A Graph Transformation-Based Semantics for Deep Metamodelling

Alessandro Rossini[1], Juan de Lara[2], Esther Guerra[2], Adrian Rutle[3], and Yngve Lamo[3]

[1] University of Bergen, Norway
rossini@ii.uib.no
[2] Universidad Autónoma de Madrid, Spain
{Juan.deLara,Esther.Guerra}@uam.es
[3] Bergen University College, Norway
{aru,yla}@hib.no

Abstract. Metamodelling is one of the pillars of model-driven engineering, used for language engineering and domain modelling. Even though metamodelling is traditionally based on a two-level approach, several researchers have pointed out limitations of this solution and proposed an alternative *deep* (also called *multi-level*) approach to obtain simpler system descriptions. However, deep metamodelling currently lacks a formalisation that can be used to explain fundamental concepts such as deep characterisation through potency and double linguistic/ontological typing. This paper provides different semantics for such fundamental concepts based on graph transformation and the Diagram Predicate Framework.

1 Introduction

Model-driven engineering (MDE) promotes the use of models as the primary assets in software development, where they are used to specify, simulate, generate and maintain software systems. Models can be specified using general-purpose languages like UML, but to fully unfold the potential of MDE, models are specified using domain-specific languages (DSLs) which are tailored to a specific domain of concern. One way to define DSLs in MDE is by specifying metamodels, which are models that describe the concepts and define the syntax of a DSL.

The OMG has proposed MOF as the standard language to specify metamodels, and some popular implementations exist, most notably the Eclipse Modeling Framework (EMF) [21]. In this approach, a system is specified using models at two metalevels: a metamodel defining allowed types and a model instantiating these types. However, this approach may have limitations [4,5,13], in particular when the metamodel includes the *type-object* pattern [4,5,13], which requires an explicit modelling of types and their instances at the same metalevel. In this case, *deep metamodelling* (also called *multi-level metamodelling*) using more than two metalevels yields simpler models [5].

Deep metamodelling was proposed in the seminal works of Atkinson and Kühne [4], and several researchers and tools have subsequently adopted this approach [1,2,16]. However, there is still a lack of formalisation of the main concepts of deep metamodelling like deep characterisation through potency and double linguistic/ontological typing.

A. Schürr, D. Varró, and G. Varró (Eds.): AGTIVE 2011, LNCS 7233, pp. 19–34, 2012.

Such a formalisation is needed in order to explain the main aspects of the approach, study the different semantic variation points and their consequences, as well as to classify the different semantics found in the tools implementing them [1,2,3,16,15].

In this paper, we present a formal approach to deep metamodelling based on the Diagram Predicate Framework (DPF) [19,20], a diagrammatic specification framework founded on category theory and graph transformation. DPF has been adopted up to now to formalise several concepts in MDE, such as (MOF-based) metamodelling, model transformation and model versioning. The proposed formalisation helps in reasoning about the different semantic variation points in the realisation of deep metamodelling as well as in classifying the existing tools according to these options.

Paper Organisation. Section 2 introduces deep metamodelling through an example in the domain of component-based web applications. Section 3 presents the basic concepts of DPF. Section 4 explains different concepts of deep metamodelling through its formalisation in DPF. Section 5 compares with related research, and Section 6 concludes.

2 Deep Metamodelling

This section introduces deep metamodelling through an example, illustrating the limitations of two metalevels when defining DSLs which incorporate the type-object pattern [4,5,13]. Moreover, it discusses some open questions that are tackled in this paper.

2.1 Overview of Deep Metamodelling

The MeTEOriC project aims at the model-driven engineering of web applications. Here we describe a small excerpt of one of the modelling problems encountered in this project. A full description of this case study is outside the scope of this paper, but is described at: `http://astreo.ii.uam.es/~jlara/metaDepth/Collab.html`.

In MeTEOriC, a DSL is adopted to define the mash-up of components (like Google Maps and Google Fusion Tables) to provide the functionality of a web application. A simplified version of this language can be defined using two metalevels (see Fig. 1(a)). The metamodel corresponds to the DSL for component-based web applications. In this metamodel, the metaclass Component defines component types having a type identifier, whereas the metaclass CInstance defines component instances having a variable name and a flag indicating whether it should be visualised. Moreover, the metaassociation type defines the typing of each component instance. The model at the adjacent metalevel below represents a component-based web application which shows the position of professors' offices on a map. In this model, the classes Map and Table are instances of the metaclass Component and represent component types, whereas the classes UAMCamp and UAMProfs are instances of the metaclass CInstance and represent component instances of Map and Table, respectively.

The type-object relation between component types and instances is represented explicitly in the metamodel by the metaassociation type between the metaclasses Component and CInstance. However, the type-object relation between allowed and actual data links is implicit since there is no explicit relation between the metaassociations datalink and dlinstance, and this may lead to several problems. Firstly, it is not possible

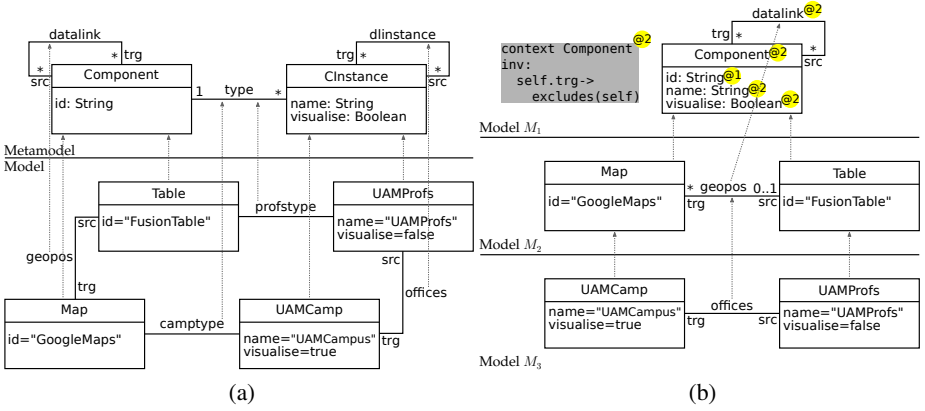


Fig. 1. A simple language for component systems in two and three metalevels

to define that the data link instance offices is typed by the data link type geopos, which could be particularly ambiguous if the model contained multiple data link types between the component types Map and Table. Moreover, it could be possible to specify a reflexive data link instance from the component instance UAMProfs to itself, which should not be allowed since the component type Table does not have any reflexive data link type. Although these errors could be detected by complementing the metamodel with attached OCL constraints, these constraints are not enough to guide the correct instantiation of each data link, in the same way as a built-in type system would do if the data link types and instances belonged to two different metalevels. In the complete definition of the DSL, the component types can define features, such as the zooming capabilities of the map component. Again, these features would be represented using the type-object pattern with metaclasses Feature (associated to Component) and FeatureInstance (associated to CInstance). These metaclasses need to be correctly instantiated and associated to the component instances, which leads to complex constraints and even more cluttered models. Hence, either one builds manually the needed machinery to emulate two metalevels within the same one, or this two-metalevel solution eventually becomes convoluted and hardly usable.

The DSL above can be defined in a simpler way using three metalevels (Fig. 1(b)) and deep characterisation, i.e., the ability to describe structure and express constraints for metalevels below the adjacent one. In this work, we adopt the deep characterisation approach described in [4], where each model element has a *potency*. In the original proposal of [4], the potency is a natural number which is attached to a model element to describe at how many subsequent metalevels this element can be instantiated. Moreover, the potency decreases in one unit at each instantiation at a deeper metalevel. When it reaches zero, a pure instance that cannot be instantiated further is obtained. In Section 4, we provide a more precise definition for potency.

In deep metamodelling, the elements in the top metalevel are pure types, the elements in the bottom metalevel are pure instances, and the elements at intermediate metalevels retain both a type and an instance facet. Because of that, they are all called *clabjects*,

which is the merge of the words class and object [5]. Moreover, since in deep metamodelling the number of metalevels may change depending on the requirements, we find it more convenient to number the metalevels from 1 onwards starting from the top-most. The model M_1 contains the definition of the DSL (Fig. 1(b)). In this model, clabject Component has potency 2, denoting that it can be instantiated at the two subsequent metalevels. Its attribute id has potency 1, denoting that it can be assigned a value when Component is instantiated at the adjacent metalevel below. Its other two attributes name and visualise have potency 2, denoting that they can be assigned a value only two metalevels below. The association datalink also has potency 2, denoting that it can be instantiated at the two subsequent metalevels. The attached OCL constraint in the model M_1 forbids to reflexively connect indirect instances of Component. This constraint has potency 2, denoting that it has to be evaluated in the model M_3 only. As elements in M_2 retain a type facet, we can add cardinality constraints to geopos, while this would need to be emulated in Fig. 1(a). The DSL in Fig. 1(b) is simpler than the one in Fig. 1(a), as it contains less model elements to define the same DSL.

The deep characterisation is very useful in the design of this DSL. For instance, in the model M_1, the designer can specify the attributes name and visualise which should be assigned a value in indirect instances of Component, i.e., UAMCamp and UAMProfs. Moreover, the model M_1 does not need to include a clabject CInstance or an association dlinstance since the clabjects UAMCamp and UAMProfs are instances of the clabjects Map and Table, respectively, which in turn are instances of the clabject Component.

The dashed grey arrows in Fig. 1(b) denote the *ontological typing* for the clabjects, as they represent instantiations within a domain; e.g., the clabjects Map and Table are ontologically typed by the clabject Component. In addition, deep metamodelling frameworks usually support an orthogonal *linguistic typing* [5,16] which refers to the metamodel of the metamodelling language used to specify the models; e.g., the clabjects Component, Map and UAMCamp are linguistically typed by Clabject, whereas the attributes id, name and visualise are linguistically typed by Attribute.

2.2 Some Open Questions in Deep Metamodelling

Deep metamodelling allows a more flexible approach to metamodelling by introducing richer modelling mechanisms. However, their semantics have to be precisely defined in order to obtain sound, robust models. Even if the literature (and this section) permits grasping an intuition of how these modelling mechanisms work, there are still open questions which require clarification.

Some works in the literature give different semantics to the potency of associations. In Fig. 1(b), the associations are instantiated like clabjects. In this case, the association datalink with potency 2 in the model M_1 is first instantiated as the association geopos with potency 1 in the model M_2, and then instantiated as the association offices with potency 0 in the model M_3; i.e., the instantiation of offices is mediated by geopos. In contrast, the attributes name and visualise with potency 2 in the model M_1 are assigned a value directly in the model M_3; i.e., the instantiation of name and visualise is not mediated. Some frameworks such as EMF [21] represent associations as Java references, so the associations could also be instantiated like attributes. In this case, the association datalink would not need to be instantiated in the model M_2 in order to be

able to instantiate it in the model M_3. This would have the effect that one could add an association between any two component instances in the model M_3, not necessarily between instances of Table and instances of Map.

Another ambiguity concerns constraints, since some works in the literature support potency on constraints [16] but others do not [3]. In Fig. 1(b), the attached OCL constraint in the model M_1 is evaluated in the model M_3 only. In other cases, it might be useful to have a potency which denotes that a constraint has to be evaluated at every metalevel. In addition, it is feasible to attach potencies to multiplicity constraints as well. In Fig. 1(b), all the multiplicity constraints are evaluated at the adjacent metalevel below. In other cases, it might be useful to attach a potency to multiplicity constraints. For instance, a potency 2 would have the effect that one could control the number of data link instances in the model M_3.

Finally, another research question concerns the relation between metamodelling stacks with and without deep characterisation. One could define constructions to *flatten* deep characterisation; e.g., given the three-metalevel stack of Fig. 1(b), one could obtain another three-metalevel stack without potencies but with some elements replicated along metalevels, making explicit the semantics of potency. This would allow the migration of deeply characterised systems into tools that do not support potency.

Altogether, we observe a lack of consensus and precise semantics for some of the aspects of deep metamodelling. The contribution of this work is the use of DPF to provide a neat semantics for different aspects of deep metamodelling: deep characterisation through potency and double linguistic/ontological typing. As a distinguishing note, we propose two possible semantics of potency for each model element, i.e., clabjects, attributes, associations and constraints. To the best of our knowledge, this is the first time that the two semantics have been recognised.

3 Diagram Predicate Framework

This section presents the basic concepts of DPF that are used in the formalisation of deep metamodelling. The interested reader can consult [9,8,10,18,19,17,20] for a more detailed presentation of the framework.

In DPF, a model is represented by a *specification* $\mathfrak{S}$. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}}: \Sigma)$ consists of an *underlying graph* S together with a set of *atomic constraints* $C^{\mathfrak{S}}$ which are specified by means of a *predicate signature* Σ. A predicate signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of *predicates* $\pi \in \Pi^{\Sigma}$, each having an arity (or shape graph) $\alpha^{\Sigma}(\pi)$. An atomic constraint (π, δ) consists of a predicate $\pi \in \Pi^{\Sigma}$ together with a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$ from the arity of the predicate to the underlying graph of a specification.

Fig. 2 shows a specification $\mathfrak{T}$ which is compliant with the requirements "a component must have exactly one identifier", "a component may be connected to other components" and "a component can not be connected to itself". In $\mathfrak{T}$, these requirements are enforced by the atomic constraints $(\texttt{[mult(1,1)]}, \delta_1 : (1 \xrightarrow{a} 2) \to (\mathsf{Component} \xrightarrow{\mathsf{id}} \mathsf{String}))$ and $(\texttt{[irreflexive]}, \delta_2 : (1 \xrightarrow{a} 1) \to (\mathsf{Component} \xrightarrow{\mathsf{datalink}} \mathsf{Component}))$.

Similar to E-graphs [11], attributes of nodes can be represented in DPF by edges from these nodes to nodes representing data types. For example, the attribute id:String of the

clabject Component in Fig. 1(b) is represented in DPF by an edge $\mathsf{Component} \xrightarrow{\mathsf{id}} \mathsf{String}$ (see Fig. 2).

The semantics of graph nodes and arrows has to be chosen in a suitable way for the corresponding modelling environment [20]. In object-oriented structural modelling, each object may be related to a set of other objects. Hence, it is appropriate to interpret nodes as sets and arrows $\mathsf{X} \xrightarrow{\mathsf{f}} \mathsf{Y}$ as multi-valued functions $f : X \to \wp(Y)$.

The semantics of predicates of the signature Σ (see Fig 2) is described using the mathematical language of set theory. In an implementation, the semantics of a predicate is typically given by the code of a corresponding validator such that the mathematical and the validator semantics should coincide. A semantic interpretation $[\![..]\!]^{\Sigma}$ of a signature Σ consists of a mapping that assigns to each predicate symbol $\pi \in \Pi^{\Sigma}$ a set $[\![\pi]\!]^{\Sigma}$ of graph homomorphisms $\iota : O \to \alpha^{\Sigma}(\pi)$, called valid instances of π, where O may vary over all graphs. $[\![\pi]\!]^{\Sigma}$ is assumed to be closed under isomorphisms.

The semantics of a specification is defined in the fibred way [8,10]; i.e., the semantics of a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ is given by the set of its instances (I, ι). To check that an atomic constraint is satisfied in a given instance of a specification $\mathfrak{S}$, it is enough to inspect only the part of $\mathfrak{S}$ which is affected by the atomic constraint. This kind of restriction to a subpart is obtained by the pullback construction [6]. An instance (I, ι) of a specification $\mathfrak{S}$ consists of a graph I and a graph homomorphism $\iota : I \to S$ such that for each atomic constraint $(\pi, \delta) \in C^{\mathfrak{S}}$ we have $\iota^* \in [\![\pi]\!]^{\Sigma}$, where the graph homomorphism $\iota^* : O^* \to \alpha^{\Sigma}(\pi)$ is given by the following pullback:

$$\begin{array}{ccc} \alpha^{\Sigma}(\pi) & \xrightarrow{\delta} & S \\ {\scriptstyle \iota^*}\uparrow & P.B. & \uparrow{\scriptstyle \iota} \\ O^* & \xrightarrow[\delta^*]{} & I \end{array}$$

In DPF, two kinds of conformance relations are distinguished: *typed by* and *conforms to*. A specification $\mathfrak{S}$ is typed by a graph T if there exists a graph homomorphism $\iota : S \to T$, called the *typing morphism*, between the underlying graph of the specification $\mathfrak{S}$ and T. A specification $\mathfrak{S}$ is said to conform to a specification $\mathfrak{T}$ if there exists a typing morphism $\iota : S \to T$ between the underlying graphs of $\mathfrak{S}$ and $\mathfrak{T}$ such that (S, ι) is a valid instance of $\mathfrak{T}$; i.e., such that ι satisfies the atomic constraints $C^{\mathfrak{T}}$.

Fig. 2 shows two specifications $\mathfrak{S}$ and $\mathfrak{S}'$, both typed by $\mathfrak{T}$. However, only $\mathfrak{S}$ conforms to $\mathfrak{T}$, since $\mathfrak{S}'$ violates the atomic constraints $C^{\mathfrak{T}}$. This is because the mis-

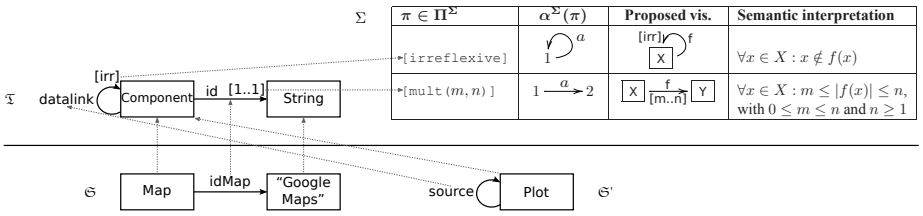


Fig. 2. A signature Σ and specifications $\mathfrak{T}$, $\mathfrak{S}$ and $\mathfrak{S}'$, where only $\mathfrak{S}$ conforms to $\mathfrak{T}$

sing id-typed edge violates the multiplicity constraint $([\texttt{mult}(1,1)], \delta_1)$, while the edge source violates the irreflexivity constraint $([\texttt{irreflexive}], \delta_2)$.

4 Formalisation of Deep Metamodelling

This section formalises different concepts of deep metamodelling through DPF. Firstly, we introduce different interpretations of potency. Secondly, we define the syntax of potency in terms of DPF. Thirdly, we define models in a deep stack together with double linguistic/ontological typing in terms of DPF. Finally, we present an operational semantics of potency in terms of constraint-aware graph transformation.

4.1 Multi- and Single-Potency

As discussed in Section 2.2, different interpretations of potency are possible. In this paper, two kinds of potency are distinguished, namely *multi-* and *single*-potency, denoted by the superscripts ▲p and △p, respectively.

A multi-potency ▲p on a clabject/reference at metalevel i denotes that this clabject/reference can be instantiated *at all metalevels from* $i+1$ *to* $i+p$ (see Fig. 3). A potency ▲p on an atomic constraint at metalevel i denotes that this constraint is evaluated at all metalevels from $i+1$ to $i+p$. Note that attributes only retain either type or instance facet but not both, therefore the multi-potency on attributes can not be considered. This "multi-" semantics is the usual semantics of potency on clabjects found in the literature.

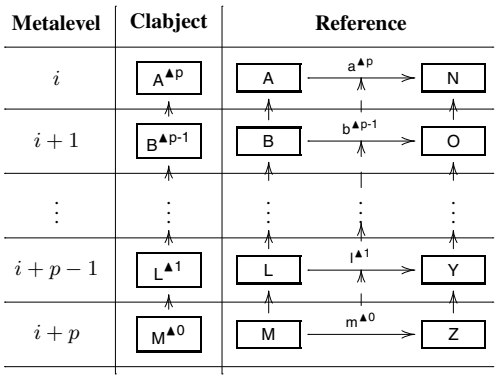


Fig. 3. Intuition on the multi-semantics of potency

In contrast, a single-potency △p on a clabject/reference at metalevel i denotes that this clabject/reference can be instantiated *at metalevel* $i+p$ *only*, but not at the intermediate metalevels (see Fig. 4). A potency △p on an attribute at metalevel i denotes that this attribute can be instantiated (i.e., can be assigned a value) at metalevel $i+p$ only. A potency △p on an atomic constraint at metalevel i denotes that this atomic constraint is evaluated at metalevel $i+p$ only.

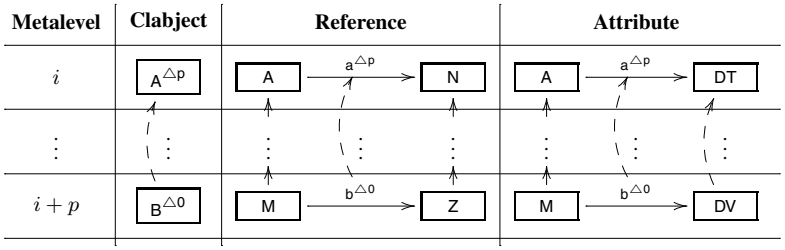


Fig. 4. Intuition on the single- semantics of potency

4.2 Syntax of Potency

The syntax of multi- and single-potencies can be represented in DPF by a *tag signature*, which has the same components of a predicate signature but a different semantic counterpart (see Section 4.3 and 4.4). A tag signature $\Psi = (\Theta^\Psi, \alpha^\Psi)$ consists of a collection of *tags* $\theta \in \Theta^\Psi$, each having an arity $\alpha^\Psi(\theta)$ and a proposed visualisation. Table 1 shows the tag signature Ψ for specifying potencies.

Table 1. The tag signature Ψ for specifying potencies

$\theta \in \Theta^\Psi$	$\alpha^\Psi(\theta)$	Proposed visual.	$\theta \in \Theta^\Psi$	$\alpha^\Psi(\theta)$	Proposed visual.
$\texttt{<multi}(p)\texttt{>}^1$	1	X▲p	$\texttt{<single}(p)\texttt{>}^1$	1	X△p
$\texttt{<multi}(p)\texttt{>}^2$	$1 \xrightarrow{a} 2$	X —f▲p→ Y	$\texttt{<single}(p)\texttt{>}^2$	$1 \xrightarrow{a} 2$	X —f△p→ Y
$\texttt{<multi}(p)\texttt{>}^\pi$	$1 \xrightarrow{a} 2$	X —f, π▲p→ Y	$\texttt{<single}(p)\texttt{>}^\pi$	$1 \xrightarrow{a} 2$	X —f, π△p→ Y

The tags $\theta \in \Theta^\Psi$ are divided into two families $\texttt{<multi}(p)\texttt{>}$ and $\texttt{<single}(p)\texttt{>}$ for multi- and single-potency, respectively. They are parametrised by the (non-negative) integer p, which represents the potency value that is attached to an element. More specifically, the tags $\texttt{<multi}(p)\texttt{>}^1$ and $\texttt{<single}(p)\texttt{>}^1$ are used for attaching potencies to clabjects, $\texttt{<multi}(p)\texttt{>}^2$ and $\texttt{<single}(p)\texttt{>}^2$ for references and attributes, and $\texttt{<multi}(p)\texttt{>}^\pi$ and $\texttt{<single}(p)\texttt{>}^\pi$ for atomic constraints (with compatible arity).

Given the tag signature Ψ, a potency (θ, γ) consists of a tag $\theta \in \Theta^\Psi$ and a graph homomorphism $\gamma : \alpha^\Psi(\theta) \to S_i$. Note that potencies can only be attached to clabjects, references, attributes and atomic constraints. This restriction can be defined by adopting typed tag signatures in which each tag is typed linguistically by a specification. This detail is omitted in this paper for brevity.

In the following, we adopt potencies to define models in a deep stack.

4.3 Double Linguistic/Ontological Typing

A model at metalevel i in a deep stack can be represented in DPF by a *(deep) specification* $\mathfrak{S}_i$. A specification $(\mathfrak{S}_i, \lambda_i, \omega_i) = (S_i, C_i \colon \Sigma, P_i \colon \Psi, \lambda_i, \omega_i)$ consists of an

underlying graph S_i, a set of atomic constraints C_i specified by means of a predicate signature Σ and a set of potencies P_i specified by means of a tag signature Ψ (see Fig. 5). Moreover, $\mathfrak{S}_i$ conforms linguistically to the specification $\mathfrak{LM}$; i.e., there exists a linguistic typing morphism $\lambda_i : S_i \to LM$ such that (S_i, λ_i) is a valid instance of $\mathfrak{LM}$. The specification $\mathfrak{LM}$ corresponds to the metamodelling language used to specify all specifications in a deep stack. Furthermore, $\mathfrak{S}_i$ conforms ontologically to the specification $\mathfrak{S}_{i-1}$; i.e., there exists an ontological typing morphism $\omega_i : S_i \to S_{i-1}$ such that (S_i, ω_i) is a valid instance of $\mathfrak{S}_{i-1}$ and the ontological typing is compatible with the linguistic typing, i.e., $\omega_i; \lambda_{i-1} = \lambda_i$.

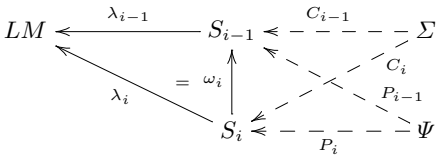


Fig. 5. Double linguistic/ontological typing

A specification $\mathfrak{S}_1$ at the top metalevel 1 of a deep stack is a special case as its elements are pure types (see Section 2). As such, $\mathfrak{S}_1$ conforms only linguistically to the specification $\mathfrak{LM}$; i.e., there is no specification $\mathfrak{S}_0$, hence, there is no ontological typing morphism $\omega_1 : S_1 \to S_0$.

Example 1 (Deep Stack). Building on the example in Section 2, Fig. 6(a) shows the specification $\mathfrak{LM}$ while Figs. 6(b), (c) and (d) show the specifications $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ of a deep stack corresponding to a simplified version of the one in Fig. 1(b). The figure also shows the ontological typings as dashed, grey arrows.

In $\mathfrak{S}_1$ the potency ▲2 on Component and datalink is specified by $(\texttt{<multi(2)>}^1, \gamma_1 : 1 \to \mathsf{Component})$ and $(\texttt{<multi(2)>}^2, \gamma_2 : (1 \xrightarrow{a} 2) \to (\mathsf{Component} \xrightarrow{\mathsf{datalink}} \mathsf{Component}))$, respectively. Similarly, the potencies △1 on id and $(\texttt{[mult(1,1)]}, \delta_1)$ are specified by $(\texttt{<single(1)>}^2, \gamma_3)$ and $(\texttt{<single(1)>}^2, \gamma_4; \delta_1)$, respectively.

The specifications $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ conform linguistically to $\mathfrak{LM}$; i.e., there exist linguistic typing morphisms $\lambda_1 : S_1 \to LM$, $\lambda_2 : S_2 \to LM$ and $\lambda_3 : S_3 \to LM$ such that (S_1, λ_1), (S_2, λ_2) and (S_3, λ_3) are valid instances of $\mathfrak{LM}$. For instance, λ_2 is defined as follows:

$\lambda_2(\mathsf{Map}) = \lambda_2(\mathsf{Table}) = \mathsf{Clabject}$
$\lambda_2(\mathsf{geopos}) = \mathsf{Reference}$
$\lambda_2(\mathsf{idMap}) = \lambda_2(\mathsf{idTable}) = \mathsf{Attribute}$
$\lambda_2(\text{“GoogleMaps”}) = \lambda_2(\text{“FusionTable”}) = \mathsf{DataType}$

Moreover, $\mathfrak{S}_2$ conforms ontologically to $\mathfrak{S}_1$; i.e., there exists an ontological typing morphism $\omega_2 : S_2 \to S_1$ such that (S_2, ω_2) is a valid instance of $\mathfrak{S}_1$:

$\omega_2(\mathsf{Map}) = \omega_2(\mathsf{Table}) = \mathsf{Component}$
$\omega_2(\mathsf{geopos}) = \mathsf{datalink}$
$\omega_2(\mathsf{idMap}) = \omega_2(\mathsf{idTable}) = \mathsf{id}$
$\omega_2(\text{“GoogleMaps”}) = \omega_2(\text{“FusionTable”}) = \mathsf{String}$

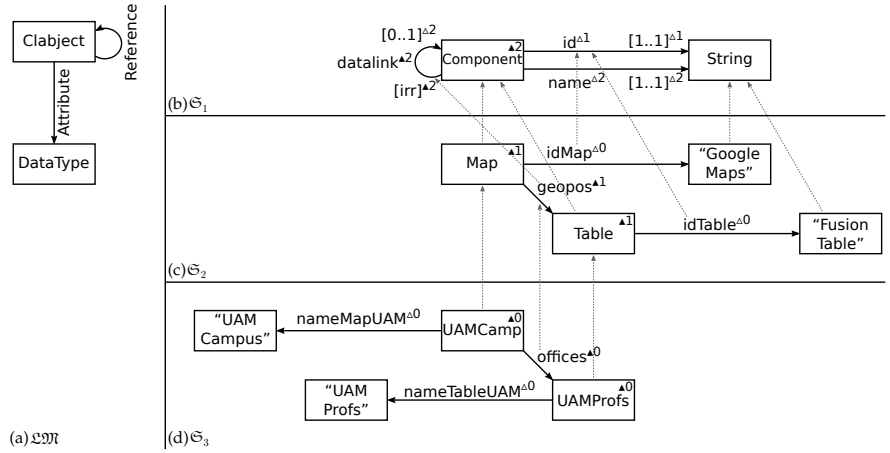


Fig. 6. The specifications $\mathfrak{LM}$, $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$

Finally, $\mathfrak{S}_3$ should conform ontologically to $\mathfrak{S}_2$, but this is not the case as the ontological typing morphism ω_3 is undefined for some elements of $\mathfrak{S}_3$:

$\omega_3(\mathsf{UAMCamp}) = \mathsf{Map}$
$\omega_3(\mathsf{UAMProfs}) = \mathsf{Table}$
$\omega_3(\mathsf{offices}) = \mathsf{geopos}$
$\omega_3(\mathsf{nameMapUAM}) = \omega_3(\mathsf{nameTableUAM}) = \omega_3(\text{“UAMCampus”}) = \omega_3(\text{“UAMProfs”}) = \emptyset$

In the following, we adopt constraint-aware graph transformation to define an operational semantics of single-potency and obtain a specification $\mathfrak{S}_3$ which conforms ontologically to $\mathfrak{S}_2$.

4.4 Semantics of Potency through Graph Transformation

Recall that a single-potency $\triangle\mathsf{p}$ on a type at metalevel i denotes that this type can be instantiated *at metalevel* $i + p$ *only*. Hence, there are always p metalevels between an instance with potency 0 and its type. However, in strict metamodelling, an instance with potency 0 at metalevel $i + p$ should be ontologically typed by a type with potency 1 at metalevel $i + p - 1$. To address this problem we define a semantics of single-potency which transforms a deep stack into a flattened stack without deep characterisation, in which an instance with potency 0 at metalevel $i+p$ has its type at metalevel $i+p-1$. This transformation adds to each metalevel $i + 1$ a replica of a type with potency decreased to $p-1$ and then deletes from metalevel i the original type, until $p = 1$. We specify this transformation with constraint-aware *transformation rules* [11].

A transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xrightarrow{r} \mathfrak{R}$ consists of three specifications $\mathfrak{L}$, $\mathfrak{K}$ and $\mathfrak{R}$. $\mathfrak{L}$ and $\mathfrak{R}$ are the *left-hand side* (LHS) and *right-hand side* (RHS) of the transformation rule, respectively, while $\mathfrak{K}$ is their interface. $\mathfrak{L} \setminus \mathfrak{K}$ describes the part of a

specification which is to be deleted, $\mathfrak{R} \setminus \mathfrak{K}$ describes the part to be added and $\mathfrak{K}$ describes the part to be preserved by the rule. Roughly speaking, an *application of transformation rule* means finding a match of the left-hand side $\mathfrak{L}$ in a source specification $\mathfrak{S}$ and replacing $\mathfrak{L}$ by $\mathfrak{R}$, leading to a target specification $\mathfrak{S}'$.

Since the transformation a specification undergoes is dependent of the potencies in the specification at the metalevel above, the transformation rules take as input and output *coupled specifications*. A coupled specification $\mathfrak{CS}_i = ((\mathfrak{S}_i, \lambda_i, \omega_i), (\mathfrak{S}_{i+1}, \lambda_{i+1}))$ consists of a specification $(\mathfrak{S}_i, \lambda_i, \omega_i)$ coupled with the specification $(\mathfrak{S}_{i+1}, \lambda_{i+1})$. This means that the application of transformation rules modifies specifications at two adjacent metalevels i and $i+1$.

Table 2 shows some of the transformation rules which define the operational semantics of single-potency. In general, all these rules follow a general pattern which adds to metalevel $i+1$ a replica of an element with single-potency decreased to $p-1$ and then deletes from metalevel i the original element; i.e.:

- Rules tc_0 and tdt_0: add to metalevel $i+1$ a replica of a clabject/data type.
- Rules tr_1 and ta_1: add to metalevel $i+1$ a replica of a reference/attribute.
- Rules $tacr_2$ and $taca_2$: add to metalevel $i+1$ a replica of an atomic constraint.
- Rules $tacr_3$ and $taca_3$: delete from metalevel i the original atomic constraint.
- Rules tr_4 and ta_4: delete from metalevel i the original reference/attribute.
- Rule tc_5: deletes from metalevel i the original clabject.

Note that the rules tc_0, tc_5, $taca_2$ and $taca_3$ are analogous to the rules tr_1, tr_4, $tacr_2$ and $tacr_3$, respectively, and are omitted from Table 2 for brevity.

The transformation uses negative application conditions (NACs) and layers [11] to control rule application. Since non-deleting rules can be applied multiple times via the same match, each rule has a NAC equal to its RHS. Moreover, since the rules are to be applied only if the matched potency is greater than 1, rules have another NAC demanding $p \leq 1$. The subscripts from 0 to 4 denote the layer to which a rule belongs, so that rules of layer 0 are applied as long as possible before rules of layer 1, etc.

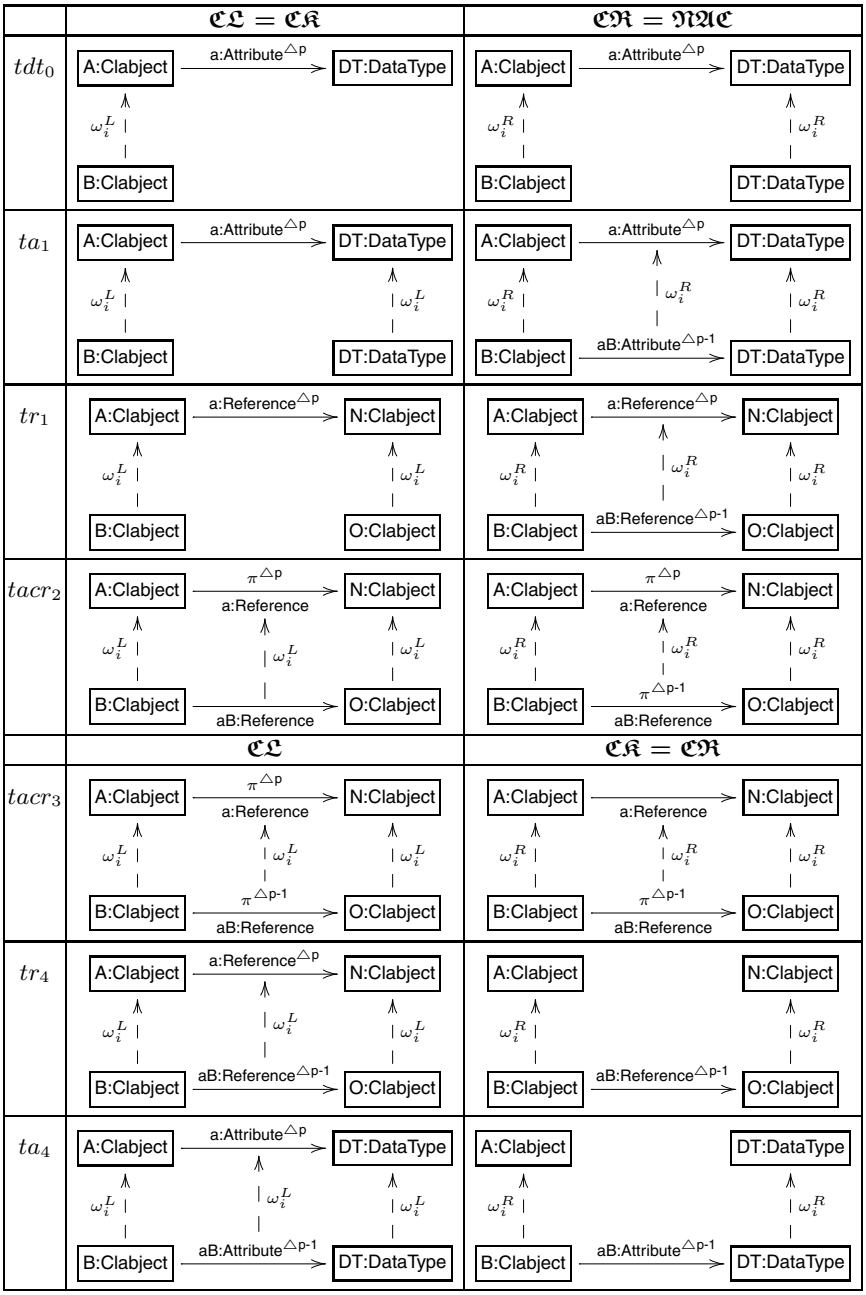
According to this layering, the transformation adds a replica of a reference only *after* it adds a replica of a clabject and *before* it deletes the original clabject. This ensures that the rule which adds a replica of a reference matches both clabjects with multi-potency and their instances as well as clabjects with single-potency and their replicas. Moreover, this ensures that the replica of the reference has as source and target an instance of the considered clabject with multi-potency or a replica of the considered clabject with single-potency. The layering of rules for data types, attributes and atomic constraints follow the same rationale.

Note that the notation 

in the rules denotes that $\lambda_i(\mathsf{A}) = \mathsf{Clabject}$, $\lambda_i(\mathsf{a}) = \mathsf{Attribute}$ and $\lambda_i(\mathsf{DT}) = \mathsf{DataType}$.

Example 2 (Deep Stack and Application of Transformation Rules). Building on Example 1, Figs. 7(b), (c) and (d) show the specifications $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$ of the deep stack, after the application of the transformation rules. In particular, the added elements are shown in Fig. 7(c) in green colour while the deleted elements are shown in Fig. 7(b) in red colour.

Table 2. The transformation rules for flattening the semantics of single-potencies



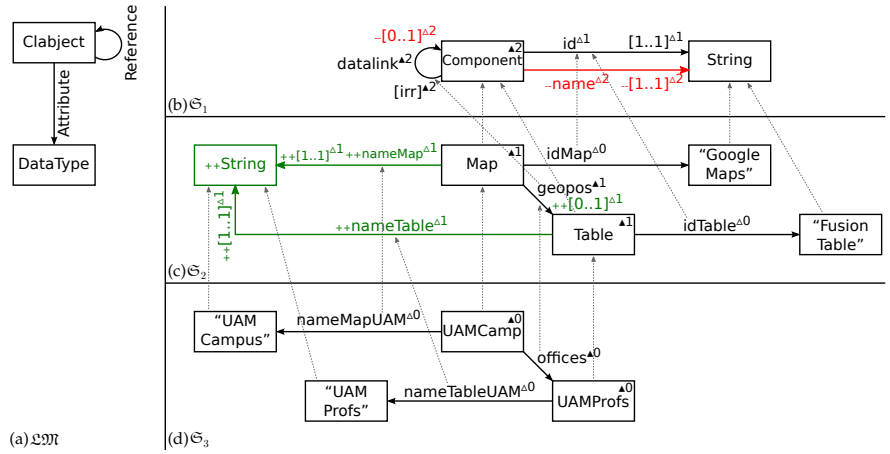


Fig. 7. The specifications $\mathfrak{S}_1$, $\mathfrak{S}_2$ and $\mathfrak{S}_3$, after applying the rules

Firstly, the application of tdt_0 and ta_1 adds to $\mathfrak{S}_2$ the node String and the edges nameMap and nameTable with potency $\triangle 1$. In this way, the ontological typing morphism ω_3 can be defined for all the elements of $\mathfrak{S}_3$, which makes $\mathfrak{S}_3$ conform ontologically to $\mathfrak{S}_2$:

$\omega_3(\mathsf{nameMapUAM}) = \mathsf{nameMap}$
$\omega_3(\mathsf{nameTableUAM}) = \mathsf{nameTable}$
$\omega_3(\text{“UAMCampus”}) = \omega_3(\text{“UAMProfs”}) = \mathsf{String}$

Secondly, the application of $tacr_2$ and $taca_2$ adds to $\mathfrak{S}_2$ the atomic constraints $(\texttt{[mult(0,1)]}, \delta_1)$, $(\texttt{[mult(1,1)]}, \delta_2)$ and $(\texttt{[mult(1,1)]}, \delta_3)$ with potency $\triangle 1$ on the edges geopos, nameMap and nameTable, respectively. In this way, these atomic constraints are evaluated at metalevel 0.

Thirdly, the application of $tacr_3$ and $taca_3$ deletes from $\mathfrak{S}_1$ the atomic constraints $(\texttt{[mult(0,1)]}, \delta_1)$ and $(\texttt{[mult(1,1)]}, \delta_3)$ on the edges datalink and name, respectively. In this way, these atomic constraints are not evaluated at metalevel 1.

Finally, the application of $taca_4$ deletes from $\mathfrak{S}_1$ the edge name. In this way, it is not possible to instantiate name at metalevel 1.

The presented flattening gives the semantics of potencies, so that an equivalent multi-level system is obtained, but without using deep characterisation. However, one can apply further flattenings using graph transformation, as shown in Fig. 8:

1. First, one can remove the double linguistic/ontological typing, keeping just one typing. This can be done by adding the linguistic metamodel on top of the ontological stack, and replicating such metamodel elements at all metalevels except 0 and 1.
2. Second, one can flatten a multi-level system into two metalevels. The first variant of this flattening is to merge all models of the ontological stack into a single model, and then consider this merged model as an instance of the linguistic metamodel.

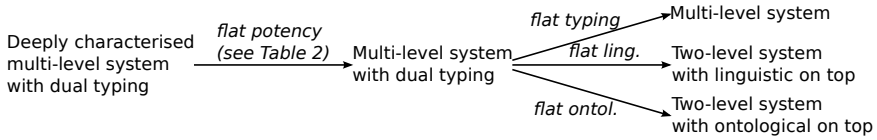


Fig. 8. Flattenings for multi-level systems

The second variant is to merge all models of the ontological stack, except the top-most one, into a single model, and then consider this merged model as an instance of the top-most model. In this variant, given a system like the one in Fig. 1(b), one would obtain a two-level system like the one in Fig. 1(a).

5 Related Work

A first strand of research focuses on multi-level metamodelling. In [12], MOF is extended with multiple metalevels to enable XML-based code generation. Nivel [2] is a double metamodelling framework based on the weighted constraint rule language (WCRL). XMF [7] is a language-driven development framework allowing an arbitrary number of metalevels. Another form of multi-level metamodelling can be achieved through powertypes [13], since instances of powertypes are also subtypes of another type and hence retain both a type and an instance facet. Multi-level metamodelling can also be emulated through stereotypes, although this is not a general modelling technique since it relies on UML to emulate the extension of its metamodel. The interested reader can consult [5] for a thorough comparison of potencies, powertypes and stereotypes.

A second strand of research focuses on deep characterisation. DeepJava [15], METADEPTH [16], and the works of Gutheil [14], Atkinson [3], and Aschauer [1] all support deep characterisation through potency. While these works agree on that clabjects are instantiated using the multi-potency semantics, they differ in other design decisions. Firstly, some works are ambiguous about the instantiation semantics for associations. In [15], the associations can be represented as Java references; hence we interpret that they are instantiated using the single-potency semantics. In [14], the connectors are explicitly represented as clabjects but their instantiation semantics is not discussed; hence we interpret that they are instantiated using the multi-potency semantics. Secondly, not all works adhere to *strict* ontological metamodelling. In [1], the ontological type of an association does not need to be in the adjacent metalevel above, but several metalevels above. Note that our single-potency semantics makes it possible to retain strict metamodelling for associations through a flattening construction that replicates these associations. Finally, some works differ in how they tackle potency on constraints and methods. Potency on constraints is not explicitly shown in [3] and not considered in [1], whereas potency on methods is only supported by DeepJava and METADEPTH.

Table 3 shows a summary of the particular semantics for deep characterisation implemented by the above mentioned works and compares it with the semantics supported by our formalisation. It is worth noting that no tool recognises the fact that multiplicity constraints are constraints as well and hence can have a potency.

Table 3. Comparison of different deep characterisation semantics

	Clabjects	**Associations**	**Strictness**	**Constraints**	**Mult. constraints**
DeepJava [15]	▲	△	yes	△	N.A.
Atkinson et al. [3]	▲	▲	yes	▲	▲1
Aschauer et al. [1]	▲	▲	no	N.A.	▲1
METADEPTH [16]	△, ▲	△, ▲	yes	△	▲1
DPF formalisation	△, ▲	△, ▲	yes	△, ▲	△, ▲

6 Conclusion and Future Work

In this paper, we presented a formalisation of concepts of deep metamodelling using DPF and graph transformation. In particular, we provided a precise definition of the double linguistic/ontological typing and two different semantics for potency on different model elements, as well as an operational semantics of potency using a flattening based on graph transformation.

We believe that distinction of two possible semantics for potency is important to achieve more flexible tools, allow their comparison and their interoperability. For instance, in our case study (see Fig. 1), the multi-potency on the association datalink has the effect that one can only add associations between instances of the component Table and instances of the component Map in the model M_3. On the contrary, a single-potency on the association datalink would have the effect that one could add associations between any two component instances in the model M_3, not necessarily between instances of Table and instances of Map. We found both semantics especially useful in this case study.

To the best of our knowledge, this work is the first attempt to clarify and formalise some aspects of the semantics of deep metamodelling. In particular, this work explains different semantic variation points available for deep metamodelling, points out new possible semantics, currently unexplored in practice, as well as classifies the existing tools according to these options.

In the future, we plan to investigate the effects of combining different potency values to interdependent model elements. This includes the investigation of the effects of overriding the potency of a clabject using inheritance, as this may lead to contradictory combinations of potencies. We also plan to formalise the linguistic extensions proposed in [16]. Finally, we will incorporate the lessons learnt from this formalisation into the METADEPTH tool, in particular the possibility of assigning potency to multiplicity constraints.

Acknowledgements. Work partially funded by the Spanish Ministry of Science (project TIN2008-02081), and the R&D programme of the Madrid Region (project S2009 /TIC-1650).

References

1. Aschauer, T., Dauenhauer, G., Pree, W.: Multi-level modeling for industrial automation systems. In: EUROMICRO 2009, pp. 490–496. IEEE Computer Society (2009)

2. Asikainen, T., Männistö, T.: Nivel: A metamodelling language with a formal semantics. Software and Systems Modeling 8(4), 521–549 (2009)
3. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. IEEE Transactions on Software Engineering 35(6), 742–755 (2009)
4. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation 12(4), 290–321 (2002)
5. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software and Systems Modeling 7(3), 345–359 (2008)
6. Barr, M., Wells, C.: Category Theory for Computing Science, 2nd edn. Prentice-Hall (1995)
7. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling: A Foundation for Language Driven Development, 2nd edn., Ceteva (2008)
8. Diskin, Z.: Mathematics of Generic Specifications for Model Management I and II. In: Encyclopedia of Database Technologies and Applications, pp. 351–366. Information Science Reference (2005)
9. Diskin, Z., Kadish, B., Piessens, F., Johnson, M.: Universal Arrow Foundations for Visual Modeling. In: Anderson, M., Cheng, P., Haarslev, V. (eds.) Diagrams 2000. LNCS (LNAI), vol. 1889, pp. 345–360. Springer, Heidelberg (2000)
10. Diskin, Z., Wolter, U.: A diagrammatic logic for object-oriented visual modeling. In: Proc. of the 2nd Workshop on Applied and Computational Category Theory (ACCAT 2007). ENTCS, vol. 203(6), pp. 19–41. Elsevier (2008)
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (March 2006)
12. Gitzel, R., Ott, I., Schader, M.: Ontological extension to the MOF metamodel as a basis for code generation. Computer Journal 50(1), 93–115 (2007)
13. Gonzalez-Perez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework. Software and Systems Modeling 5(1), 72–90 (2006)
14. Gutheil, M., Kennel, B., Atkinson, C.: A Systematic Approach to Connectors in a Multi-level Modeling Environment. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 843–857. Springer, Heidelberg (2008)
15. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style? Multi-level programming with DeepJava. In: OOPSLA 2007: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 229–244. ACM (2007)
16. de Lara, J., Guerra, E.: Deep Meta-modelling with METADEPTH. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 1–20. Springer, Heidelberg (2010)
17. Rossini, A.: Diagram Predicate Framework Meets Model Versioning and Deep Metamodelling. Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2011)
18. Rossini, A., Rutle, A., Lamo, Y., Wolter, U.: A formalisation of the copy-modify-merge approach to version control in MDE. Journal of Logic and Algebraic Programming 79(7), 636–658 (2010)
19. Rutle, A.: Diagram Predicate Framework: A Formal Approach to MDE. Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2010)
20. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A formal approach to the specification and transformation of constraints in MDE. Journal of Logic and Algebraic Programming 81(4), 422–457 (2012)
21. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0., 2nd edn. Addison-Wesley Professional (2008)