Domain-Specific Textual Meta-Modelling Languages for Model Driven Engineering

Juan de Lara and Esther Guerra

Universidad Autónoma de Madrid (Spain)

Abstract. Domain-specific modelling languages are normally defined through general-purpose meta-modelling languages like the MOF. While this is satisfactory for many Model-Driven Engineering (MDE) projects, several researchers have identified the need for *domain-specific meta-modelling* (DSMM) languages providing customised meta-modelling primitives aimed at the definition of modelling languages in a specific domain, as well as the construction of meta-model families.

In this paper, we discuss the potential of *multi-level meta-modelling* for the systematic engineering of DSMM architectures. For this purpose, we present: (i) several primitives and techniques to control the meta-modelling facilities offered to the users of the DSMM languages, (ii) a flexible approach to define textual concrete syntaxes for DSMM languages, (iii) extensions to model management languages enabling the practical use of DSMM in MDE, and (iv) an implementation of these ideas in the METADEPTH tool.

Keywords: Model-Driven Engineering, Deep Languages, Domain-Specific Meta-Modelling, Textual Concrete Syntax, Multi-Level Transformations

1 Introduction

Model-Driven Engineering (MDE) promotes an active use of models throughout the software development. These models are sometimes defined using general-purpose languages like the UML, but for restricted, well-known domains, it is also frequent the use of Domain-Specific Modelling Languages (DSMLs).

In current MDE practice, DSMLs are built by the language designer using a meta-model defined with a general-purpose meta-modelling language, like the MOF. This meta-model describes the instances that the users of the language can build in the immediate meta-level below. Thus, DSMLs usually comprise two meta-levels: the definition of the DSML and its usage. More recently, several researchers [9, 16] have pointed out the utility of using domain-specific meta-modelling (DSMM) languages as a means to provide domain-specific meta-modelling primitives to customize families of similar DSMLs, e.g., for expressing traceability [16], variability [16] or to define domain-specific process modelling notations [9] and DSML profiles [13]. In this case, the language spans three meta-levels: definition of the DSMM language for a specific domain, definition of the DSML by using the constructs provided by the DSMM language, and usage of the DSML. Unfortunately, existing approaches to DSMM are generally based on a two meta-level setting and the definition of ad-hoc "promotion" transformations between models and meta-models, which makes the adoption of DSMM cumbersome in practice. Moreover, there is no general framework for defining DSMM languages with integrated MDE support.

In this paper, we propose multi-level meta-modelling [5] as an underlying framework for DSMM, and discuss mechanisms to facilitate its adoption in MDE projects. Multi-level meta-modelling allows the definition of *deep* languages, which can be instantiated in more than one meta-level. In this way, the users of the language perform DSMM as, in each meta-level, the constructed models are instances of the upper meta-level but also meta-models w.r.t. the meta-level below. In our context, this means that a DSML is naturally defined as an instance of a DSMM language, and at the same time, it acts as a meta-model for lower meta-levels (i.e., it defines a language). Moreover, we provide: (a) means to customize the meta-modelling features that will be offered to the users of the DSMM languages, (b) a flexible way to define textual concrete syntaxes at every meta-level, and (c) model management languages able to work in a multi-level setting, enabling the use of DSMM in MDE projects (for space constraints we just show the use of model transformations). The framework is supported by MetaDepth [6], a multi-level meta-modelling tool supporting deep characterization through potency [5], and dual ontological/linguistic typing.
Paper Organization. Sec. 2 discusses related research, exposing motivations and needs in the area. Sec. 3 applies multi-level meta-modelling to DSMM and identifies some challenges: how to customise the DSMLs in a DSMM framework (Sec. 4), how to define a concrete syntax for the DSMLs (Sec. 5), and how to manipulate models in a multi-level setting (Sec. 6). Finally, Sec. 7 concludes.

2 Related work

Several researchers have pointed out the benefits of using DSMM languages. For example, the traceability modelling language [16] (TML) is a DSMM language used to express the allowed traces and constraints between several meta-models. Its rationale is that TML users do not need the full power of EMF or MOF to construct trace meta-models, but they benefit from specific meta-modelling primitives like `Trace` and `TraceLink`. Other DSMM languages are described in [16] to express variability over DSMLs, and to extend DSMLs with interfaces for model reuse. However, no general framework for defining such DSMM languages is proposed. Instead, they use two meta-levels and define ad-hoc "promotion" transformations between models (e.g., a TML model) and meta-models (the resulting trace meta-model). These transformations are a way to emulate three meta-levels within two, hindering the construction of DSMM languages.

In [8], the authors present a language to declare component types with ports, which can be instantiated choosing a number of port instances. This DSMM language is defined in a two meta-level framework extended with capabilities to instantiate the components, emulating the existence of two meta-levels within one.

The price to pay is that one has to manually encode support for the definition of class/features/data types and their instantiation, the definition and evaluation of constraints, and the emulation of inheritance within a single meta-level.

In [13], the UML profiling mechanism is adapted for EMF-based DSMLs. This is another example of DSMM as users need a language to define new profiles and apply them at the meta-level below. Again, a two meta-level setting forces the use of workarounds. In this case, they emulate the existence of attribute instances at the lowest meta-level by the run-time adaptation of the meta-model, injecting new attribute types and classes.

Instead of emulating several meta-levels within two [8] or using artificial workarounds [13, 16], we claim that a more natural way to define DSMM languages is the native use of multi-level meta-modelling, also known as deep meta-modelling [6]. Previously, Jablonski et al. [9] used multiple levels to build domain-specific process modelling notations. However, their approach is restricted to meta-modelling, and does not consider the language concrete syntax or its manipulation through model management languages, hindering its use in MDE.

Here, we propose some mechanisms to handle these deficiencies based on some multi-level meta-modelling techniques developed originally by Atkinson and Kühne [3]. There are several multi-level meta-modelling frameworks [1, 2, 12]. For example, DeepJava [12] extends Java to allow multiple instantiations, whereas the main concern in [1] is the efficient navigation between meta-levels. In [2], the authors discuss the visualization of multi-level languages but do not consider linguistic extensions or the integration with model management languages. All these frameworks either do not consider concrete syntaxes [1, 12] or do not integrate model manipulation languages enabling their use for MDE [2].

Although there are many approaches to define textual concrete syntaxes for DSMLs [7, 10], their definition for DSMM languages poses new challenges. For instance, there is the need to define the concrete syntax for models several meta-levels below, for which the concrete types that will be available in the models are unknown in advance. Sometimes, it is also necessary to extend the predefined concrete syntax for a particular DSML built using a DSMM language. This enables a progressive refinement of concrete syntaxes at different meta-levels.

Altogether, our contribution is a comprehensive framework to define DSMM language environments based on multi-level meta-modelling. Our approach covers the definition of textual concrete syntaxes, a fine grained customization of the meta-modelling facilities offered to the DSMM language users, and model management languages tailored to a multi-level setting.

3 Deep meta-modelling for domain-specific meta-modelling

In DSMM, users are not given the full power of a general-purpose meta-modelling language, but a more suitable meta-modelling language that contains primitives of the domain, and that is restricted for a particular meta-modelling task.

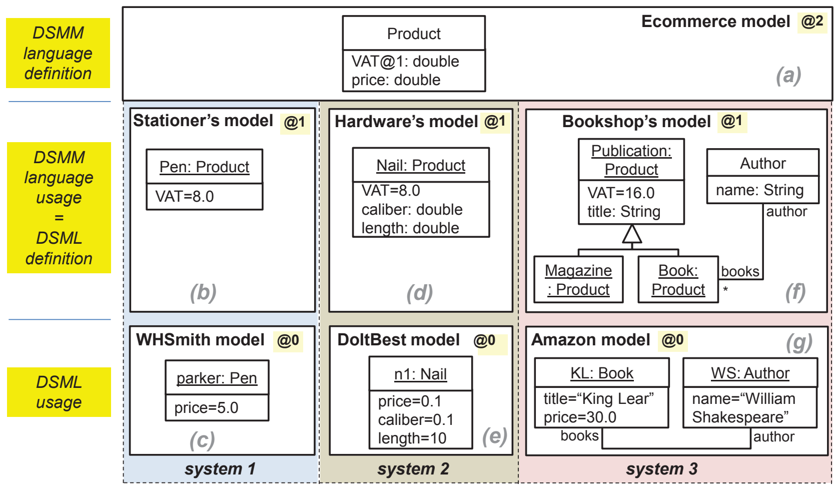

Fig. 1. Definition of a DSMM language for e-commerce (a). Using the language with increasing degrees of extension: no extension (b,c), property extensions (d,e), concept extensions (f,g).

For instance, assume we need to model information systems for e-commerce in various domains. For this purpose, we can build a specialized meta-modelling language that facilitates the construction of DSMLs for each one of these domains. An over-simplified definition of such a meta-modelling language and some of its uses for different scenarios are shown in Fig. 1. Model (a) defines the DSMM language, which is made of a single class `Product`. This language can be used to define a DSML for a stationer system (model (b)), for which we just use the primitives of the domain (e.g., we create an instance of `Product` called `Pen`). Finally, this DSML can be used to define the items in a particular stationery store (i.e., we can create instances of `Pen`, as done in model (c)).

In this way, the definition of a DSML spans three meta-levels: the `Stationer` model is an instance of `Ecommerce`, and `WHSmith` is an instance of `Stationer`. Therefore, it is natural to use a multi-level framework to support the definition and usage of our DSMM language, as these frameworks natively support instantiation across several meta-levels without recurring to artificial workarounds. In a multi-level framework, elements retain both a *type facet* which allows their instantiation in the next meta-level, and an *instance facet* as they are instances of an element at the meta-level above. Thus, model elements become *clabjects* (from the union of "class" and "object") enabling a more uniform way of modelling [3].

DSMM languages normally comprise three meta-levels. To enforce this depth in a multi-level framework, we can use *deep characterization* through the concept of *potency* [3]. The potency is an integer number that can be attached to models, clabjects, attributes and references. If an element is not explicitly given a potency, it receives the one of its immediate container. The potency of an element gets decremented at each meta-level, and when it reaches zero, the element cannot be instantiated further. Thus, the definition of our DSMM language has potency 2 (see model (a) in Fig. 1, its potency is indicated by '@2'), it gets instantiated into models with potency 1 (middle models), and the instances of these have potency 0 and therefore cannot be instantiated in subsequent meta-levels. In this way, the DSMM language user is effectively performing DSMM because he builds models with potency 1, which are instantiated as models of potency 0.

The potency is also a way for the deep characterization of properties, in order to control the meta-level in which they can be assigned a value. For example, in our DSMM language, all products will receive a price. Hence, `Product` declares an attribute `price` with potency 2, so that it will receive a value two meta-levels below (i.e., each pen has its own price). The potency of the attribute is not explicit, but it is received from the enclosing model. In contrast, the VAT is the same for all products of the same type, hence it has potency 1.

DSMM languages are used to build meta-models for related but different domains. Hence, a particular domain may need to extend the meta-modelling concepts offered by the DSMM language with new domain-specific properties. For example, model (d) in Fig. 1 shows that, in the hardware domain, we need to increase the attributes offered by `Product`. In particular, `Nail`s need to define their `caliber` and `length`. These two attributes are specific for nails and therefore cannot be included in the definition of `Product` as they are not general for every domain. Similarly, we may also need to declare domain-specific constraints, e.g., stating that the caliber should be larger than 0.1. Finally, some domains may need to make available new primitives to the users of the DSMLs. For instance, in the bookshop domain, the manipulated products are `Book`s, which have exactly one `Author` (see model (f) in Fig. 1). The concept of `Author` is not included in the DSMM language, and hence we need to include it in the meta-model for bookshops. This is only possible if the DSMM language provides facilities to define new clabjects, references and multiplicities. Moreover, one may wish to group several products in an inheritance hierarchy. For example, both `Magazine`s and `Book`s have a `title` and share the same `VAT` value.

The previous *linguistic extensions* can be supported by a multi-level framework if we use a dual ontological/linguistic typing for the model elements. The ontological typing is a relation within the domain, and refers to the type of which an element is instance. For example, the ontological type of `Pen` is `Product`, and the ontological type of `parker` is `Pen`. Hence, *ontological meta-modelling* is concerned with describing the concepts in a certain domain and their properties [4]. All elements in the top-most model (model (a) in Fig. 1) and some elements in the domain-specific meta-models (e.g., `Author`) may not have ontological type. In contrast, all elements have a linguistic type, which refers to the meta-modelling primitive used to create the element. For example, the linguistic type of `Product`, `Pen` and `parker` is clabject, while the linguistic type of `books` is reference.

One can interpret the union of the three models in each column of Fig. 1 as being conformant to a linguistic meta-model, as shown in Fig. 2(a) (the linguistic meta-model is only partially shown). In our approach, a *linguistic extension* is

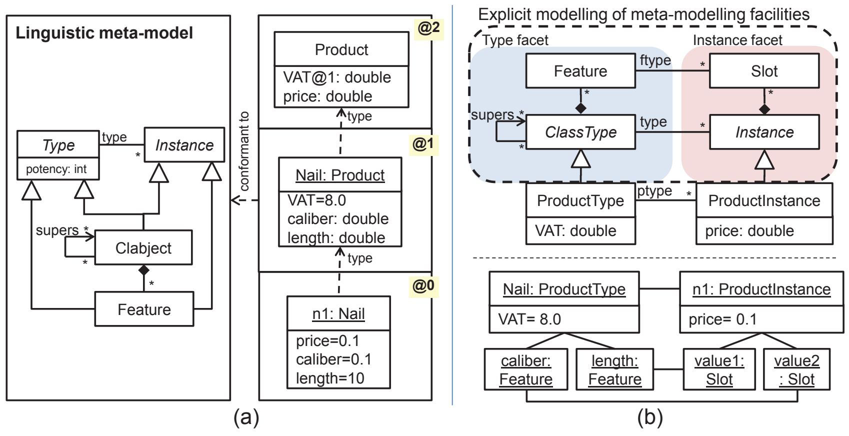

Fig. 2. Defining a DSMM language using: (a) 3 levels and dual typing, (b) 2 levels.

an element without ontological typing, like `Author` or the `caliber` attribute in `Nail`. The dual ontological/linguistic typing is very convenient for DSMM as it makes available standard meta-modelling facilities at each meta-level.

Alternatively, Fig. 2(b) shows the definition of our DSMM language using only two meta-levels. This solution makes necessary to explicitly model the desired meta-modelling facilities, and to manually encode the machinery to emulate built-in support for instantiation and constraint checking. Thus, one should build mechanisms taking care of type conformance, data types, definition and evaluation of constraints, and so on.

Altogether, deep meta-modelling facilitates the construction of DSMM languages. However, the following challenges remain:

– We need mechanisms to control the linguistic extensions offered by the DSMM languages, as not any extension may be valid in any domain.
– To be usable in practice, we need to provide a suitable concrete syntax for the DSMM languages and for the DSMLs defined with them. Ideally, both syntaxes should be defined once together with the DSMM language definition, and it should be possible to refine or extend them to take into account the particularities of specific domains.
– To enable the integration of DSMM in MDE projects, we need appropriate model management languages able to work in this multi-level setting.

These three challenges are tackled in the next three sections.

4 Customising the meta-modelling facilities

Designers need to control the way in which the designed DSMM languages will be used and extended. For this purpose, we propose the use of tags to identify the non-extendable language elements, and the use of constraints to ensure a certain extensibility degree. We will illustrate both control mechanisms using the textual syntax of the METADEPTH [6] tool. For example, the listing shown in Fig. 3 defines our DSMM language for e-commerce systems (lines 1–7), its usage to define a stationer's model (lines 8–10), and an instance of this (lines 12–14), corresponding to models (a, b, c) in Fig. 1.

```
1  strict Model Ecommerce@2 {
2    strict Node Product {
3      VAT@1 : double;
4      price : double;
5      minPrice: $self.price > 0$
6    }
7  }
8  Ecommerce Stationer {
9    Product Pen { VAT = 8; }
10 }
11
12 Stationer WHSmith {
13   Pen parker { price = 5.0; }
14 }
```

Fig. 3. Definition and use of the DSMM language for e-commerce in METADEPTH

The top-model `Ecommerce` lacks ontological type and hence is declared using the keyword `Model` (line 1). This model defines clabject `Product` using the keyword `Node` (line 2). Potencies are specified using the "@" symbol. Constraints can be defined using Java or the Epsilon Object Language (EOL), a variant of OCL that permits side effects [11]. For example, the constraint `minPrice` in line 5 demands a positive price for the products. It receives potency 2 from the model, therefore it will be evaluated two meta-levels below. The model instantiated in lines 8–10 has `Ecommerce` as ontological type, which is used instead of the keyword `Model`.

By default, the meta-models built with a DSMM language can be extended with new primitives (i.e., new clabjects), and any element in the meta-models can be extended with new features. To fine tune the extensibility of a DSMM language, our first proposal is a tagging control mechanism to identify the non-extensible elements. In this way, if the model with the DSMM language definition is tagged as `strict`, it will not be possible to add clabjects without an ontological typing in the next meta-level. If a clabject is tagged `strict`, their instances are forbidden to define new attributes, references or constraints. In the previous listing, both the `Ecommerce` model and the `Product` node are `strict`. Thus, we can use the DSMM language to build the stationer's model in Fig. 1, but not the hardware model (as `Product` instances cannot be extended) or the bookshop model (as `Author` has no ontological type).

If an element is not tagged `strict`, then we may need to control its allowed linguistic extensions. For example, we may like each `Product` instance at potency 1 to declare an attribute acting as identifier, which will receive a value at potency 0. Even though we could declare such a field at meta-level 2 with potency 2, here we may wish to let the decision of the attribute name and type (e.g., `String` or `int`) to the meta-level 1. For this purpose, we propose defining constraints that can make use of facilities of the linguistic meta-model. Fig. 4 shows a constraint, with potency 1, demanding the linguistic extension of all `Product` instances with some attribute tagged as identifier. The method `newFields` belongs to the API of METADEPTH's linguistic meta-model, and returns a collection with the new attributes declared in a meta-level. The method `isId` checks if a field is an identifier. As this constraint has potency 1, it will be evaluated at the next meta-level, where the DSMM language is used.

```
Node Product {
  ...
  extid@1: $self.newFields(). exists(f | f.isId())$
}
```

Fig. 4. Constraint demanding a linguistic extension

Finally, as the next section shows, we can also control the allowed linguistic extensions syntactically through the design of an appropriate concrete syntax.

5 Designing the concrete textual syntax

Even though deep meta-modelling enables DSMM, our goal is building DSMM languages, and therefore we need to design a concrete syntax for them (in addition to their abstract syntax). In the previous section, we used the default textual concrete syntax that METADEPTH makes available to model uniformly at every meta-level. However, this syntax usually leads to verbose model specifications, while we may prefer a more compact, domain-specific representation. For example, instead of creating instances of `Product` using `Product Pen{VAT=8;}`, we may like a more concise syntax for product instantiation like `Pen(8%)`.

Should the designer only had to define the syntax of the DSMM language, he may use existing tools for describing textual syntaxes like xText [15], TCS [10] or ANTLR [14]. However, as Fig. 5 illustrates, a multi-level architecture poses some challenges that these tools are not able to deal with, since the designer has to provide a syntax for the languages built with the DSMM language as well. In this way, when defining a DSMM language, the designer has to provide both the syntax of the models at meta-level 1 (i.e., of the domain-specific meta-models) and the syntax of the models at level 0 (i.e., of the meta-model instances). For this purpose, we assign to each concrete syntax definition a potency governing the meta-level at which it is to be used. Thus, the syntax with potency 1 will be used in the next

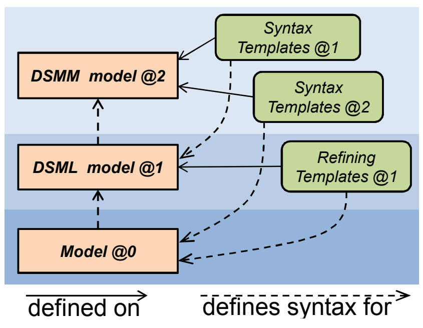

Fig. 5. Defining the concrete syntax.

meta-level, and the one with potency 2 will be used two meta-levels below. Moreover, it should be possible to refine the syntax initially defined for the models at meta-level 0, in order to introduce domain-specific constructs and describe the syntax of any linguistic extension.

Following this idea, we have created a template-based language to define textual concrete syntaxes in metaDepth. Using this language, the syntax of each clabject is defined through a template, which has a potency attached, controlling the meta-level at which the template will be applied. Another template declares the syntax of the model itself. As an example, Fig. 6 shows to the left the definition of the concrete syntax for our example DSMM language, whereas the right corresponds to the syntax for models at meta-level 0.

```
1  Syntax for Ecommerce [".ecommerce_mm"] {          7  Syntax for Ecommerce [".ecommerce"] {
2   template@1 TEcommerce for Model Ecommerce:       8   template@2 DeepProds for Model Ecommerce:
3      "id '{' &TProduct* '}'"                       9      "typename id '{' &DeepProd* '}'"
4   template@1 TProduct for Node Product:           10   template@2 DeepProd for Node Product:
5      "id '(' #VAT '%' ')'"                        11      "typename id '(' #price '€' ')'"
6  }                                                12  }
```

Fig. 6. Defining the concrete syntax for models at levels 1 (left) and 0 (right)

The first line in the definition to the left declares the language to which the syntax applies (the `Ecommerce` model) as well as the associated file extension (*ecommerce_mm*). Its two templates have potency 1, therefore they correspond to the syntax of the DSMM language (i.e., the templates will be used in the next meta-level). In particular, lines 2–3 define the syntax of the instances of the `Ecommerce` model, whereas lines 4–5 define the syntax of the clabject `Product`. The keyword "id" stands for the identifier of an element (see lines 3 and 5), whereas the attributes of a clabject can be referenced by using the prefix "#" (like `VAT` in line 5). Templates can refer to other templates, as in line 3, where it is indicated that the instances of `Ecommerce` can contain zero or more `Product` instances ("&TProduct*"). Using this textual syntax, we can specify the `Stationer` model as `Stationer{ Pen(8%) }`.

The right of the same listing declares the syntax for the models at meta-level 0, which will be stored in a separate file with extension *ecommerce*. In this case, all templates have potency 2. At this point, we do not know the model type for which the syntax is defined, but we only know that it will be an indirect instance of `Ecommerce` (line 8). To access the name of the concrete type we use the keyword "typename", which is interpreted by the parser to check that it is an indirect instance of `Ecommerce` (line 9). The same applies to the definition of templates for clabjects (lines 10–11). With this syntax we can write `Stationer WHSmith { Pen parker(5.0€) }` to instantiate model `Stationer`.

Templates can include "semantic" actions (e.g., to initialize fields of the created clabjects) and may have several syntactic expressions. For instance, we can insert in line 12 of the previous listing `''typename id'' with ''#price = 0''` to permit defining products without a price, which gets initialized to 0.

Finally, similar to [7], we can use a single template to define the syntax of several clabjects in the same inheritance hierarchy. Thus, if a clabject does not have an associated template, it uses the template of its closest ancestor in the inheritance hierarchy. A useful keyword in this case is "type", which gets substituted by the name of the clabject. For example, given a clabject "B" inheriting from "A", and a template attached to "A" with body `''type id''`, then writing "`A a`" creates an `A` instance, while typing "`B b`" creates a `B` instance. As a difference with "typename", "type" is used for direct types (i.e., at adjacent meta-levels) expecting exactly `A` or `B`. In contrast, "typename" is used for indirect types (i.e., at non adjacent meta-levels) and induces a checking that the name typed in place of "typename" is an indirect instance of the clabject the template is attached to.

5.1 Customising the meta-modelling facilities at the syntax level

In Section 4, we showed how to customise the extensibility of a DSMM language at the abstract syntax by identifying strict (i.e., non-extensible) elements and constraining the kind of allowed extensions. These design decisions should be reflected in the concrete syntax of the DSMM language as well, to discard forbidden extensions syntactically even before than semantically.

Our template language provides the following keywords to customise the allowed extensions for a DSMM language at the concrete syntax: `flingext` to allow declaring new fields with no ontological type, `lingext` to allow the addition of new clabjects with no ontological type, `constraint` for declaring constraints, and `super` to define new inheritance relations for clabjects. Moreover, two additional keywords allow defining how these extensions should be instantiated at level 0: `flinginst` for field instances and `linginst` for clabject instances.

For example, the listing shown to the left of Fig. 7 provides a concrete syntax enabling the definition of new fields and constraints in the instances of `Product`, due to the expression in line 6. Moreover, line 10 enables the instantiation of those extra fields in indirect instances of `Product`. Note that, this time, the concrete syntaxes of models at levels 0 and 1 are defined together, and have associated the same file extension. The listing to the right of the figure shows the definition of two models using this concrete syntax. The model `Hardware` in lines 1–7 (in the column to the right of Fig. 7) is an instance of `Ecommerce`, while the model in lines 9–11 is an instance of `Hardware`.

The listing to the left of Fig. 8 illustrates the use of `lingext` to allow the definition of new clabjects at meta-level 1 (line 4), and the use of `supers` to allow inheritance between instances of `Product` (line 7). The right of the same figure uses this syntax to define model (f) in Fig. 1. In the listing, `Magazine` (line 18) and `Book` (line 19) inherit from `Publication`, `Book` defines a new field `author` (line 20), and there is a new clabject `Author` with no ontological type (lines 22–24).

```
Syntax for Ecommerce [".ecommerce"] {
 template@1 TEcommerce for Model Ecommerce:
   "id '{' &TProduct∗ '}'"
 template@1 TProduct for Node Product:
   "id '(' #VAT '%' ')' '{'
          (flingext ';' | constraint)∗ '}' "
 template@2 DeepProds for Model Ecommerce:
   "typename id { &DeepProd∗ }"
 template@2 DeepProd for Node Product:
   "typename id '(' #price '€' flinginst∗ ')' "
}
```

```
Hardware {
   Nail (8.0%){
      caliber : double;
      length : double;
      bigger : $self.caliber>=0.1$
   }
}

Hardware DoItBest {
   Nail n1(0.1€ caliber=0.1 length=10)
}
```

Fig. 7. Extensible textual syntax (left), and its use (right)

```
Syntax for Ecommerce [".ecommerce"] {

 template@1 TEcommerce for Model Ecommerce:
   "id '{' (&TProduct | lingext )∗ '}' "

 template@1 TProduct for Node Product:
   "id ('(' #VAT '%' ')')? ('extends' supers)?
       (';' | '{' (flingext ';' | constraint)∗ '}') "

 ...
}
```

```
Bookshop {
   Publication (16%){
      title : String;
   }
   Magazine extends Publication;
   Book extends Publication {
      author : Author;
   }
   Node Author {
      name : String;
   }
}
```

Fig. 8. Syntax template allowing inheritance and new clabjects (left), and its use (right)

5.2 Refining the syntax of domain-specific modelling languages

Even if the DSMM language defines a syntax for the instances of the created domain-specific meta-models, the builder of a particular domain-specific meta-model may wish to design a special concrete textual syntax for some of the instantiated clabjects or for the linguistic extensions (see Fig. 5).

For example, we can design a template especially for `Nails`, as Fig. 9 shows. This template would be defined by the builder of the stationer's domain-specific meta-model at meta-level 1, and attached to it. The template refines the default one defined for products, so that the instances of `Nail` can be defined using this more specialised syntax (in addition to the default one). Hence, we can write `n1(0.1€, 0.1, 0.1)`, in addition to `Nail n1(0.1€ caliber=0.1 length=0.1)`. In order to disable the instantiation of nails using the latter, more general syntax, we should add the modifier `overwrite` to template `TNail`.

```
template@1 TNail for Node Nail:
  "id '(' #price '€', #caliber, #length ')' "
```

Fig. 9. Refining the syntax template for nails at level 1

To conclude, as an implementation note, our template language for specifying concrete syntaxes has been implemented using a meta-model, whereas its concrete syntax has been specified with itself through bootstrapping. The parser generation relies on ANTLR [14]. Moreover, METADEPTH includes a registry of parsers and automatically selects the appropriate parser according to the file extension of the model to be loaded.

6 Model management for DSMM languages

To integrate DSMM languages in MDE, we need to provide suitable model management languages able to deal with multiple meta-levels. In METADEPTH we have adapted the Epsilon family of model management languages[1] to work in a multi-level setting. Hence, we can define model manipulation operations and constraints for DSMM languages using the Epsilon Object Language (EOL), code generators working at several meta-levels using the Epsilon Generation Language (EGL), and model-to-model transformations spanning several meta-levels using the Epsilon Transformation Language (ETL). As the working scheme and challenges are similar in all cases, we will illustrate our solution only in the context of model-to-model transformations.

As an example, assume we want to generate a graphical user interface that allows the customers of an e-commerce system to select the products (at level 0) they want to buy. For this purpose, we need to transform the products to a model representation of the graphical user interface, from which we can generate code for different platforms like Java Swing or HTML. Based on this example, in the following we illustrate the four typical transformation scenarios in a multi-level setting: *deep transformations*, *co-transformations*, *refining transformations* and *reflective and linguistic transformations.*

Deep transformations. Oftentimes, a transformation needs to be defined using the meta-model of the DSMM language, and applied to the instances of the DSMLs built with it (i.e., at the bottom level). This scenario is depicted to the right of Fig. 10. In this case, the transformation definition needs to use indirect types because the direct types at level 1 are unknown when the transformation is defined. For example, if we want to generate a graphical user interface for any model of products at level 0, we would like to define the transformation only once at meta-level 2 together with the DSMM language definition. The left of Fig. 10 shows the ETL deep transformation to achieve this goal, which will be executed on indirect instances of the `Ecommerce` model. Rule `Product2CheckButton` creates a `CheckButton` for each indirect instance of `Product` (lines 3–8). The rule is annotated with the top-level meta-model needed by the transformation (line 1), and the level at which the transformation is to be executed (line 2). The *post* block (lines 10–15), which is executed when the transformation finishes, creates the *GroupBox* for the checkbuttons and the container *Window*.

Co-transformations. In this kind of transformations, a model and its meta-model need to be transformed at the same time, as the right of Fig. 11 illustrates.

[1] see `http://www.eclipse.org/epsilon/`

```
1  @metamodel(name=Ecommerce,file=Ecommerce)
2  @model(potency=0)
3  rule Product2CheckButton
4    transform pr : Source!Product
5    to cb : Target!CheckButton {
6      cb.name := pr.name()+'_cbutton';
7      cb.text := pr.name()+'('+pr.price+')';
8    }
9
10 post {
11   var wd: Target!Window := new Target!Window();
12   var gb: Target!GroupBox := new Target!GroupBox();
13   wd.children := gb;
14   gb.children.addAll(Target!CheckButton.all());
15 }
```

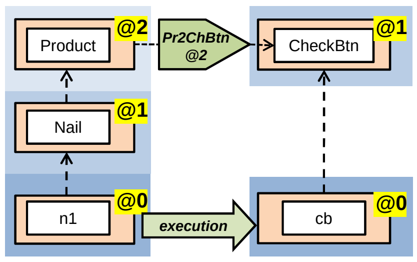

Fig. 10. Deep transformation example (left) and scheme (right)

Here, the same transformation has to deal with direct and indirect instances of the clabjects in the meta-model of the DSMM language; therefore, a mechanism is needed to select the level at which the rules will be applied.

As an example, we may wish to generate a menu for each product type defined at level 1, and checkbuttons for each product instance at level 0. For this purpose, we can use the transformation in Fig. 11. Line 1 imports the previous transformation which transforms the products at level 0. Then, rule `ProductType2Menu` is executed for each `Product` at level 1. The level at which the rule is executed is specified by the model alias, before the '!' symbol (see line 4). Hence, we use `Level0` for a model with potency 0 and `Level1` for a model with potency 1. We can also use the alias `Source` to refer to the source model regardless its potency. This is the alias used in the listing of Fig. 10, where the annotation in line 2 forces the execution of the transformation on models with potency 0. Hence, our framework implicitly makes available all (meta-)*-models of the context model for the transformation.

Refining transformations. Sometimes, a deep transformation needs to be refined for particular instances defined at level 1. This situation is depicted to the right of Fig. 12. For example, if we decide to transform the instances of `Nail` in a different way to consider the specific attributes that we added to it

```
1  import 'file:///Prod2GUI.etl';
2
3  rule ProductType2Menu
4    transform pr : Level1!Product
5    to mn : Target!Menu {
6      mn.name := pr.name()+'_menu';
7      mn.text := pr.name()+'('+pr.VAT+')';
8    }
```

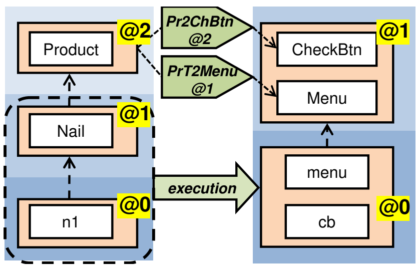

Fig. 11. Co-transformation example (left) and scheme (right)

```
import 'file:///Prod2GUIDeep.etl';

@metamodel(name=Ecommerce,file=Ecommerce.mdepth)
@model(potency=0)
rule Nail2CheckButton
  transform pr : Source!Nail
  to mn : Target!CheckButton
  extends Product2CheckButton {
    mn.name := pr.name()+'_check_nail';
    mn.text := pr.name()+'('+pr.price+',
      caliber='+pr.caliber+', length='+pr.length+')';
  }
```

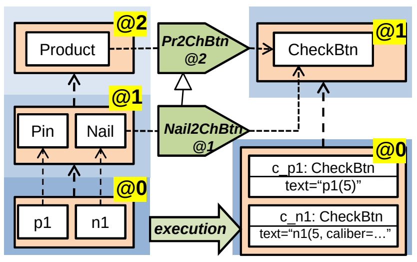

Fig. 12. Refining transformation example (left) and scheme (right)

(`caliber` and `length`), we need to refine the transformation rule defined for `Products` in Fig. 10. The refined rule is shown in Fig. 12. The rule extends `Product2CheckButton`, but it is refined for type `Nail`. To support this kind of transformations, we adapted ETL to allow extending a rule if the child rule transforms a direct or indirect instance of the clabject type transformed by the parent rule. The child rule will be applied whenever is possible, executing the body of the rules of both parent and child. In our example, rule `Nail2CheckButton` will be executed for instances of `Nail`, whereas rule `Product2CheckButton` will be executed for indirect instances of `Product` that are not instances of `Nail`.

Reflective and linguistic transformations. When defining a deep transformation, we may want to account for the linguistic extensions that can be performed at level 1. For this purpose, the transformation language needs reflective capabilities to access any new declared field, and it has to be possible to perform queries using linguistic types (i.e., `Node`, `Edge` and `Model`). The combination of these two capabilities enables the construction of generic transformations, applicable at any meta-level, and to elements of any ontological type. The working scheme of this kind of transformations is shown to the right of Fig. 13.

The listing in Fig. 13 shows a transformation with one reflective rule and another one defined on a linguistic type. Rule `Product2CheckButton` is reflective. It gets executed for each indirect instance of `Product` at level 0, creating a `CheckButton`. The rule takes into account that `Product` instances at level 1 may have been extended with new attributes. Thus, the rule iterates on the new attributes in line 6 (returned by `newFields`), concatenating their name and value. Technically, this reflection is possible because ETL is also reflective, being able to call transparently methods of the metaDepth API.

In its turn, rule `Node2Label` uses linguistic typing, being applicable to all `Node` instances (all elements) of potency 0 which are not indirect instances of `Product` (forbidden by the guard in line 15). In this way, if we apply this transformation to the `Amazon` model in Fig. 1, we obtain one `CheckButton` (the transformation of the `KL` book by rule `Product2CheckButton`) and one `Label` (the transformation of the `WS` author by `Node2Label`).

In the presented transformation examples, the target language has two meta-levels. We also allow DSMM languages as target, and currently we only support

```
rule Product2CheckButton
  transform pr : Level0!Product
  to cb : Target!CheckButton {
    cb.name := pr.name()+'_cbutton';
    cb.text := pr.name()+'( price='+pr.price+' ';
    for (f in pr.newFields())
        cb.text := cb.text+f.name()+'= '+
                    f.getValue()+' ';
    cb.text := cb.text+')';
  }

rule Node2Label
   transform pr : Level0!Node
   to cb : Target!Label {
    guard: not pr.isKindOf(Level0!Product)
    cb.name := pr.name()+'_label';
    cb.text := pr.name()+'(';
    for (f in pr.fields())
        cb.text := cb.text+f.name()+'= '+
                    f.getValue()+' ';
    cb.text := cb.text+')';
  }
```

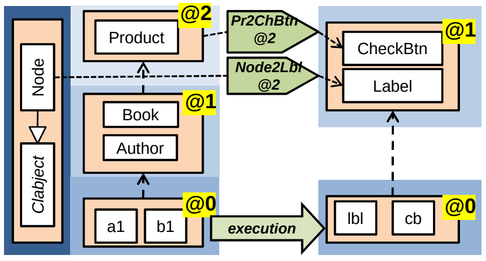

Fig. 13. Linguistic transformation example (left) and scheme (right)

rules specifying the creation of direct instances of clabjects. One may consider *abstract* rules specifying the creation of indirect instances, which would need to be refined at level 1 stating which clabject to instantiate. This is left for future work.

7 Discussion and future work

In this paper, we have presented our approach to define DSMM languages supporting the flexible definition of a textual concrete syntax, a fine control of the exposed meta-modelling facilities, and integration in MDE projects by making available multi-level model management languages.

We also discussed the typical transformation scenarios in a multi-level setting (deep transformations, co-transformations, refining transformations and linguistic/reflective transformations) and illustrated their support using ETL. These scenarios apply to other model management languages and tasks as well. In particular, they apply to the definition of textual syntaxes: at the top-level, we can define syntactic templates for level 0 models (similar to deep transformations), or for both level 0 and level 1 models (similar to co-transformations); we can add refining templates at level 1 (like in refining transformations); and we can define templates dealing with linguistic extensions (as in linguistic transformations). Each model management language needs to provide appropriate constructs to deal with each scenario, namely: the ability to select the meta-level at which a certain operation is to be applied (e.g., potencies for rules and templates), the ability to select clabjects of specific meta-levels (e.g., aliases `Level0` and `Level1` in rules), the possibility to obtain indirect instances of clabjects (transparently in our case), to access clabjects by their linguistic type (e.g., `Node`) and to reflectively access linguistic extensions (e.g., method `newFields`).

We are currently using metaDepth to define DSMM languages in different domains: component-based systems, web engineering and mobile devices. We are also exploring the definition of visual syntaxes for DSMM languages, and extending the integration of the tool with multi-level meta-modelling languages.

Acknowledgements. This work was funded by the Spanish Ministry of Economy and Competitivity (project "Go Lite" TIN2011-24139) and the R&D programme of the Madrid Region (project "e-Madrid" S2009/TIC-1650).

References

1. T. Aschauer, G. Dauenhauer, and W. Pree. Representation and traversal of large clabject models. In *MoDELS'09*, volume 5795 of *LNCS*, pages 17–31, 2009.
2. C. Atkinson, M. Gutheil, and B. Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Trans. Soft. Eng.*, 35(6):742–755, 2009.
3. C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.
4. C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.
5. C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Software and System Modeling*, 7(3):345–359, 2008.
6. J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS'10*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010.
7. J. Espinazo-Pagán, M. M. Tortosa, and J. G. Molina. Metamodel syntactic sheets: An approach for defining textual concrete syntaxes. In *ECMDA-FA'08*, volume 5095 of *LNCS*, pages 185–199. Springer, 2008.
8. M. Herrmannsdörfer and B. Hummel. Library concepts for model reuse. *Electron. Notes Theor. Comput. Sci.*, 253:121–134, September 2010.
9. S. Jablonski, B. Volz, and S. Dornstauder. A meta modeling framework for domain specific process management. In *COMPSAC'08*, pages 1011 –1016, 2008.
10. F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE*. ACM, 2006.
11. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
12. T. Kühne and D. Schreiber. Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. In *OOPSLA'07*, pages 229–244, 2007.
13. P. Langer, K. Wieland, M. Wimmer, and J. Cabot. From UML profiles to EMF profiles and beyond. In *TOOLS*, volume 6705 of *LNCS*, pages 52–67, 2011.
14. T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007. See also: `http://www.antlr.org/`.
15. xText. `http://xtext.org`.
16. S. Zschaler, D. S. Kolovos, N. Drivalos, R. F. Paige, and A. Rashid. Domain-specific metamodelling languages for software language engineering. In *SLE*, volume 5969 of *LNCS*, pages 334–353, 2009.