This document is published in:

Vitek, J. (ed.) (2010) *Objects, Models, Components, Patterns: 48th International Conference, TOOLS 2010, Málaga, Spain, June 28–July 2, 2010. Proceedings*. (Lecture Notes in Computer Science, 6141). Springer, 1-20.
DOI: http://dx.doi.org/ 10.1007/978-3-642-13953-6_1

Deep Meta-modelling with MetaDepth

Juan de Lara[1] and Esther Guerra[2]

[1] Universidad Autónoma de Madrid, Spain
Juan.deLara@uam.es
[2] Universidad Carlos III de Madrid, Spain
eguerra@inf.uc3m.es

Abstract. Meta-modelling is at the core of Model-Driven Engineering, where it is used for language engineering and domain modelling. The OMG's Meta-Object Facility is the standard framework for building and instantiating meta-models. However, in the last few years, several researchers have identified limitations and rigidities in such a scheme, most notably concerning the consideration of only two meta-modelling levels *at the same time*.

In this paper we present MetaDepth, a novel framework that supports a dual linguistic/ontological instantiation and permits building systems with an arbitrary number of meta-levels through deep meta-modelling. The framework implements advanced modelling concepts allowing the specification and evaluation of derived attributes and constraints across multiple meta-levels, linguistic extensions of ontological instance models, transactions, and hosting different constraint and action languages.

1 Introduction

Model-Driven Engineering (MDE) is a software development paradigm aiming at speeding up development times, while increasing quality and maintainability. MDE pursues these goals by treating models as the key assets of the process, being no longer mere documentation but used actively to (re-)generate code, as well as for validation and verification. Therefore, these activities demand computer-processable models with precise syntax. In MDE, models' syntax is defined through a meta-model that describes the set of valid models. Hence, meta-modelling is one of the pillars of MDE, being used for language engineering and domain modelling, and it is also at the core of other related approaches like product lines, feature-oriented development [6] and method engineering [12].

The OMG has proposed the Meta-Object Facility (MOF) [22] as the meta-modelling approach in the Model-Driven Architecture (MDA) [19], a particular incarnation of MDE. MOF has been adopted as a standard by many meta-modelling tools and frameworks, most notably by the Eclipse Modelling Framework (EMF) [24]. MDA proposes a four layer, linear meta-modelling architecture and a style of meta-modelling called *strict* in which an element of a meta-layer is the instance of exactly one element at the upper meta-level. Several authors have pointed out limitations of this approach [4,5,11,12], in particular concerning the existence of only one kind of instantiation relation and the constraint of considering only two adjacent meta-levels at the same time. Two meta-levels are enough to cover the *linguistic* case, where an object is an instance of exactly one class, but cannot capture in addition *ontological* instantiation relations within a domain. Hence, engineers are often forced to squeeze into two meta-modelling layers concepts that would naturally span several layers, resulting in more complex and cluttered models [5]. Moreover, the lack of uniformity employed in the concepts at the different layers in most approaches (e.g. UML associations are structurally different from links) makes it difficult to treat in a uniform way meta-models and models, as well as to link models in different meta-levels (since "meta-" is a relative term and meta-models are also models).

Several solutions have been proposed to these problems [5,11,12]. Their common idea is to increase the flexibility of the meta-modelling architecture by allowing an arbitrary number of meta-levels. In [5] a mechanism called *potency* was proposed, so that one model can control the properties of models that are indirect instances of it. In [1,4] a dual ontological and linguistic instantiation is proposed, allowing an element to be a linguistic instance (e.g. be an instance of `Class` in the upper linguistic meta-level) and also an instance of some domain concept (e.g. be an instance of `ProductType` in the upper ontological meta-level).

This paper presents MetaDepth, a new meta-modelling environment that allows modelling with an arbitrary number of ontological meta-levels. It implements the *potency* concept and permits a dual ontological and linguistic instantiation. It is distinguished from other similar frameworks [2,3] in that it supports advanced modelling concepts, like (OCL) constraints, derived attributes and transactions, and allows controlling whether ontological instance models can be linguistically extended. The purpose of the framework is to permit experimentation with this alternative way of meta-modelling, but at the same time provide a scalable, efficient system that permits its industrial use. Hence, MetaDepth can work in an interpreted mode, where a stack of models can be kept and worked with, and then allows compilation to obtain specialized code (in the line of JMI [25]) and optimized performance. The framework is integrated with the Epsilon languages [10], which permits using the Epsilon Object Language (EOL) [16] as an action language to define behaviour for meta-models, as well as the Epsilon Validation Language (EVL) [17] for expressing constraints. Both EOL and EVL are extensions of OCL. To the best of our knowledge, no framework with similar characteristics exists. Moreover, the interplay of potency with constraints, actions, multiplicities and association ends has not been properly addressed in the literature, nor have the mechanisms and benefits for controlling linguistic extensions. We also aim to contribute to the clarification of these issues.

Paper organization. Section 2 reviews multi-level meta-modelling and the concept of potency. Section 3 details the architecture of MetaDepth. Section 4 presents two examples that show the benefits of our approach. In the first one, we define a multi-level language through a unique meta-model. For example, one can think that in UML it is natural that `Objects` are instances of `Classes`, and hence should belong to a lower meta-level (so that an object diagram is an instance of a class diagram). Our framework naturally allows this, with the benefit of a less complex language definition. In the second case study, we solve the impedance mismatch arising when one needs to relate models at different meta-levels (a complicated technical issue in two-level frameworks such as EMF). Section 5 compares with related approaches and Section 6 concludes. A beta version of the tool can be downloaded from `http://astreo.ii.uam.es/∼jlara/metaDepth/`

2 Deep Meta-modelling

Some authors have pointed out the limitations of considering only two meta-modelling levels at the same time [5,11], either for language engineering or for domain modelling. A common example is the item-description or the type object pattern [18], where one needs to design a language containing both *ProductTypes* (e.g. Books) and *Products* (e.g. the book "Moby Dick"). In the classical meta-modelling approach, one would propose a two-level solution like the one to the left of Fig. 1. Although this solution is valid, it has some drawbacks. First, the user has to manually maintain the `type` links between each instance of `Product` and its `ProductType` at the model level. These links are indeed a (manually maintained) form of ontological instantiation relation for which the system does not provide automatic conformance checks. Should we have inheritance between types at the model level, it would have to be emulated manually too. Hence, this solution squeezes three meta-levels into two.

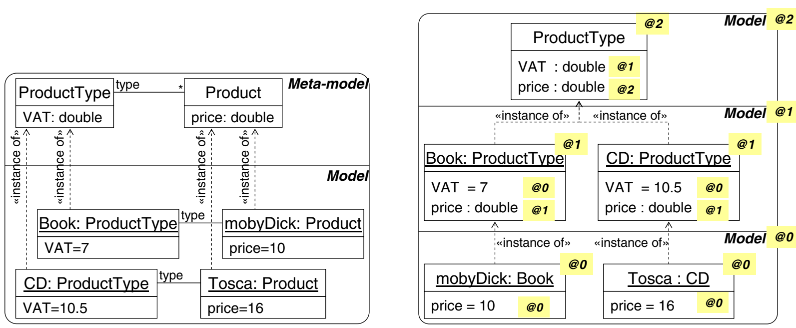

Fig. 1. A meta-model and a model including the *Type Object* pattern (left). The same system using deep meta-modelling (right), adapted from [18].

The solution to the right explicitly organizes the domain concepts into three levels. In this way, the `ProductType` is declared at the top-level, the different kinds of `ProductTypes` at the following meta-level, and the instances of these at the bottom level. This solution reduces accidental complexity, as one does not need the artificial class `Product` that the solution to the left introduced[1].

[1] One can still eliminate `Product` in the two-level solution by moving `Book` and `CD` to the top meta-model and setting them as subclasses of `ProductType`. However, this solution is not valid if we need to add new kinds of `ProductTypes` at run time.

Moreover, the instantiation of ProductTypes, Books and CDs is handled by the system thus enabling automatic conformance checks. Note that this pattern is ubiquitous in the definition of many languages, for example in UML (classes/objects), in web modelling languages (node types/node instances), role access control languages (user types/users) and so on. We call this style of meta-modelling, which considers more than two levels, *deep* or *multi-level meta-modelling*.

The solution meta-model to the left of Fig. 1 is able to control the attributes that instances of Product (mobyDick, Tosca) have. Hence, in deep meta-modelling we would expect the language designer to have the same level of control over *indirect* instances two or more meta-levels below. For this purpose, *potency* was proposed in [4] as a way to express how many times a property needs to be instantiated down the meta-levels before we get a *plain* instance and hence we have to assign it a value. The potency is a natural number that is assigned to properties, and which gets decremented each time we go down a meta-level. Hence, in our example, property VAT is assigned a value in the next meta-level, and price two meta-levels below. Not only properties can have potency, but also classes and associations. As we will see later, MetaDepth allows assigning potency to models, constraints and derived attributes as well.

Considering the solution to the right of Fig. 1, one realizes that the elements in the middle meta-level have both type and instance facets. This is so because they are instances of ProductType and, as they have potency bigger than zero, can be instantiated in turn. The term *clabject* was coined in [4] to refer to elements with a dual type/instance facet.

The ≪instance of≫ relation between the elements in different meta-levels is *ontological*, as this is a relation within the domain (i.e. mobyDick is a Book and this a ProductType). At the same time, as we have to use a modelling language to build the models, we can argue that Book is an instance of Class and mobyDick an instance of Object (if such concepts exist in the language). One way to support this duality is to introduce another instantiation dimension, called *linguistic*, and to have a meta-model that governs the linguistic constructions used by the models at the different meta-levels. While the ontological instantiation is a relation within a domain, the linguistic instantiation refers to implementation and provides the underlying modelling primitives. This situation is depicted in Fig. 2, where the linguistic meta-model contains concept Clabject with property potency to allow its instantiation in any meta-level. An important issue is that the union of the models in the three ontological meta-levels is a *strict* instance of the linguistic meta-model.

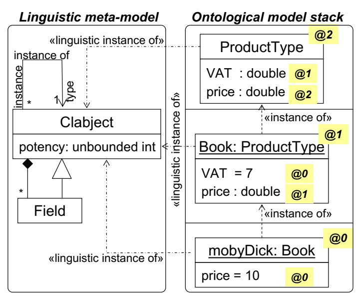

Fig. 2. Dual classification

After having introduced the basics of deep meta-modelling, there are still missing details. For example, how could constraints be introduced in the different ontological models? Should these and other elements like association ends be given a potency? Finally, one may wonder whether, as elements `Book` and `CD` have a type facet, we could declare new attributes for them, or whether we could *linguistically extend* a certain ontological model by introducing new types, instances of elements of the linguistic meta-model. In other words, do we demand strictness of the ontological ≪`instance of`≫ relation? The next section introduces MetaDepth's architecture, and discusses these issues.

3 The Architecture of MetaDepth

MetaDepth is a new meta-modelling system that we started to develop in 2008, based on the experience we gained with AToM^3 [8] in previous years. AToM^3 is a Python-based tool for the definition of the syntax of visual languages by meta-modelling and their semantics by graph transformation. MetaDepth is a completely rebuilt kernel, written in Java, which uses the deep meta-modelling approach presented in Section 2. It can work in two ontological instantiation modes: *strict* and *extensible*, as shown in Fig. 3.

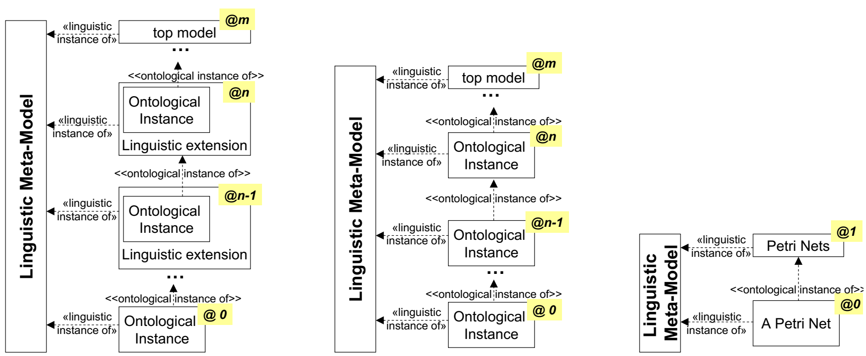

Fig. 3. MetaDepth instantiation schemes: extensible ontological instantiation (left) and strict ontological instantiation (center). Example of strict instantiation (right).

In the *extensible* case, each ontological instance model can be linguistically extended using the "horizontal" instantiation provided by the linguistic meta-model. Hence, instances of elements marked as `ext` can be extended with new attributes. A complete model can also be marked as `ext`, which means that new types can be added and that all its elements (except those explicitly marked as `strict`) can be extended. This situation is shown to the left of Fig. 3, which depicts how two models can be linguistically extended. In all cases strictness is kept for the linguistic instantiation dimension.

The *strict* case is closer to standard meta-modelling environments, where the top-level meta-model hard-codes all language concepts and can be subsequently instantiated ontologically, but such instances cannot be linguistically extended. This situation is represented in the center of Fig. 3. In this mode one could use the highest meta-level to describe the MOF meta-model with potency 2, such model could be ontologically instantiated to describe meta-models for languages at potency 1, which in turn could be instantiated to models of potency 0. Hence, the strict mode is similar to most meta-modelling environments (although without restrictions on the number of meta-levels). The right of the figure shows a simple case where the linguistic meta-model is directly used to define a meta-model for Petri nets, which is instantiated into a Petri net model.

Note that allowing linguistic extensions adds extra flexibility to this meta-modelling framework in two senses. First, at any potency bigger than zero clabjects retain its *type* facet, and hence can be allowed for linguistic extension (i.e. to define attribute types). On the other hand, it is often convenient to extend models by allowing the introduction of new linguistic elements, e.g. to adapt languages to particular usages, as we will see throughout the paper.

3.1 The Linguistic Meta-model

MetaDepth's linguistic meta-model took MOF as inspiration, but we have modified it to accommodate an arbitrary number of meta-levels, deep instantiation and potency. Fig. 4 shows a fraction of it, where the uncoloured concrete classes are those the designer typically instantiates when building a model (i.e. `Model`, `Node`, `Edge`, `Field` and `DerivedField`).

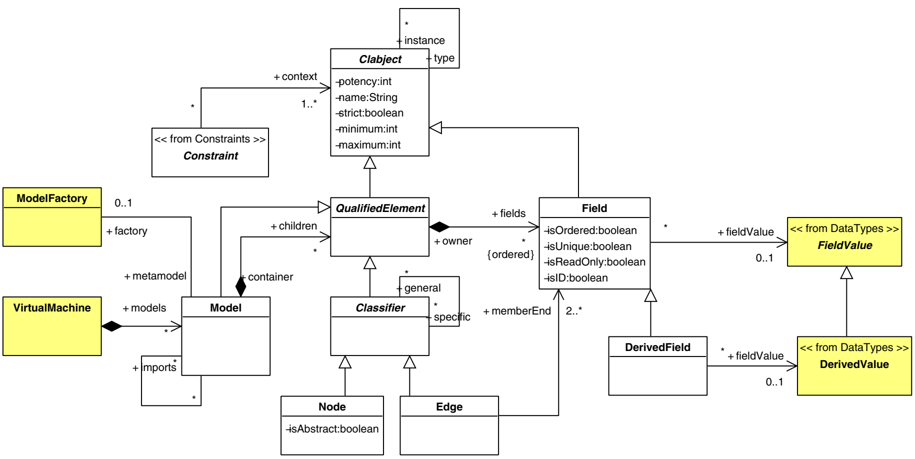

Fig. 4. MetaDepth's linguistic meta-model, partially shown

The root class `Clabject` takes responsibility of handling the dual type/object facet of elements. As such, it holds a potency value, as well as links to its type and instances. The potency can be unlimited, and in that case the instances of the clabject can have arbitrary potency (included unlimited). `Clabjects` also define a minimum and maximum multiplicity to control the cardinality of its instances within a given context. `Constraints` can be attached to clabjects, have a potency, and can specify on which events they should be evaluated (e.g. when creating or deleting the clabject). `QualifiedElements` are `Clabjects` owning some field. `Models`, `Nodes` and `Edges` are all `QualifiedElements`. `Classifiers` are a special kind of `QualifiedElement` that can form general/specific hierarchies, and are refined into `Nodes` and `Edges`. The latter have two or more association ends modelled as `Fields`. Finally, derived fields are fields of which their value is automatically calculated.

MetaDepth's modularity mechanism is based on the notion of `Model`, which can be nested, as shown by the composite pattern used. `Models` are `QualifiedElements` and hence can own `Fields` and have associated `Constraints`. Each model with potency bigger than zero has an associated `ModelFactory` in charge of instantiating the clabjects defined in the model. All working models are managed by a `VirtualMachine` container, which is a *singleton* object.

The framework supports the usual *atomic* data types, like integers, floating point numbers, strings, etc.; user-defined enumerations, as well as ordered and unordered collections with unique or non-unique elements.

3.2 Tool Support, Compiled and Interpreted Modes

MetaDepth models can be built through the provided Java API, or through a `CommandShell` and a textual syntax – similar to the Human Usable Textual Notation (HUTN) [21] – that we have built with ANTLR [23]. Loading and storing models is also done in this format. As an example, Listing 1 shows how the three models in Fig. 2 are defined with the textual syntax. The top-most model `Store` is assigned potency 2, which means it can be instantiated in the following two lower meta-levels. All elements defined inside `Store` have the potency of the container clabject, hence we only need to explicitly declare potencies different from 2 (field `VAT` in this case). Please note also that we give an initial value to the fields of `ProductType`. Although all elements in the listing are given an explicit name, this is not mandatory: we can declare *anonymous* clabjects like `Book {price=10;}` (i.e. the book is not given a name) and the system assigns them a unique UUID-based identifier.

The framework is fully integrated with Epsilon, a family of languages built on top of the Epsilon Object Language (EOL) [16], which extends OCL with expressions permitting secondary effects such as assignments and methods. The integration was possible because EOL communicates with the models through a connectivity layer. Thus, EOL can work with EMF models, but also with any other model technology that implements the interface of this connectivity layer. We implemented such an interface and provided support to make EOL aware of the multiple ontological levels. The solution is very practical as one can use EOL programs to build models as in Listing 2. The listing shows a typical interaction with the command line interpreter. In line 1 we enter in the context of model *MyLibrary* (we could have created a new model as well). In line 2 we begin writing the EOL program (which could be loaded from a file as well). Then the program inserts 1000 new books in the model and initializes their price.

```
Model Store@2 {
  Node ProductType {
    VAT@1 : double = 7.5;
    price : double = 10;
  }
}
Store Library {
  ProductType Book { VAT = 7; }
}
Library MyLibrary {
  Book mobyDick { price = 10; }
}
```

Listing 1. A simple METADEPTH three-model stack

```
context MyLibrary
::entering context MyLibrary
# EOL
:: entering eol execution mode
for (i in Sequence{1..1000}) {
  var b : new Book;
  b.price := 10+i/500;
}
```

Listing 2. A simple EOL program to populate a METADEPTH model

We can use EOL not only for initializing models, but also to define their behaviour. As an example, Listing 6 shows a Petri net simulator we have built for Petri net models. Moreover, we can use the rest of the Epsilon languages with our METADEPTH models so that it is possible to transform models with the Epsilon Transformation Language [10].

Another feature of METADEPTH concerns undoability of actions. Using the *Command* pattern, all API calls are recorded in an event list, and each command provides an appropriate undo function. This allows undo/redo of any action on models, and permits integration with the transaction syntax of EOL.

By default, METADEPTH works in interpreted mode. This allows for flexible modelling and is useful for rapid prototyping of languages, as one can evolve models and meta-models at the same time. One can also create several independent models in the same `VirtualMachine` (i.e. models do not need to be related through instantiation). Once the meta-models are ready, they can be *compiled* if so desired. We have built a code generator that produces specialized classes inheriting from the classes in the meta-model of Fig. 4, as well as interfaces declaring getter, setter and creation methods that follow the JMI specification [25]. This enables interface compatibility with applications that handle JMI meta-data (like those of the EMF), and improves performance as we generate optimized code which improves, e.g., constraint evaluation, object creation and field access. However, compiled meta-models are less flexible because they can no longer be modified, even though all their properties are readily accessible. The compilation we have implemented is more complex than in normal two-level meta-modelling frameworks, since compiling a model with certain potency implies compiling all direct and indirect models above in the ontological meta-level hierarchy. The compilation also generates a specialized command shell that initializes the `VirtualMachine` with the compiled meta-models and allows their instantiation.

3.3 Constraints and Derived Attributes

Constraints and actions can be defined using Java or EOL. Constraints and derived attributes have an assigned potency that governs the meta-level at which they have to be evaluated. For example, Listing 3 modifies the running example by adding a few constraints and a derived attribute on the top-level model.

```
1  Model Store@2 {
2   Node ProductType@2 {
3    VAT@1      : double = 7.5;
4    price@2    : double = 10;
5    discount@2: double = 0;
6    minVat@1   : $self.VAT>0$
7    minPrice@2: $self.price>0$
8    maxDisc@2 : $self.VAT*self.price*0.01+self.price<self.discount$
9    /finalPrice@2:double=$self.VAT*self.price/100+self.price-self.discount$;
10  }
11 }
```

Listing 3. Constraints and derived attributes in MetaDepth

The previous listing adds property `discount` to `ProductType`, declares three constraints in lines 6, 7 and 8, and defines a derived field in line 9. Constraints are specified between two “$” symbols, preceded by their identifier, and can be declared inside the context of a clabject (as done in this case), or be declared outside and then explicitly assigned to one or more clabjects, promoting reusability. The constraint in line 6 has potency 1, therefore it will be evaluated in the next meta-level below. This constraint cannot access the value of fields with potency bigger than 1, like `price`, as these may not have a value[2]. The default language for constraints is EOL, but one can also use Java. For example, the equivalent Java code to the constraint in line 6 is `minVat[Java]@1: $((Integer)self.getValue("VAT"))>0$`, which is more verbose but permits interacting with external Java programs.

Constraint `maxDisc` is more interesting as it uses fields with potency 1 and 2. This is allowed as, from the point of view of action and constraint languages, fields whose value is given in a type are accessed in the same way as fields whose value is given in the instance. In our example, `mobyDick` interprets `VAT` like a static field for which its value was set at the upper meta-level. This feature simplifies writing constraints spawning several meta-levels.

[2] In our case `price` does have a value as it has been initialized with the value 10, but this is not the general case.

```
Store Library {
  ProductType Book {
    VAT    = 7;
    title  : String;
    author : Author;
  }
  Node Author {
    name   : String;
    nonRep : $Author.allInstances().forAll(x|x<>self implies
                                        x.name<>self.name)$
    books : Book[1..*]{unique};
  }
  Edge writer (Book.author, Author.books) {
    year : int;
  }
}
```

Listing 4. Linguistic extensions and associations in MetaDepth

Finally, `ProductType` defines the derived field `finalPrice`. The declaration of derived fields is similar to the declaration of normal fields, but these are preceded by a backslash, and include a calculation function in EOL or Java that can use fields with equal or lower potency. Our current implementation calculates derived field values in a lazy way, whenever they are accessed by some getter function. This works well in textual modelling environments, but we foresee the need for a change propagation algorithm in case some exogenous observer (e.g. a graphical visualization) needs the value.

3.4 Controlling Linguistic Extensions

MetaDepth supports both *strict* and *extensible* ontological instantiation, the latter being the default. Linguistic extension is interesting to permit unforeseen extensions to Domain Specific Languages (DSLs) spawning more than one level, as our running example. In these languages, the top-most meta-model is usually highly generic, and hence extensions at lower levels are often required.

Listing 4 shows an extension of the running example, where an extensible instantiation of model `Store` is used to define a `Library`. In this usage scenario we are interested in associating an author to `ProductType` instances (i.e. to `Books`). Thus, we add to the library a new node `Author`, instance of `Node` in the linguistic meta-model. For the sake of illustration, `Author` is provided with the constraint `nonRep` that forbids replicating names. This shows that *allInstances* effectively returns all ontological instances of `Author`. As an alternative, we could have just assigned the modifier {`id`} to the field `name` to obtain the same behaviour. Please note that `Library` is still a strict instance of the linguistic meta-model.

`Authors` are related to one or more `Books`, which is modelled through their field `books`. The {`unique`} modifier ensures that a given `Author` is not related to the same `Book` twice. Other supported modifiers are `id` (ensures uniqueness

```
strict Model PetriNets@1{
  abstract Node NamedElement { name : String{id}; }
  Node Place : NamedElement {
    tokens    : int = 0;
    outTrans  : Transition[*] {ordered,unique};
    inTrans   : Transition[*] {ordered,unique};
    minTokens : $self.tokens>0$
  }
  Node Transition : NamedElement {
    inPlaces  : Place[*] {ordered,unique};
    outPlaces : Place[*] {ordered,unique};
  }
  Edge ArcPT(Place.outTrans,Transition.inPlaces){ weight : int = 1; }
  Edge ArcTP(Transition.outPlaces,Place.inTrans){ weight : int = 1; }
  minWeight(ArcPT, ArcTP) : $self.weight>0$
  minPlaces : $Place.allInstances()->size()>0$
}
```

Listing 5. A meta-model for Petri nets

of values among all clabjects in the same context), `ordered` (retains the order of elements) and `readOnly` (forbids changing the value).

Associations can be provided with fields (i.e. similar to association classes) by explicitly defining an `Edge` between their association ends. An example is shown in lines 12–14 of Listing 4, where the `Edge` relating books and authors includes the year in which the book was written. As in UML, declaring such an `Edge` has the effect of allowing the navigation from an `Author` to all its edges through `self.writer`, while the direct navigation from an `Author` to its `Books` is done by `self.books`. In the context of the `writer` edge, it is possible to navigate to the `Author` and `Books` through the `author` and `books` ends.

In the example we have made reference to the ontological types `Author` and `Book` to declare associations. However, in specific situations, it is useful to refer to linguistic types, like `Node`, when defining association ends. This makes sense if

```
while (Transition.allInstances()->exists(t |
                                  t.enabled() and t.fire())) {}
operation Transition enabled() : Boolean {
  return self.ArcPT->forAll(arc | arc.inPlaces.tokens>=arc.weight);
}
operation Transition fire() : Boolean {
  for (arc in self.ArcPT)
    arc.inPlaces.tokens := arc.inPlaces.tokens-arc.weight;
  for (arc in self.ArcTP)
    arc.outPlaces.tokens := arc.outPlaces.tokens+arc.weight;
  return true;
}
```

Listing 6. A simulator for Petri nets defined with EOL we want to specify that a certain association end is to be taken by any (linguistic) instance of `Node`. As the next section will show, this is very useful if we want to relate models of different potency.

Listing 5 shows an example of *strict* meta-modelling. It is a meta-model for Petri nets containing `Places` and `Transitions` (both inheriting from `NamedElement`), as well as weighted arcs. All these elements inherit the *strict* modifier from their container model. In the example, `NamedElement` and its children have the same potency 1, but MetaDepth also allows clabjects in a hierarchy to define different potencies. A clabject keeps the biggest potency of all its ancestors.

The Petri net meta-model defines several constraints. In the context of the model, `minPlaces` restricts nets to have at least 1 place (line 16). Most meta-modelling approaches do not allow global constraints; however some constraints (like `minPlaces`) are inherently global and do not fit in the context of any class in the meta-model. As all MetaDepth elements have built-in cardinalities (see Fig. 4), we can obtain the same restriction as `minPlaces` by replacing line 3 with "**Node** `Place[1..*] : NamedElement {`". Constraint `minWeight` is also defined globally, but it is assigned to both kinds of arcs to enforce a positive weight (line 15). This has the advantage of promoting reusability as the constraint does not have to be defined twice.

For the sake of completeness, Listing 6 shows a simulator for Petri nets specified through EOL. This language allows adding operations on meta-classes, and we have used this feature to define the operations `enabled` and `fire` on node `Transition`. The first one contains pure OCL code, which checks if the transition is enabled. For this purpose it iterates through all incoming arcs checking that the number of tokens in the pre-place is bigger or equal than the arc's weight. Operation `fire` has secondary effects: the removal and addition of tokens to the pre- and post-places of the transition. The main simulation loop is defined in line 1, which iterates on all transition instances while there is some enabled, and then fires it.

4 Examples

This section presents two examples that show the usefulness of MetaDepth and help to illustrate some of its distinguishing features, such as the use of linguistic types or the interplay of potency, multiplicities and association ends.

4.1 Defining Multi-level Languages

The first example shows the use of deep meta-modelling for defining DSLs spawning more than one meta-level. This is the case of many languages that implement the *Type Object* pattern. For example, UML defines class and object diagrams as two different structural diagrams. However, UML defines both in the same meta-level, with the drawback that one has to maintain explicit relations between objects and their classifiers and ensure that they remain consistent. Instead, one can use deep meta-modelling to simplify the language definition and to automate the maintenance of consistency between classes and objects.

Listing 7 shows how a simple language containing class and object diagrams is defined in MetaDepth. The idea is to specify a three-level meta-modelling architecture where the top-most level contains the definition of class diagrams and potency 2 (Model `ClassDiagram` in Listing 7). In this way, in the next meta-level we can build class diagrams (e.g. `Zoo` in Listing 8), and in the bottom meta-level we can build object diagrams and the built-in meta-modelling infrastructure of MetaDepth handles the type checking with respect to the class diagram that the object diagrams instantiate. In this way, a stack of two languages is defined with just the model in Listing 7. This model is *strict* to avoid the creation of new linguistic types in class and object diagrams, and permit only the creation of `Class`es and `Assoc`s. On the contrary, classes and associations are *extensible* to allow their instances to define new fields in them. Node `Class` contains field `isAbstract` to designate whether the class is abstract or not, and constraint `noAbsObjects` ensures that object diagrams (two levels below) do not contain objects whose class is abstract. The constraint is evaluated two levels below, on all indirect instances of `Class`, because the constraint has potency 2 (as it is not explicitly specified, it receives the potency of the owner clabject).

```
strict Model ClassDiagram@2 {
  ext Node Class {
    isAbstract@1 : boolean = false;
    in           : Class[*];
    out          : Class[*];
    noAbsObjects : $self.isAbstract=false$
  }
  ext Edge Assoc(Class.out, Class.in);
}
```

Listing 7. A meta-model for class and object diagrams (meta-level 2)

Listing 8 shows an instance of `ClassDiagram`, namely a class diagram named `Zoo` which declares two classes and one association. Class `Person` declares one field (`name`) and one association end (`pet`). The field `name` has been added as a linguistic extension to `Person` (that is, `name` is not an instance of any feature in the upper ontological meta-level). The association end `pet` is an ontological instance of the association end `out` defined for `Class` in the upper meta-level. This fact is indicated with modifier {`out`} in line 4 of Listing 8. The listing also shows an instance (i.e. an object diagram) of the defined class diagram called `myZoo`.

Revisiting our language specification in Listing 7, we can see that fields `in` and `out` have an (unbounded) multiplicity. This multiplicity constrains the meta-level immediately below, and hence one can have an unbounded number of instances of them in a class diagram. These instances can declare their own multiplicity, which affects their instantiation in object diagrams. However, please note that

```
ClassDiagram Zoo {
  Class Person {
    name : String {id};
    pet  : Animal[*] {out};
  }
  Class Animal {
    kind  : String {id};
    owner : Person[1..*] {in};
  }
  Assoc hasPet (Person.pet, Animal.owner) { since : int; }
}
Zoo myZoo {
  Person p { name = "Juan"; }
  Animal a { kind = "monkey"; }
  hasPet(p,a) { since = 2010; }
}
```

Listing 8. A class and an object diagram (meta-levels 1 and 0, respectively)

```
Model Graphics@2 {
  abstract Node Figure {
    x : int;
    y : int;
    rotation : double = 0;
    scale    : double = 1;
    refersTo : Node;
  }
  Node Rectangle : Figure {
    width@1  : int;
    height@1 : int;
  }
}
```

Listing 9. The Graphics meta-model

association ends `in` and `out` have potency 2 and are also available in object diagrams, storing the content of all their instances. For example in Listing 8, `p.out` evaluates to `[a]` because `p.pet` evaluates to `[a]`. This shows that the instantiation semantics for fields is coherent with the way of handling indirect instances of all other elements, like `Nodes` and `Edges`, as e.g., both `p` and `a` are indirect instances of `Class`.

4.2 Relating Models at Different Meta-levels

Next we illustrate how to relate models at different meta-levels in MetaDepth. This is of practical importance as, in standard approaches like in EMF, models cannot be meta-models at the same time. In EMF, models can be treated as meta-models by passing through a transformation called *promotion*, thus making it difficult to link elements of models with elements of meta-models. As an example we show a multi-level language to define the graphical concrete syntax of meta-models. The purpose of the example is not to show advanced concrete syntax concepts, nor to discuss how instances would be graphically rendered, but only show how models can be naturally put in relation with meta-models. In order to keep the example simple we restrict our discussion to the visual representation of `Nodes` as rectangles.

Listing 9 shows the meta-model for the two-level language called `Graphics`. When instantiated in the next meta-level it allows defining visualizations for a meta-model *M*. Then, by instantiating it again we obtain instances with the rendering information about the instances of *M*. This situation is depicted in Fig. 5, which shows that `Graphics` models are associated to the definition of a language, and the rendering information to models of this language. The `Graphics` definition contains an abstract clabject `Figure` with fields `x` and `y` that store the absolute position of instances of figures two meta-levels below, as well as their rotation and scaling. Figures also contain field `refersTo` to point to the `Node` that the figure is representing. A real-world model for concrete syntax would include several types of figures (circles, text, etc.) that could be composed to form the visual concrete syntax of a `Node`. In our case, for space constraints, we consider just `Rectangles` which can be configured with their dimension.

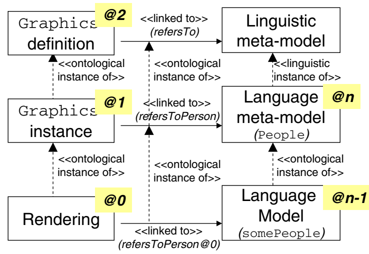

Fig. 5. Scheme of the example

```
1  Model People@1 {
2    Node Person { name : String; }
3  }
4  Graphics ConcreteSyntax imports People {
5    Rectangle iconPerson {
6      width  = 15;
7      height = 10;
8      refersToPerson: Person{refersTo};
9    }
10 }
```

Listing 10. Defining the concrete syntax for a language meta-model

Consider the simple language defined by the meta-model `People` in Listing 10. We can instantiate the `Graphics` meta-model to define a visual representation for `Person` in the language meta-model. This is possible because we have defined association end `refersToPerson` in line 8, which instantiates the association end `refersTo`. Association end `refersToPerson` points to `Person`, which is a linguistic instance of `Node`, hence being type compatible with the definition of `refersTo` in line 7 of Listing 9. Moreover, since meta-model `Graphics` is generic,

```
People somePeople {
  Person e {
    name = "Esther";
  }
}
ConcreteSyntax concreteSyntax01 imports somePeople {
  iconPerson {
    x = 10; rotation = 90;
    y = 10; scale = .5;
    refersToPerson = e;
  }
}
```

Listing 11. Instantiating the meta-model and its concrete syntax

we can also define a visualization for the class diagram `Zoo` in Listing 8 by just replacing "People" by "Zoo" in line 4 of Listing 10. This shows the flexibility of MetaDepth to relate models at different potencies and meta-levels, and the advantages of treating uniformly models at different meta-levels.

Finally, we can instantiate the `People` meta-model and its associated concrete syntax as shown in Listing 11.

5 Related Work

There are two main lines of related research: those works following a MOF-like way of meta-modelling, where only two adjacent levels are considered at the same time, and those following a deep meta-modelling approach.

Regarding the first group, MOF [22] is the OMG's language to specify meta-models and the most adopted approach in practice. The MOF specification is divided in two parts: a basic one called Essential MOF (EMOF), and a more advanced one called Complete MOF (CMOF). The specification claims that it can be used with as many meta-levels as users demand. However, there are conceptual problems, e.g. when one needs to introduce data type instances, as the basic data types would have to be replicated across the different meta-levels. Moreover, current implementations only allow handling two levels at the same time. Even though MOF specifies a set of reflective services, the specification neglects that considering three or more meta-levels at the same time requires some entities to simultaneously have both type and object facets. The main implementation of MOF is integrated within EMF [24] and is called *Ecore*. It forces a tree-based edition of models, and only supports EMOF, therefore lacking useful constructs like a proper concept of association enabling the definition of associative classes. Neither EMOF nor CMOF specify how to define constraints or actions to calculate derived attributes.

Many current meta-modelling research efforts revolve around MOF. For example, KM3 [14] is a DSL to specify meta-models based on MOF. KM3 has a textual front-end for meta-modelling frameworks, and as such can be compared to the textual notation we have developed for MetaDepth, but does not introduce new meta-modelling concepts. Kermeta [15] is another textual language whose purpose is to specify behaviours for EMF meta-models. Hence, its role for EMF meta-models is equivalent to our use of EOL to specify behaviour. In [9], Smalltalk is used to implement a reengineering framework based on EMF, which allows using Smalltalk also as an executable meta-language.

XMF [7] is a complete MDE framework which allows language-driven development. Hence, XMF permits defining the abstract and concrete syntax of languages (both textual and diagrammatic), their semantics as well as transformations. In XMF, behaviour is expressed by means of XOCL, an extension of OCL with imperative constructs in the line of EOL. XMF has meta-modelling facilities supporting an arbitrary number of meta-levels, a meta-object protocol and a language for expressing mappings between meta-models. However, even though XMF permits an arbitrary number of meta-levels, it lacks constructs (like potency) to control the structure of deep instances of a model, making it difficult to define deep languages with it.

Concerning deep meta-modelling, several efforts can be recently found directed to a practical test of the seminal ideas of Atkinson and Kühne [4]. For example, DeepJava [18] is an extension of Java with the concept of potency and, as such, it cannot be considered a meta-modelling framework. It provides methods with potency, but has to use special keywords to navigate up the type hierarchy in order to find attribute values. On the contrary, our constraints and computations for derived attributes can access type fields in a uniform way. This is similar to considering that a type attribute value is like a *static attribute* with respect to an instance, and has the advantage that one does not need to know exactly how many meta-levels up the given field was given a value, and facilitates the integration with constraint and action languages.

The work in [3] is another recent proposal for deep meta-modelling. The tool is currently being developed, based on *Ecore*. They consider multi-level constraints, and propose extending OCL to cope with multiple ontological meta-levels. This is similar to our approach, but we assign *potency* to constraints, making them easier to define. This is so because potency layers constraints, and hence they do not have to explicitly invoke *allInstances()* a predefined number of times, it is enough to implement in the OCL interpreter the ability to recognise indirect ontological instances of clabjects, and to interpret fields of ontological types similar to static attributes in Java. Another difference concerns relations, as they do not consider association ends, but add this information inside the relation class itself. The main motivation for this is graphical visualization in concrete syntax and uniformity of structure between ontological meta-levels. On the contrary, the design of our framework was not driven by concrete syntax issues, as we foresee building systems supporting graphical syntax as well as a more sophisticated textual syntax, probably posing different challenges. We retain association ends (similar to MOF and UML), as this allows us to reuse the multiplicity semantics of structural features. Most importantly, it makes easier the practical integration with navigation languages like OCL. We also agree on the importance of uniformity of relations at the different levels, and hence we retain association ends (i.e. fields) at all levels. Finally, [3] does not consider linguistic extensions, transactions, derived attributes, nor an action language, whereas we can use the Epsilon languages, which enable manipulation and transformation of models. Our dual working scheme of interpretation/compilation allows rapid prototyping of languages, and enables the generation of stand-alone, efficient domain-specific tools.

Nivel [2] is a deep meta-modelling framework based on the weighted constraint rule language (WCRL). It implements the concept of potency and the dual linguistic/ontological classification. It brings some interesting ideas from conceptual modelling, like the possibility of several classes to implement an association role. Nivel's semantics is given by its translation into WCRL, which allows some form of automated reasoning, but the kind of reasoning and its usefulness was not shown in [2]. The language lacks constraints and action languages (except WCRL itself), which hinders its use in practical MDE.

Other works that have influenced MetaDepth include Amulet [20], whose prototype/instance concept is similar to our linguistic clabject extensions.

6 Conclusions and Future Work

In this paper we have presented MetaDepth, a novel framework for deep meta-modelling. The tool supports the concept of potency, allowing an arbitrary number of ontological meta-levels. It provides advanced features like multi-level constraints, derived attributes and linguistic extensions at lower meta-levels. The framework can work either in interpreted or compiled modes, favouring flexibility or efficiency. The current implementation offers a textual syntax, inspired by HUTN, and is integrated with the Epsilon languages. In particular, we have shown the use of EOL for specifying actions and EVL for constraints.

Concerning future work, we will continue improving MetaDepth in the near future. For example, we would like to allow the framework to run in client/server mode, so that the kernel can be accessed through web services. We would also like to build a system to support a graphical concrete syntax, in the spirit of the old $AToM^3$ [8], but allowing interaction through the web navigator. Even though we can use ETL now, the plan is to incorporate a formal model transformation language into it, and for this purpose we are working on an implementation of our pattern-based transformation language [13]. It could be interesting to study the implications of deep meta-modelling for model transformation, and for this we would need a formalization of the framework. Finally, we believe that deep meta-modelling allows improving the current practice of MDE. We are currently exploring idioms, and identifying good practices and patterns for deep meta-modelling and multi-level language engineering.

Acknowledgements. This work has been partially sponsored by the Spanish Ministry of Science and Innovation, under project "METEORIC" (TIN2008-02081), and mobility grants JC2009-00015 and PR2009-0019, as well as by the

R&D programme of the Community of Madrid, project “e-Madrid” (S2009/TIC-1650). The authors are grateful to the five anonymous reviewers that helped in improving the paper, as well as the research group of Prof. R. Paige at the University of York, and Prof. H. Vangheluwe for their comments and discussions.

References

1. Álvarez, J.M., Evans, A., Sammut, P.: Mapping between levels in the metamodel architecture. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 34–46. Springer, Heidelberg (2001)
2. Asikainen, T., Männistö, T.: Nivel: a metamodelling language with a formal semantics. SoSyM 8(4), 521–549 (2009)
3. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. IEEE Trans. Soft. Eng. 35(6), 742–755 (2009)
4. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Trans. Model. Comput. Simul. 12(4), 290–321 (2002)
5. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. SoSyM 7(3), 345–359 (2008)
6. Batory, D.S.: Multilevel models in model-driven engineering, product lines, and metaprogramming. IBM Systems Journal 45(3), 527–540 (2006)
7. Clark, A., Sammut, P., Willans, J.: Applied Metamodelling: A Foundation for Language Driven Development, 2nd edn., Ceteva (2008), `http://itcentre.tvu.ac.uk/~clark/Publications.html`
8. de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
9. Ducasse, S., Girba, T., Kuhn, A., Renggli, L.: Meta-environment and executable meta-language using Smalltalk: An experience report. SoSyM 8, 5–19 (2009)
10. Epsilon (2009), `http://www.eclipse.org/gmt/epsilon/`
11. González-Pérez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework. SoSyM 5(1), 72–90 (2006)
12. González-Pérez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley, Chichester (2008)
13. Guerra, E., de Lara, J., Orejas, F.: Pattern-based model-to-model transformation: Handling attribute conditions. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 83–99. Springer, Heidelberg (2009)
14. Jouault, F., Bézivin, J.: KM3: A DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
15. Kermeta, `http://www.kermeta.org/`
16. Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
17. Kolovos, D.S., Paige, R.F., Polack, F.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 204–218. Springer, Heidelberg (2009)
18. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. In: OOPSLA’07, pp. 229–244. ACM, New York (2007)

19. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled. Addison-Wesley Object Technology Series (2004)
20. Myers, B., McDaniel, R., Miller, R., Ferrency, A., Faulring, A., Kyle, B., Mickish, A., Klimovitski, A., Doane, P.: The Amulet environment: New models for effective user interface software development. IEEE Trans. Soft. Eng. 23(6), 347–365 (1997)
21. OMG. HUTN (2009), http://www.omg.org/cgi-bin/doc?formal/2004-08-01
22. OMG. MOF 2.0 (2009), http://www.omg.org/spec/MOF/2.0/
23. Parr, T.: ANTLR (2010), http://www.antlr.org
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading (2008), http://www.eclipse.org/modeling/emf/
25. Sun. Java Metadata Interface, http://java.sun.com/products/jmi/index.jsp