepts at any meta-level (see the linguistic meta-model in the figure). This meta-model contains notions like Model, which can be nested, Node (elements with both type and instance facet), and Edge (bidirectional associations). All elements inherit from abstract class Element, which defines the potency, cardinality (minimum, maximum), name, and optional type of the element.

The base API includes the linguistic meta-model and a set of additional classes (marked as "runtime"; bottom-left corner). The figure shows some methods of the API that are used as join points to produce events (marked with the jp stereotype), like createNode (in ModelFactory), execute (in Verify) or setValue (in QualifiedElement). The list of join points is shown in Table 1. Please note that, as

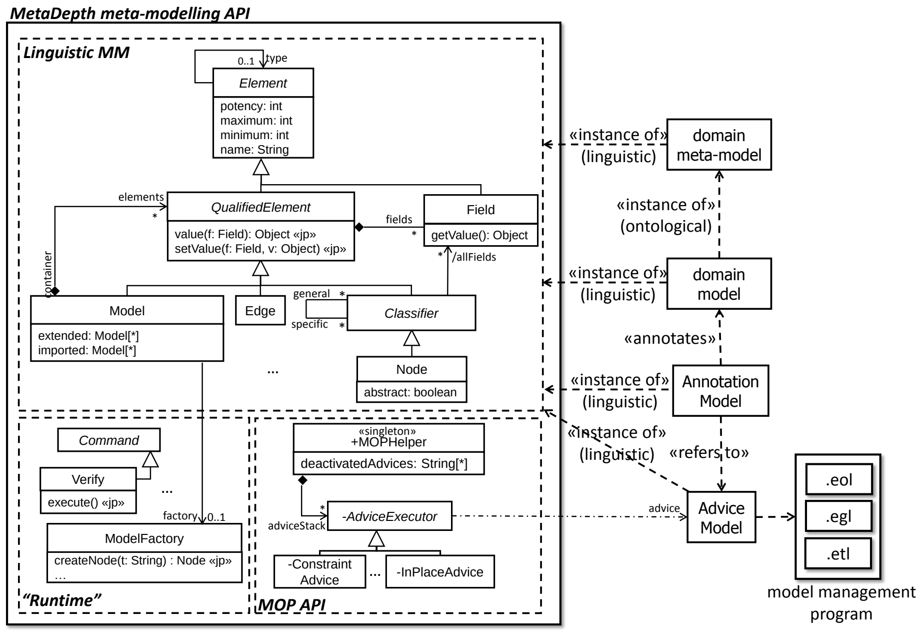


Figure 9: Scheme of the MetaDepth *open* meta-modelling framework

we use a meta-model for advices, which has enumerations for the meta-modelling event to be captured, the user making the extension does not need to know the specific API method that needs to be adviced. Instead, as explained before, the meta-modelling joint points are fixed and are made available as event types.

Advices can be written using the Epsilon languages. This is possible because those languages rely on Java reflection when using methods which do not belong to the Epsilon framework itself. Hence, the MetaDepth API is implicitly exposed.

The MOP API includes infrastructure to execute advices of different types, as well as a singleton class MOPHelper, which maintains an index with the active advices, and offers an API for level and temporal control of the MOP. Such API permits to activate/deactivate an advice, or to check whether the system is executing an advice. Such object is exposed and can be used from the Epsilon languages, while the classes to execute the different types of advices are hidden to the user.

7.2. An EMF-based implementation

We have also implemented the architecture described in the previous sections for EMF. To this end we have used AspectJ to inject the behaviour required to make explicit the occurrence of the events of interest. Alternatively, we could have modified EMF's source code, but this would make our implementation incompatible with most EMF installations. Please note that the built-in notification mechanism of EMF is not enough to implement a MOP since only a few events are available (e.g., instantiation and model loading notifications are not considered in EMF) and they are only notified after they have happened. Moreover, it would not be possible to install event handlers globally (e.g., for any feature set) since notification listeners are installed per model element or EMF resource.

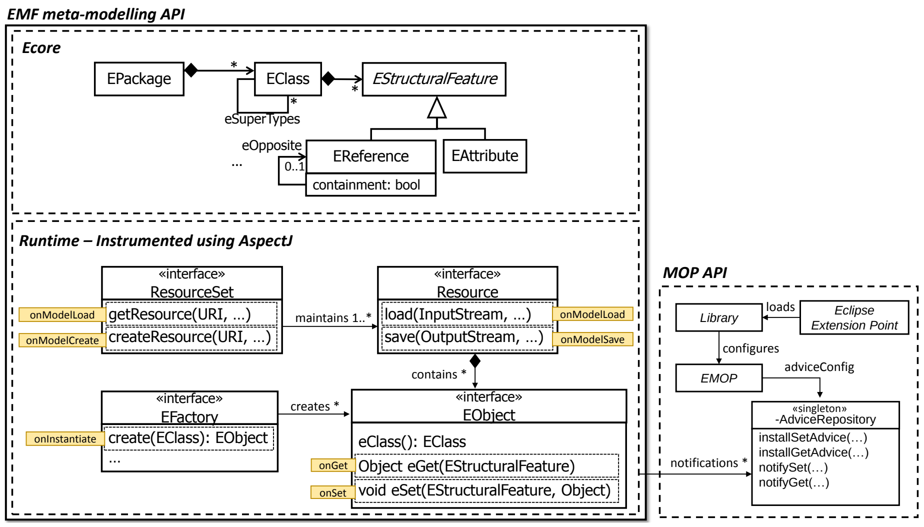


Figure 10: Architecture of our implementation for EMF

Figure 10 shows the architecture of our solution. We package a concrete MOP configuration into a library. A library is integrated into Eclipse using an extension point, and can be activated or deactivated by the user. The library is in charge of checking the presence of annotations if needed and setting up the EMOP object on the required events using the fluent API presented in Section 5.1. The advices configured with the EMOP API are installed in a global context (AdviceRepository). To enable the notification of events to the AdviceRepository we have instrumented relevant parts of EMF using AspectJ. The "Runtime" part of Figure 10 shows some of the methods that we have advised with additional behaviour to notify about the execution of the meta-modelling events.

8. Evaluation

We have evaluated our approach by extending EMF and MetaDepth with modelling services not foreseen initially when they were created. A summary of them is shown in Table 2. The table lists the extension name, a brief description, the approach used for the advice (programmatic, in place, code generation, constraint, model-to-model) the meta-modelling join point where the advices are activated, and the platform where they are available.

Table 2: Extension libraries available in our implementations

Name	**Description**	**Type**	**Join Point**	**Platform**
Auto-instantiate	Provides annotations to indicate that some classes have to be automatically instantiated	In-place	Create Model, Instantiate object	EMF, MetaDepth
Class cardinality	Provides annotations to set the maximum number of instances of a type	Constraint	Validate, Instantiate object	EMF
Disable validation	Removes diagnostics related to annotated elements	Program.	Validate	EMF
Prototype	Enables a form of prototype-based modelling	Program.	Get feature	EMF
Resource type	Maintains an index of loaded meta-models and their meta-models	Program.	Load model, Set feature	EMF
Compute subtypes	Fills a "subclasses" reference for each EClass	In-place	Set feature	EMF
Static semantics library	Library of OCL constraints expressed as annotations, e.g., to model composition, link commutativity, xor of links, etc.	Constraint	Validate	MetaDepth
Meta-model/model co-evolution	Signals if changes in a meta-model affect the conformity of existing models and fixes the models	Program.	Load/save model, Set feature	EMF
Multi-level facilities library	Library of different multi-level modelling facilities (powertypes, deep cardinalities, etc)	In-place, Constraint	Create model, instantiate object	MetaDepth
Bottom-up modelling	Permits creating untyped objects, and construct their types a-posteriori	In-place	Create model, instantiate object	MetaDepth
Type object	Annotations to indicate the roles of the type-object pattern and automatic promotion transformations	M2M	Model save, Validation	EMF
Visualization	Visualizes a model using graphviz	Code gen.	Create Model	MetaDepth

The implementation of these services demonstrates the usefulness of our proposal to address unforeseen extensions. In the following we describe in more detail some representative extensions for MetaDepth (Section 8.1) and EMF (Section 8.2), and finish in Section 8.3 with a comparison of the techniques used, a discussion of strengths and limitations of the approach and the degree of the fulfilment of the requirements for open meta-modelling frameworks suggested in Section 2.1.

8.1. *Extending* MetaDepth

We illustrate advices using different model management languages, selected among those of Table 2. In particular, we show a bottom-up meta-modelling facility that permits generating types after the instances using in-place transformation advices (Section 8.1.1), and a deep cardinality facility that extends cardinality checking to several meta-levels below using a constraint advice (Section 8.1.2).

8.1.1. Bottom-up meta-modelling (an in-place transformation advice)

In mainstream meta-modelling, types are created before their instances. However, some modelling scenarios would benefit from creating example instances first, and then deriving types from the shape of those instances. This approach is called bottom-up or example-based meta-modelling [40]. While MetaDepth permits creating untyped objects (so called linguistic extensions), it does not natively support the automated creation of a type given a prototype instance. Our goal with this extension is to enable this functionality.

Listing 8 shows on the left (lines 1-18) a meta-model and an instance model. The instance model has an untyped object named Pet (lines 9–11), for which we want to create a type. Similarly, the object Juan has an untyped field age (line 16), for which we want to create a field type, too. Fields with no type (at the higher meta-level) but just a value can be created by declaring a datatype the value is expected to conform to, and a value. Our approach is to create a set of annotations (createType, createFieldType) applicable to nodes and fields. An advice will then become activated on such annotations, creating the corresponding types (see lines 22-31 of the Listing) and removing the original annotations.

```
 1 Model Persons {
 2   Node Person {
 3     name : String;
 4   }
 5 }
 6
 7 Persons MyModel {
 8   @createType
 9   Node Pet {
10     kind : String = "canary";
11   }
12
13   Person Juan {
14     name = "Juan";
15     @createFieldType
16     age : int = 25;
17   }
18 }
19
20
21
```

```
22 // Models after advices are processed
23 Model Persons{
24   Node Person {
25     name:String;
26     age:int=25;
27   }
28   Node Pet {
29     kind:String="canary";
30   }
31 }
32
33 Persons MyModel{
34   Pet PetInst {
35     kind="canary";
36   }
37   Person Juan {
38     name="Juan";
39     age=25;
40   }
41 }
```

Listing 8: Example model (left). Model after advices are processed (right)

The right of Listing 8 shows the models once the advice has been processed. As a result, a new node Pet is created in the meta-model (lines 28-30) with a kind field, which takes the default value from the instance. Similarly, an age field is created in node Person (line 26). At the model level, Pet is automatically renamed to PetInst and made an instance of Pet, while Juan.age becomes an instance of Person.age.

The advice is expressed as an in-place transformation with EOL, an excerpt of which is shown in Listing 9. Operation main is the advice entry point, which is called when a node is annotated with createType. The annotation clones the node using the createType helper operation, defined in lines 10-16 in the context of Node, and then sets the type of the instance to the created node (line 6), removes the annotation (line 7) and changes the identifier of the instance. The createType operation (lines 11-16) clones the node and its fields, increasing their potency; while operation setTypeTo (code omitted) sets the type of the context node, including all its fields. The advice illustrates the transparent access to the methods of the MetaDepth API (e.g., operation clone() and removeAnnotation()) and the features of the meta-classes of the linguistic meta-model (e.g., features potency and container).

```
1  /* Advice attached to createType annotation */
2  operation main(node: Node) {
3    var clone := node.createType();
4    var modelType := node.container.type;
5    modelType.addChildren(clone);
6    node.setTypeTo(clone);
7    node.removeAnnotation('createType');
8    node.name(node.name+'Inst');
9  }
10
11 operation Node createType() : Node {
12   var clone := self.clone();
13   clone.potency := self.potency+1;
14   for (f in clone.fields)
15     f.potency := f.potency+1;
16   return clone;
17 }
```

Listing 9: EOL advice (excerpt) for bottom-up meta-modelling

The alternative to a MOP-based mechanism to implement this facility would have been to extend MetaDepth with a new console command. However, this approach is intrusive, requiring the modification of MetaDepth itself. Instead, using a MOP approach, the extension can be defined externally, using EOL, and shared with other engineers.

8.1.2. Deep cardinality checkings (a constraint advice)

In standard two-level modelling, reference cardinalities apply to the next meta-level below. In multi-level modelling, we might be interested in defining cardinalities applicable to instances of instances of a reference, i.e., to meta-levels beyond the next one. Some works, like [23] have recently proposed a notion of cardinality that is decorated with a potency, indicating at which meta-level the constraint needs to be evaluated. This feature is not available in MetaDepth, and hence will be defined as an extension.

The models in Listing 10 illustrate the need for this feature. This example is a slight variation of the example presented in [23]. This way, the top-most model (named CarTypesModel) is used to define types of cars, which then can be subsequently instantiated to define specific cars. Hence, we declare a CarType which has two WheelTypes. A specific CarType can be automatic (line 6), while instances of instances of WheelType have a serial number (line 9). The CarTypesModel is instantiated in lines 13-21. The SomeCarTypes model declares a Beetle car type, which is not automatic and which instantiates reference wheel twice: front and rear. Please note that the default cardinality of wheel (2..2) applies to this level, and is satisfied. In their turn, front and rear define their own cardinality (2..2). Both front and rear refer to SteelWheel, which is an instance of WheelType. A Beetle car may have a flowerHolder, which is declared as an untyped field in line 18.

Finally, the bottom-most model (named MyCar, in lines 23-42), defines a specific Beetle car, named B53, which has a flowerHolder and two specific SteelWheels instances (each with a specific serial number) at the front and the rear.

```
Model CarTypesModel@2 {
  Node CarType {
    @deepCardinality(potMin=2, potMax=2,
                     min=4, max=4)
    wheel : WheelType[2];
    automatic@1: boolean;
  }
  Node WheelType {
    serialNo : String;
  }
}

CarTypesModel SomeCarTypes {
  CarType Beetle {
    automatic = false;
    front : SteelWheel[2] {wheel};
    rear : SteelWheel[2] {wheel};
    flowerHolder : boolean;
  }
  WheelType SteelWheel {}
}

SomeCarTypes MyCar {
  Beetle B53 {
    flowerHolder = true;
    front = [w1, w2];
    rear = [w3, w4];
  }

  SteelWheel w1 {
    serialNo = "FSTN2017_3141516";
  }
  SteelWheel w2 {
    serialNo = "FSTN2017_3141517";
  }
  SteelWheel w3 {
    serialNo = "FSTN2016_3141516";
  }
  SteelWheel w4 {
    serialNo = "FSTN2016_3141517";
  }
}
```

Listing 10: Models with deep cardinality annotations

At the top-most level, we would like to express that, two levels below, every instance of instance of CarType needs to have exactly four wheels. However, this cannot be ensured by standard cardinalities, since they apply to the next level below only. Hence, we have designed annotation deepCardinality to perform this task. It defines an interval of levels potMin...potMax where the cardinality should be applicable, and a cardinality interval min...max. A level interval 1..1 is equivalent to standard potency. The interval of the listing (2..2) checks that collectively, all instances of instances of wheel have 4 elements, which is satisfied by the example. Alternatively, the parameter collective=false in the annotation would make the check individually on B53.front and B43.rear.
Listing 11 shows an excerpt of a constraint advice, expressed with EOL. The entry point is operation main, which receives the node in which the annotation is placed (parameter target) and the field name (parameter refName). The operation obtains the annotation (which is not located in the direct type, but several levels above) using method getOwnedIndirectAnnotation, and then the value of each annotation parameter. Then, if the level of the target node is between the expected interval, it checks the cardinality either collectively (line 14) of individually (line 15). The code for these methods is omitted for simplicity.

```
1  operation main(target: Node, refName : String) : Boolean {
2    var annotation = target.getOwnedIndirectAnnotation('deepCardinality', refName);
3    var potMin = annotation.get('potMin');
4    var potMax = annotation.get('potMax');
5
6    var min = annotation.get('min');
7    var max = annotation.get('max');
8
9    var collective = annotation.get('collective');
10
11   var levels = target.getLevelsFromOwningAnnotationDefi('deepCardinality', refName);
12
13   if ( potMin<=levels and (levels<=potMax or potMax==−1) ) {
14       if (collective) return target.checkCardinalityCollective(refName, min, max);
15       else return target.checkCardinalityIndividual(refName, min, max);
```

```
    }
    return true;
}
```

Listing 11: EOL constraint advice (excerpt) for deep cardinalities

The alternative to a MOP-based mechanism to implement this facility would have been to modify MetaDepth's textual language syntax and semantics, to incorporate deep constraints. This would require complex modifications within MetaDepth's source code, and might even cause (syntactical) incompatibilities with previous MetaDepth's versions. Instead, a MOP mechanism permits an external implementation of the semantics, and the syntactical expression of the new facility using annotations (and hence avoiding the modification of the language syntax). This way, a core meta-modelling language can be grown and adapted by incorporating new annotations.

8.2. Extending EMF

This section presents a case study related to the use of a MOP to extend EMF with support for (meta-)model co-evolution. As an additional example, the supplemental material contains the full description of the use of our MOP to implement multi-level modelling in EMF via the type-object pattern and promotion transformations (see `http://miso.es/tools/mop`).

8.2.1. Extending EMF with co-evolution capabilities

Model co-evolution upon meta-model changes is a widely studied theme in MDE research [41]. Actually, it is a recurring theme in other areas, like object-oriented databases [42], relational databases, ontologies and XML schemas [43], where long-lived data needs to become consistent with their types when these evolve. There exists a variety of tools which target the migration of models or other MDE artefacts when a meta-model changes [44]. However, these tools are not automatic. They require copies of both the original meta-model and evolved metamodels and need to be activated on demand by the user, on each non-evolved model.

As a concrete example, a well-known annoyance of EMF is that deleting an EClass in an .ecore meta-model invalidates all .xmi files containing instances of such an EClass. As noted before, migration tools are useless if the user does not take care of having copies of the meta-model and every possibly affected model. Thus, there is an underlying technical problem related to the fact that EMF does not have any service to perform automatic co-evolution.

In this scenario a MOP facility like ours enables the creation of migration services without the need of modifying the meta-modelling framework, EMF in this case. In order to assess the usefulness of our approach to design complex services like this, we have implemented a co-evolution prototype for EMF. This case study describes the design space for this problem and the implementation on top of EMOP. Please note that the purpose of the case study is not to provide a complete solution to the co-evolution problem, but to show how a MOP-based approach facilitates its implementation and integration within EMF.

To create the co-evolution library we need to take into account the following concerns.

- EMF does not have a specific notion of model or meta-model, but there are just resources containing model elements. Thus, a mechanism is needed to relate resources according to their conformance relationship.
- A resource can be loaded in memory or serialized. The same serialized (meta-)model can be loaded multiple times in memory at the same time.
- Changes between two versions of the same resource can be gathered by tracking editing changes to the resource or by comparing the new and the old version of a resource using comparison tools like EMF Compare.

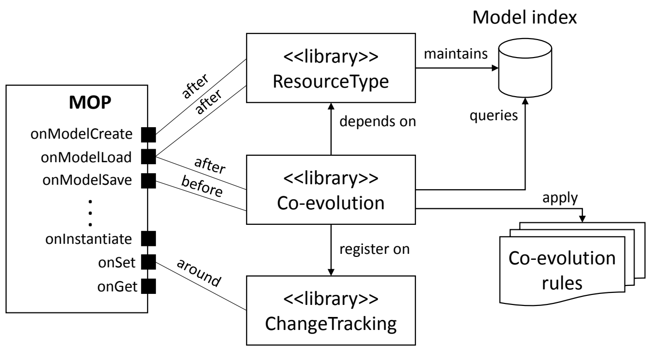


Figure 11: Architecture of the co-evolution library

Given these constraints we opt for splitting the problem into three MOP libraries. The architecture is depicted in Figure 11. We define a library called ResourceType which detects which resources contains the EClasses required by a loaded or newly created resource. This library maintains a model index that describes the conformance relationship between resources. Such model index can be queried by other libraries to determine which is the meta-model of a given model, or which models conform to a given meta-model. We also define another library called ChangeTracking that detects modifications in a resource and allow listeners to be notified about them. This library uses around to extend the onSet event so that a client can decide to reject a change (e.g., in our case when a change breaks conformance). Please note that both ResourceType and ChangeTracking are useful libraries on their own and can be used to build other services on top of them.
The co-evolution library is built on top of these two reusable libraries. In addition the library makes use of an extensible repository of co-evolution rules. Given a meta-model change notified by ChangeTracking the library behaves as follows:

1. *Select co-evolution rule.* The system looks up the rule repository in order to identify a suitable co-evolution rule. We classify rules as non-breaking,

breaking and resolvable and breaking and non-resolvable (i.e., when user intervention is required) as in the taxonomy of Cicchetti and collaborators [41].

2. *Handle loaded models.* Query the model index to get all models conforming to the changed meta-model which are currently loaded. For each model, we identify elements affected by the co-evolution rule. For each element, the rule generates one or more change proposals. If there is only one proposal and can be automatically applied, the system does so. If there are more than one or requires user intervention, the user is asked for information.

3. *Handle persisted models.* This step is similar to the previous one, but now the system looks in the workspace for .xmi files conforming to the changed meta-model. Then, it loads the file with the old meta-model and performs the co-evolution as before. An alternative is to generate an update script so that the user can apply it manually, for instance to other models not stored in the workspace.

Listing 12 shows an small excerpt of the implementation of the live co-evolution scenario, focussing on how the MOP is used to install the main co-evolution callback (lines 5–7), which delegates the task of detecting changes to the ChangeTracking library (lines 17–19). This library notifies about model changes which are used by processChange method (lines 25–27) to perform the actual co-evolution task. The implementation of ChangeTracking consists of installing a callback around set events, so that each time a model feature changes the corresponding change object is computed and the MetamodelEvolution library is notified.

```
 1 public class MetamodelEvolution extends GlobalLibrary {
 2
 3   /** Process is called automatically when the library is activated
 4        by the user */
 5   @Override public void process(EMOP mop) {
 6     mop.onModelLoad().after(r -> load(r));
 7   }
 8
 9   private void load(Resource r) {
10     if ( ! isMetamodel(r) )
11       return;
12
13     /* Create a new MOP specific to the source */
14     EMOPResource emop = EMOP.resource(r);
15
16     /* Activate the change tracking library for the resource */
17     new ChangeTracking().
18         onChange(this::processChange).
19         process(emop);
20   }
21
22   /* Given a change, locates all models affected by the change
23      using a query over the ModelIndex provided by the ResourceType
24      library, and then apply co-evolution rules */
25   private boolean processChange(Change change) {
26     ...
27   }
28 }
29
```

```java

public class ChangeTracking extends LocalLibrary {
  private ArrayList<ChangeCallback> callbacks = new ArrayList<>();

  public ChangeTracking process(EMOPResource mop) {
    mop.onSet().around(this::notifyChanged);
    return this;
  }

  private void notifyChanged(EStructuralFeature f, EObject obj, Object new_, Object old_) {
    Change c = computeChange(f, obj, new_, old_);
    for (ChangeCallback callback : callbacks) {
      if ( ! callback.apply(c) ) {
        throw new InterruptAround();
      }
    }
  }
}
```

Listing 12: Definition of the co-evolution library

Figure 12 shows a screenshot of the tool. It records the changes made to the meta-model and computes co-evolution proposals, so that the user can interactively select how to fix the co-evolution issue.

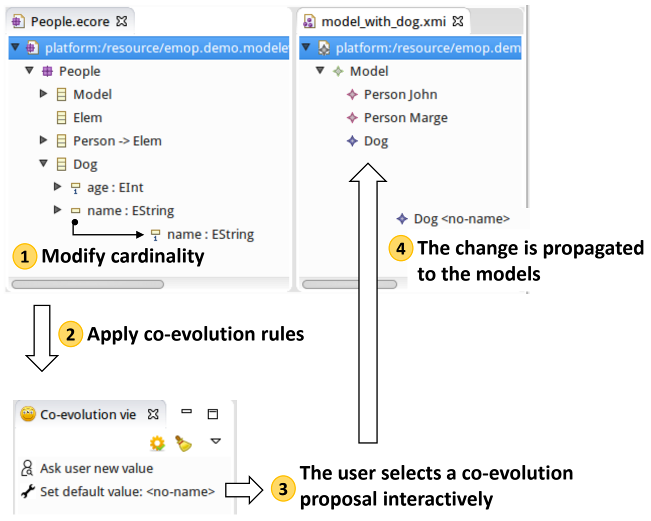

People.ecore
platform:/resource/emop.demo.modele
People
Model
Elem
Person -> Elem
Dog
age : EInt
name : EString
name : EString
1 Modify cardinality
model_with_dog.xmi
platform:/resource/emop.dem
Model
Person John
Person Marge
Dog
Dog <no-name>
4 The change is propagated to the models
2 Apply co-evolution rules
Co-evolution vie
Ask user new value
Set default value: <no-name>
3 The user selects a co-evolution proposal interactively

Figure 12: Screenshot of the prototype co-evolution tool

In order to assess the practical applicability of our implementation, we have applied it to simulate the evolution of the AUTOSAR meta-model implemented by Artop (AUTOSAR Tool Platform User Group; `https://www.artop.org/`) between versions 2.2.0 and 3.0.0. These are large meta-models, with 746 and 675 classes respectively, more than 1,500 features and around 10,000 EAnnotations. Version 3.0.0 is a refactored version to make the meta-model smaller, which required 8,642 changes (according to EMF Compare). It introduces many non-breaking changes (i.e., the modification or deletion of EAnnotations which describe meta-data) but it also contains a number of breaking and resolvable

changes like the removal of classes (e.g., SwCalprm and SwCalprmRef which are classes to represent calibration parameters). To keep models consistent with the new version, we implemented a rule to automatically remove the corresponding EObjects, and another rule to handle meta-property modifications. Overall, our MOP was able to handle the evolution of real-world meta-models seamlessly, since it is built by intercepting calls to the regular EMF infrastructure.
In conclusion, the implementation of this case study required developing all the co-evolution machinery (the essential complexity of the problem), but our MOP approach made it possible to integrate it with EMF smoothly in a way that was not possible before.

8.3. Discussion

In this subsection we first evaluate our implementations against the requirements set in Section 2.1 (Section 8.3.1), discuss on efficiency considerations (Section 8.3.2) and argue on workarounds and effort needed to achieve similar functionality without MOPs (Section 8.3.3).

8.3.1. Summary and lessons learnt

We have realized the concept of *open meta-modelling framework* with two different implementations, and we have used them to create several MOP-based facilities. Based on this experience, we can now evaluate the implementations against the requirements set in Section 2.1, and reflect on the lessons learned which we hope are useful for the reader. The way the requirements have been addressed is explained in Table 3.

Table 3: Approaches to address the requirements for open meta-modelling frameworks

	MetaDepth		EMOP	
	Implement.	**Remark**	**Implement.**	**Remark**
R1: API	MetaDepth API, integrated MOPHelper	Source code modified	EMF API, Fluent EMOP API	Created external library, which used AOP
R2: Events	Internal	Instrumented code so that advices can be added	External, AOP	Based on AOP mechanisms
R3: Extensions	Implicit, annotations. Model-based	Advices are activated by annotations	User Interface	Explicit manual activation
R4: Selection	Annotations	Native	Programmatic, annotations	Combination of EAnnotations and promotion transformation
R5: Advice Languages	Model management (Epsilon)	Access to the API through reflection	Java	Languages like ATL, Acceleo or Epsilon could be integrated if the developer writes the appropriate glue code

Both implementations realize the same conceptual model but in different ways. Regarding the meta-modelling API, both are built on top of existing APIs. However, while the MetaDepth source code was modified to integrate MOP facilities, the EMF framework was left untouched and the EMOP was implemented externally using AspectJ.
Regarding requirement 2 (events), both implementations offer mechanisms to add advices on relevant events. In the case of MetaDepth, these are selected using the EventType enumeration (see Figure 8), while in EMOP they are selected based on the on⟨event⟩ methods of the EMOP API (see Figure 3).
Extensions (requirement 3) can be loaded on demand in both cases: explicitly in the case of EMOP (through a user interface), or implicitly (due to annotated models) in the case of MetaDepth. The way to configure an extension is model-based in case of MetaDepth (i.e., using the Advices and Annotations meta-models), while a Java fluent API must be used in case of EMOP.
The elements over which the extensions need to be executed (requirement 4) are selected through native annotations, and also programmatically in case of EMOP. Annotations are native in MetaDepth (but built explicitly for this work), while they have been emulated in EMF through promotion transformations. One limitation of the EMF solution is that models can only be annotated via external models (i.e., a regular EObject does not support EAnnotations). Another drawback of using a promotion-based approach is that it requires code generation to enable the manipulation of the promoted model. On the other hand, a current limitation of MetaDepth is the lack of an API that permits global advices. As presented in Section 5 both approaches, annotation-based and programmatic, are useful and required in a complete MOP.
Regarding languages to specify advices (requirement 5), they are natively implemented in MetaDepth using MDE artefacts like transformations and code generators. This is possible because it is an integrated environment, and the Epsilon languages can resort to (Java) reflection to transparently access the MetaDepth API from them. In contrast, in the EMF ecosystem there is no standard way of integrating tools. Thus, we need to give explicit support for each tool of interest in order to use their model management operations as part of the MOP or if we want to use the MOP to configure the tool. So far have experimented with support for ATL for model transformations and a specialization of the MOP for Xtext to activate libraries only when the text is processed by an Eclipse job.
Some of the libraries implemented for EMF are facilities already available in MetaDepth, like support for the type-object pattern, class cardinalities and explicit conformance relationships between (meta-)models. Our experience implementing them shows how a MOP can be useful to overcome certain limitations of a meta-modelling framework. Moreover, this experience provides some insight on the idea of having a *minimal meta-modelling framework* plus a MOP to support the different modelling styles and requirements as extensions.

8.3.2. Efficiency

MOPs may cause performance penalties as modelling operations need to be intercepted to check for attached advices. To evaluate this aspect, we performed some experiments to assess the penalties incurred by MOPs. The first experiment evaluates the MOP implementation in MetaDepth, while the second one deals with EMOP. Interestingly, in both experiments we encountered similar effects.

Fig. 13 shows a comparison of efficiency for the deep cardinality MOP presented in Section 8.1.2, with respect to an ad-hoc implementation using pure EOL constraints and no MOP mechanism. In both cases, we performed experiments with models ranging from 10 to 16,000 objects (repeating 8 times each experiment and taking the median time), where half the objects had cardinality errors. The experiments were performed on a Windows 10 computer with i7-6500U processor and 16Gb of RAM memory. It can be seen that EOL constraints are more efficient for small-to-medium models (up to sizes of around 10,000 objects), while for larger models the MOP mechanism becomes more scalable. This effect may be attributed to the overhead that the MOP mechanism introduces, which only pays off for larger model sizes.

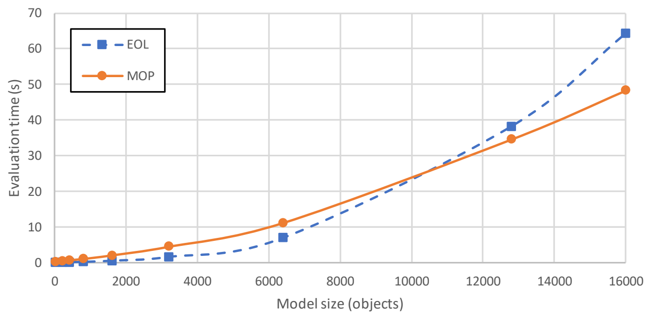


Figure 13: Comparing efficiency of MetaDepth implementations of deep-cardinality checks using MOPs and EOL constraints over models of increasing size

A similar experiment has been performed for EMOP, our implementation of top of EMF, in this case for the prototype extension. We created models of increasing size in which objects are a prototype of another object, and then we accessed to values from such objects in order to exercise obtaining values from the prototype. In the case of EMF we simulate this by having a hash map linking an object with its prototype, and copying values from the prototype as required. The results are consistent with the first experiment, as shown in Fig. 14. As the size increases, the MOP overhead pays off since it internally implements a more sophisticated prototype resolution mechanism, based on lazy loading to retrieve values from a prototype.

Overall, our prototypes have not been implemented with performance as the main design criterion, but we aimed at flexibility. It is future work to analyse optimization possibilities, for example taking inspiration of recent advances in virtual machines [45].

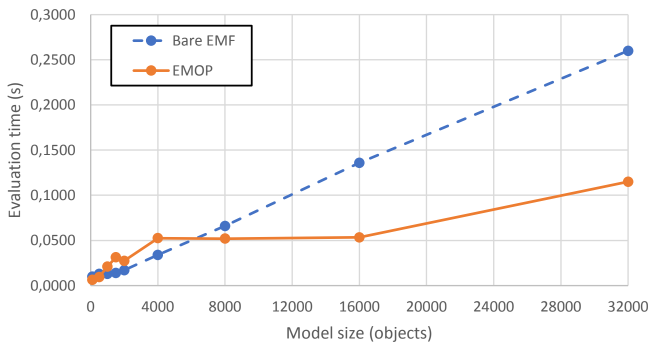


Figure 14: Comparing efficiency of EMF implementations of the prototype extension using EMOP and bare EMF over models of increasing size

Table 4: Mechanisms needed to implement the extensions without MOPs

Name	**Modify environment**	**Modify Ling. MM**	**Add command**	**Manual execution**	**AOP**	**Framework extensions**
Auto-instantiate						×
Class cardinality	×	×				
Disable validation					×	
Prototype					×	
Resource type					×	
Compute subtypes		×			×	
Static semantics library	×	×				
Meta-model/model conformance reconciliation			×	×		
Multi-level facilities library	×	×				
Bottom-up modelling	×		×			
Type object			×			
Visualization				×		

8.3.3. MOPs vs. manual implementation

Overall, creating the services described in Table 2 without MOP support would have involved several workarounds, discussed next and summarized in Table 4.

- *Modification of the source code of the meta-modelling environment.* In the case of MetaDepth, implementing some extensions, like the facilities for bottom-up meta-modelling, would require the modification of the tool source code. In contrast, a MOP-based mechanism permits adding such functionality in an external way.

- *Creating framework extensions.* Frameworks like EMF allow developers to create custom extensions. For instance, in EMF one can create particular types of resources to implement special loading or serialization capabilities. That is, one can use subclassing and method overriding to modify the default behaviour. This solution is possible for infrastructure elements (like resources), or to customize framework implementation classes when using the compiled approach (i.e., a custom implementation of the EObject interface from which the code of generated classes are set to inherit). This strategy can be used to implement the auto-instantiation example.

- *Aspect oriented programming.* In case the source code is not available or cannot be modified (as in EMF), one would need to resort to AOP. This solution is generally applicable, but it requires knowledge of both EMF internals to define join points and some AOP technology like AspectJ, which may be beyond the skills of normal modelling users.

- *Modification of the linguistic meta-model.* Without MOPs or annotations, one would typically need to modify the linguistic meta-model to include such extra information. In the case of the deep cardinalities, the linguistic meta-model would be modified to include the range of potencies associated to a cardinality. In the case of MetaDepth, this solution implies in addition modifying the textual concrete syntax of the language.

- *Adding a command.* MOP extensions are triggered on the occurrence of events. Hence, without MOPs a mechanism to activate the extensions is needed. In EMF, this requires adding a menu in the Eclipse user interface, while in MetaDepth it requires creating a new console command.

- *Manual execution of transformations.* Advices can be expressed using model management languages, which in case of MOPs are automatically triggered. Without MOPs, they should be manually executed.

Hence, we see that without MOPs one needs to rely on suboptimal solutions, like the need to modify the internals of the environment, obtain lower-quality solutions (e.g., implying manual execution of transformations) or resort to complex alternatives based on AOP. Creating a framework extension is often the solution of choice (e.g., for persistence [46]). However, this is not the best approach when one needs to mix several extensions together (e.g., two extensions which define their own EMF resources are incompatible) or when the compiled approach is too heavyweight. In addition, framework extensions may not be reusable across several meta-models if a placement mechanism, like our annotations, is not implemented as well. Instead, using a MOP it has been possible to focus on the essential complexity of the problem, that is, identify the meta-modelling events on which advices must be added and then implementing the extension itself. In case of MetaDepth these advices can be implemented using model management languages. Moreover, advices were organized into reusable libraries, which can be used to build further services.
To have an indication of the effort needed by a manual implementation to emulate a MOP-based facility, we re-implemented the MOP for bottom-up modelling (see Section 8.1.1) as a new console command of MetaDepth. The results are summarized in Table 5, using Lines of Code (LOC) as an (indirect) measurement of effort. For the manual implementation, we had to extend the base code of MetaDepth with two new commands (to create node types and attribute types). This resulted in around 450 LOC of Java. Defining a MOP for the

same functionality can be done externally using EOL as we have shown in Section 8.1.1. Both extensions resulted in around 45 lines of EOL code, and the annotation definition in around 30 lines of MetaDepth code.

Table 5: LOC for a manual vs. MOP-based implementation of the bottom-up modelling functionality in MetaDepth.

Manual Implementation			MOP-based implementation		
Artefact	**LOCs**	**Language**	**Artefact**	**LOCs**	**Language**
Refactoring.java	121	Java	pullUp.eol	36	EOL
CreateType.java	130	Java	pullUpField.eol	9	EOL
CreateAttribType.java	126	Java	pullUp.annotations	27	MetaDepth
AttribRefactoring.java	78	Java			
Total	**455**		**Total**	**72**	

While comparing effort by LOC (of heterogeneous technologies) may be misleading, we just use them as an indication of a typical size of an extension, to be performed by environment extenders (with or without MOP-based mechanisms). A conclusion is that sophisticated extensions can be defined in a succinct way using MOPs, as extenders may use a model management language for model manipulation, while a Java implementation may become more verbose (as the table shows). Moreover, the Java implementation is intrusive, requiring changing the MetaDepth code base. This means creating a new distribution of the tool itself. In contrast, a MOP implementation is totally external. Users can make available these extensions by means of libraries, which do not require a new tool version. An intrusive solution requires good knowledge of the code organization and internal working scheme of the meta-modelling tool. In contrast, a MOP implementation is external, and requires knowing the MOP API exposed by the meta-modelling tool. If well designed, such API will be smaller and easier to learn than the complete code base of the meta-modelling tool.
Altogether, we claim that an *open* design for a meta-modelling framework based on the notion of MOP is a way to improve the flexibility and extensibility of current, *closed* approaches which is worth pursuing by the MDE community.

9. Related work

We organize the review of related research in two parts. First, Section 9.1 overviews existing (meta-)modelling frameworks and tools, analysing the degree in which a MOP-based mechanism could be applicable and useful.
In a second part (Section 9.2), we revise works that have inspired us, in areas of programming and modelling including reflection, meta-object protocols, aspect-oriented programming and aspect-oriented modelling.

9.1. Extensibility in meta-modelling frameworks and tools

Many meta-modelling tools [6, 8, 9, 10], languages [3, 4, 5] and frameworks [7] have been proposed along the years, each one of them supporting different functionalities. These have greatly influenced the effort needed to build tools and services on top of them for specific purposes, like co-evolution [41], meta-model

Table 6: Existing tools and approaches which could have been benefited from using a MOP

Name	**References**	**Observations**
Auto-instantiate	[7, 5]	• The UML plug-in automatically creates objects upon profile loading [5]. • EMF [7] could use this functionality to automatically instantiate root classes.
Class cardinality	[7, 49]	• In EMF [7], root classes have by convention a [1..1] cardinality. • Model finders [49] need to define a global search scope that could be specified by cardinality intervals.
Disable validation	[47, 50, 11]	• Meta-model testing frameworks need to accept invalid models[47]. • Flexible modelling environments [50, 11] need to relax conformance in early phases of modelling.
Prototype-based modelling	[18, 50, 11]	• An initial design based on prototypes could have been tested using MOPs [18]. • Proptotype-based meta-modelling tools could have used this facility to keep compatibility with EMF [50, 11].
Resource type	[51]	• Keeping track of referenced meta-models is useful for composing meta-models, like in Melangue [51].
Efficient subtype computation	[52, 53, 54]	• Useful for any analysis tool handling meta-models, model-based metric tools, or model-based refactoring tools.
Static semantics library	[40, 55]	• Useful to externally define libraries of constraints, which then can annotate model elements.
Meta-model/model co-evolution	[41, 56]	• Useful to enhance any migration tool to provide live co-evolution.
Multi-level facilities library	[57, 58, 59, 60]	• Multi-level modelling tools, like Melanee, XMF, Xmodeller, MultEcore, would benefit from MOPs to permit interoperability and adaptation.
Bottom-up modelling	[40, 61]	• Flexible modelling approaches and tools could use this facility to create types from objects.
Type-object	[60, 62, 63]	• Any promotion-based approach would benefit (see [15] for additional examples).
Visualization	[48, 64]	• Automated synchronization of textual and graphical concrete syntaxes.

testing [47], or synchronization of visual and textual concrete syntaxes [48]. Our purpose in this section is to analyse the benefits that extensibility mechanisms, like those based on MOP would have brought to users of these systems. To focus our comparison, we base on the functionalities offered by the MOPs we have developed, indicating meta-modelling tools that would have benefited from having them available. A summary is shown in Table 6, and we discuss them next:

- *Auto-instantiate.* Several modelling applications require the automatic instantiation of objects. For example, UML needs to create objects when a profile is loaded [5], while languages for (algebraic, boolean) expressions may need to create objects representing constants, as we have seen in this paper. Moreover, in frameworks like the EMF [7], there is the tacit convention of having a *root* class in meta-models, an object of which needs to contain directly or indirectly all other objects in the model. Currently such class needs to be instantiated manually, but an extension like ours could make this automatic. While automatic instantiation seems to be a recurring need, we are not aware of meta-modelling frameworks with native support for it. Hence, a MOP extension like the one we have defined could provide this automation.

- *Class cardinality.* Related to the previous functionality, we may want to provide an instantiability interval to classes. Some frameworks, like the EMF, rely on cardinality of container references for this. However, this is not enough when we need to restrict globally the instances of a class (e.g., to define a search scope in model finding applications [49]), or when the class is the root (and hence it is not contained in any composition reference). Some meta-modelling frameworks, like MetaDepth [9] or MetaEdit+ [6] natively support class instantiation cardinalities. Some model finders like the USE validator [49] are based on UML, and hence need to extend the language with class cardinalities. These could have been externally defined with a MOP mechanism.

- *Disable validation.* Meta-modelling frameworks typically enforce the conformance relation between models and meta-models at all times. Some of them, like the EMF take this to the extreme, so that they may not even be able to persist incorrect models. Being able to defer validation is necessary for meta-modelling testing frameworks, where incorrect models need to be input as test cases [47]. The workaround proposed in [47] to solve this lack of support in EMF is the creation of a custom meta-modelling language. Another scenario for tolerating inconsistencies is in early phases of modelling, where support for discussions between developers are more important than model correctness, which can be achieved at a later stage [50, 11]. Again, the workarounds proposed in [50, 11] was the construction of custom meta-modelling languages. MOP mechanisms could make EMF fit for this purpose, though.

- *Prototype-based modelling.* Most meta-modelling approaches follow a class-based approach [7, 6, 8, 9, 10, 3, 5, 7] while just a few have adopted a prototype-based approach [50, 11]. The latter typically results in more flexible systems, which may be needed for some scenarios [18]. A MOP mechanism would permit enjoying the flexibility of prototype-based solutions in class-based systems (e.g., in EMF for compatibility), and the combination of both modelling styles.

- *Resource type.* Some frameworks, like the EMF, lack the concept of *model*, and hence complicate keeping track of what is the meta-model(s) a set of objects in a resource conform to. This is especially important in systems aiming at the composition of domain-specific languages (DSLs), like Melange [51]. A MOP-based approach on top of EMF could have helped in the construction of this kind of systems.

- *Efficient subtype computation.* In some systems, like the EMF, classes store their direct supertypes. Other systems [4, 5], use an intermediate class Generalization. Hence, meta-modelling systems have typically a preferred inheritance navigation direction, which is more efficient than the other: from subtypes to supertypes [5, 7], or from supertypes to subtypes [4]. However, some applications may need to traverse frequently the inheritance relations in both directions, like static analysis systems [52], systems to calculate metrics [53], or model-based refactoring tools [54]. MOPs to cache (direct and indirect) super- or sub-types would make these systems more efficient and easier to build.

- *Static semantics library.* Most meta-modelling systems support auxiliary languages to express constraints, like OCL [65] or EOL [35]. Writing these constraints is sometimes hard, while some of them are recurring (e.g., to express reflexivity or acyclicity of relations, or to enforce commutativity of references). Systems like the Diagram Predicate Framework (DPF) [55] or metaBUP [40] support the creation of constraint libraries, which can then be applied to meta-model elements via annotations. This mechanism could have been more easily built using a MOP mechanism with annotation-based placement.

- *Meta-model/model co-evolution.* A few meta-modelling systems have built-in capabilities to tackle the meta-model/model co-evolution problem [6], while most don't [7, 9, 8, 10]. A considerable amount of research efforts have been conducted to automate model co-evolution upon meta-model changes [56, 41]. While these typically target off-line co-evolution, on-line co-evolution is a typical scenario e.g., in database systems [66]. As we have seen in Section 8.2, MOP mechanisms could be used to perform live meta-model/model co-evolution, and could have simplified and enhanced the construction of those systems.

- *Multi-level facilities library.* Several multi-level modelling primitives, like potency [19], dual potencies [22], durability and mutability [21], and deep association cardinalities [23] have been proposed in the literature, while a few multi-level modelling flavours exists (e.g., based on powertypes [17], based on deep characterization [67]). These primitives and approaches could be defined using a MOP-based mechanism, facilitating tool interoperability, and a more flexible experimentation.

- *Bottom-up modelling.* Most meta-modelling approaches follow a top-down approach: types are created first, and then instantiated. However, some approaches favour the creation of example models even before the meta-model is built [40, 61]. Then, these examples are used to drive and automate meta-model construction. As we have seen in Section 8.1.1, MOPs could have been used to automate this process and simplify the construction of these systems.

- *Type-object.* Several applications emulate multiple meta-levels using promotion transformations, which transform objects into types [62, 63]. Promotion transformations are also the basis of some meta-modelling tools, like Multecore [60]. A MOP mechanism could add facilities to automatically trigger the transformation of objects into types for frameworks lacking multi-level support, like EMF.

- *Visualization.* The need to update concrete syntax upon changes in the model is a common need. Some systems support several concrete syntaxes (e.g., graphical and textual) [64], but many times specific synchronization functionality needs to be developed [48]. A MOP can simplify the construction of this synchronization service by taking care of concrete syntax updates.

9.2. Meta-Object protocols and aspect orientation

Our work takes inspiration from meta-object protocols, reflection, aspect-oriented modelling and aspect oriented programming, areas which are reviewed next.

9.2.1. Meta-object protocols and reflection

MOPs originated in the OOP community, as a way to open language implementations, so that they become more flexible and extensible [12, 13]. Today languages like Groovy have native support for MOPs, while it is common for languages to allow some weak form of MOP.

We have taken inspiration from works of the OOP community. For example, in [30] the authors suggest making a singleton object available, exposing an API to configure extensions with placement control, and the possibility to activate/deactivate them. We took this idea in MetaDepth to implement the MOPHelper object. Such object is also accessible from model management languages. Reflection and MOPs provide flexibility, at the cost of performance. Hence, some works [45, 31] are directed to perform optimizations for meta-programming. In MetaDepth, we just permit enabling or disabling the MOP mechanism, but optimizations are left to future work. Similarly, the activation of extensions is manual in EMOP.

MOPs have been recently proposed for language interpreters by Cazzola and collaborators [68], and realized in the Neverlang tool. Neverlang permits the modular definition of the syntax and semantics of programming languages. A MOP mechanism enables the user to override the default semantics of the interpreter. A DSL was designed for placement control (based on tree pattern matching) and specifying the overriding semantics by accessing the Neverlang's exposed API. While this approach adapts specific interpreters, our proposal works at a higher meta-level, extending and adapting the meta-modelling facilities themselves. This way, Neverlang extensions are interpreter-specific, while in our case extensions are typically meta-model independent and hence can be reused across domain meta-models.

Some works have addressed reflection in the context of meta-modelling. In [69] the authors signal some drawbacks in the MOF, like the lack of explicit notions of model and meta-model, and the lack of explicit instance-of relations. For this purpose, they propose *sNets*, a meta-modelling formalism that incorporates

the notion of model/meta-model and reifies the instance-of relation between nodes. However, their proposal is still tied to two-level modelling, and does not consider the possibility of adding a MOP. Instead, MetaDepth's linguistic meta-model contains the notion of (nested) model, and explicitly reifies the instance-of relation between fields, objects and models.
The Open Meta-Modelling Environment (OMME) [70] proposes a more flexible meta-modelling approach than standard ones like MOF, by permitting multi-level modelling and advanced constructs like powertypes. Despite the name, OMME is not *open* in the sense defined in this paper as it does not allow user extensions of the offered meta-modelling facilities. Overall, we could not find works on using MOPs within meta-modelling environments. Please note that EMF supports event notifications, but it only covers a subset of the events that we consider in Table 1, and as mentioned in Section 4, this constitutes only a weak extension mechanism. Essentially, the EMF notification mechanism covers setting properties (directly setting a mono-valued property or adding elements to a multi-valued property), querying the original and the old value in a setting and watching if elements has been removed from a property. Events like getting a property value or object instantiation are not covered. Moreover, advices can only be executed *after* the event has occurred. The EMOP API provides a common and simpler access mechanism for a wider range of events than EMF notifications, although internally uses notifications when possible.
OpenPonk is a modelling tool based on Pharo Smalltalk [71]. Being based on Smalltalk it inherits its reflective capabilities, like for instance the ability to configure model notifications by intercepting calls to infrastructure methods. Similarly, Moose is a Smalltalk-based reengineering platform based on the FAME meta-modelling framework [72]. Both OpenPonk and FAME do not define a MOP, but given Smalltalk support for reflection, our approach could be integrated there as well.
Some model transformation languages have exploited reflection to some extent, or allow extensibility using MOP-like techniques. RubyTL [73] was a transformation language in which extensions were created by implementing advices attached to pre-defined extension points. Mistral [74] reifies a set of transformation events like rule execution, or slot assignment. They are handled by meta-rules and mechanisms for meta-rule ordering are available. Our aspect-based implementation approach could be used to implement similar extensions for existing languages, like ATL. Finally, event-driven grammars [75] extended graph transformation with the possibility to match meta-modelling events like creating or connecting elements, and be triggered by those events.

9.2.2. Aspect-oriented modelling

Aspect-oriented modelling lifts ideas from AOP to modelling [27]. Hence, the objective is to identify crosscutting concerns at the model level, specifying them in a separate way, and then weave those aspects with the base model [29]. This way, both the base model and the advice are models. Alternatively, some approaches to AOM rely on generating code targeting an aspect-oriented language (like AspectJ), and then rely on weaver provided by the target language to deal with crosscutting aspects [76]. Some approaches to AOM are related to a systematic identification of early aspects – already in the requirements and design phases – which then need to be expressed in notations for requirements, architecture and design [77, 78]. In our case, advices are expressed either programmatically or using model management languages. This is needed because our advices effect both on structural join points (just like AOM), and on the occurrence of designated meta-modelling events. The latter is the distinctive feature of our approach with respect to AOM.

Several approaches have been proposed for weaving the aspect model into the base model. In [29] it is done by expressing the advice using graph transformation rules. This way, the pointcut is expressed by pattern matching. In [79] it is based on explicitly marking the UML models involved. In our case we use both a programmatic and annotation-based approach. We leave for future work a declarative expression of pointcuts, e.g., based on OCL expressions.

Cazzola et al. proposed the reflective object oriented analysis [80] (ROOA) as a way to allow the non-functional properties of a model to be moved into a "meta-object" (i.e., objects describing properties of another objects). This approach can also be applied at the architectural level [81]. In MetaDepth it would be possible to imitate ROOA by combining multi-level modelling (to define the analysis meta-level) and our MOP approach to inject the behaviour at this meta-level.

9.2.3. Aspect-oriented programming

Aspect-oriented programming (AOP) [26] aims at the modularization of crosscutting concerns by allowing custom actions (advices) to be plugged into designated execution points (join points). Typically, AOP frameworks provide languages to describe join points by means of pointcuts. In our case, we use AOP as an implementation mechanism in the case of our EMF-based prototype, EMOP.

A variant of AOP is event-based aspect oriented programming [82], in which the possible pointcuts are predefined as a set of event types. Our implementation is similar, as our MOP also exposes a fixed set of events.

In [83] the authors propose Meta-AspectJ (MAJ), a language for generating AspectJ programs using code templates. One of the applications of MAJ is to design small domain-specific extensions of Java based on annotations. The annotations trigger the execution of MAJ generators, which produce aspect code woven with the base program. In our case, annotations may have associated modelling advices, but these are not restricted to be code generators. Moreover, the trigger of those advices is associated with events exposed by the meta-modelling framework.

Aspect orientation has pervaded different domains, leading to the proposal of Domain-Specific Aspect Languages (DSALs) [84]. For example, DSALs have been proposed to optimize image processing systems, to create design rule checkers for .Net, or to attach transactional properties to an application. Different parts of an aspect may have domain-specific concepts, like the join points, pointcuts or the advices. In our case, our join points are specific to meta-modelling

and include both events like model load, and structure like an object or a feature. Pointcuts are expressed either using annotations or programmatically, while advices can be express using general purpose languages or model management languages.
Related to annotations, in [85] an approach to derive a language to specify types of tags for a given DSL is presented, focussing on the generation of a specific textual syntax. Our proposal is agnostic of the concrete syntax and if used with a multi-level framework, does not require a generation step. In [86], we proposed a DSL to describe the syntax of Java annotations. From such work, we took the idea of the requires and forbids relations between annotations. In [87], the authors produce custom visualization for Java annotations, using a MOP approach.
Hence, altogether, the use of MOPs as a way to extend meta-modelling frameworks is novel. While some ideas were taken from the OOP community, the application to the context of meta-modelling required several adaptations. First, the placement mechanism based on annotation proved to be very adequate. In contrast, in OOP, the placement mechanism is normally based on naming specific methods or classes, perhaps resorting to regular expressions. Second, we could resort to model management languages to implement advices. Finally, we had to consider join points in meta-modelling frameworks.

10. Conclusions and future work

In this paper, we have presented the architecture of an open meta-modelling framework, a design approach which enables framework extensions through a MOP. We have shown its main elements, including a novel annotation-based approach for placement control. The concepts presented in this work have been validated by two implementations and several ready-to-use libraries built with them, which altogether show the feasibility and generality of our proposal. To the best of our knowledge this is the first proposal for *open* meta-modelling frameworks and change the common way of implementing them. The insights provided by this work are useful for designers and implementors of meta-modelling frameworks.
As future work we plan to improve our support for EMF and MetaDepth (e.g., by providing better support for transactions) and look into how to deal with the ordering and potential conflicts of extensions. We plan also to add support for OCL expressions as a means to express structural pointcuts. In the long term we plan to build a minimal meta-modelling framework extensible through a built-in MOP. A related topic is to explore the use of aspect-oriented techniques to the extensibility of model management languages. While this was studied in Mistral for the case of model-to-model transformation, our aim is to extend this idea to model management language families like Epsilon.

Acknowledgments

Work partially funded by projects RECOM and FLEXOR (Spanish MINECO, TIN2015-73968-JIN (AEI/FEDER/UE) and TIN2014-52129-R) and the R&D programme of the Madrid Region (S2013/ICE-3006). We thank the reviewers for their constructive comments, which helped us in improving the paper.

References

[1] D. C. Schmidt, Guest editor's introduction: Model-driven engineering, Computer 39 (2) (2006) 25–31.

[2] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2012.

[3] OMG, Meta Object Facility, `http://www.omg.org/mof/` (2015).

[4] C. Gonzalez-Perez, A conceptual modelling language for the humanities and social sciences, in: RCIS, IEEE, 2012, pp. 1–6.

[5] OMG, Unified Modelling Language, `http://www.omg.org/spec/UML/` (2015).

[6] S. Kelly, J. Tolvanen, Domain-Specific Modeling - Enabling Full Code Generation, Wiley, 2008, see also `http://www.metacase.com/`.

[7] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2^{nd} Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2008.

[8] J. de Lara, H. Vangheluwe, AToM^3: A tool for multi-formalism and meta-modelling, in: FASE, Vol. 2306 of Lecture Notes in Computer Science, Springer, 2002, pp. 174–188.

[9] J. de Lara, E. Guerra, Deep meta-modelling with MetaDepth, in: TOOLS'10, Vol. 6141 of LNCS, Springer, 2010, pp. 1–20, see also `http://metaDepth.org`.

[10] C. Atkinson, M. Gutheil, B. Kennel, A flexible infrastructure for multilevel language engineering, IEEE Trans. Software Eng. 35 (6) (2009) 742–755.

[11] N. Hili, A metamodeling framework for promoting flexibility and creativity over strict model conformance, in: FlexMDE@MoDELS, Vol. 1694 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 2–11.

[12] G. Kiczales, J. D. Rivieres, The Art of the Metaobject Protocol, MIT Press, Cambridge, MA, USA, 1991.

[13] G. Kiczales, J. Ashley, L. Rodriguez, A. Vahdat, D. G. Bobrow, Metaobject protocols: Why we want them and what else they can do, in: A. Paepcke (Ed.), Object-Oriented Programming: The CLOS Perspective, The MIT press, 1993, pp. 101–118.

[14] L. G. DeMichiel, R. P. Gabriel, The common lisp object system: An overview, in: ECOOP, Vol. 276 of Lecture Notes in Computer Science, Springer, 1987, pp. 151–170.

[15] J. de Lara, E. Guerra, J. S. Cuadrado, When and how to use multilevel modelling, ACM Trans. Softw. Eng. Methodol. 24 (2) (2014) 12:1–12:46.

[16] D. Ungar, R. B. Smith, SELF: the power of simplicity, Lisp and Symbolic Computation 4 (3) (1991) 187–205.

[17] C. Gonzalez-Perez, B. Henderson-Sellers, A powertype-based metamodelling framework, Software and System Modeling 5 (1) (2006) 72–90.

[18] T. Aschauer, G. Dauenhauer, W. Pree, A modeling language's evolution driven by tight interaction between academia and industry, in: ICSE'10, ACM, 2010, pp. 49–58.

[19] C. Atkinson, Meta-modeling for distributed object environments, in: EDOC, IEEE Computer Society, 1997, p. 90.

[20] C. Atkinson, T. Kühne, The essence of multilevel metamodeling, in: UML, Vol. 2185 of Lecture Notes in Computer Science, Springer, 2001, pp. 19–33.

[21] C. Atkinson, R. Gerbig, Flexible deep modeling with melanee, in: Modellierung, Vol. 255 of LNI, GI, 2016, pp. 117–122.

[22] B. Neumayr, M. A. Jeusfeld, M. Schrefl, C. G. Schütz, Dual deep instantiation and its ConceptBase implementation, in: CAiSE, Vol. 8484 of Lecture Notes in Computer Science, Springer, 2014, pp. 503–517.

[23] C. Atkinson, R. Gerbig, T. Kühne, A unifying approach to connections for multi-level modeling, in: MoDELS, IEEE Computer Society, 2015, pp. 216–225.

[24] R. Gerbig, C. Atkinson, J. de Lara, E. Guerra, A feature-based comparison of Melanee and MetaDepth, in: MULTI@MODELS, Vol. 1722 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 25–34.

[25] P. Maes, Concepts and experiments in computational reflection, in: OOPSLA, ACM, 1987, pp. 147–155.

[26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: ECOOP, 1997, pp. 220–242.

[27] M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, E. Kapsammer, A survey on UML-based aspect-oriented design modeling, ACM Comput. Surv. 43 (4) (2011) 28:1–28:33.

[28] L. Fuentes, P. Sánchez, Designing and weaving aspect-oriented executable UML models, Journal of Object Technology 6 (7) (2007) 109–136.

[29] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, J. Araújo, MATA: A unified approach for composing UML aspect models based on graph transformation, Trans. Aspect-Oriented Software Development 6 (2009) 191–237.

[30] N. Papoulias, M. Denker, S. Ducasse, L. Fabresse, Reifying the reflectogram: towards explicit control for implicit reflection, in: SAC'15, ACM, 2015, pp. 1978–1985.

[31] É. Tanter, J. Noyé, D. Caromel, P. Cointe, Partial behavioral reflection: spatial and temporal selection of reification, in: OOPSLA'03, ACM, 2003, pp. 27–46.

[32] J. de Lara, E. Guerra, *A Posteriori* typing for model-driven engineering: Concepts, analysis, and applications, ACM Trans. Softw. Eng. Methodol. 25 (4) (2017) 31:1–31:60.

[33] E. Clayberg, D. Rubel, Eclipse plug-ins, Addison-Wesley professional, 2014.

[34] C. Atkinson, T. Kühne, Reducing accidental complexity in domain models, Software and System Modeling 7 (3) (2008) 345–359.

[35] D. S. Kolovos, R. F. Paige, F. Polack, The Epsilon Object Language (EOL), in: ECMDA-FA, Vol. 4066 of LNCS, Springer, 2006, pp. 128–142.

[36] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, F. A. C. Polack, The design of a conceptual framework and technical infrastructure for model management language engineering, in: ICECCS, IEEE Computer Society, 2009, pp. 162–171.

[37] J. de Lara, E. Guerra, R. Cobos, J. Moreno-Llorena, Extending deep metamodelling for practical model-driven engineering, Comput. J. 57 (1) (2014) 36–58.

[38] L. M. Rose, R. F. Paige, D. S. Kolovos, F. Polack, The Epsilon Generation Language, in: ECMDA-FA, Vol. 5095 of Lecture Notes in Computer Science, Springer, 2008, pp. 1–16.

[39] D. S. Kolovos, R. F. Paige, F. Polack, The Epsilon Transformation Language, in: Proc. ICMT, Vol. 5063 of Lecture Notes in Computer Science, Springer, 2008, pp. 46–60.

[40] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, J. de Lara, Example-driven meta-model development, Software and System Modeling 14 (4) (2015) 1323–1347.

[41] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, Automating co-evolution in model-driven engineering, in: Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE, IEEE, 2008, pp. 222–231.

[42] K. T. Claypool, J. Jin, E. A. Rundensteiner, SERF: schema evolution through an extensible, re-usable and flexible framework, in: ACM CIKM, ACM, 1998, pp. 314–321.

[43] M. Hartung, J. Terwilliger, E. Rahm, Recent advances in schema and ontology evolution, in: Z. Bellahsene, A. Bonifati, E. Rahm (Eds.), Schema Matching and Mapping, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 149–190.

[44] R. F. Paige, N. Matragkas, L. M. Rose, Evolving models in model-driven engineering: State-of-the-art and future challenges, Journal of Systems and Software 111 (2016) 272–280.

[45] S. Marr, C. Seaton, S. Ducasse, Zero-overhead metaprogramming: reflection and metaobject protocols fast and without compromises, in: PLDI, ACM, 2015, pp. 545–554.

[46] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, D. Launay, Neo4emf, a scalable persistence layer for emf models, in: European Conference on Modelling Foundations and Applications, Springer, 2014, pp. 230–241.

[47] J. J. López-Fernández, E. Guerra, J. de Lara, Combining unit and specification-based testing for meta-model validation and verification, Inf. Syst. 62 (2016) 104–135.

[48] L. Addazi, F. Ciccozzi, P. Langer, E. Posse, Towards seamless hybrid graphical-textual modelling for UML and profiles, in: ECMFA, Vol. 10376 of Lecture Notes in Computer Science, Springer, 2017, pp. 20–33.

[49] M. Kuhlmann, M. Gogolla, From UML and OCL to Relational Logic and Back, in: Proc. of MODELS'12, Vol. 7590 of LNCS, Springer, Berlin, Heidelberg, 2012, pp. 415–431.

[50] J. Sottet, N. Biri, JSMF: a javascript flexible modelling framework, in: FlexMDE, Vol. 1694, CEUR, 2016, pp. 42–51.

[51] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Melange: A meta-language for modular and reusable development of dsls, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, 2015, pp. 25–36.

[52] J. S. Cuadrado, E. Guerra, J. de Lara, Static analysis of model transformations, IEEE Trans. Software Eng. 43 (9) (2017) 868–897.

[53] C. Hein, M. Engelhardt, T. Ritter, M. Wagner, Generation of formal model metrics for MOF based domain specific languages, ECEASST 24.

[54] T. Arendt, G. Taentzer, A tool environment for quality assurance based on the eclipse modeling framework, Autom. Softw. Eng. 20 (2) (2013) 141–184.

[55] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, A. Rutle, Dpf workbench: A diagrammatic multi-layer domain specific (meta-)modelling environment, in: Computer and Information Science 2012, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 37–52.

[56] M. Herrmannsdoerfer, S. Benz, E. Jürgens, COPE - automating coupled evolution of metamodels and models, in: ECOOP, Vol. 5653 of Lecture Notes in Computer Science, Springer, 2009, pp. 52–76.

[57] C. Atkinson, R. Gerbig, M. Fritzsche, A multi-level approach to modeling language extension in the enterprise systems domain, Inf. Syst. 54 (2015) 289–307.

[58] T. Clark, P. Sammut, J. S. Willans, Super-languages: Developing languages and applications with XMF (second edition), CoRR abs/1506.03363.

[59] U. Frank, Designing models and systems to support IT management: A case for multilevel modeling, in: Proceedings MULTI@MODELS, Vol. 1722 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 3–24.

[60] F. Macías, A. Rutle, V. Stolz, Multecore: Combining the best of fixed-level and multilevel metamodelling, in: MoDELS, Vol. 1722 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 66–75.

[61] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, C. Guychard, Using free modeling as an agile method for developing domain specific modeling languages, in: MoDELS, ACM, 2016, pp. 24–34.

[62] J. Steel, K. Duddy, R. Drogemuller, A transformation workbench for building information models, in: ICMT, Vol. 6707 of Lecture Notes in Computer Science, Springer, 2011, pp. 93–107.

[63] N. Drivalos, D. S. Kolovos, R. F. Paige, K. J. Fernandes, Engineering a DSL for software traceability, in: SLE, Vol. 5452 of Lecture Notes in Computer Science, Springer, 2008, pp. 151–167.

[64] M. Garzón, H. I. Aljamaan, T. C. Lethbridge, Umple: A framework for model driven development of object-oriented systems, in: SANER, IEEE Computer Society, 2015, pp. 494–498.

[65] OCL, `http://www.omg.org/spec/OCL/` (2014).

[66] G. H. Sockut, B. R. Iyer, Online reorganization of databases, ACM Comput. Surv. 41 (3) (2009) 14:1–14:136.

[67] C. Atkinson, T. Kühne, Model-driven development: A metamodeling foundation, IEEE Software 20 (5) (2003) 36–41.

[68] W. Cazzola, A. Shaqiri, Open programming language interpreters, The Art, Science, and Engineering of Programming Journal 1 (2017) 1–34.

[69] J. Bézivin, R. Lemesle, Towards a true reflective modeling scheme, in: Reflection and Software Engineering, Papers from OORaSE 1999, 1st OOPSLA Workshop on Reflection and Software Engineering, Denver, CO, USA, November 1999, Vol. 1826 of Lecture Notes in Computer Science, Springer, 1999, pp. 21–38.

[70] B. Volz, S. Jablonski, Towards an open meta modeling environment, in: Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM '10, ACM, New York, NY, USA, 2010, pp. 17:1–17:6.

[71] P. Uhnák, R. Pergl, The OpenPonk modeling platform, in: Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies, ACM, 2016, p. 14.

[72] S. Ducasse, T. Gîrba, A. Kuhn, L. Renggli, Meta-environment and executable meta-language using Smalltalk: an experience report, Software and Systems Modeling 8 (1) (2009) 5–19.

[73] J. S. Cuadrado, J. G. Molina, A plugin-based language to experiment with model transformation, in: MoDELS'06, Springer-Verlag, 2006, pp. 336–350.

[74] I. Kurtev, Application of reflection in a model transformation language, Software and System Modeling 9 (3) (2010) 311–333.

[75] E. Guerra, J. de Lara, Event-driven grammars: relating abstract and concrete levels of visual languages, Software and System Modeling 6 (3) (2007) 317–347.

[76] A. Mehmood, D. N. A. Jawawi, Aspect-oriented model-driven code generation: A systematic mapping study, Information & Software Technology 55 (2) (2013) 395–411.

[77] P. Sánchez, A. Moreira, L. Fuentes, J. Araújo, J. Magno, Model-driven development for early aspects, Information & Software Technology 52 (3) (2010) 249–273.

[78] O. Alam, J. Kienzle, G. Mussbacher, Modelling a family of systems for crisis management with concern-oriented reuse, Softw., Pract. Exper. 47 (7) (2017) 985–999.

[79] L. Fuentes, M. Pinto, P. Sánchez, Generating CAM aspect-oriented architectures using model-driven development, Information & Software Technology 50 (12) (2008) 1248–1265.

[80] W. Cazzola, A. Sosio, F. Tisato, Shifting up reflection from the implementation to the analysis level, in: Reflection and Software Engineering, Springer, 1999, pp. 1–20.

[81] F. Tisato, A. Savigni, W. Cazzola, A. Sosio, Architectural reflection realising software architectures via reflective activities, Engineering Distributed Objects (2001) 102–115.

[82] R. Douence, O. Motelet, M. Südholt, A formal definition of crosscuts, in: REFLECTION, Vol. 2192 of Lecture Notes in Computer Science, Springer, 2001, pp. 170–186.

[83] S. S. Huang, D. Zook, Y. Smaragdakis, Domain-specific languages and program generation with meta-AspectJ, ACM Trans. Softw. Eng. Methodol. 18 (2) (2008) 6:1–6:32.

[84] J. Fabry, T. Dinkelaker, J. Noyé, E. Tanter, A taxonomy of domain-specific aspect languages, ACM Comput. Surv. 47 (3) (2015) 40:1–40:44.

[85] T. Greifenberg, M. Look, S. Roidl, B. Rumpe, Engineering tagging languages for DSLs, in: MoDELS'15, IEEE, 2015, pp. 34–43.

[86] I. Córdoba-Sánchez, J. de Lara, Ann: A domain-specific language for the effective design and validation of Java annotations, Computer Languages, Systems & Structures 45 (2016) 164–190.

[87] A. D. Eisenberg, G. Kiczales, Expressive programs through presentation extension, in: AOSD'07, Vol. 208, ACM, 2007, pp. 73–84.