Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Model-Driven Engineering with Domain-Specific Meta-Modelling Languages

Juan de Lara*, Esther Guerra, Jesús Sánchez Cuadrado

Universidad Autónoma de Madrid (Spain),
e-mail: {Juan.deLara, Esther.Guerra, Jesus.Sanchez.Cuadrado}@uam.es

Received: date / Revised version: date

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 projects, several researchers have identified the need for *domain-specific meta-modelling* (DSMM) languages. These provide customised domain-specific meta-modelling primitives aimed at the definition of modelling languages for a specific domain, as well as the construction of meta-model families. Unfortunately, current approaches to DSMM rely on ad-hoc methods which add unnecessary complexity to the realization of DSMM in practice.

Hence, the goal of this paper is to simplify the definition and usage of DSMM languages. For this purpose, we apply *multi-level meta-modelling* for the systematic engineering of DSMM architectures. Our method integrates techniques to control the meta-modelling primitives offered to the users of the DSMM languages, provides a flexible approach to define textual concrete syntaxes for DSMM languages, and extends existing model management languages (for model-to-model transformation, in-place transformation and code generation) to work in a multi-level setting, thus enabling the practical use of DSMM in Model-Driven Engineering.

As a proof of concept, we report on a working implementation of these ideas in the MetaDepth tool.

Key words Model-Driven Engineering – Multi-Level Meta-Modelling – Domain-Specific Meta-Modelling – Textual Concrete Syntax – MetaDepth

1 Introduction

Model-Driven Engineering (MDE) promotes an active use of models throughout the software development process, leading to an automated generation of the final application. These models are sometimes defined using general-purpose modelling languages like the UML, but for restricted, well-known domains, it is also frequent the use of Domain-Specific Modelling Languages (DSMLs) tailored to the application domain and objectives of the project [28].

In current MDE practice, DSMLs are built by the language designer using a meta-model expressed with a general-purpose meta-modelling language, like the MOF [37]. This meta-model describes the instances that the users of the language can build at the immediate meta-level below. Thus, DSMLs usually comprise two meta-levels: the definition of the DSML and its usage.

More recently, several researchers [25,50] 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. For example, DSMM languages have been designed for expressing traceability [17], variability [50], to define and instantiate feature models [16], to define domain-specific process modelling notations [25] and DSML profiles [32].

A DSMM language spans three meta-levels: (1) the definition of the DSMM language for a specific domain, (2) the definition of the DSML by using the domain-specific constructs provided by the DSMM language, and (3) the usage of the DSML. However, existing approaches to DSMM are generally based on a two meta-level setting and different workarounds, like the definition of ad-hoc "promotion" transformations between models and meta-models [46], or the merging of models at two adjacent meta-levels into a single one (see Figure 1). These workarounds make the adoption of DSMM cumbersome in practice, adding unnecessary complexity to the resulting models [11] or to the supporting architecture. Moreover, currently, there is no general DSMM framework with integrated support for model management operations across the different meta-levels, e.g., to manipulate models in-place, to define model-to-model transformations or to generate code.

* *Present address:* Computer Science Department, Universidad Autónoma de Madrid, 28049 Madrid (Spain)

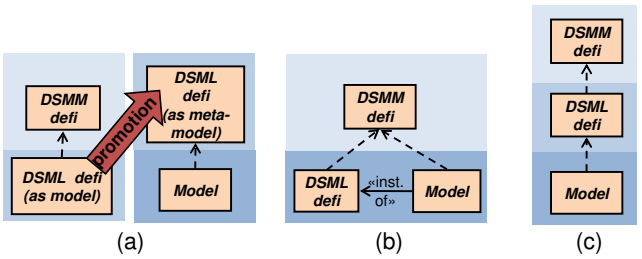

Fig. 1 Some alternatives to define DSMM architectures. (a) Promotion transformations. (b) Merging the DSML meta-model and its instances in a single meta-level, to emulate the "*instance of*" relation. (c) Use of multi-level meta-modelling.

In this paper we propose multi-level meta-modelling as the underlying concept behind DSMM, and discuss mechanisms to facilitate the construction of DSMM languages. Multi-level meta-modelling was originally proposed in the seminal work of Atkinson and Kühne [4,8,11]. It allows the definition of *deep* languages [4] that can be instantiated in more than one meta-level. In this way, at each meta-level, the constructed models are instances of the upper meta-level but also meta-models with respect to 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), as shown in Figure 1(c). Moreover, our framework provides: (i) means to customize the meta-modelling features that will be offered to the users of the DSMM languages, (ii) a flexible way to define textual concrete syntaxes at every meta-level, and (iii) model management languages (for code generation, in-place and model-to-model transformations) able to work in a multi-level setting, thus enabling the use of DSMM in MDE projects. The framework is supported by our multi-level meta-modelling tool MetaDepth [12], and is integrated with the Epsilon languages for model manipulation [18], modified to work in a multi-level setting.

This paper is an extended version of [14], where we have included more detailed discussions, a more realistic and challenging running example, a formalization of the main multi-level meta-modelling concepts, an improved language to define the concrete syntax, extensions to consider multi-level target domains in model-to-model transformations, as well as multi-level support for new model management languages besides model-to-model transformation (which was presented in [14]): for model manipulation with the Epsilon Object Language (EOL) [29] and for code generation with the Epsilon Generation Language (EGL) [40].

The rest of this paper is organized as follows. Section 2 overviews multi-level meta-modelling and its application to DSMM, identifying some challenges. Section 3 presents the MetaDepth tool, which will be used to illustrate the concepts introduced in this work. Then, the identified challenges are addressed in the next sections: Section 4 explains how to customise the meta-modelling facilities offered by the DSMM languages, Section 5 discusses how to define a concrete syntax for the DSMLs, and Section 6 shows how to manipulate models in a multi-level setting. Finally, Section 7 discusses related research and Section 8 concludes the paper. An appendix presents a formalization of the main concepts in multi-level meta-modelling and shows the complete listings of Section 5.

2 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 is restricted for a particular meta-modelling task or application. This section provides a motivating example for DSMM languages (subsection 2.1), proposes a solution based on deep meta-modelling (subsection 2.2), and compares this solution with the existing alternative approaches summarized in Figure 1 (subsection 2.3). The section concludes by identifying the challenges that need to be addressed to realize our solution.

2.1 Domain-specific process modelling

Assume we need to build process models for particular domains [24,25]. Our aim is therefore to define a meta-modelling language facilitating the construction of process modelling languages for specialized application areas, like software engineering, logistics or education. A simplified definition of such a meta-modelling language is model (a) in Figure 2. This DSMM language defines a class `Task`, which can be carried out by actors (class `Performer`) and can produce and consume `Artefacts`. Tasks can be connected through different kinds of `Gateways`, to account for different relation semantics: sequential (`Seq`) and parallel (`Fork` and `Join`).

We can use this language to define a DSML for educational processes, which will be used by educators to plan learning paths for different courses. A general-purpose process modelling language would have a too broad scope for this endeavour, whereas a specialized modelling language can provide educators with concepts close to educational processes, like `Lesson`, `Project`, `Evaluation`, `Self-Study`, `Tutorial class`, `Laboratory class`, and so on. Model (b) in Figure 2 shows the definition of a simplified version of such a DSML. The process model defines the set of possible educational tasks (lessons, projects and evaluation) and the allowed plannings (lessons can be given in sequence using `NextClass`, or in parallel with projects and evaluations using `Split` and then `Sync` to synchronize). Moreover, in every educational process, the final

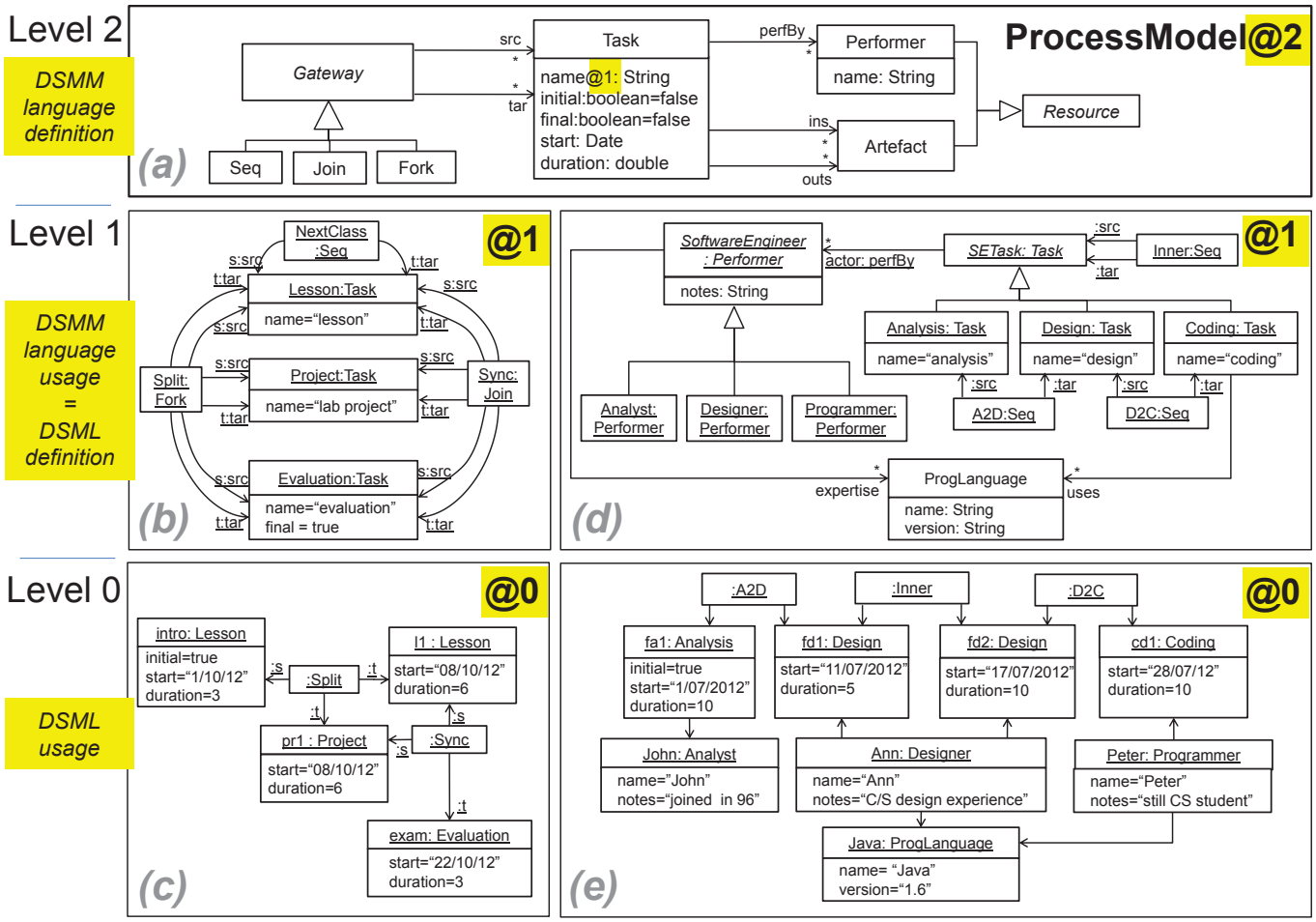

Fig. 2 Definition of a DSMM language for domain-specific process modelling (a). Definition of an educational process modelling language (b). A software engineering process modelling language (d). Model instances (c) and (e).

task is always some kind of evaluation. For simplicity, the model does not include performers (students, professors) or artefacts (course material, grades). To build this DSML, we have only used primitives of the domain (i.e., we have created instances of classes defined in the meta-level above, like `Task` and `Seq`). These domain-specific primitives make process modelling more natural than using a general-purpose meta-modelling language like the MOF [20,25]. The model in Figure 2 (b) actually defines a DSML for the educational domain, therefore we can use it to plan the activities in a course (model in Figure 2 (c)).

The defined DSMM language can be used to build DSMLs for other domains, like *software* process modelling, as shown in model (d) in Figure 2. This DSML includes a vocabulary for the different tasks relevant for software engineering, the different performer roles, and the resulting products (the latter have been omitted for space constraints). In this case, the definition of the DSML uses more advanced meta-modelling primitives, like inheritance and the definition of abstract entities (e.g., `SoftwareEngineer`). Moreover, it defines new attribute types (e.g., `notes`) and concepts (e.g., `ProgLanguage`) that are not available in the definition of the DSMM language because they are specific to this particular application (i.e., software engineering) within the process modelling domain. This customized DSML can then be used to build project development plans like the one in the model of Figure 2 (e).

DSMM languages are expected to be defined by model engineers with knowledge of the domain, who know the fundamental features needed by different applications within the domain (e.g., "*task*" and "*gateway*" are basic concepts in most kinds of process models). In this process, he may need to analyse existing DSMLs to identify commonalities, which would be added to the DSMM language. Then, DSML designers use DSMM languages to build DSMLs in the particular areas they are experts in (e.g., software process models). The same person can play both roles of DSMM language designer and DSML designer. Finally, the users of the DSMLs are the end users in the specific application domain (e.g., project managers or educators).

2.2 Multi-level modelling

The previous example shows that the definition of a DSML spans three meta-levels: the model in Figure 2 (b) is an instance of the model in Figure 2 (a), and the model in Figure 2 (c) is an instance of the one in Figure 2 (b). The model in Figure 2 (b) defines the types of tasks to be used in educational applications (`Lesson`, `Project`, `Evaluation`), which are themselves instances of meta-class `Task` defined in the model of Figure 2 (a). Therefore, it is natural to use a multi-level approach to support the definition and usage of our DSMM language, as this approach natively supports instantiation across several meta-levels without recurring to artificial workarounds, like ad-hoc transformations or manual encodings of several meta-levels into a single one, as shown in Figures 1(a) and (b). In a multi-level framework, elements retain both a *type facet* that 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 the terms "class" and "object") enabling a more uniform way of modelling [8].

DSMM languages normally comprise three meta-levels. To enforce this architecture in a multi-level framework, we can use *deep characterization* through the concept of *potency* [8,9]. The potency is a natural number (including zero) that can be attached to models[1], clabjects, fields[2] and associations. If an element is not explicitly given a potency, it receives the one of its immediate container. The potency of the instances of an element is equal to the potency of the element minus one. Finally, when the potency of an element reaches zero, it cannot be instantiated in lower meta-levels, becoming a pure instance. Thus, the definition of our DSMM language has potency 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. We use the notation '@X' to mean "potency X", and it can be attached to models, clabjects, associations and fields. In this way, the DSMM language user is effectively performing domain-specific meta-modelling because he builds models with potency 1 (the DSML), which are instantiated as models of potency 0.

In two-level meta-modelling frameworks, a class can only define the properties of its immediate instances, one meta-level below. On the other hand, in multi-level frameworks, the potency can be used to define the properties of (indirect) clabject instances, several meta-levels below. In this way, potency for fields works in a similar way as for clabjects. For example, in our DSMM language, all task instances at meta-level 1 have a name. Hence, `Task` declares a field `name` with potency 1 (indicated by '@1') so that it will receive a value one meta-level below. On the other hand, all indirect instances of `Task` two meta-levels below (i.e., the instances of instances of `Task`) have a `start` date, a `duration`, and can be declared to be `initial` and `final`. Thus, these fields are declared in the model of Figure 2 (a) with potency 2, received from the enclosing model (and therefore omitted in the figure). Similar to clabjects, fields with potency 2 are automatically instantiated at potency 1 when the owner clabject is instantiated (e.g., `final` gets instantiated in clabject `Evaluation` in the model of Figure 2 (b)); then, the field is instantiated once more at potency 0 (e.g., in clabject `exam`). Fields with potency equal to zero are pure instances and must be assigned a value, like field `start` in all clabjects of model (c) in Figure 2. If no value is assigned to a field with potency zero, then a default value declared in the upper meta-levels is sought, like in the case of field `final` in the clabjects of model (c). Fields with potency bigger than zero may receive a value or not. In the latter case, the field is not shown in the model. In the first case, the value is the default for its instances. This means that, in the example, all instances of `Evaluation` are final unless they explicitly override the default *true* value.

In addition, it may be necessary to extend the meta-modelling primitives initially offered by the DSMM language with new properties and concepts specific to the particular application within the domain (e.g., specific primitives to software processes in the process modelling domain). Moreover, it is sometimes desirable to offer users more sophisticated meta-modelling facilities to obtain simpler models. For example, model (d) in Figure 2 makes use of inheritance to define common properties for software engineers (in clabject `SoftwareEngineer`) and software engineering tasks (in clabject `SETask`). Both clabjects are declared abstract, hence they cannot be instantiated. Software engineers are enriched with an additional field `notes`, and a new concept `ProgLanguage` to represent programming languages has been included. These new fields and concepts could not be foreseen by the DSMM language designer, as the DSMM language is meant to be applicable to related but nonetheless different applications within the domain. However, these extensions are only possible if the DSMM language provides facilities to define new clabjects, associations and multiplicities.

Multi-level frameworks can naturally support these extensions by using a dual ontological/linguistic typing for the model elements. This *Orthogonal Classification Architecture* (OCA) was originally proposed in [10]. 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 `Analysis` is `Task` (see model (d) in Figure 2), and the ontological type of `John` is `Analyst` (see model (e) in Figure 2). Thus, ontological meta-modelling is concerned with describ-

[1] Potency for models is termed "level" in [8].

[2] In a multi-level setting, we use the term "field" instead of "attribute" or "slot" as fields have both a type and an instance facet.

ing the concepts in a certain domain and their properties [10]. In its turn, the linguistic type of an element refers to the meta-modelling primitive used to create the element. For example, the linguistic type of `Task`, `Analysis` and `Analyst` is *Clabject*, while the linguistic type of `version` is *Field*. All elements in the topmost model (model (a) in Figure 2) and some elements in the domain-specific meta-models (e.g., `ProgLanguage`) may not have ontological type. In contrast, all elements have a linguistic type.

One can interpret the union of the three models in each column of Figure 2 as being conformant to a linguistic meta-model, as Figure 3 shows[3]. In our approach, a *linguistic extension*, or simply an *extension*[4] is an element with a linguistic type but without any ontological type, like clabject `ProgLanguage` in model (d) of Figure 2, or the `subject` field in clabject `Evaluation` in Figure 3. Ontological instance models of a model M are allowed to have linguistic extensions (i.e., elements with no ontological typing), and are still considered valid instances of M. This is the case of the model with potency 1 in Figure 3, which is an ontological instance of the one with potency 2. A formal definition of these concepts is presented in Appendix A.

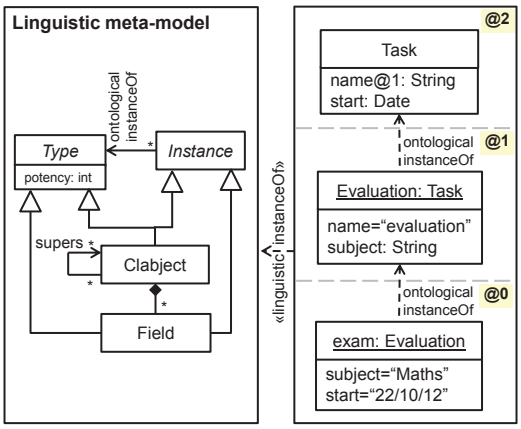

Fig. 3 DSMM language definition and usage using 3 levels and dual typing.

The dual ontological/linguistic typing is very convenient for DSMM as it makes available standard meta-modelling facilities at each meta-level. Otherwise, these facilities would need to be replicated at every meta-level bigger than zero.

[3] The linguistic meta-model shown in Figure 3 is a simplification of a prototypical one. We will provide more details for the linguistic meta-model of our MetaDepth tool in Section 4. The models in this figure are also simplified/adapted with respect to Figure 2.

[4] *Linguistic extension* comes from the fact that we use the linguistic meta-model to create an element, which therefore has only linguistic type, but no ontological type.

2.3 Comparison with other approaches and challenges

As illustrated in Figures 1(a) and 1(b), so far there have been two main alternative proposals for implementing DSMM, both relying in just two meta-levels: (i) emulating the instance-of relation using a regular association, and (ii) promotion transformations. Next, we introduce these approaches and highlight the differences with our proposal.

2.3.1 Emulation of the instance-of relation. Figure 4 shows how to define part of our example DSMM language using two meta-levels and encoding the instance-of relation as a regular association.

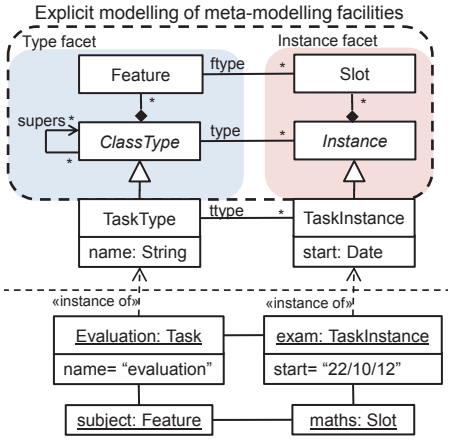

Fig. 4 Defining a DSMM language using 2 levels.

This solution requires each class in the language meta-model to be split into two: one for the type facet (e.g., `TaskType`) and one for the instance facet (e.g., `TaskInstance`), related through a regular association representing the instance-of relation (e.g., `ttype`). The type and instance facets of classes need to be made explicit through additional classes in the meta-model (`ClassType` and `Instance`). More importantly, this solution requires to explicitly model the desired meta-modelling facilities within the language meta-model (e.g., classes `Feature` and `Slot`), and manually encoding the machinery to emulate built-in support for instantiation (i.e., type conformance, data types and suitable data values, inheritance and its semantics, etc.) and for constraint checking within model-based tools. This is challenging as regular model-based tools (e.g., model transformation languages) are not prepared to interpret this instantiation relation built ad-hoc within the same meta-level. As a simple example, to obtain all instances of `Evaluation` in Figure 4, we need to write the OCL expression `TaskInstance.allInstances->select(i|i.ttype.name = 'evaluation')`. Thus, we are indeed loosely type-checking that a `Task` is actually an `Evaluation`. In the presence of advanced features,

such as inheritance, these expressions become more complex. Using our solution based on multi-level meta-modelling shown in Figure 3, we just need to write `Evaluation.allInstances`. Finally, the creation of objects with instance facet, like `TaskInstance`, require explicitly creating `Slots` to assign a value to the features defined by the task type, probably using an imperative language, and subsequently checking that slots contain correct values according to the feature's data type. In our approach, these facilities are automatically provided by the multi-level framework, they do not have to be built ad-hoc.

2.3.2 Promotion transformations. Figure 5 shows the encoding of part of our DSMM language for educational process modelling using two levels and promotion transformations.

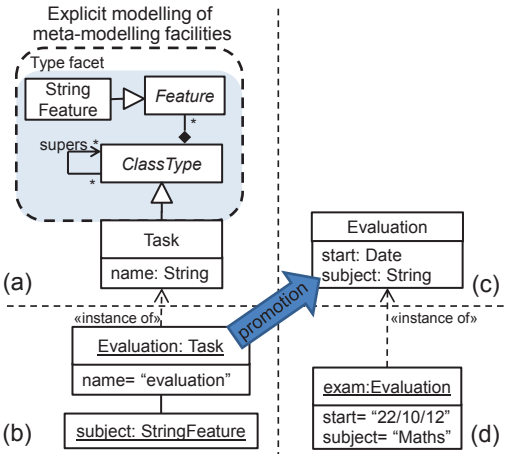

Fig. 5 Defining a DSMM language using 2 levels and a promotion transformation.

In this case, the language meta-model (tagged (a)) has to include classes to simulate the definition of elements with type facet at the next meta-level (model tagged (b)). That is, the meta-model has to include classes to model types (`ClassType`), features of different kinds (e.g., `StringFeature`), inheritance (`supers`), constraints, multiplicities and so on. Then, a DSML is defined as an instance of this meta-model (model tagged (b)). From this definition, we need to derive a meta-model (tagged (c)) that can be instantiated to create models, instances of the DSML. We call this process *promotion transformation*. In our example, it involves creating a meta-class for each `Task` instance and an attribute for each `Feature` instance. In general, it involves the transformation of the constraints, multiplicities and inheritance relationships included in the DSML definition model as well. Moreover, sometimes, the promotion transformation also needs to encode domain information; for instance, the fact that any kind of task should have a start date is encoded by the promotion transformation, which adds this attribute to `Evaluation`. This means that we may need to build different promotion transformations depending on the particular DSMM language. This is so as, while fields with potency 1 (like *name*) are explicitly present in the meta-model (a), the fields with potency 2 are not. Hence, the transformation needs to explicitly create fields with potency 2 (like *start*) in the corresponding elements in model (c), and this depends on the particular domain that model (a) is representing. Alternatively, one could use some mechanism, like annotations or aspects, to inject simple domain information in the transformation.

A promotion-based approach does not need to make explicit in meta-model (a) the instance facet of elements (as in Figure 4). However, there is a disconnection between models (b) and (c), as the type-facet information is missing in (b), while the instance-facet information is missing in (c). This may lead to consistency problems. For example, model (b) has only instance facet, and therefore it does not support linguistic extensions (like the definition of new meta-classes). Any linguistic extension should be made directly on meta-model (c). This could be a problem if model (b) is modified and therefore meta-model (c) needs to be regenerated, as the manual modifications made to (c) could be overwritten. On the other hand, meta-model (c) only has type facet, and hence cannot access to class attributes like `name` in model (b). Our proposal based on multi-level meta-modelling does not have these problems, as any model can be seen as an instance of the meta-level above, but can also be instantiated.

Moreover, the price to pay in a promotion-based approach is the need to create an ad-hoc promotion transformation for each DSML. Intuitively, a promotion transformation adds a type facet to the elements in model (b). This is automatic in multi-level meta-modelling frameworks, where there is no need to promote models into meta-models.

2.3.3 Multi-level meta-modelling: challenges. Altogether, an architecture supporting deep meta-modelling facilitates the construction of DSMM languages: the meta-model with potency 2 in Figure 3 is simpler than those in Figures 4 and 5. Moreover, the solutions based on two levels become increasingly involved in scenarios using more than three meta-levels. On the contrary, multi-level frameworks offer a generic solution which is independent of the number of meta-levels.

However, domain-specific meta-modelling using a multi-level approach also adds some challenges, like:

- Mechanisms are needed to control the linguistic extensions offered by the DSMM languages, as not any extension may be appropriate for every domain, or we may wish to restrict the extension capabilities offered to the users.
- To be usable in practice, a suitable concrete syntax for the DSMM languages (which will be used at meta-level 1) and for the DSMLs defined with them (which

will be used at meta-level 0) is needed. Ideally, both syntaxes should be specified 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 applications within domains.
– To enable the integration of DSMM in MDE projects, appropriate model management languages able to work in this multi-level setting are needed, notably for code generation, in-place or model-to-model transformations.

We will tackle these three challenges in Sections 4, 5 and 6. Before, we introduce our MetaDepth tool in the next section, as we will use it to illustrate our solutions.

3 Domain-Specific Meta-Modelling with MetaDepth

MetaDepth [12,14,15] is a multi-level meta-modelling tool that supports textual modelling and implements deep characterization through potency. Listing 1 shows the definition of our DSMM language for domain-specific process models in MetaDepth. The top-model `ProcessModel` lacks ontological type and hence is declared using the keyword `Model` (line 1). This model defines clabjects like `Task`, `Resource` or `Performer` using the keyword `Node`. Potencies are specified using the "@" symbol. If an element does not specify a potency, it takes the one from its enclosing container. There is no restriction on the number of meta-levels that MetaDepth can handle, even though in this paper we restrict to 3, as this is the most common scenario for DSMM languages.

Constraints can be defined using Java or the Epsilon Object Language (EOL), a variant of OCL that permits encoding side effects [29]. For example, the constraint `minDuration` in line 13 demands a positive duration for the tasks. It receives potency 2 from the model, therefore it will be evaluated two meta-levels below.

```
1  Model ProcessModel@2 {
2    Node Task {
3      name@1 : String[0..1];
4      initial : boolean = false;
5      final : boolean = false;
6      start : Date;
7      duration : double;
8
9      perfBy : Performer[*];
10     ins : Artefact[*];
11     outs : Artefact[*];
12
13     minDuration: $ self.duration>0 $
14   }
15
16   abstract Node Resource {}
17   Node Performer : Resource {
18     name : String;
19   }
20   Node Artefact : Resource {}
21
22   abstract Node Gateway {
23     src : Task[*];
24     tar : Task[*];
25   }
26   Node Seq : Gateway {}
27   Node Join : Gateway {}
28   Node Fork : Gateway {}
29 }
```

Listing 1 Defining the DSMM language for domain-specific process modelling in MetaDepth (model (a) in Figure 2).

Listing 2 shows the usage of the previous DSMM language to define the software process model shown in Figure 2 (model (d)), enriched with some constraints that will be evaluated at level 0. The instantiated model has `ProcessModel` as ontological type, which is used instead of the keyword `Model` in line 1. Line 2 declares an instance of `Task` named `SETask`, which is used as the base clabject from which all types of software engineering tasks inherit (`Analysis`, `Design` and `Coding`). Clabject `SETask` is declared abstract, therefore it cannot be instantiated at the next meta-level.

```
1  ProcessModel SoftwareProcess {
2    abstract Task SETask {
3      actor : SoftwareEngineer[*] {perfBy};
4    }
5
6    Task Analysis : SETask {
7      name = "requirements, analysis";
8      analyst : $ self.actor.forAll(a | a.isKindOf(Analyst)) $
9    }
10
11   Task Design : SETask {
12     name = "high-level design, low-level design";
13     designer : $ self.actor.forAll(a | a.isKindOf(Designer)) $
14   }
15
16   Task Coding : SETask {
17     name = "coding, unit testing";
18     uses : ProgLanguage[*];
19     programmer : $ self.actor.forAll(a | a.isKindOf(Programmer)) $
20   }
21
22   Node ProgLanguage {
23     name : String;
24     version : String;
25   }
26
27   abstract Performer SoftwareEngineer {
28     notes : String;
29     expertise : ProgLanguage[*];
30   }
31   Performer Analyst : SoftwareEngineer {}
32   Performer Designer : SoftwareEngineer {}
33   Performer Programmer : SoftwareEngineer {}
34
35   Seq A2D {
36     from: Analysis{src};
37     to : Design{tar};
38   }
39   Seq D2C {
40     from : Design{src};
41     to : Coding{tar};
42   }
43   Seq Inner {
44     from : SETask{src};
45     to : SETask{tar};
46     equal : $ self.from.type=self.to.type $
47   }
48
49   instOnce : $ Analysis.allInstances().size()>0
50                and Design.allInstances().size()>0
51                and Coding.allInstances().size()>0 $
52 }
```

Listing 2 Using the DSMM language defined in Listing 1 (model (d) in Figure 2, extended with some constraints).

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. For instance, clabject `ProgLanguage` in line 22 has no ontological type, clabjects `Coding` and `SoftwareEngineer` declare collections of `ProgLanguage` in lines 18 and 29, and clabject `SoftwareEngineer` in line 27 is an abstract instance of `Performer` which declares a new field `notes` in line 28. In addition, three `Seq` instances are created: `A2D` to move from an `Analysis` task into a `Design` task (line 35), `D2C` to move from a `Design` task into a `Coding` task (line 39), and `Inner` to move between tasks of the same type (line 43). This latter constraint, modelled in line 46, makes use of the field `type` from the MetaDepth API, which will be explained in the next section.

MetaDepth allows the definition of constraints local to clabjects or at the model level. For example, each `Task` instance defines a constraint regarding the role of the involved actors, so that `Analysis` tasks can only be performed by `Analysts` (line 8), `Design` tasks can only be performed by `Designers` (line 13), and so on. Constraint `instOnce` in lines 49–51 has a global context, and demands at least one instance of `Analysis`, `Design` and `Coding` in the instance models at the next meta-level. All these constraints have potency 1. For constraints, the potency indicates the number of meta-levels below at which a constraint will be evaluated. Hence constraints with potency 1 are evaluated at the next meta-level, while constraints with potency 2 are evaluated 2 meta-levels below.

References are instantiated similar to clabjects, as illustrated in Figure 6. The top meta-level, with potency 2, shows the definition of a reference `src` allowing the connection of `Seq` instances to zero or more `Task` instances. The next meta-level shows an instance of the reference `src`, named `from`, connecting `A2D` and `Analysis`. The `from` reference has potency 1, and hence has a type facet. This means that it can be added a multiplicity (`1..1` in this case) and be instantiated in the next meta-level. The figure shows to the right the concrete syntax for this definition, where `from` is declared as a reference type (`from: Analysis`). In this example, we can omit the multiplicity definition `[1..1]` (as we have done in Listing 2, line 36) because it is the default in MetaDepth. Finally, the fact that `from` is an instance of `src` is indicated between curly braces.

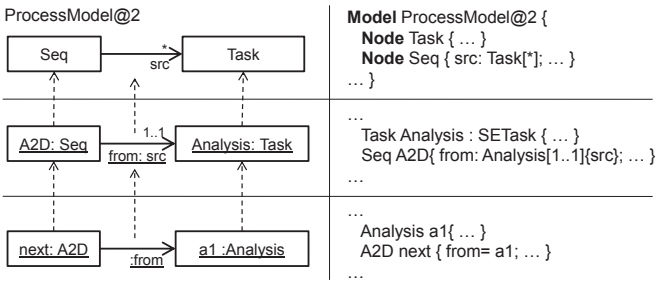

Fig. 6 Instantiation of references.

4 Customising the Meta-Modelling Facilities

Designers need to control the way in which the designed DSMM languages will be used and extended, as not any extension may be appropriate for a certain language. For this purpose, we propose two control mechanisms: the use of modifiers to identify the non-extendable language elements, and the use of constraints to ensure a certain extensibility degree.

More in detail, our first proposal to fine tune the extensibility of a DSMM language is the use of the `strict` modifier to tag 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`, its instances are forbidden to define new fields, references or constraints. For example, if we tag the `Performer` node `strict` in Listing 1, we cannot define the new field `notes` in line 28 of Listing 2. Similarly, if we mark the `ProcessModel` model `strict` in Listing 1, we cannot define the new clabject `ProgLanguage` in Listing 2, because it has no ontological typing.

If an element is not tagged `strict`, then we may need to control its allowed linguistic extensions. For example, we may ask each `Resource` instance at potency 1 to declare some field acting as identifier (in MetaDepth this is indicated with the {`id`} modifier), which will receive a value at potency 0. Even though we could declare a field in `Resource` at meta-level 2 with potency 2, here we wish to let the decision of the field name and type (e.g., `String` or `int`) to the DSML designer at meta-level 1. This is useful, as this identifier could be defined to be the social security number for `Performer` instances, and an internal code for `Artefact` instances. For this purpose, we propose defining constraints that can make use of facilities of the linguistic meta-model, like seamless navigation to upper meta-levels and access to linguistic extensions of model elements.

Figure 7 shows a simplified version of MetaDepth's linguistic meta-model. It contains the main fields and methods that can be accessed uniformly from within constraints and model management operations. The metamodel only shows the elements used in this paper, further classes and methods have been omitted.

The base class in the hierarchy is `Element`, from which any element that may have both type and instance facets inherits. It has name, potency, cardinality and strictness. The `name` property acts as unique identifier within its container, and if no value is explicitly given, then it is automatically generated. The `QualifiedElement` class contains fields and declares the derived properties `references`, which holds the fields with non-primitive type, and `newFields`, to hold the linguistic extensions (i.e., fields without ontological type). Its method `references` returns all

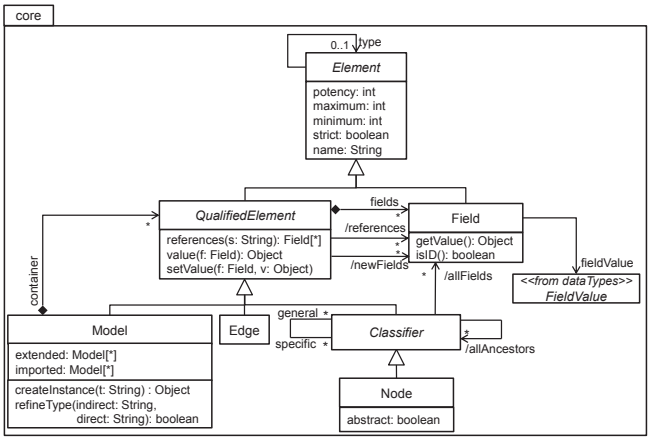

Fig. 7 MetaDepth's linguistic meta-model (simplified `core` package).

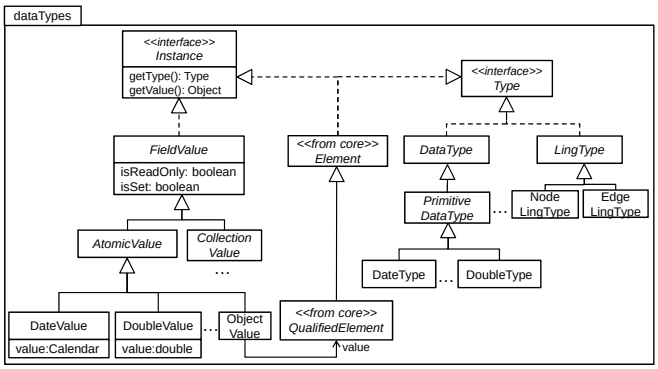

Fig. 8 MetaDepth's data types (excerpt of `dataTypes` package).

instances of the given reference type, while `value` returns the value of a given field. A `Classifier` can participate in inheritance relations as defined by the `general/specific` reference. The leaf modelling elements are `Models`, `Nodes`, `Edges` (similar to bidirectional associative classes) and `Fields`. Method `createInstance` in class `Model` is used to instantiate a given type, while `refineType` checks if a given type is (direct or indirect) instance of another one, and is useful in reflective operations for the creation of indirect instances (more details will be given in Section 6.1). The concept of *"clabject"*, in the sense of Atkinson and Kühne [8], corresponds to class `Node` in this linguistic meta-model. `Nodes` can be abstract. All subclasses of `Element` inherit a cardinality (attributes `maximum` and `minimum`) to restrict the minimum and maximum number of instances within an enclosing container. For multi-valued fields, it restricts the multiplicity of their values; for nodes, it restricts the number of instances in a model; and for models, the number of instance models within the system.

Method `getValue` in class `Field` returns the field value and `isID` checks if the field is an identifier. Field values are stored in subclasses of `FieldValue`, as shown in Figure 8. Values can be atomic or collections, and we distinguish primitive data types, enumeration types (not shown) and linguistic types. Object references are kept by class `ObjectValue`, which owns a reference to a `QualifiedElement`. The latter are also considered types by their conformance to the `Type` interface.

When invoking an operation over a clabject within Epsilon, the engine first checks if it is a built-in OCL operation, like *allInstances*. Otherwise, we use a transparent reflection mechanism to access the method in the underlying MetaDepth framework. For usability reasons, a method without parameters can be invoked similar to accessing a field; this is how we have implemented all derived fields in the meta-model of Figure 7. If the name of a field in the model collides with a name in the linguistic meta-model, we can prefix this name by "^" to refer to the linguistic one. For instance, if we have the declaration `Task Analysis { name = ``requirements and analysis''; }`, writing `Analysis.name` returns "requirements and analysis", while `Analysis.^name` returns the object identifier "Analysis".

As an example of accessing the linguistic meta-model API within a constraint, the next snippet shows a constraint `existsId` which demands all `Resource` instances to be extended with some field tagged as identifier, so that its value is unique at the next meta-level. The constraint has potency 1 and hence it will be evaluated in direct instances of `Resource`. In this way, when evaluated in a specific instance, the method `newFields` will obtain a collection with the new fields declared in the instance, and the method `isID` will check if the field is an identifier.

```
1 abstract Node Resource {
2   ...
3   existsId@1: $ self.newFields().exists(f | f.isID()) $
4 }
```

To satisfy this constraint, Listing 2 needs to be modified by adding an identifier field to all instances of `Resource` as follows:

```
1 abstract Performer SoftwareEngineer {
2   sSN : String{id};
3   ...
4 }
```

This kind of constraints is also very useful to control the instantiation of references. For example, in the process model DSMM language, instances of `Seq` should contain exactly one instance of `src` and one instance of `tar`, both with a maximum cardinality of one, like in `A2D`, `D2C` and `Inner` in Listing 2. In their turn, `Fork` instances should have at most one instance of `src` with a maximum cardinality of one, while in `Join` instances the same restriction applies to the `tar` instances.

The next snippet shows how to declare these constraints. In MetaDepth, constraints can be declared in the scope of models and then be assigned to several elements, promoting reuse of the constraint code.

Thus, we declare two constraints `singleSource` and `singleTarget` with potency 1, which we assign to `Seq` and `Fork` on the one hand, and to `Seq` and `Join` on the other. The method `references` belongs to the linguistic meta-model and returns the collection of instances of the given reference defined by a clabject, while `getMaximum` returns the maximum cardinality declared by a field.

```
1 Model ProcessModel@2{
2   ...
3   singleSource@1(Seq, Fork): $ self.references("src").size()=1 and
4                 self.references("src").forAll( r | r.getMaximum() = 1) $
5   singleTarget@1(Seq, Join): $ self.references("tar").size()=1 and
6                 self.references("tar").forAll( r | r.getMaximum() = 1) $
7 }
```

As a main difference with standard OCL constraints on a two meta-level setting [38], our constraints can be assigned a potency bigger than one, and hence be evaluated several meta-levels below. This is possible by using what we call *indirect types*. For example, the direct type of `fa1` in model (e) in Figure 2 is `Analysis`, whereas its indirect type at the top meta-level is `Task`. If we write a constraint at meta-level 2 including the expression `Task.allInstances()`, evaluating the expression at meta-level 1 yields the direct instances of `Task`, whereas its evaluation at meta-level 0 returns the instances of every direct instance of `Task` (which we call *indirect instances*).

Another difference of our constraints with respect to those of standard OCL is that we can use constraints to assess a certain structure in the type facet of clabjects. Thus, our constraints can use methods and attributes of the linguistic meta-model, like `type`. In contrast, standard OCL constrains normally work with the instance facet of elements only.

As the next section will show, we can also control the allowed linguistic extensions syntactically through the design of an appropriate concrete syntax, resulting in a more fine-grained control.

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 may lead to verbose model specifications, while we may prefer a more compact, domain-specific representation. For example, instead of the heavy instantiation syntax of `Seq` clabjects, where the `from` and `to` references have to be instantiated as well (see lines 35–47 of Listing 2), we may offer a lighter syntax like: `seq a2d: Analysis -> Design`.

If the designer only had to define the concrete syntax of the DSMM language and use it at the immediate meta-level below, he might use existing standard tools for describing textual syntaxes like Xtext[5], TCS [27] or ANTLR[6]. However, as Figure 9 illustrates, a multi-level architecture poses some challenges that these tools are not able to deal with, since there is the need to define a syntax for the DSMM language as well as for the languages built with it. 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., their instances). Moreover, it should be possible to refine the syntax initially defined for the models at meta-level 0, to describe the syntax of any defined linguistic extension. This can be necessary when new constructs for a specific application area are introduced, like `ProgLanguage` in software process modelling.

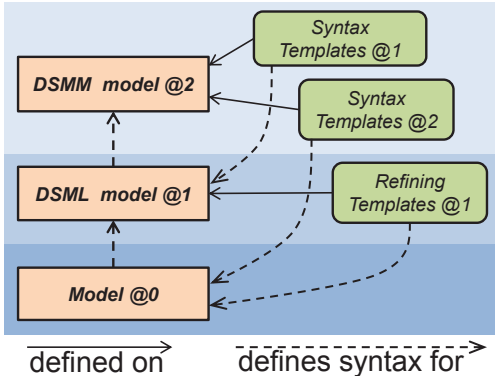

Fig. 9 Defining the concrete syntax in a multi-level setting.

Following this idea, in MetaDepth we have created a template-based language to define textual concrete syntaxes for multiple meta-levels. Using this language, the syntax of each clabject is defined through a template which has a potency controlling the meta-level at which the template will be applied. Thus, the syntax template with potency 1 will be used in the next meta-level, and the one with potency 2 will be used two meta-levels below.

Figure 10 shows an excerpt of the meta-model that describes the abstract syntax of our template language. A syntax definition is composed of templates (`TemplateRule`) that refer to a model or a clabject (`ModelTemplate` and `NodeTemplate` respectively, which have a reference to a `Model` or a `Node`, omitted in the figure for simplicity). There must be a template associated to the model, which acts as entry point of the syntactic definition. A template consists of one or more sequences of template elements. If there are multiple sequences, they are considered alternatives. A `TemplateElement` specifies how to parse and serialize a model element. There are five types. A `Group` aggregates a sequence of elements that are repeated one or

[5] `http://www.eclipse.org/Xtext/`
[6] `http://www.antlr.org/`

more times, zero or more times, or that are optional (this is controlled by the `kind` attribute). A `Token` specifies a keyword or a symbol. A `TemplateRef` is a "call" to another template, and the result can be assigned to a field of the clabject associated to the template. A `FieldRef` references a field, and a recognizer is generated according to its type. For instance, for primitive fields, if the field type is integer, only values satisfying the `[0-9]+` regular expression will be valid. In the case of references (i.e., non-primitive fields), an identifier is expected and an element of a compatible type is looked up in the name space (i.e., the current model plus all the imported models).

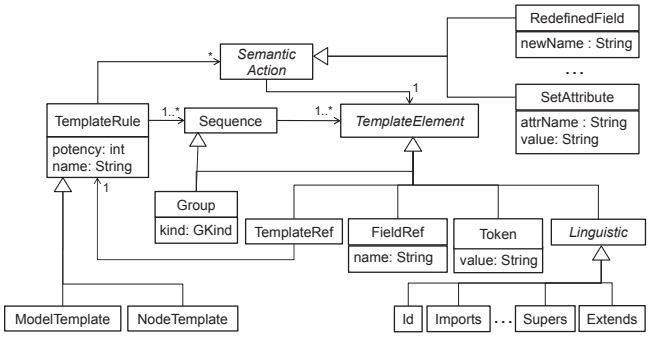

Fig. 10 Meta-model of the abstract syntax of the template language.

We have also defined several *linguistic* constructs to interact with the linguistic layer of models (see Figure 7) and control the possible linguistic extensions. These constructs are summarized in Table 1, and they will be explained in detail in Section 5.3.

Keywords for models:	
Extends	Allows extending a model
Imports	Allows importing a model
LingElements	Allows defining clabjects without ontological type (linguistic extensions)
Constraints	Allows defining new constraints (linguistic ex.)
Keywords for clabjects:	
Extends	Allows extending a clabject
Id	Assigns the identifier of a new clabject
Type	Allows using the direct type of the template (permits reuse across the inheritance hierarchy)
Typename	Allows using an indirect type of the template
Fields	Allows defining new fields (linguistic ex.)
Constraints	Allows defining new constraints (linguistic ex.)
Supers	Allows defining inheritance relations (linguistic ex.)
FieldValues	Allows assigning values to new fields (see Fields)
Instances	Allows instantiation of new clabjects (see LingElements)

Table 1 Keywords to access MetaDepth's linguistic layer.

Finally, templates may have semantic actions and syntactic predicates associated to the template elements. An action is triggered when some element of the template is recognized. For instance, one of such actions triggers the assignment of a literal value to a property when a given token is recognized (`SetAttribute` in the meta-model). The property can belong to a model or to the linguistic meta-model (see Figure 7). In the latter case, its name should be prefixed by "^". A syntactic predicate specifies how elements should be parsed or serialised. For instance, the `stdInt` predicate indicates that the parsed value of a given field should have an integer format, even if the property type is, e.g., string. Automatic conversions are performed when needed. Table 2 gathers the semantic actions and predicates currently supported.

5.1 Defining a concrete syntax to specify meta-models

Our template language allows the creation of a textual concrete syntax specifically designed to build meta-models for some domain. As an example, Listing 3 shows an excerpt of the definition of the concrete syntax for our example DSMM language, that is, a specialized syntax to describe process models at level 1.

```
1  Syntax DSPM_MM for ProcessModel [".pm_mm"] {
2    model template Process_Syntax@1 for ProcessModel
3      "process" ^Id "{"
4          (_:TaskTemplate | _:PerformerTemplate | _:SeqTemplate)*
5      "}";
6
7    node template TaskTemplate@1 for Task
8      ("final")? "task" ^Id #name
9    with "final" set final = true ;
10
11   node template PerformerTemplate@1 for Performer
12       "performer" #name
13   with name is id ;
14
15   node template SeqTemplate@1 for Seq
16     "seq" ^Id ":" #src "->" #tar
17   with src redefinedBy from
18          tar redefinedBy to;
19 }
```

Listing 3 Defining the concrete syntax for meta-level 1.

First of all, in line 1, the DSMM meta-model to which the syntax applies (the `ProcessModel` model) is declared, as well as the file extension for this language (*pm_mm*). All 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 2–5 define the syntax of the instances of the `ProcessModel` model. A syntax definition must have a model template acting as entry syntax rule to ensure that a model is created. Afterwards, lines 7–9 define the syntax of the clabject `Task`. The keyword `^Id` stands for the identifier of an element (see line 8), whereas the fields of a clabject can be referenced by using the prefix "#" (like `name` in line 8). Templates can refer to other templates, as in line 4, where it is indicated that the instances of `ProcessModel` can contain zero or more `Task`, `Performer` or `Seq` instances (we omit the templates for `Fork` and `Join` for brevity). The underscore in "_:TaskTemplate" means that the element created by the referred template does not need to be assigned to any property, which is typically the case for templates

Name	Type	Syntax	Meaning
set	Action	`"token" set property = value`	A property is set to a value when a token is recognized
redefinedBy	Action	`x redefinedBy y`	A reference type x is instantiated by reference y, which has also type facet
stdIdentifier	Pred.	`property is stdIdentifier`	A string property is recognized as an identifier (i.e., without quotes)
stdInt	Pred.	`property is stdInt`	A property is parsed as an integer and converted to the property type (e.g., Integer to Float conversion)
stdDecimal	Pred.	`property is stdDecimal`	A property is parsed as a decimal and converted to the property type (e.g., Decimal to String conversion)
fieldIsId	Pred.	`x is id`	A field is the key of the clabject: the linguistic Id is automatically set
openBlock	Pred.	`"begin" openBlock`	Establishes that a token opens a block for layout purposes
closeBlock	Pred.	`"end" closeBlock`	Establishes that a token closes a block

Table 2 Semantic actions and syntactic predicates supported in MetaDepth.

called from the root model template. The `with` keyword (lines 9, 13 and 17) introduces semantic actions and predicates. In the first case (line 9), the semantic action sets the `final` property to true when the "final" token is recognized[7]. In the second case (line 13), the predicate establishes that the value given to the `name` field will be used as the identifier (at the linguistic level) for the created clabject. In the third case (line 17), there are two semantic actions triggered when the identifier of a clabject assignable to `src` and `tar` is read. In this case, both references are instantiated creating the `from` and `to` references, as explained in Section 3.

Using this syntactic template, we can specify the meta-model for the DSML to describe software engineering processes as shown in Listing 4. This is a concrete syntax specifically designed to build DSMLs for process modelling. In this simplified example, we have defined three types of `Tasks`: `Analysis`, `Design`, and `Coding` (lines 2–4), which are put in sequence by two `Seq` clabjects: `a2d` and `d2c` (lines 10–11). Additionally, three instances of `Performer` are declared in lines 6–8.

```
1  process SoftwareProcess {
2    task Analysis "requirements and analysis"
3    task Design "high and low level design"
4    task Coding "coding and unit testing"
5
6    performer Analyst
7    performer Designer
8    performer Programmer
9
10   seq a2d: Analysis -> Design
11   seq d2c: Design -> Coding
12 }
```

Listing 4 Using the concrete syntax defined in Listing 3.

5.2 Defining a default concrete syntax for models

The user of the DSML also needs to be provided with a concrete syntax to describe the models at level 0. Initially, we define this concrete syntax at meta-level 2, when the DSMM is defined, and reuse it for all DSMLs in the domain.

Listing 5 declares such a syntax for our example. In this case, models will be stored in a separate file with extension *pm*, and all templates have potency 2. At this point, however, 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 `ProcessModel` (line 2). To access the name of the concrete type we use the keyword `^Typename`, which is interpreted by the parser to check that the recognized identifier refers to a type that is an instance of `ProcessModel` (line 3). The same applies to the definition of templates for clabjects (lines 8, 12 and 15). This definition provides a generic syntax for models at level 0, but specifically adapted for process models. As we will show later, it is possible to further specialize it for specific kinds of process models (e.g., in the software process or educational domains).

```
1  Syntax DeepDSPM for ProcessModel [".pm"] {
2    model template DeepProcess@2 for "ProcessModel"
3      ^Typename ^Id "{"
4        ( _:DeepTask | _:DeepPerformer | _:DeepSeq )*
5      "}" ;
6
7    node template DeepTask@2 for Task
8      "*" ^Typename ("!")? ^Id ("by" #perfBy)?
9    with "!" set initial = true ;
10
11   node template DeepPerformer@2 for Performer
12     "-" ^Typename ^Id ;
13
14   node template DeepSeq@2 for Seq
15     ^Typename ":" #src "->" #tar;
16 }
```

Listing 5 Defining the concrete syntax for meta-level 0.

Listing 6 shows the definition of a `SoftwareProcess` model using this syntax. By default, the syntax supports Java-like comments (lines 2, 6 and 10), but the style and symbols for delimiting comments can be configured. For instance, for one-line OCL-like comments and no multi-line comments, we add the following definition to line 1 in Listing 5: "... ["pm"], comments[one-line="- -", multi-line=off] {".

```
1  SoftwareProcess MyProcess {
2    // People involved
3    - Programmer Richard
4    - Designer Steve
5    - Analyst Bill
6    // Tasks
7    * Analysis doAnalysis by Bill
8    * Design designProduct by Steve
9    * Coding codeEverything by Richard
10   // Sequence of steps
11   a2d: doAnalysis -> designProduct
12   d2c: designProduct -> codeEverything
13 }
```

Listing 6 Using the concrete syntax defined in Listing 6.

[7] Although *final* has potency 2, it can be assigned a value at level 1, acting as default value for all instances at level 0.

Finally, similar to [19], 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, we use 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 writing "`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`. Its usefulness is a more compact description of the textual concrete syntax by taking advantage of the *inheritance* relation, so that e.g. we could define a unique template for all `Gateway` subtypes. 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. Therefore, it makes use of the *instantiation* relation.

5.3 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 level 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 at the concrete syntax level, before than at the abstract syntax level.

Our template language provides access to the linguistic layer of MetaDepth through the keywords summarized in Table 1. Some of these keywords allow selecting and including linguistic elements at the concrete syntax level of the DSMM language. These extensions can be accessed using the following keywords: `^Fields` to allow declaring new fields with no ontological type, `^LingElements` to allow new clabjects with no ontological type, `^Constraints` for declaring constraints (using EOL as constraint language), and `^Supers` to permit defining new inheritance relations for clabjects. Moreover, two additional keywords allow defining how these extensions should be instantiated at level 0: `^FieldValues` for field instances and `^Instances` for clabject instances. Finally, with the "set" semantic action, it is possible to set properties accessing the API of the linguistic meta-model in Figure 7.

For example, the concrete syntax definition of Listing 3 can be extended to allow defining new linguistic types like `ProgLanguage`, inheritance relationships between `Task` types, and new fields and constraints for tasks. Listing 7 shows the syntactic template that allows these linguistic extensions when creating DSMLs for process modelling, and Listing 8 shows its use. Line 5 in Listing 7 uses `^LingElements` to allow defining new clabjects as part of a process model, so that we can define the clabject `ProgLanguage` in Listing 8 (lines 19–22). Line 9 in Listing 7 uses `^Supers` to automatically include the syntax (and associated semantics) to inherit from other tasks. This allows `Analysis`, `Design` and `Coding` to inherit from `SETask` in Listing 8 (lines 6, 10, 14). In lines 10–11 and 17–18 of Listing 7, we use `^Fields` and `^Constraints` to permit adding new fields and constraints to tasks and performers. For instance, in Listing 8, new fields are added to the `SoftwareEngineer` performer in lines 25–26, while constraints are added to every task to check that a proper performer is assigned. The semantic actions in lines 13 and 20 of Listing 7 set the clabject as abstract (by accessing the linguistic layer) when the token "abstract" is read.

```
1  Syntax DSPM_MM for ProcessModel [ ".pm_mm" ] {
2    model template DSPM_MM_Syntax@1 for "ProcessModel"
3      "process" ^Id "{"
4        ( _:TaskTemplate | _:PerformerTemplate | _:SeqTemplate |
5          ^LingElements )+
6      "}" ;
7
8      node template TaskTemplate@1 for Task
9        ("abstract")? "task" ^Id #name ^Supers "{"
10         ^Fields
11         ^Constraints
12       "}"
13     with "abstract" set ^abstract = true ;
14
15     node template PerformerTemplate@1 for Performer
16       ("abstract")? "performer" ^Id #name ^Supers "{"
17           ^Fields
18           ^Constraints
19       "}" ;
20     with "abstract" set ^abstract = true ;
21
22     node template SeqTemplate@1 for Seq
23         "seq" ^Id ":" #src "->" #tar
24     with src redefinedBy from
25          tar redefinedBy to ;
26 }
```

Listing 7 Defining a concrete syntax for meta-level 1 which allows linguistic extensions.

```
1  process SoftwareProcess {
2    abstract task SETask "preparation" {
3      actor : SoftwareEngineer[*] { perfBy };
4    }
5
6    task Analysis "requirements and analysis" : SETask {
7      analyst : $ self.actor.forAll(a | a.isKindOf(Analyst)) $;
8    }
9
10   task Design "high and low level design" : SETask {
11     designer : $ self.actor.forAll(a | a.isKindOf(Designer)) $;
12   }
13
14   task Coding "coding and unit testing" : SETask {
15     uses : ProgLanguage[*];
16     programmer : $ self.actor.forAll(a | a.isKindOf(Programmer)) $;
17   }
18
19   Node ProgLanguage {
20     name : String;
21     version : String;
22   }
23
24   abstract performer SoftwareEngineer "se" {
25     notes : String;
26     expertise : ProgLanguage[*];
27   }
28
```

```
29    performer Analyst "analyst" : SoftwareEngineer {}
30    performer Designer "designer" : SoftwareEngineer {}
31    performer Programmer "programmer" : SoftwareEngineer {}
32
33    seq a2d: Analysis -> Design
34    seq d2c: Design -> Coding
35    seq inner: SETask -> SETask
36  }
```

Listing 8 Using the concrete syntax defined in Listing 7.

5.4 Refining the syntax of domain-specific modelling languages

Even if the DSMM language defines a concrete 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 syntax for some of the clabjects or for the linguistic extensions. For example, he may want to design a template especially for `SoftwareEngineer` so that the languages the engineer is expert in can be specified. For this purpose, either a new syntax is defined from scratch, or the default syntax is refined. Listing 9 shows the latter option applied to the running example.

```
1  Syntax SE for SoftwareEngineering imports DeepDSPM [".se"] {
2
3    override node template DeepPerformer@2 for Performer
4      _:AnalystTemplate | _:DesignerTemplate | _:ProgrammerTemplate ;
5
6    node template AnalystTemplate@1 for Analyst
7      "-" ^Id "the" "analyst" "knows" #expertise ("," #expertise)*
8
9    node template DesignerTemplate@1 for Designer
10     "-" ^Id "the" "designer" "likes" #expertise ("," #expertise)*
11
12   node template ProgrammerTemplate@1 for Programmer
13     "-" ^Id "the" "programmer" "masters" #expertise ("," #expertise)*
14 }
```

Listing 9 Refining the syntax for software engineering processes at level 1.

This syntax definition overrides the default template for `Performers` (line 3), calling to specific templates for `Analyst`, `Designer` and `Programmer`. This permits a reference to the `expertise` field defined at level 1, also using a different syntax in each case. Listing 10 shows the use of the refined part of the syntax.

```
1  SoftwareProcess MyProcess {
2    // People involved
3    - Richard the programmer masters Lisp, C
4    - Steve the designer likes Objective-C
5    - Bill the analyst knows Basic
6
7    ...
8  }
```

Listing 10 Using the refined concrete syntax in Listing 9.

5.5 Implementation remarks

As an implementation note, the presented template language for specifying concrete syntaxes has been implemented in MetaDepth using the meta-model shown in Figure 9, whereas its concrete syntax has been specified with itself, achieving bootstrapping. The parser generation relies on ANTLR. Any error concerning the resulting grammar (e.g., due to left-recursion) is appropriately caught and reported to the user, but currently we do not support tracing back errors from the generated parser to the template definition. Finally, MetaDepth has been enhanced with a registry of parsers, so that the parser to be used is selected by the file extension of the model to be loaded.

For the interested reader, all the complete listings of the running example are available in the Appendix B.

6 Model Management for DSMM Languages

In order to integrate the use of DSMM languages in MDE, we need to provide suitable model management languages able to deal with multiple meta-levels and the dual ontological/linguistic typing. In MetaDepth, we have adapted the Epsilon family of languages [18] to access transparently the linguistic API (as explained in Section 4), and 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) [29], code generators working at several meta-levels using the Epsilon Generation Language (EGL) [40], and model-to-model transformations spanning several meta-levels using the Epsilon Transformation Language (ETL) [30].

Figure 11 depicts the different scenarios encountered for model management operations in a multi-level setting. In the scenario (a), a model management operation is defined using the types of the DSMM language meta-model, and should be applicable to models with potency 0, two meta-levels below. The challenge here is being able to access indirect instances (with potency 0) of a given clabject with potency 2. The scenario (b) is a generalization of the previous one, where the operation in addition needs to access elements of potency 1. In this case, the challenge is to provide seamless navigation means from elements of potency 0 to their clabject types at the meta-level above. In the scenario (c), an operation defined using types with potency 2 and applicable at potency 0 is (totally or partially) redefined at meta-level 1. For this purpose, the languages should provide adequate overriding mechanisms, which typically are extensions of those used in combination with inheritance hierarchies. In scenario (d), the operation is defined using linguistic types, and hence it becomes applicable to clabjects of arbitrary ontological type and meta-level. In this case, the challenge is being able to mix ontological and linguistic typings in the model management languages. Finally, in all cases, operations may need to use reflection to consider the possible linguistic extensions at meta-level 1 and access the linguistic meta-model seamlessly.

Interestingly, the construction of textual syntaxes for DSMM languages also involved similar scenarios: the

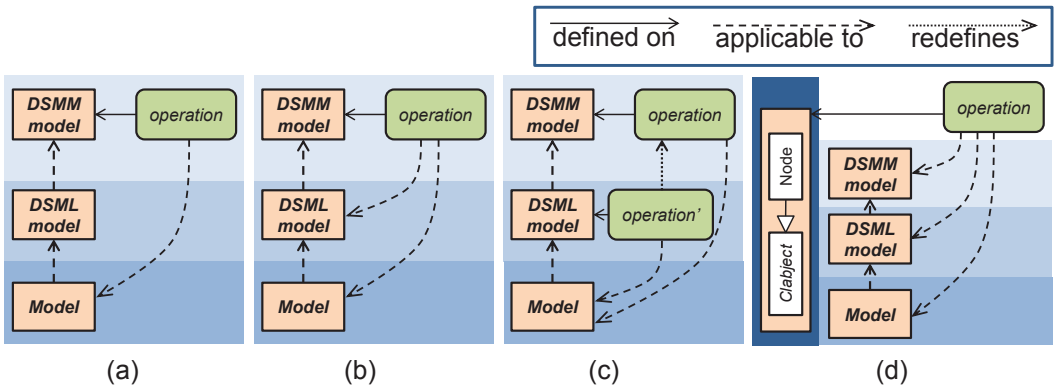

Fig. 11 Scenarios for model management operations in a multi-level setting. (a) Deep operation, applied two levels below. (b) Operation accessing multiple meta-levels. (c) Operation redefinition. (d) Operation defined over linguistic types.

definition of the syntax for meta-levels 0 and 1 (scenarios (a) and (b)), refining templates (scenario (c)), and the use of keywords like `LingElements` and `Instances` to account for linguistic extensions. The definition of generic concrete textual syntaxes using linguistic types is also possible, but experimenting with this feature is left for future work.

In the next subsections we concretize such scenarios for the different Epsilon model management languages, explaining how the different challenges posed by the multi-level setting have been solved.

6.1 Multi-level model-to-model transformation

Assume we want to transform process models into models of a given project management system (PMS), like *Redmine*[8]. The meta-model for such a PMS is shown in Figure 12 and in Listing 11 in MetaDepth's syntax. A PMS contains `Tickets` organized in `Categories` and assigned to `Users`. All these elements can be annotated by using `Tags`, which are pairs (key, value).

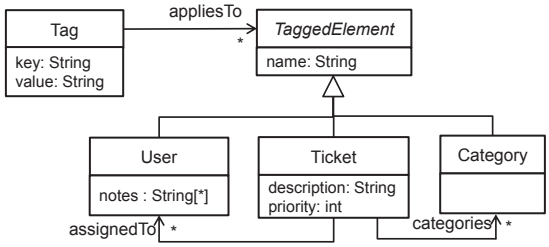

Fig. 12 Meta-model of a project management system.

```
1  Model PMS@1 {
2    abstract Node TaggedElement {
3      name : String;
4    }
5
6    Node User : TaggedElement {
7      notes : String[*];
8    }
9
10   Node Ticket : TaggedElement {
11     description : String;
12     priority : int;
13     assignedTo : User[*];
14     categories : Category[*];
15   }
16
17   Node Category : TaggedElement {}
18
19   Node Tag {
20     key : String;
21     value : String;
22     appliesTo : TaggedElement[*];
23   }
24 }
```

Listing 11 Meta-model of a project management system.

We would like to transform indirect `Task` instances at meta-level 0 into `Tickets`, which have as category the type of the task, and are assigned to the users who are performers of the task. 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*. Moreover, we also report on additional challenges found when the target domain of a transformation is multi-level.

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 in Figure 13. In this case, the transformation is applied on indirect types but the direct types at level 1 are unknown. For example, we would like to define a transformation from process models to PMS models once at meta-level 2, together with the definition of the DSMM language definition, and let the transformation be applicable to any process model at level 0.

Listing 12 shows the ETL deep transformation to achieve this objective, which will be executed on indirect instances of the `ProcessModel` model. The transformation has annotations indicating the source and target meta-models as well as the source potency

[8] `http://www.redmine.org/`

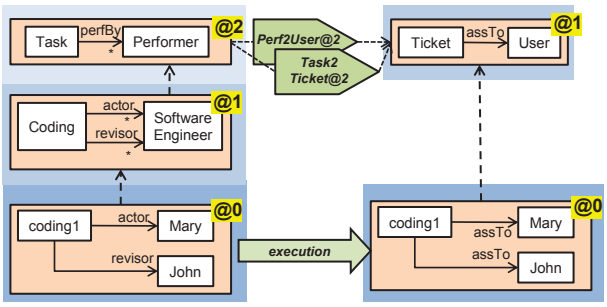

Fig. 13 Deep transformation scheme.

where the transformation is to be executed (lines 1–3).

```
1  @metamodel(name=ProcessModel,domain=source)
2  @metamodel(name=PMS,domain=target)
3  @model(potency=0)
4  rule Task2Ticket
5    transform task : Source!Task
6    to ticket : Target!Ticket
7  {
8      ticket.name := task.^name;
9      ticket.description := task.name;
10     ticket.priority := task.duration-1;
11     for (ref in task.references("perfBy"))
12         ticket.assignedTo ::= task.value(ref);
13 }
14
15 @lazy
16 rule Performer2User
17   transform per : Source!Performer
18   to usr : Target!User
19 {
20     usr.name := per.name;
21 }
```

Listing 12 Deep transformation example.

Rule `Task2Ticket` creates a `Ticket` for each indirect instance of `Task` (lines 4–13). The name of the created ticket is the object identifier of the task (line 8), its description is the name of the task (line 9), and its priority is equals to the `duration` of the task minus one (line 10). Recall that, while `task.^name` accesses the attribute *name* in the linguistic meta-model, `task.name` accesses the field `name` defined in upper ontological meta-levels.

Then, we need to transform performers into users and assign them tickets. At meta-level 2, `Tasks` and `Performers` are related through reference `perfBy`. However, this reference can be instantiated an arbitrary number of times at meta-level 1, and we do not know the names of such level 1 reference types beforehand. In the case of Figure 13, `perfBy` is instantiated twice, by `actor` and `revisor`, which themselves can be instantiated at level 0. For this purpose, the method `references` is used to return all instances of reference `perfBy` at level 1 (line 11). Lines 11–12 iterate over all such reference types, obtaining their value. The standard '::=' ETL operator assigns to the field `assignedTo` the result of transforming the content of the reference. If more than one instance of `perfBy` exists, their content is transformed and then added to the `assignedTo` collection. In particular, the instances of `perfBy` contain indirect instances of `Performer`, and therefore ETL invokes the *lazy* rule `Performer2User` to obtain the *equivalent* target objects of the source ones. This rule simply transforms indirect instances of `Performer` into `Users` (lines 15–21).

Please note that, for this *deep transformation* scenario, we had to modify ETL's rule matching semantics to consider indirect types in a transparent way. For example, the pattern `Source!Performer`, when applied to a model with potency 0, returns the accessible indirect instances of `Performer`.

Co-transformations. In this kind of transformations, one needs to transform a model and its meta-model at the same time, as Figure 14 illustrates. Here, the same transformation has to deal with direct and indirect instances of the clabjects in the meta-model of the DSMM language, and therefore a mechanism is needed to select the level at which the rules and queries will be applied, and to navigate up in the ontological hierarchy (i.e., from clabjects in meta-level 0 to clabjects in meta-level 1).

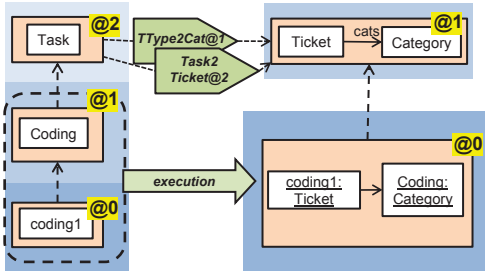

Fig. 14 Co-transformation scheme.

As an example, we want to assign each ticket a category, which is taken from the task types at potency 1. Listing 13 shows a modification of rule `Task2Ticket` and a new rule that implement this behaviour. Thus, we assign to `categories` the result of transforming the ontological type of the task (line 10). The level at which we seek the clabjects is specified by the model alias, before the '!' symbol (see lines 2, 3, 15 and 16). Hence, we use `SLevel0` for a model with potency 0 in the source domain and `SLevel1` for a model with potency 1 in the source domain. We can also use the alias `Source` to refer to the source model regardless its potency. Indeed, this is the alias used in Listing 12, where the annotation in line 3 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. A similar convention applies to the models in the target domain (i.e., `TLevel0`, `TLevel1` and so on). Additionally, using the linguistic API, it is possible to access the

type clabject of another one using `^type`, as shown in line 10.

```
1  rule Task2Ticket
2    transform task : SLevel0!Task
3    to ticket : Target!Ticket
4  {
5      ticket.name := task.^name;
6      ticket.description := task.name;
7      ticket.priority := task.duration−1;
8      for (ref in task.references("perfBy"))
9          ticket.assignedTo ::= task.value(ref);
10     ticket.categories ::= task.^type;
11 }
12
13 @lazy
14 rule TaskType2Category
15     transform task : SLevel1!Task
16     to category : Target!Category
17 {
18     category.name := task.name;
19 }
```

Listing 13 Co-transformation example.

Deep transformations and co-transformations are a form of *generic*, reusable transformations [43], as they are defined once and can be applied to *a family* of DSMLs: those that are conformant to the DSMM language meta-model at level 2. Co-transformations are generic, as they do not make assumptions on specific types to be defined at meta-level 1. Instead, these are accessed either from clabjects at meta-level 0 (via the linguistic attribute `type` as in line 10 of Listing 13), or by iterating over the types at meta-level 1 by using the `SLevel1` alias and the types defined at the top-most meta-level (like in line 15 of Listing 13).

Refining transformations. For DSMLs in some application domains, a deep transformation may not be reused "as is", but may need to be refined for particular clabjects defined at level 1. This situation is depicted in Figure 15.

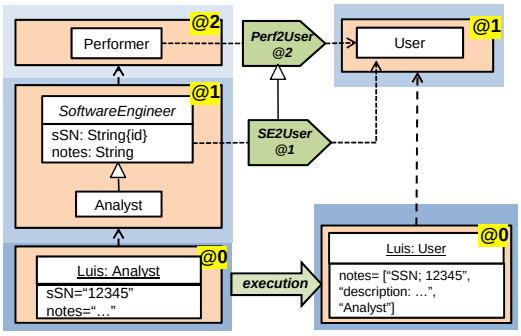

Fig. 15 Refining transformation scheme.

For example, if in the software engineering application domain, we decide to transform the instances of `SoftwareEngineer` in a different way to consider the specific fields that we added to it (i.e., `sSN` and `notes`), we need to refine the transformation rule defined for `Performers` in Listing 12. The refined rule is shown in Listing 14. The rule extends `Performer2User`, but it is refined for type `SoftwareEngineer` and will be executed at meta-level 0 (as indicated by `SLevel0!SoftwareEngineer` in line 3).

```
1  @greedy
2  rule SoftwareEngineer2User
3    transform se : SLevel0!SoftwareEngineer
4    to usr : Target!User
5    extends Performer2User
6  {
7      // The := operator applied to collections concatenates
8      usr.notes := "SSN "+se.sSN;
9      usr.notes := "description: "+se.notes;
10     usr.notes := se.type.toString();
11 }
```

Listing 14 Refining transformation example.

In this case, we needed to adapt the ETL rule extensibility mechanism [49]. In particular, we allow extending a rule if the child rule transforms a direct or indirect *instance* (and not just a subclass) of the clabject type transformed by the parent rule. The child rule will be applied whenever is possible, resulting in the execution of the body of the rules of both parent and child. In our example, rule `SoftwareEngineer2User` will be executed for instances of `SoftwareEngineer`, and by the overriding mechanism, the body of `Performer2User` will be executed as well. Rule `Performer2User` will be executed for indirect instances of `Performer` that are not instances of `SoftwareEngineer`. Finally, note that by default ETL rules match source elements of exactly the type given in the from part. By using the `@greedy` annotation we indicate that elements of the same type or a subtype should also be matched.

Reflective and linguistic transformations. When defining a deep transformation, we may want to account for the linguistic extensions 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 another form of generic transformations, applicable at any meta-level and to elements of any ontological type. A particular scenario of this kind of transformations is shown in Figure 16.

Listing 15 shows a transformation with one reflective rule and another rule defined on a linguistic type. Rule `Performer2User` has been modified with reflection to store in the `notes` collection all linguistic extensions defined in `Performer` instances (lines 7–8). Hence, the rule takes into account that `Performer` instances at level 1 may have been extended with new fields. The rule iterates on the new fields returned by `newFields` (line 7), adding a string representation of each field value to the `notes` collection. As previously stated, this reflection is pos-

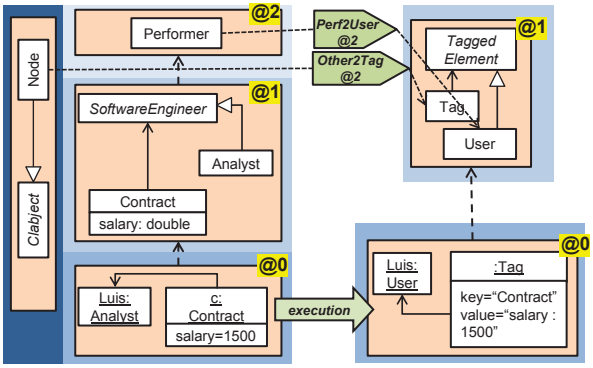

Fig. 16 Linguistic transformation scheme.

sible because EOL (and hence ETL) implements a transparent reflection mechanism, which we have modified to be able to call transparently methods of the MetaDepth API. Using this reflective rule makes the refined rule `SoftwareEngineer2User` of Listing 14 unnecessary. Nonetheless, in general, refining rules are still useful to implement domain-specific semantics that cannot be foreseen at the top-most meta-level.

```
1  @lazy
2  rule Performer2User
3    transform per : SLevel0!Performer
4    to usr : Target!User
5  {
6      usr.name := per.name;
7      for (f in per.newFields())
8          usr.notes := per.value(f).toString();
9  }
10
11 @greedy
12 rule Other2Tag
13   transform obj : SLevel0!Node
14   to tag : Target!Tag
15 {
16   guard: not obj.isKindOf(SLevel0!Task) and
17            not obj.isKindOf(SLevel0!Resource) and
18            not obj.isKindOf(SLevel0!Gateway)
19
20   tag.key := obj.^type.^name;
21   for (f in obj.fields())
22       tag.value:=tag.value+' '+f.name()+': '+f.getValue();
23   tag.appliesTo ::= obj.getConnectedClabjects();
24 }
25
26 operation Any getConnectedClabjects() : Sequence {
27     var allConn : Sequence;
28     for (f in self.references())
29       allConn.add( f.getValue() );
30     return allConn;
31 }
```

Listing 15 Linguistic transformation example.

In its turn, rule `Other2Tag` (lines 11–24) uses linguistic typing. The rule creates a `Tag` from all `Node` instances (i.e., from all elements) which are not indirect instances of `Task`, `Resource` and `Gateway` (forbidden by the guard in lines 16–18). If we apply this rule to the model excerpt shown in Figure 16, where a new clabject `Contract` has been defined at level 1, we obtain a `Tag` object from this clabject. The key of the tag is the identifier of the object ontological type (line 20), while its value is the concatenation of each field name and value (lines 21–22).

The method `fields` used in line 21 returns all fields owned by the given clabject. The created `Tag` is attached to the objects resulting from the translation of the clabjects connected to the transformed node (line 23). The connected clabjects are obtained reflectively by the helper method in lines 26–31.

While in deep and co-transformations the genericity comes from defining the transformations over types defined at meta-level 2, so that the transformations become applicable to a family of DSMLs, in linguistic transformations the genericity is obtained by using linguistic types, so that the transformations become applicable to arbitrary models. As we have seen, it is possible to mix both types of genericity in the same transformation definition.

Multi-level target. In our running example, the target language has two meta-levels, but one can find scenarios in which the target language is multi-level as well. For example, assume we want to transform from PMS models to process models, as shown in Figure 17. Thus, we want to create a task type at meta-level 1 for each category, and an instance of the created task type for each ticket in that category.

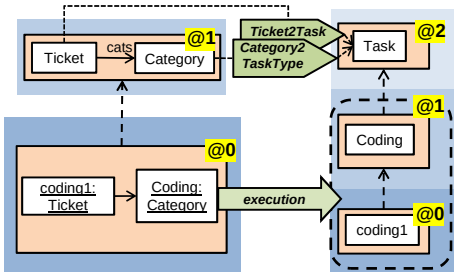

Fig. 17 Scheme of transformation with multi-level target.

This scenario poses an additional challenge: the transformation needs to produce models at two different meta-levels (1 and 0), co-creating types and their instances, but the type of the instances is unknown at design time, and will only be known at run time. Although the linguistic layer of MetaDepth offers methods to create instances of a given type, the challenge is to design rules whose target type is unknown beforehand. An example of how we tackle this issue is shown in Listing 16.

```
1  @metamodel(name=PMS,domain=source)
2  @metamodel(name=ProcessModel,domain=target)
3  rule Category2TaskType
4    transform category : Source!Category
5    to task : TLevel1!Task
6  {
7    task.^name := category.^name;
8    task.name := category.name;
9  }
10
11 rule Ticket2Task
12   transform ticket : Source!Ticket
13   to task : TLevel0!Task
14 {
15   guard: TLevel0!Model.refineType('Task',
16            ticket.categories.first().^name)
```

```
17
18    task.duration := ticket.priority+1;
19 }
```

Listing 16 Transformation with multi-level target example.

Rule `Category2TaskType` transforms each category into a `Task` instance at meta-level 1, as indicated by the `TLevel1` alias. The created task has as name identifier (linguistic field) the identifier of the category (line 7), and as name (ontological field) the name of the category (line 8). Then, rule `Ticket2Task` transforms each ticket into a `Task` indirect instance at meta-level 0, as indicated by the `TLevel0` alias. As the direct type of this instance is unknown beforehand, the only type information we can provide in the rule signature is `TLevel0!Task`. The rule guard ensures that the rule is executed only if there is an appropriate direct type, in which case the rule assigns this type to the created task instance. This is done by method `refineType` from MetaDepth's API, which receives the indirect and direct type names as parameters. In our case, the direct type is the identifier `^name` of the first category assigned to the ticket. For clarity, we have used the standard syntax of ETL to invoke `refineType` in the guard, although we could design a special syntax to express these refinements.

6.1.1 Comparison with standard two-level approaches. Once we have reviewed the possible multi-level scenarios, it is interesting to reflect on how they could be solved in the workarounds shown in Figure 1 using standard two-level model transformation languages (workaround 1: model/meta-model at the same level; workaround 2: promotion transformation).

A general drawback of workaround 1, which applies to every scenario, is that the instantiation of clabjects with linguistic extensions cannot be done directly, but needs from a special instantiation mechanism. Such a mechanism needs to traverse all fields with type facet of the clabject type (the linguistic extensions), creating slots with appropriate data type in the instance.

Deep transformations can be easily implemented in workaround 1, as the meta-model explicitly defines elements with instance facet, and a transformation can be defined over those elements. However, this scenario is more challenging in workaround 2, as a transformation should be defined over the initial meta-model (model (a) in Figure 5), but the promotion generates a new meta-model (model (c) in Figure 5), and the transformation would not be applicable on its instances.

Co-transformations access multiple meta-levels. This is possible in workaround 1, as *"instance of"* relations are plain links at the model level, and can be traversed by a transformation by writing ad-hoc navigation expressions. Implementing co-transformations in workaround 2 becomes more complex, as one needs to navigate between two disconnected models at the instance level (model (d) and (b) in Figure 5). A transformation should have both models as input, and we would need to write navigation expressions relating them as part of the transformation logic.

Refining transformations are challenging in both workarounds. Workaround 1 requires a mechanism to refine a rule defined over an element with type facet, for elements with instance facet. However, refinement mechanisms in standard transformation languages are only available if the elements are in a subtype relation, which is not the case. The same challenge arises in workaround 2.

Reflective and linguistic transformations are possible in both workarounds whenever the transformation language provides reflective and generic support. Finally, the multi-level target scenario could be implemented in both workarounds, but the implementation should face similar challenges to the ones presented here for the dynamic creation of types and their instances.

Altogether, from the model management perspective, workarounds 1 and 2 have some drawbacks compared to a multi-level approach. These could be overcome by heavy modifications of existing transformation languages, or by creating a domain-specific transformation language for the given workaround.

6.2 Multi-level in-place model transformations

In this subsection, we illustrate multi-level in-place transformation by defining a simulator for process models. The simulator uses the types defined at meta-level 2, and is applicable to model instances at potency 0. The implemented semantics is similar to that of Petri nets [34]. First, initial task instances are located, which constitute the set of *enabled* tasks. Then, the following simulation loop starts. One enabled task is selected at random, and if it is initial, it is initialized. The initialization consists in the creation of the task input artefacts. By default, we create an arbitrary number of artefact instances between the minimum and the maximum allowed. This initialized task becomes part of the set of *active* tasks. An active task can be *completed*, and then its output artefacts are created similarly to the creation of input artefacts. The output artefacts become input artefacts of any consecutive task, and any task that has all needed input artefacts becomes enabled. The simulation proceeds by iterating in this loop and finishes when there is no enabled task that can become active, and there is no active task than can be completed.

We use EOL to define in-place transformations. In this language, it is possible to define operations on a global scope and to attach operations to meta-classes. We have adapted EOL to work in a multi-level setting by assigning potencies to operations, and implementing a method overriding mechanism across meta-levels. Hence, when a method is invoked on a particular clabject, its

type at the immediate meta-level above is sought. At this meta-level, method lookup works as in standard object oriented programming languages, using dynamic dispatch [1] (i.e., by an upwards traversal of the inheritance hierarchy). If the method is not overridden at this meta-level, then it is sought in the next upper meta-level following the same procedure, and so on.

Figure 18 shows an example of this mechanism. To build our simulator, we have defined operations `createInps()` and `createOutps()` in clabject `Task` at meta-level 2. They are used to create the input and output artefacts of a task, and contain a default implementation via reflection. These methods can be overridden at the next meta-level. For example, the figure shows the overriding of some of them in `Analysis`, `Design` and `Coding`, to create the appropriate artefacts for these kinds of task. An invocation to `createInps` on an instance of `Analysis` results in the execution of the method overridden in `Analysis`, whereas an invocation on an instance of `Coding` results in the execution of the method defined in `Task`.

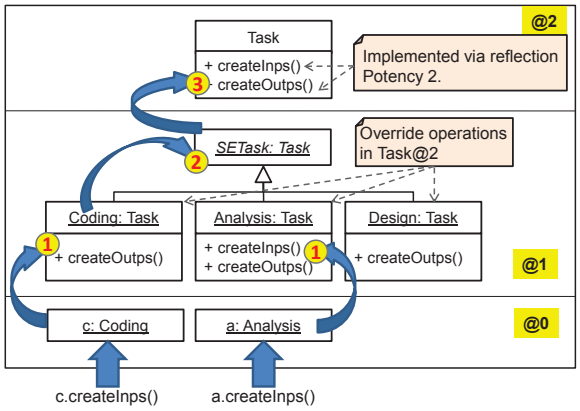

Fig. 18 Method overriding in a multi-level environment.

Interestingly, this mechanism permits a natural generalization to enable the overriding of methods at potency 0, defined for specific clabjects with no type facet. However, we leave this extension for future work.

We have also modified EOL to permit the declaration of `abstract` operations. An abstract operation with potency 1 has the same semantics as in standard object oriented programming languages: every concrete clabject, or some ancestor in the inheritance hierarchy, needs to override it. However, an abstract operation with potency 2 can also be overridden at the lower meta-level. As usual, each instance of the clabject needs to have access to an implementation of the abstract method, which is sought using the lookup mechanism described before. Currently, this is not checked statically at modelling time, but errors are reported at run-time using appropriate exceptions.

Listing 17 shows a small excerpt of the simulator. The operation `main` contains the main simulation logic, and is annotated with the needed meta-model (line 1) and its potency (line 2). Lines 4–8 iterate on all indirect task instances at level 0, creating three boolean auxiliary fields to store their state: `active`, `completed` and `enabled`. Prefixing attributes by ∼ is the standard way in EOL to create auxiliary fields. Then, lines 10 and 11 create the set of enabled and active tasks, and lines 12–18 initialize the set of enabled tasks with all initial ones. One of such initial tasks is selected (line 20) and then initialized (line 24). Then, the main simulation loop in lines 25–29 is run while some enabled or active task remains.

```
1  @metamodel(name=ProcessModel,file=DSPM.mdepth)
2  @potency(value=2)
3  operation main() {
4    for (t in Task.all()) { // create auxiliary fields to record task state
5      t.~active := false;
6      t.~completed := false;
7      t.~enabled := false;
8    }
9
10   var enabledTasks : Set(Task);
11   var activeTasks : Set(Task);
12   var initTasks := Task.all().select( t | t.initial );
13
14   //...
15   for ( t in initTasks ) {
16     t.~enabled := true;
17     enabledTasks.add(t);
18   }
19
20   var tinit := initTasks.random();
21   activeTasks.add(tinit);
22
23   var step := 0;
24   tinit.initializeTask();
25   while (enabledTasks.size()>0 or activeTasks.size()>0) {
26     completeTask(tinit);
27     step := step+1;
28     //...
29   }
30   'Simulation finished!'.println();
31 }
32
33 @potency(value=2)
34 operation Task createInps( ) {
35     for ( ref in self.references("ins") ) {
36       var ttype := self.type.value(ref);
37
38       var max : Integer;
39       if (ref.getMaximum()=-1) max := 10;
40       else max := ref.getMaximum();
41
42       var num := Sequence{1..max}.random();
43
44       for (i in Sequence{1..num}) {
45         var ainst := self.container.createInstance(ttype.toString());
46         self.setValue(ref, ainst);
47       }
48   }
49 }
50 ...
```

Listing 17 Excerpt of a simulator for process models.

Lines 33–49 show the operation `createInps` defined for clabject `Task` with potency 2. The operation reflectively iterates on all instances of reference `ins` (line 35), accesses the type of the reference end (line 36), instantiates the type a random number of times up to the maximum cardinality (the maximum is 10 if the cardinality is unbounded), and assigns the created instances to the reference (line 46).

While the previous listing uses a default, reflective implementation for `createInps`, Listing 18 shows the redefinition of method `createOutps` for clabjects `Analysis` and `Design`. In the listing, `RequirementsDoc` and `DesignDoc` are both instances of `Artefact`. This mechanism enables the implementation of application-specific code, overriding the simulator defaults.

```
1 operation Analysis createOutps() {
2     var req := new RequirementsDoc();
3     self.output := req;
4 }
5
6 operation Design createOutps() {
7     var dd := new DesignDoc();
8     self.output := dd;
9 }
```

Listing 18 Overriding some operations for software process models.

6.3 Multi-level code generation

In this subsection, we illustrate code generation in a multi-level setting by showing a code generator from process models into XML logs in the eXtensible Event Stream (XES) format [22]. This is the log format adopted by the IEEE task force on process mining, and was designed with extensibility in mind. In this way, it is possible to extend the standard for special requirements, like specific application domains or specific tool implementations.

In XES, event *logs* are made of an arbitrary number of *traces*. Each trace describes the execution of one specific instance, or case, of the logged process, and is made of a number of *events*. Logs, traces and events can be described by means of attributes. It is also possible to define *classifiers*, which contain attribute types, and are used to describe events. Hence, XES logs serialize information found at meta-levels 0 and 1.

Listing 19 shows an excerpt of the XES code generator that has been implemented using the EGL language. EGL is a template-oriented language, similar to Java Server Pages (JSP), in which the text to be emitted is written in a template. The language allows inserting EOL code enclosed between "[%" and "%]", and the result of an EOL expression *exp* resulting in a string can be emitted by using the syntax "[%=*exp*%]".

Lines 1–6 in Listing 19 write a preamble, while lines 7–10 define the attributes for the trace. In this case, there is only one attribute that will contain the name of the model. The attributes for event types are defined in lines 11-23, first the common attributes to all tasks (lines 12–13) and then the extensions declared at level 1 (lines 15–22). In particular, line 16 iterates on all non-abstract tasks at level 1 using `SLevel1!` as prefix, and line 17 loops on every new field (i.e., linguistic extension) with a basic data type. To avoid name collisions, the name of the field to be serialized is the concatenation of the names of the task and the field. Next, lines 24–28 declare the classifiers by selecting all non-abstract tasks at level 1. The attributes belonging to each classifier are calculated by function `getAllFields` defined on `Task` (lines 46–51).

```
1  <?xml version="1.0" encoding="UTF−8" ?>
2  <log xes.version="1.0" xes.features=""
3       xmlns="http://code.deckfour.org/xes">
4     <extension name="Time" prefix="time"
5      uri="http://code.deckfour.org/xes/time.xesext"/>
6  <!−− ...>
7    <global scope="trace">
8       <!−− traces have a name (the name of the model) −−>
9     <string key="concept:name" value="name"/>
10   </global>
11   <global scope="event">
12    <string key="concept:name" value="name"/>
13    <boolean key="initial" value="false"/>
14 // ...
15 [% // iterate on linguistic extensions of each direct instance of Task...
16   for (t in SLevel1!Task.all().select( t | not t.abstract ) ) {
17     for (f in t.newFields().select( f | f.isDataType() ) ) { %]
18       <[%=f.fieldType.toString().toXESType()%]
19         key="[%=t.toString()+'_'+f.name()%]"
20         value="[%=f.fieldType.toString().toXESType()%]"/>
21 [% }
22   }%]
23   </global>
24 [% // Now define the classifiers... These are the Tasks at level 1...
25    for (t in SLevel1!Task.all().select( t | not t.abstract)) {%]
26      <classifier name="[%=t.name()%]"
27        keys="[%=t.getAllFields()%]"/>
28 [% }%]
29    <trace>
30      <string key="concept:name" value="[%=context.name()%]"/>
31 [% for (t in SLevel0!Task.all()){%]
32        <event>
33          <string key="concept:name" value="[%=t.name%]"/>
34 //...
35 [% for (f in t.type.newFields().select( f | f.isDataType() ) ) { %]
36            <[%=f.fieldType.toString().toXESType()%]
37              key="[%=t.type.toString()+'_'+f.name()%]"
38              value="[%=t.value(f.name())%]"/>
39 [% }%]
40        </event>
41 [%}%]
42    </trace>
43 </log>
44
45 [%
46 operation Task getAllFields() : String {
47   var fields : String := "concept:name initial final start duration";
48   for (f in self.newFields())
49     fields := fields+' '+self.toString()+'_'+f.name();
50   return fields;
51 }
52
53 operation String toXESType() : String {
54   if (self.toString()=="String") return "string";
55   //...
56 }
57 %]
```

Listing 19 A multi-level code generator for XES.

What we have explained up to now is the serialization of the information from level 1. The model at level 0 is serialized in lines 29–43: line 30 stores the name of the model, and line 31 iterates on all tasks at level 0 and serializes the value for each field.

Altogether, this example shows that the scenarios found for other model management operations also apply here. We needed to access the models at meta-levels 1 and 0, to navigate from elements in meta-level 0 to meta-level 1, and to query reflectively the linguistic extensions. These mechanisms are basically the same as

in EOL. Code generators could also make use of refinement operations that override EOL operations (e.g., `getAllFields`) for certain types at level 1, although it was not needed for this particular example.

7 Related Work

In this section, we review related research. We start by reviewing the uses, applications and approaches to DSMM. Then, we follow with a discussion of multi-level modelling environments, with special attention to their integration with model management languages. As a model operation defined over a DSMM language can be reused by the DSMLs built with it, we continue by revising reutilization approaches for model management operations. Finally, we finish with an overview of relevant approaches to the definition of textual concrete syntaxes and a summary of our contributions.

7.1 Approaches to domain-specific meta-modelling

Several researchers have pointed out the benefits of using DSMM languages. For example, the traceability modelling language [17] (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 [50] to express variability over DSMLs, and to extend DSMLs with interfaces for model reuse. However, no general framework for defining 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 [23], the authors present a language to declare component types with ports, which can be instantiated choosing a number of port instances. The language is extended with generic library mechanisms for the purpose of defining libraries of reusable components. Such libraries emulate to some extent the existence of two meta-levels within one. While the authors neatly show how to build this support in a generic and minimally invasive way for its integration with existing meta-models, the proposed prototype-cloning mechanism does not emulate fundamental aspects of instantiation, like being able to define attributes in meta-models and providing values for them in models, or being able to define integrity constraints in meta-models which are evaluated in models. Including such aspects in a cloning mechanism would lead to the construction of a type system emulating two meta-levels in one, which is the strategy followed in [5] to implement a multi-level system in EMF.

In [32], the UML profiling mechanism is adapted for EMF-based DSMLs. This is an example of DSMM as users need a language to define 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. Such a complex mechanism is unnecessary in multi-level frameworks supporting deep characterization, as we have seen.

Instead of emulating several meta-levels within two [23] or using artificial workarounds [32,50], we claim that a more natural way to define DSMM languages is the native use of multi-level meta-modelling. Following this philosophy, in [25], multiple levels are used to define domain-specific process modelling notations. However, the approach is restricted to meta-modelling only, and does not consider the language concrete syntax or its manipulation through model management languages, making its use difficult in MDE.

7.2 Approaches to multi-level meta-modelling

In this work, we have proposed using multi-level meta-modelling to handle the current limitations in DSMM frameworks. Multi-level meta-modelling was initially proposed by Atkinson and Kühne [4,9]. Apart from MetaDepth, there are other multi-level meta-modelling frameworks, which we review next.

MelAniE [5–7] is a graphical multi-level modelling tool based on EMF and GMF. Regarding modelling features, MelAniE provides fields with so-called traits, like *value mutability*, which defines over how many levels the value may be changed from the default. Modelling elements must define a *level*, which in our case we indicate through the *potency* of the model. While this avoids a manual definition of the level for every element, all elements in the same model are restricted to have the same level. The authors do not mention support for constraints, as we have in MetaDepth. Regarding model management, in [6], the authors present the first steps towards extending the transformation language ATL to work in a multi-level setting. They discuss challenges of the two-level to multi-level and multi-level to two-level transformation scenarios; however, the paper handles only the multi-level to two-levels, deep transformation scenario, that is, the transformation is always executed on the bottom-most model of the source domain. This limitation makes unnecessary the use of potencies for rules. MelAnIE provides more intrusive mechanisms to access ontological fields (using "._o_." as a prefix to every field) and the linguistic layer (using "._l_." as a prefix), than those we have presented here. Currently, MelAniE does not integrate other model management languages (e.g., for code generation or in-place transformation) as MetaDepth does, although the authors plan to do so.

Being graphical, MelAniE [5] supports the possibility to mix a default graphical concrete syntax similar to object diagrams with a domain-specific one. This approach is similar to our refining templates for textual syntax.

DeepJava [31] extends Java with the ability to perform multiple instantiations, and allows potency on Java classes, attributes and methods. This approach is comparable to our adaptation of EOL to work in a multi-level setting. There are a few differences, though. While in DeepJava one can use a field *type* to access the clabject ontological type, there seems to be no further support for accessing a linguistic layer reflectively (e.g., one cannot check which are the new linguistic extensions of a clabject). The reason is that DeepJava programs are pre-processed and compiled into plain Java, while MetaDepth offers run-time support. Both tools allow overriding methods defined at higher meta-levels, but we could not find the details on the working scheme of dynamic dispatch across meta-levels in DeepJava. Another difference is that we avoid instantiation of clabjects by declaring them abstract, while in DeepJava one should assign potency 0 to those clabjects. However, clabjects with potency 0 cannot be an instance of another clabject (i.e., they should be linguistic extensions), as their potency cannot be defined arbitrarily, but need to be one less than the potency of their type clabject. Finally, other differences (e.g., lack of control of linguistic extensions and lack of dedicated model management languages) are due to the fact that DeepJava is a programming language whereas MetaDepth is a modelling language.

The cross-layer modeller (XLM) [16] supports an arbitrary number of modelling layers by using an embedding in UML and giving semantics to instantiation as OCL constraints. In particular, the designer needs to specify templatized OCL constraints to control the instantiation of associations. XLM does not provide advanced instantiation mechanisms, like deep instantiation, or meta-modelling features like attributes/links, or inheritance.

In other multi-level frameworks, like [2], the main concern is the efficient navigation between meta-levels, but they do not consider the concrete syntax of model management languages. Nivel [3] is a multi-level meta-modelling framework based on the weighted constraint rule language (WCRL). OMME [48] is a prototype that implements advanced modelling constructs like a dual ontological/linguistic typing, deep characterization and powertypes. However, none of these frameworks are connected to model management languages.

Powertypes are another means of multi-level meta-modelling [36]. The powertype pattern consists in a pair of classes in which one is a powertype of the other. Roughly, while one class describes the type facet of the instances, the other one describes their instance facet. Instantiating a powertype means subtyping from the class containing the type facet and instantiating from the class containing the instance facet. Hence, both relations "instance-of" and "subtype" are allowed to cross meta-levels [20]. While powertypes can assign both slots and attribute types to entities in the meta-level immediately below, the potency allows assigning attribute types to elements at arbitrary meta-levels below.

Powertypes are used in [20] to define meta-models for software methodologies. This proposal models both the process and product aspects of methodologies, while proposals like SPEM only consider the process aspect [21]. In our example, the product aspect can also be modelled by instantiating `Artefact` and in combination with linguistic extensions. Since the approach in [21] does not consider an orthogonal linguistic typing, it needs to explicitly include the primitives for product modelling (e.g., `Model`, `ModelUnit`) in the top-level meta-model.

Approaches to multi-level meta-modelling can be traced back to the eighties in knowledge-based systems like Telos [35] and deductive object base managers like ConceptBase [26]. ConceptBase implements the object model of a Datalog-based variant of Telos. It supports instantiation chains of arbitrary length and definition of Event-Condition-Action (ECA) rules and integrity constraints, but not deep characterisation (i.e., the ability to influence meta-levels below the immediate one). Coming from the database tradition, it is not integrated with model management languages. However, some experiments for the use of ECA rules for model-to-model transformations have been recently proposed [33], but not considering the multi-level setting.

7.3 Reuse of model management operations

A DSMM language capitalizes on the knowledge in a certain domain (e.g., process modelling), and permits defining DSMLs for different application areas (e.g., educational modelling, software process modelling). A model management operation defined over the DSMM language becomes reusable for the DSMLs. Next, we briefly review other reutilization approaches for model management operations.

In our own previous work on *generic* model management [13,44], we can define reusable model operations over so-called concepts. Concepts are similar to meta-models, but their elements (classes, fields, references) are variables that need to be bound to the elements of a meta-model. Hence, an element in the meta-model can be related to at most one element in the concept. This binding induces a re-typing of the model operation, which becomes applicable to the bound meta-model. In this approach, meta-models can exist even before defining the concept, and reuse is achieved by binding the concept to the meta-model. In the multi-level approach presented in this paper, the meta-model of the DSMM language plays the role of "concept", the DSMLs can only exist after creating the DSMM language (as they

are instances of it), and reuse is achieved directly without the need of an explicit binding. In contrast to a binding, the *type* relation enables many-to-one relations from the DSML to the DSMM language.

In other approaches like [45], reuse is obtained by adapting the meta-models to which an existing transformation is to be applied. The aim of adapting the meta-model is to make it a subtype of the expected input meta-model of the transformation [47], so that the transformation can be applied without changes.

7.4 Approaches to the definition of concrete syntaxes

Although there are many approaches to define textual concrete syntaxes for DSMLs [19,27], 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.

7.5 Summary

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. Finally, while this paper is focused on DSMM with three meta-levels, MetaDepth is not restricted to work with three levels, but it supports an arbitrary number of levels.

8 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, linguistic/reflective transformations and single-level to multi-level) and illustrated their support using the Epsilon model management languages. The same scenarios apply to the definition of textual syntaxes as well: 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, which in our experience are: the ability to select the meta-level at which a certain operation is to be applied (e.g., potencies for operations and templates), the ability to select clabjects of specific meta-levels (e.g., alias `SLevel0` and `SLevel1` in rules), the possibility to obtain indirect instances of clabjects (transparently in our case), to execute a certain operation/rule/constraint several meta-levels below (by attaching a potency), to access clabjects by their linguistic type (e.g., `Node`) and to reflectively access linguistic extensions (e.g., through the `newFields()` method). Table 3 summarizes the different techniques we have implemented, indicating their usefulness for each scenario.

Model management operations defined over the DSMM language meta-model become generic, reusable in a domain, and applicable to the instances of the family of DSMLs defined by DSMM language. Further support for genericity in this sense can be obtained by the use of linguistic types and access to the linguistic layer.

We are currently using MetaDepth to define DSMM languages in different domains: component-based systems, web engineering and mobile devices. We also aim at studying processes and methodologies for defining DSMM languages, including their concrete syntax. We foresee combining a top-down approach (consider the top-most meta-model first) with a bottom-up approach (draft examples at the bottom-level, and then generalizing) [42]. We are exploring the definition of visual syntaxes for DSMM languages, as well as improving the DSMM support by enabling the definition of operations on pure clabject instances, controlling the extensibility of Epsilon rules and operations, improving our templates for textual syntax (e.g., enabling a more detailed customization of the syntax for linguistic extensions) and performing additional static checks to catch more errors at design time. As we showed in Section 6.1, the definition of a special syntax for multi-level model-to-model transformations is also desirable. We plan to tackle this issue using the Eclectic framework [41]. Finally, we are currently planning a detailed comparison of the multi-level approach and possible workarounds, from the modelling and model management perspective, using a real-life case study and an empirical evaluation.

Acknowledgements. We thank the referees for their detailed and useful comments. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139), and the R&D programme of Madrid Region with project "eMadrid" (S2009/TIC-1650).

Scenario	Purpose	Technique
(a)	Access to instances of indirect types	Transparent modification of semantics of standard operators like *X.allInstances()*.
(a)	Define constraints/operations/rules/syntactic templates applicable to indirect instances of a given clabject	Add potency to constraints/operations/rules/syntactic templates.
(b)	Navigate to clabject type	Transparent access to linguistic layer using *X.type*.
(b)	Iterate over clabject instances at a particular meta-level	Special semantics for model aliases (*SLevel0*, *TLevel0*, *SLevel1*...)
(c)	Operation/rule/syntactic template refinement at lower meta-levels	Operation overriding across meta-levels (EOL and EGL), rule inheritance (ETL), template overriding.
(d)	Control of linguistic extensions	Use of tag *strict* and access to linguistic layer from constraints.
(d)	Reflective access to linguistic extensions	Transparent access to API of linguistic layer.
(d)	Use of linguistic types	Special semantics for linguistic types, in combination with query operations like *allInstances()*. Transparent access to API of linguistic layer.
(e)	A rule needs to create a clabject with a type unknown at model transformation design time	Reflective mechanism for type refinement (*refineType*).

Table 3 Summary of techniques to enable DSMM using multi-level meta-modelling. Scenario (a) refers to deep operations, (b) to operations requiring access to multiple meta-levels, (c) to refining operations at lower meta-levels, (d) to access to linguistic extensions reflectively and the use of linguistic types, and (e) to model transformations with multi-level target.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Monographs on Computer Science, 1996.
2. T. Aschauer, G. Dauenhauer, and W. Pree. Representation and traversal of large clabject models. In *MoDELS'09*, volume 5795 of *LNCS*, pages 17–31. Springer, 2009.
3. T. Asikainen and T. Männistö. Nivel: a metamodelling language with a formal semantics. *Software and System Modeling*, 8(4):521–549, 2009.
4. C. Atkinson. Meta-modeling for distributed object environments. In *EDOC*, pages 90–101. IEEE Computer Society, 1997.
5. C. Atkinson, R. Gerbig, and B. Kennel. Symbiotic general-purpose and domain-specific languages. In *ICSE'12 (New Ideas and Emerging Results track)*, pages 1269–1272, 2012.
6. C. Atkinson, R. Gerbig, and C. Tunjic. Towards multi-level aware model transformations. In *ICMT'12*, volume 7307 of *LNCS*, pages 208–223. Springer, 2012.
7. C. Atkinson, M. Gutheil, and B. Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Trans. Soft. Eng.*, 35(6):742–755, 2009.
8. C. Atkinson and T. Kühne. The essence of multilevel metamodeling. In *UML*, volume 2185 of *LNCS*, pages 19–33. Springer, 2001.
9. C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.
10. C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.
11. C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Software and System Modeling*, 7(3):345–359, 2008.
12. J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS'10*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010. See also `http://astreo.ii.uam.es/~jlara/metaDepth`.
13. J. de Lara and E. Guerra. From types to type requirements: Genericity for model-driven engineering. *Software and System Modeling*, page in press, 2011.
14. J. de Lara and E. Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In *ECMDA-FA'12*, volume 7349 of *LNCS*, pages 259–274. Springer, 2012.
15. J. de Lara, E. Guerra, R. Cobos, and J. Moreno-Llorena. Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal*, In press, 2012.
16. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking. In *SIGSOFT FSE*, pages 452–455. ACM, 2011.
17. N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes. Engineering a DSL for software traceability. In *SLE'08*, volume 5452 of *LNCS*, pages 151–167. Springer, 2008.
18. Epsilon. `http://www.eclipse.org/epsilon/`, 2012.
19. 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.
20. C. Gonzalez-Perez and B. Henderson-Sellers. A powertype-based metamodelling framework. *Software and System Modeling*, 5(1):72–90, 2006.
21. C. Gonzalez-Perez and B. Henderson-Sellers. Modelling software development methodologies: A conceptual foundation. *Journal of Systems and Software*, 80(11):1778–1796, 2007.
22. C. W. Günther. Xes 1.0 extensible event system standard definition. Technical report, See also `http://www.xes-standard.org/`, 2009.
23. M. Herrmannsdörfer and B. Hummel. Library concepts for model reuse. *Electron. Notes Theor. Comput. Sci.*, 253:121–134, September 2010.
24. J. Holt. *A Pragmatic Guide to Business Process Modelling (2nd Ed)*. British Informatics Society Ltd, 2009.
25. S. Jablonski, B. Volz, and S. Dornstauder. A meta modeling framework for domain specific process management. In *COMPSAC'08*, pages 1011 –1016. IEEE Computer Society, 2008.
26. M. Jarke, R. Gallersdörfer, M. A. Jeusfeld, and M. Staudt. ConceptBase - a deductive object base for meta data management. *J. Intell. Inf. Syst.*, 4(2):167–192, 1995.

27. F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE'06*, pages 249–254. ACM, 2006.
28. S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE CS, 2008.
29. 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.
30. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT'08*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
31. 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. ACM, 2007.
32. P. Langer, K. Wieland, M. Wimmer, and J. Cabot. From UML profiles to EMF profiles and beyond. In *TOOLS'11*, volume 6705 of *LNCS*, pages 52–67. Springer, 2011.
33. L. Liu and M. A. Jeusfeld. Suitability of active rules for model transformation. In *CAiSE Forum*, volume 855 of *CEUR Workshop Proceedings*, pages 131–138, 2012.
34. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
35. J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Trans. Inf. Syst.*, 8(4):325–362, 1990.
36. J. Odell. Power types. *JOOP*, 7(2):8–12, 1994.
37. OMG. MOF 2.4.1. `http://www.omg.org/spec/MOF/`, 2011.
38. OMG. OCL 2.3.1. `http://www.omg.org/spec/OCL/`, 2012.
39. OMG. MDA home page. `http://www.omg.org/mda/`, 2013.
40. L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack. The Epsilon Generation Language. In *ECMDA-FA'08*, volume 5095 of *LNCS*, pages 1–16. Springer, 2008.
41. J. Sánchez Cuadrado. Towards a family of model transformation languages. In *ICMT'12*, volume 7307 of *LNCS*, pages 176–191. Springer, 2012. See also `http://sanchezcuadrado.es/projects/eclectic/`.
42. J. Sánchez Cuadrado, J. de Lara, and E. Guerra. Bottom-up meta-modelling: An interactive approach. In *MoDELS*, volume 7590 of *LNCS*, pages 3–19. Springer, 2012.
43. J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Generic model transformations: *write once, reuse everywhere*. In *ICMT*, volume 6707 of *LNCS*, pages 62–77. Springer, 2011.
44. J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Flexible model-to-model transformation templates: An application to ATL. *Journal of Object Technology*, 11(2):4: 1–28, 2012.
45. S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. *Software and System Modeling*, 11(1):111–125, 2012.
46. J. Steel, K. Duddy, and R. Drogemuller. A transformation workbench for building information models. In *ICMT*, volume 6707 of *LNCS*, pages 93–107. Springer, 2011.
47. J. Steel and J.-M. Jézéquel. On model typing. *Software and System Modeling*, 6(4):401–413, 2007.
48. B. Volz and S. Jablonski. Towards an open meta modeling environment. In *10th Workshop on Domain-Specific Modeling*, 2010.
49. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, D. Kolovos, R. Paige, M. Lauder, A. Schürr, and D. Wagelaar. Surveying rule inheritance in model-to-model transformation languages. *The Journal of Object Technology*, 11, 2012.
50. S. Zschaler, D. S. Kolovos, N. Drivalos, R. F. Paige, and A. Rashid. Domain-specific metamodelling languages for software language engineering. In *SLE'09*, volume 5969 of *LNCS*, pages 334–353. Springer, 2009.

Appendix A

This appendix provides a formal definition of potency, linguistic typing and ontological typing. For simplicity, we just consider clabjects and fields, which suffices to illustrate the main concepts of our framework.

Definition 1 (Model) *A model $M = \langle CL, fields, FI\rangle$ is made of:*

- *a set CL of clabjects,*
- *a set FI of fields,*
- *a total function $fields\colon CL \to 2^{FI}$ assigning to each clabject c a (perhaps empty) set $fields(c)$ of its owned fields, s.t. $\forall f \in FI\ \exists_1 c \in CL$ with $f \in fields(c)$ (every field belongs to exactly one clabject).*

Remark. We use 2^{FI} to denote the set of all subsets of FI (its powerset). We sometimes use $cl(M)$ to denote CL, the set of clabjects of M, and $fi(M)$ to denote FI, the set of fields included in clabjects of M.

We write $U = \{M_i = \langle CL_i, fields_i, FI_i\rangle\}_{i\in I}$ for the set of all possible models, which we sometimes call the *universal set of models*.

Next, we define the potency for models, clabjects and fields as a family of functions to the natural numbers, including zero. The potency of clabjects should not be bigger than the potency of their enclosing model, and the potency of fields should not be bigger than the potency of their owner clabject. Having fields with potency bigger than their owning clabject may be useful in models with inheritance [15], but we omit inheritance in this formalization for simplicity.

Definition 2 (Potency) *Given the universal set U, the potency $pot_U = \langle pot, \{pot^i\}_{i\in I}\rangle$ is defined as:*

- *a total function $pot\colon U \to \mathbb{N}_0$ assigning each model M_i a positive number, or zero;*
- *a family of total functions $\{pot^i\colon CL_i \cup FI_i \to \mathbb{N}_0\}_{i\in I}$ for each model $M_i \in U$, assigning each clabject and field a positive number or zero, s.t. $\forall c \in CL_i$, $pot^i(c) \leq pot(M_i)$, and $\forall f \in fields_i(c)$, $pot^i(f) \leq pot^i(c)$.*

Remark. Potency for models is named level in [8].

In order to define the linguistic typing, we assume a very simple linguistic meta-model $L_{MM} = \{MODEL, CLABJECT, FIELD\}$. Then, the linguistic type is a family of total functions with co-domain L_{MM}, as shown in the next definition.

Definition 3 (Linguistic typing) *Given the universal set U, the linguistic typing $ltype_U = \langle ltype, \{ltype^i\}_{i\in I}\rangle$ is defined as:*

- *a total function $ltype\colon U \to L_{MM}$, s.t. $\forall M_i \in U, ltype(M_i) = MODEL$;*
- *a family of total functions $\{ltype^i\colon CL_i \cup FI_i \to L_{MM}\}_{i\in I}$ for each model $M_i \in U$, s.t. $\forall c \in CL_i$ $ltype^i(c) = CLABJECT$, and $\forall f \in FI_i$ $ltype^i(f) = FIELD$.*

Remark. For the potency and the linguistic typing, we sometimes use *pot* and *ltype* (without the superindex) for clabjects and fields when no confusion arises.

Next, we define the ontological typing as a family of partial functions.

Definition 4 (Ontological typing) *Given the universal set $U = \{M_i = \langle CL_i, fields_i, FI_i\rangle\}_{i\in I}$, with $CL = \uplus_{i\in I} CL_i$, $FI = \uplus_{i\in I} FI_i$, the ontological typing $otype_U = \langle otype, \{otype^i\}_{i\in I}\rangle$ is made of:*

- *a partial function $otype\colon U \to U$ assigning each model M_i its ontological model type $otype(M_i)$;*
- *a family of partial functions $\{otype^i\colon CL \cup FI \to CL \cup FI\}_{i\in I}$,*

s.t. $otype_U$ fulfills the following well-formedness rules, $\forall M_i \in U$:

1. $otype(M_i) = M' \Rightarrow pot(M_i) + 1 = pot(M')$,

2. $\forall c \in CL_i$, $otype^i(c) = c' \Rightarrow$
   $c' \in CL$ *(clabjects are typed by clabjects)*,

3. $\forall f \in FI_i$, $otype^i(f) = f' \Rightarrow$
   $f' \in FI$ *(fields are typed by fields)*,

4. $\forall c \in CL_i$, $otype^i(c) = c' \Rightarrow$
   $[c' \in cl(otype(M_i)) \wedge pot(c) + 1 = pot(c') \wedge$
   $\forall f' \in fields(c')$ *s.t.* $pot(f') > 0$
   $\exists_1 f \in fields(c)$ *s.t.*
   $[otype^i(f) = f' \ \wedge \ pot(f) + 1 = pot(f')]\ ]$

5. $\forall f \in FI_i$, $otype^i(f) = f' \Rightarrow$
   $[\ \exists c, c'$ *s.t.* $otype(c) = c' \ \wedge$
   $f \in fields(c) \ \wedge \ f' \in fields(c')$
   $\wedge \ pot(f) + 1 = pot(f')]$

6. $pot(M_i) = 0 \Rightarrow \exists M' \in U$ *s.t.* $otype(M_i) = M'$

7. $\forall c \in CL_i\ [\ pot^i(c) = 0 \Rightarrow$
   $\exists c' \in cl(otype(M_i))$ *s.t.* $otype(c) = c'\ ]$

8. $\forall f \in FI_i\ [\ pot^i(f) = 0 \Rightarrow$
   $\exists f' \in fi(otype(M_i))$ *s.t.* $otype(f) = f'\ ]$

Remark. We use $\uplus$ for disjoint union.

Remark. The first condition states that the potency of M_i is one less than the potency of its ontological model type M'. The second and third conditions state that clabjects and fields should be typed by clabjects and fields, respectively. The second condition is subsumed by the fourth, but we explicitly state it for clarity. The fourth condition states that if a clabject c has c' as ontological type, then its potency is one less. As the codomain of the potency is $\mathbb{N}_0$, we have that elements with potency zero cannot be instantiated. The fourth condition also requires every field of c' with potency bigger than zero to be instantiated in c. The fifth condition is a compatibility condition specifying that if the ontological type of a field f is f', then the ontological type of the owner of f is the owner of f'. The last three conditions state that elements with potency zero should have an ontological type.

The previous definition of ontological typing reduces to the standard conformance relation in a two metalevels setting, if we restrict the potency function to the set $\{0, 1\} \subset \mathbb{N}_0$, where models with potency 1 are the meta-models and those with potency 0 the models. However, model self-typing (i.e., a model whose ontological type is itself, as suggested in the OMG's MDA [39]) is not allowed in this formalization due to the requirement of the potency decrease in the model instances. Instead, we require top-level models to lack an ontological type and have a linguistic type only. Hence, the linguistic metamodel plays the role of the MOF in this architecture.

Next, we use the previous definitions to characterize linguistic extensions as elements with no ontological type.

Definition 5 (Linguistic extension) *Given the universal set $U = \{M_i = \langle CL_i, fields_i, FI_i\rangle\}_{i\in I}$, and an ontological typing $otype_U = \langle otype, \{otype^i\}_{i\in I}\rangle$, the linguistic extensions LE_j of $M_j \in U$ are defined as the tuple $LE_j = \langle CL'_j \subseteq CL_j, fields_j|_{CL'_j}, FI'_j \subseteq FI_j\rangle$ s.t.:*

- *$\forall c' \in CL'_j$ $otype(c')$ is undefined,*
- *$\forall c \in CL_j \setminus CL'_j$ $otype(c)$ is defined,*
- *$\forall f' \in FI'_j$ $otype(f')$ is undefined,*
- *$\forall f \in FI_j \setminus FI'_j$ $otype(f)$ is defined.*

Remark. Function $otype^i$ partitions CL_i and FI_i into the set of ontologically typed clabjects and fields ($CL_i \setminus CL'_i$ and $FI_i \setminus FI'_i$) and the linguistic extensions (CL'_i and FI'_i). Note that LE_j may fail to be a model, according to Definition 1, because some field without ontological type (belonging to FI'_j) may be owned by a clabject with ontological typing (not belonging to CL'_j). In any case, every element $M_i \in U$ is a correct model according to our framework.

Due to the fifth condition in Definition 4, we have that if a clabject has no ontological type, then no owned

field can have an ontological type. Conversely, it is allowed to have fields with no ontological type, owned by clabjects with ontological type.

Appendix B

This appendix provides complete listings for the running example presented in Section 5.

```
// Loads the meta−model
load "DSPM"

Syntax DSPM_MM for ProcessModel [ ".pm_mm" ] {
    permitsLoad = true;
    backtracking = true;

    model template DSPM_MM_Syntax@1 for "ProcessModel"
      "process" ^Id "{"
          ( _:TaskTemplate | _:PerformerTemplate | _:ArtefactTemplate |
        _:SeqTemplate | _:JoinTemplate | _:ForkTemplate | ^LingElements)+
      "}" ;

    node template TaskTemplate@1 for Task
      "task" ^Id #name ^Supers
       ("inputs" (#ins)+ )?
       ("outputs" (#outs)+ )?
       "{"
       ^Fields
       ^Constraints
       "}"
       with ins redefinedBy inputArtefacts
            outs redefinedBy outputArtefacts ;

    node template PerformerTemplate@1 for Performer
       "performer" ^Id #name ^Supers
       "{"
       ^Fields
       ^Constraints
        "}" ;

    node template ArtefactTemplate@1 for Artefact
       "artefact" ^Id
       ;

    node template SeqTemplate@1 for Seq
        "seq" ^Id ":" #src "−>" #tar
    with src redefinedBy from
          tar redefinedBy to
          ;

    node template JoinTemplate@1 for Join
        "join" ^Id ":" "(" #src ("," #src)+ ")" "−>" #tar
    with src redefinedBy from
          tar redefinedBy to
          ;

    node template ForkTemplate@1 for Fork
        "fork" ^Id ":" #src "−>" "(" #tar ("," #tar)+ ")"
    with src redefinedBy from
          tar redefinedBy to
          ;
}

Syntax DSPM_Model_Syntax for ProcessModel [".pm"] {
    permitsLoad = true;
    backtracking = true;

   model template DeepProcess@2 for "ProcessModel"
     ^Typename ^Id "{"
         ( _:DeepPerformer | _:DeepTask | _:DeepArtefact |
           _:DeepSeq | _:DeepJoin | _:DeepFork | ^LingInsts)∗
     "}" ;

   node template DeepPerformer@2 for Performer
     "−" ^Typename ^Id
     ;

   node template DeepArtefact@2 for Artefact
     "o" ^Typename ^Id #description
     ;

   node template DeepTask@2 for Task
     "∗" ^Typename ("!")? ^Id ("by" #perfBy)?
        ("requiring" #ins)∗
        ("delivering" #outs)∗
     with "!" set initial = true
     ;

   node template DeepSeq@2 for Seq
      ^Typename ":" #src "−>" #tar;

   node template DeepJoin@2 for Join
      ^Typename ":" "(" #src ("," #src)+ ")" "−>" #tar;

   node template DeepFork@2 for Fork
      ^Typename ":" #src "−>" "(" #tar ("," #tar)+ ")";
}
```

Listing 20 Defining the concrete syntax for meta-level 1.

```
process Waterfall {

  task SETask "preparation" {
    actor : SoftwareEngineer[∗] { perfBy };
  }

  task Analysis "requirements and analysis" : SETask {
    analyst : $self.actor.forAll( a | a.isKindOf(Analyst) )$;
    outreqs : $self.output.forAll( a | a.isKindOf(RequirementsDoc) )$;
  }

  task Design "high and low level design" : SETask {
    designer : $self.actor.forAll( a | a.isKindOf(Designer))$;
  }

  artefact Script
  artefact Documentation
  artefact SourceCode

  task Coding "coding and unit testing" : SETask
  inputs Documentation
  outputs SourceCode {
    uses : ProgLanguage[∗];
    programmer : $self.actor.forAll( a | a.isKindOf(Programmer))$;
  }

  task Testing "test" {
    tester : $self.actor.forAll( a | a.isKindOf(Tester))$;
  }

  task UnitTesting "unit" : SETask, Testing { }
  task AcceptanceTesting "accept" : Testing {
    actor : User[∗] { perfBy };
  }

  performer SoftwareEngineer "se" {
     SSN : String{id};
     notes : String;
     expertise : ProgLanguage[∗];
  }

  performer Analyst "analyst" : SoftwareEngineer {}
  performer Designer "designer" : SoftwareEngineer {}
  performer Programmer "programmer" : SoftwareEngineer {}
  performer User "user" { }

  seq a2d: Analysis −> Design
  seq d2c: Design −> Coding
  seq inner: SETask −> SETask

  fork c2t: Coding −> (UnitTesting, AcceptanceTesting)

  Node ProgLanguage {
     name : String;
     version : String;
  }

  Node Contract {
     startDate : String;
     salary : double;
     iRPF : double;
     employee : SoftwareEngineer;
  }
}
```

Listing 21 Using the concrete syntax to define the meta-model for the DSML.

```
1  Syntax SE for SoftwareEngineering imports DeepDSPM [".se"] {
2
3    override node template DeepPerformer@2 for Performer
4      _:AnalystTemplate | _:DesignerTemplate |
5      _:ProgrammerTemplate | _:UserTemplate;
6
7    node template AnalystTemplate@1 for Analyst
8      "–" ^Id "the" "analyst" "knows" #expertise ("," #expertise)*
9
10   node template DesignerTemplate@1 for Designer
11     "–" ^Id "the" "designer" "likes" #expertise ("," #expertise)*
12
13   node template ProgrammerTemplate@1 for Programmer
14     "–" ^Id "the" "programmer" "masters" #expertise ("," #expertise)*
15
16   node template UserTemplate@1 for User
17     "–" "A" "user" ^Id
18 }
```

Listing 22 Extending the concrete syntax.

```
1  Waterfall Project1_DefaultSyntax {
2
3    // Staff
4    – Richard the programmer masters Lisp, C
5    – Steve the designer likes ObjectiveC
6    – Bill the analyst knows Basic
7    – A user Me
8
9    // Artefacts
10   o Documentation aDocument "of a really bad window system"
11   o SourceCode aProgram "written in Lisp"
12
13   // Tasks
14   * Analysis doAnalysis by Bill
15   * Design designProduct by Steve
16   * Coding codeEverything by Richard
17     requiring aDocument
18     delivering aProgram
19   * UnitTesting unit by Richard
20   * AcceptanceTesting accept by Me
21
22   // Flow
23   a2d: doAnalysis –> designProduct
24   d2c: designProduct –> codeEverything
25   c2t: codeEverything –> (accept, unit)
26   inner: unit –> unit
27
28   // Programming languages
29   ProgLanguage Lisp {
30       name = "Common Lisp";
31   }
32
33   ProgLanguage C {
34       name = "Ansi C";
35   }
36
37   ProgLanguage ObjectiveC {
38       name = "Objective–C";
39   }
40
41   ProgLanguage Basic {
42       name = "Basic";
43   }
44 }
```

Listing 23 Using the concrete syntax to create models.