When and How to Use Multi-Level Modelling

JUAN DE LARA, Universidad Autónoma de Madrid (Spain)
ESTHER GUERRA, Universidad Autónoma de Madrid (Spain)
JESÚS SÁNCHEZ CUADRADO, Universidad Autónoma de Madrid (Spain)

Model-Driven Engineering (MDE) promotes models as the primary artefacts in the software development process, from which code for the final application is derived. Standard approaches to MDE (like those based on MOF or EMF) advocate a two-level meta-modelling setting where Domain-Specific Modelling Languages (DSMLs) are defined through a meta-model, which is instantiated to build models at the meta-level below.

Multi-level modelling – also called *deep meta-modelling* – extends the standard approach to meta-modelling by enabling modelling at an arbitrary number of meta-levels, not necessarily two. Proposers of multi-level modelling claim that this leads to simpler model descriptions in some situations, although its applicability has been scarcely evaluated. Thus, practitioners may find it difficult to discern when to use it and how to implement multi-level solutions in practice.

In this paper, we discuss the situations where the use of multi-level modelling is beneficial, and identify recurring patterns and idioms. Moreover, in order to assess how often the identified patterns arise in practice, we have analysed a wide range of existing two-level DSMLs from different sources and domains, to detect when their elements could be rearranged in more than two meta-levels. The results show that this scenario is not uncommon, while in some application domains (like software architecture and enterprise/process modelling) is pervasive, with a high average number of pattern occurrences per meta-model.

Categories and Subject Descriptors: D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods; D.3.2 [**Language Classifications**]: Design languages; Extensible languages; D.3.3 [**Language Constructs and Features**]: Patterns; Classes and objects

Additional Key Words and Phrases: Model-driven engineering, meta-modelling, domain-specific modelling languages, multi-level modelling, meta-modelling patterns

ACM Reference Format:
Juan de Lara, Esther Guerra and Jesús Sánchez Cuadrado, 2014. When and how to use multi-level modelling. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 46 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Models and modelling play a cornerstone role in the Model-Driven Engineering (MDE) development paradigm [Völter and Stahl 2006]. In contrast to code-centric development approaches, MDE advocates a pervasive use of models throughout the development cycle, and their automated processing for simulating, verifying and generating code for the final system.

This work is supported by the Spanish Ministry of Economy and Competitivity with project Go-Lite (TIN2011-24139) and the EU commission with project MONDO (FP7-ICT-2013-10, #611125).
Author's addresses: J. de Lara, Computer Science Department, Universidad Autónoma de Madrid (Spain); E. Guerra, Computer Science Department, Universidad Autónoma de Madrid (Spain); J. Sánchez Cuadrado, Computer Science Department, Universidad Autónoma de Madrid (Spain).

DOI:http://dx.doi.org/10.1145/0000000.0000000

In MDE, models frequently describe the systems to be built from the problem perspective, in contrast to algorithmic or solution-oriented descriptions. Therefore, as a central asset for the development process, MDE promotes the construction of Domain-Specific Modelling Languages (DSMLs) specially tailored and highly effective for particular domains [Völter 2013]. DSMLs provide powerful, expressive primitives of the domain, while hiding any accidental complexity. Hence, domain-specific models become simpler and easier to understand than a description based on algorithmic constructions. In general, techniques, patterns and tools to strip models of accidental details are of great importance in MDE, enabling models to be truly intensional to better reflect the intrinsic structure of systems.

Standard approaches to MDE rely on a two meta-level architecture[1] to define meta-models for DSMLs and instantiate those into models. This architecture, implemented in widely used frameworks like the Eclipse Modelling Framework (EMF) [Steinberg et al. 2008], forces the description of a domain (a meta-model) within one meta-level using the natively available meta-modelling facilities, like type definition, type inheritance, data types, definition of features and cardinalities. Such facilities are not available at the model level; hence, whenever they need to be used in models, they must be explicitly modelled at the meta-model level, resulting in unnecessary accidental complexity [de Lara et al. 2014a].

Several researchers have proposed *multi-level modelling* as a means to overcome this limitation by making it possible to define deep languages that span more than two meta-levels [Atkinson and Kühne 2003; González-Pérez and Henderson-Sellers 2006]. Thus, this approach recognises the fact that some model elements may have a dual type/instance facet, and hence it makes some meta-modelling facilities available at every meta-level. In some situations, this may result in simpler models as the engineer does not need to explicitly model and give semantics to those meta-modelling facilities or resort to artificial workarounds [Atkinson and Kühne 2008].

Altogether, multi-level modelling has become a promising technology for DSML engineering. Unfortunately, there are scarce applications of multi-level modelling in realistic scenarios [de Lara et al. 2014b], and few attempts to show the range of problems where multi-level modelling provides benefits compared to other meta-modelling approaches. Without such know-how, practitioners frequently find it difficult to discern when it is practical to use it, and how to implement multi-level solutions.

This paper aims at filling this gap by identifying a set of patterns and idioms where the use of multiple levels makes sense. For each pattern, we enumerate the advantages and disadvantages of different modelling alternatives considering both a multi-level setting and a two-level architecture, provide guidelines of the more appropriate solution depending on the modelling goal and scenario, and illustrate their usage in real scenarios. In order to assess the applicability of our patterns, we have analysed a large corpus of more than 400 existing meta-models and detected a number of them that could benefit from a multi-level architecture. The meta-models come from different sources (most notably, the ATL meta-model zoo [AtlanEcore 2014], the ReMoDD repository [ReMoDD 2014] and OMG specifications [OMG 2014]) and domains (like software architecture, process modelling, software design, web engineering, software modernization and health). Interestingly, for some domains like software architecture and enterprise/process modelling, some of our patterns are pervasive for most DSMLs in the domain, with a high number of pattern occurrences in each meta-model (especially in the enterprise/process modelling domain). Moreover, for some sources, like

[1]Although the OMG proposes a four meta-layer architecture, in practice, MDE practitioners use just two layers at a time: one to build meta-models for DSMLs and another one to instantiate those in the form of models.

OMG specifications, their frequency of occurrence is high (more than 35%). Therefore, we can conclude that multi-level modelling is applicable and potentially beneficial in real scenarios, and therefore a relevant technology for MDE in practice.

The rest of the paper is organized as follows. Firstly, we use an example to illustrate different modelling idioms based on a two-level architecture in Section 2, and using multi-level modelling in Section 3. Section 4 presents different meta-modelling patterns and alternative modelling solutions for them, including both multi-level and two-level, and discusses advantages and disadvantages. Then, Section 5 evaluates the extent to which these patterns occur in practice by analysing a wide corpus of existing meta-models. In view of the results of this analysis, Section 6 discusses the benefits of using a multi-level modelling solution instead of two-level solutions, giving some guidelines on how to refactor into multi-level and proposing requirements and challenges for multi-level modelling tools in order to address the different situations found. Finally, Section 7 reviews related research, and Section 8 draws some conclusions and lines for future work.

2. MOTIVATING EXAMPLE: DYNAMIC MODELLING OF OPTIMIZATION PROBLEMS

Assume we are interested in describing combinatorial optimization problems (e.g., determining the optimal way to deliver packages) and their solution techniques [Papadimitriou and Steiglitz 1998]. Our system is aimed at being extensible, and hence its description should enable a flexible, dynamic addition of new kinds of problems, like the Travelling Salesman Problem (TSP) or the Chinese Postman Problem (CPP) [Gutin et al. 2002]. Moreover, it would be desirable to classify problems according to their configuration features. For example, both TSP and CPP are problems on graphs, and so both need to be configured with the graph on which the problem should be solved.

We also want to be able to create new solution methods for a given problem dynamically, classified according to their working scheme, as some methods are direct (like those based on “brute force”, combinatorial solutions), while others are transformation-based, relying on transforming a particular instance of a problem into a different one [Papadimitriou and Steiglitz 1998]. For example, we may add a heuristic, direct solution of the TSP based on multi-agent systems [Wooldridge 2009] (e.g., the Ant Colony Optimization [Dorigo and Gambardella 1997]), or define a transformation-based method which translates CPP instances into TSP instances [Laporte 1997].

Finally, it should be possible to instantiate a problem kind (like solving the TSP on a graph of 24 cities) using an appropriate solution method (like a multi-agent system with 24 agents).

The following list summarizes the desired features of the system:

(1) Dynamic addition of problem types.
(2) Dynamic addition of direct and transformation-based solution methods.
(3) Description of features for problem types (e.g., number of graph nodes) and solution types (e.g., number of agents), factoring out commonalities.
(4) Instantiation of problem types and their solution methods.

We will use different meta-modelling styles to solve this problem. A meta-model MM is a model that describes the structure of a set S_{MM} of models considered valid. Each model in the set S_{MM} is said to be a valid instance, or conform to the meta-model MM. Meta-models contain the definition of classes, attributes and relations that can be used (instantiated) in instance models. Additionally, meta-models may include further integrity constraints that instance models need to satisfy. Different meta-modelling languages and technologies can be used for describing meta-models, like MOF (implemented in popular frameworks like EMF Ecore [Steinberg et al. 2008]), KM3 [Jouault and Bézivin 2006], or UML class diagrams. Frequently, integrity constraints are specified using the Object Constraint Language (OCL) [OMG 2012d].

In the following, when using standard two-level meta-modelling, we assume Ecore-based meta-models unless explicitly stated otherwise[2], and in some occasions, we report on unsupported features of Ecore that would be needed for a particular example.

2.1. First solution: Static types

A first attempt to modelling the example system is shown in Figure 1. The upper part (labelled (a)) shows a meta-model of the problem, while the lower part (label (b)) depicts an instance model. In this attempt, we have modelled problem and solution types as meta-classes. At first sight, this seems natural, and enables a hierarchical classification of problem types through inheritance, reflecting the fact that both the `ChinesePostman` and `TravellingSalesman` problem types are graph optimization problems (as they are subclasses of `GraphOptimProblem`) and share common properties like the number of nodes in the graph[3]. Similarly, we can define transformation-based and direct solution methods, like `MultiAgentSystem`, each defining their own configuration properties by means of attributes, like the number of agents used in the multi-agent system solution. This successfully addresses requirement 3 of our list. Below, the figure shows a model defining an instance of the TSP to be solved using the multi-agent system solution method instantiated with 24 agents. Thus, this solution addresses successfully requirement 4.

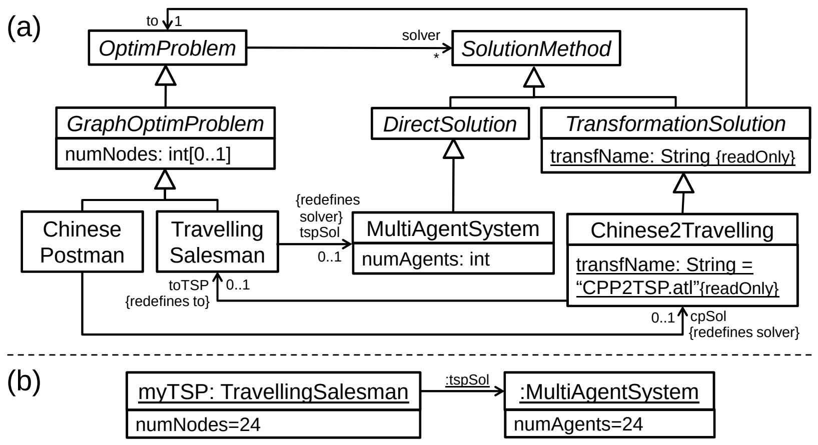


Fig. 1. Modelling optimization problems and solution types as meta-classes.

However, this modelling solution does not cover requirements 1 and 2 in our list. The reason is that, to add a new problem type or solution method, one needs to modify the meta-model, which is normally rigid and non-modifiable at run-time in meta-modelling environments. While we can "dynamically create" *objects* at the model level, this is not so for new *classes* at the meta-level, for which we most probably need a recompilation step for the meta-model, or resort to complex reflective mechanisms.

[2]The main implication is that Ecore does not provide associations, but references, and so cardinalities are only attached to the target reference ends. Or otherwise stated, the source cardinality of references can be assumed to be *. However, associations can be emulated with opposite references.

[3]We have simplified the problem, allowing the configuration of the number of nodes in the graph, but not the number of edges or their weight.

Another problem arises when trying to constrain the valid solution methods for specific problem types. For instance, to specify that `MultiAgentSystem` is a valid solution method for the `TravellingSalesman` problem, we have to *redefine* the `solver` reference defined from `OptimProblem` to `SolutionMethod`. This solution becomes particularly tricky if we want to define several solution methods for the same problem, like being able to use a `MultiAgentSystem` or a solution based on exhaustive combinatorial search to solve the `TravellingSalesman` problem. This is so because it requires the meta-modelling framework to support advanced overriding mechanisms that are not available in popular frameworks like EMF, but that to some extent exist in (Complete) MOF [OMG 2013i] and UML [OMG 2011g]. A similar problem arises when defining transformation-based solution methods, as references `solver` and `to` need to be redefined. Moreover, if we want to store the name of the file with the code implementing the transformation, we need a mechanism to provide a value for attributes defined in meta-classes (attribute `transfName` in `TransformationSolution`), and forbid their modification in instances. In the example, this is done using a static, read-only attribute. However, some popular meta-modelling frameworks like EMF lack this overriding mechanism.

2.2. Second solution: Explicit dynamic types

On reflection, we may realise that the system requirements demand meta-classes to have object-like features: dynamic creation of meta-classes for the definition of new types of problems and solutions, and valuation of references and attributes at the meta-model level. The previous solution emulates such object-specific features with type-specific mechanisms (subclassing and reference/attribute overriding). Hence, next we explore an alternative approach modelling those elements as objects.

Figure 2 shows a simplified version of the new solution using what we call *explicit dynamic types*, where we model problem types (like `TravellingSalesman`) and solution types (like `MultiAgentSystem`) as objects. In this way, requirements 1 and 2 are satisfied because we can create new types dynamically. However, we need to include in the meta-model elements to emulate such a *type* facet. In particular, in order to address requirement 3, we need to add a meta-class `Feature` to permit the definition of features for new problem and solution types, a relation `extends` to allow organizing problem and solution types along hierarchies, and an attribute `isAbstract` for problem and solution types (e.g., `GraphOptimProblem` is abstract). For clarity, the figure only shows these elements for problem types.

Once we model problem types as objects, we can no longer use the native instantiation mechanism of the meta-modelling framework to instantiate them, but instantiation needs to be emulated. For this purpose, the meta-model includes different meta-classes to model problem types (`OptimProblemType`) and their instances (`OptimProblem`), together with a reference `type` for the *instantiation* relation. In addition, since problem types may define features, their instances need to define `Slots` to assign each feature a value. Thus, although not shown in the figure, the meta-model needs to include OCL constraints checking that each problem instance has one slot for each feature in the problem type, and the value in the slot is correct according to the feature's type (*String*, *int* and so on). A similar problem arises for the modelling of solution method types (subclasses of `SolutionMethodType`) and their instances.

Altogether, while this solution addresses all system requirements, it adds much more complexity than the previous one. The reason is that we need to assign type-facet features to instances. Hence, we need to explicitly model and provide semantics to meta-modelling facilities (inheritance, instantiation, data types) that a meta-modelling framework natively provides. However, such facilities are only available at the meta-model level (where types reside) and not at the model level (where instances

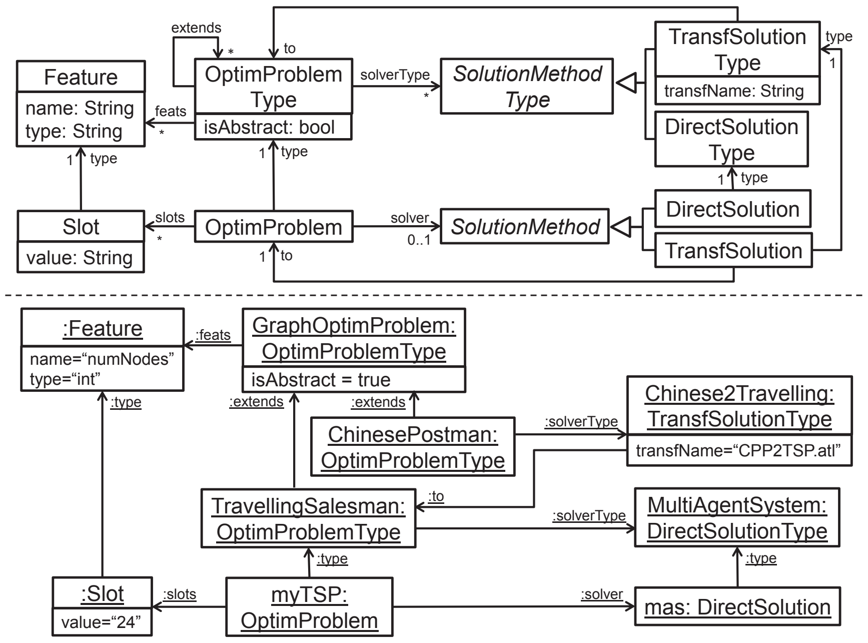


Fig. 2. Modelling optimization problems and solution types as objects.

arise). The next section introduces a third solution based on the uniform modelling of types and instances and the possibility of using more than two meta-levels to describe the problem.

3. MULTI-LEVEL MODELLING AND THE ORTHOGONAL CLASSIFICATION ARCHITECTURE

In order to alleviate the drawbacks of the solutions proposed in the previous section, we introduce two techniques that extend the standard two-level meta-modelling approach: potency-based multi-level modelling [Atkinson 1997; Atkinson and Kühne 2001] and the orthogonal classification architecture (OCA) [Atkinson and Kühne 2002; Atkinson et al. 2010].

3.1. Potency-based multi-level modelling

Potency-based multi-level modelling (also called deep meta-modelling[4]) was introduced by Atkinson and Kühne in [Atkinson 1997; Atkinson and Kühne 2001]. It enables the use of an arbitrary number of meta-levels when describing a problem. In this way, elements within a model have a dual facet: they are instances with respect to some element in the meta-model above, and they are types with respect to elements in the meta-level below. For this reason, they are often called *clabjects* (from *cla*ss+*object*) [Atkinson 1997]. This approach solves the duality of the two previous solutions, as clabjects have both type and instance characteristics at the same time.

Figure 3 shows the multi-level solution for our running example. At the upper meta-level, we define meta-classes representing types of problems, transformation-based solutions and direct solutions. These meta-classes are instantiated in the intermediate meta-level in order to dynamically create new problem types (like the

[4]For simplicity, we will use the term "multi-level modelling" to refer to potency-based multi-level modelling, when no confusion can arise.

`TravellingSalesman` problem) and solution method types (like `MultiAgentSystem`). These instances are indeed clabjects and thus can be instantiated. In this way, the bottom-most model enables the configuration of specific problems and solution methods, just like the model in Figure 1 (b).

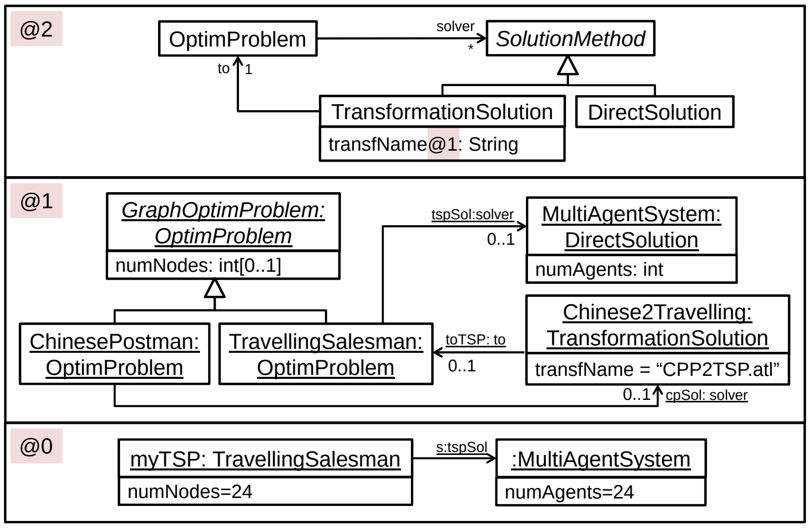


Fig. 3. Modelling optimization problems and solution types using deep meta-modelling.

In the figure, the clabjects in the middle meta-level have a dual type/instance facet: `TravellingSalesman` is an instance of `OptimProblem`, and it is the type of `myTSP`. This duality applies also to references. For example, `tspSol` is an instance of `solver`, and is also a type for the `s` link at the bottom model. Being an instance, `tspSol` should obey the cardinalities at the level above (`*` in this case); being a type, it can define its own cardinalities (`0..1`). Hence, similar to the solution based on explicit dynamic types, new problem types and solution methods can be dynamically created at the middle meta-level, therefore addressing requirements 1 and 2.

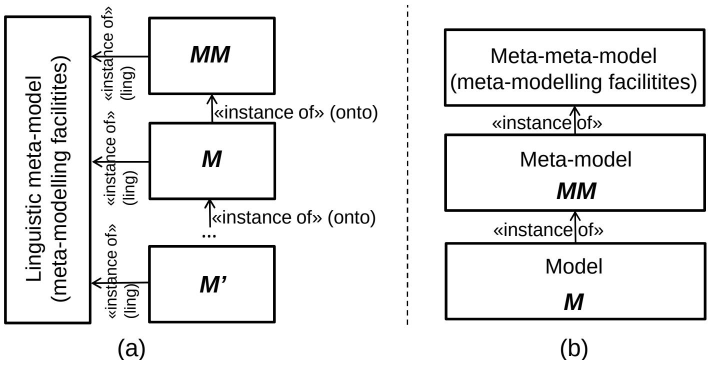
As multi-level modelling spans several meta-levels, it becomes necessary to control the instantiation depth of elements and the features of instances beyond the adjacent meta-level below. For this purpose, the *potency* permits specifying in how many meta-levels an element can be instantiated. The potency is a positive number (or zero) that can be attached to models, clabjects, fields and references. It gets automatically decremented in the instances at each deeper meta-level, and when it reaches zero, it is not possible to instantiate the element in lower meta-levels.

In Figure 3, we use '@' to denote the potency of elements, following the syntax of the METADEPTH multi-level modelling tool [de Lara and Guerra 2010]. If an element is not tagged with a potency, then it receives the potency from its immediate container, and ultimately from the model. In the example, the top-level model has potency 2 and therefore can be instantiated in two meta-levels. All elements in this model receive the same potency, except the field `transfName`, which explicitly declares potency 1. This means that the instances of `TransformationSolution` should provide a value for it at the next meta-level. If the field were defined with potency 2, it should receive a value two levels below, though it could still receive a value at the intermediate level acting as default for the instances at the level below. Hence, while in standard two-level modelling a type can only control the features of its instances at the immediate meta-level below, the potency is a way for deep characterization, enabling the declaration of properties for elements several meta-levels below. An example of fields with potency 2 is given in the running example of Section 4. We take the convention of hiding the fields with primitive type and potency bigger than 0 at intermediate meta-levels.

The multi-level solution addresses all requirements of the running example, in contrast to the static type solution in Figure 1. Moreover, it is conceptually simpler than the explicit dynamic type solution in Figure 2, as it has less modelling elements (11 clabjects vs 19 classes/objects, and 6 references vs 22 references) and there is no need to define OCL constraints to make explicit the instantiation semantics of any element.

3.2. The orthogonal classification architecture: Linguistic and ontological typings

Technically, one may wonder how it is possible to define inheritance relationships or fields (like `numNodes`) in the intermediate meta-level of our example. While two-level meta-modelling frameworks make available meta-modelling facilities (like inheritance or the ability to define abstract classes) only at the top meta-model, in multi-level modelling these facilities are available at every meta-level by adhering to the orthogonal classification architecture (OCA), as Figure 4(a) shows. This architecture was originally proposed in [Atkinson and Kühne 2002].



Linguistic meta-model (meta-modelling facilitites)
«instance of» (ling)
MM
«instance of» (onto)
M
...
M'
(a)
Meta-meta-model (meta-modelling facilitites)
«instance of»
Meta-model
MM
Model
M
(b)

Fig. 4. (a) Orthogonal classification architecture. (b) Standard meta-modelling architecture.

OCA distinguishes two orthogonal typings for model elements: *ontological* and *linguistic*. The ontological classification structure or logical dimension of an element expresses instantiation within a domain. For instance, `DirectSolution` is the ontological type of `MultiAgentSystem`. The linguistic typing or physical dimension of an element refers to the meta-modelling facility used for the construction of the element. Here, one assumes a linguistic meta-model which includes concepts like `Clabject`, `Field` and `Reference`, and makes available meta-modelling facilities like inheritance and instantiation. For example, the linguistic type of `MultiAgentSystem` is `Clabject`, and the linguistic type of `toTSP` is `Reference`. While all elements in a model have a linguistic typing, they may lack the ontological typing. This is the case of the field `numNodes`. The set of elements of a model with only linguistic typing is called *linguistic extension* [de Lara and Guerra 2010].

In contrast, standard two-level meta-modelling architectures do not distinguish between ontological and linguistic typings, but there is only one typing, as Figure 4(b) shows. This architecture makes available meta-modelling facilities only at the meta-model level. However, we have seen that making available these facilities at every meta-level avoids the need to explicitly model them, as we did in the solution of Figure 2. The instantiation relation between MM and the meta-meta-model is linguistic because it provides the meta-modelling facilities to build MM. Thus, the meta-meta-model plays the role of the linguistic meta-model in the OCA architecture. The instantiation relation between M and MM is of ontological nature.

So far, we have illustrated that multi-level modelling can help in building systems in a simpler way. However, when facing the development of a new system, the question still remains of whether using multi-level modelling for that particular case would be useful. To answer this question, in the next section we systematically identify different meta-modelling patterns, analyse potential solutions, and discuss the advantages and disadvantages of each solution depending on the scenario.

4. MULTI-LEVEL META-MODELLING PATTERNS

This section introduces different meta-modelling patterns where multiple levels are implicit, and discusses the advantages and disadvantages of different modelling solutions, including both two-level and multi-level. The patterns emulate different meta-modelling facilities, like the *instantiation* relation and the *type-object* facet of elements, the dynamic creation and instantiation of features, the definition and instantiation of attributed entities, the configuration of references, and the classification of elements.

The patterns we present emerged from both our own experience building multi-level systems and by encoding such systems in a two-level setting following alternative approaches. As we will show in Section 5, it was also crucial to perform an extensive analysis of the literature and existing meta-models, like the ATL meta-model zoo or the OMG specifications. This study made evident that these patterns were actually modelling different meta-modelling facilities using different techniques.

We use a presentation of the patterns in the style of [Gamma et al. 1994], including first a brief statement of the pattern goal ("*intent*" section), followed by additional names that practitioners use for the pattern ("*also known as*" section), a motivating example revolving around the classical problem of dynamically modelling product types and their instances (section "*motivation*"), several solutions (section "*solutions*"), the structure of the pattern (section "*structure*") and some conclusions with the trade-offs of each solution (section "*conclusions*"). As a difference with [Gamma et al. 1994], in the solutions section we discuss the merits of both two-level and multi-level solutions.

4.1. Modelling types and instances: The type-object pattern

Intent. This pattern allows the explicit modelling of types and their instances, where types are not static but can be added dynamically. Types and their instances may define features, which are fixed and known a priori.

Also known as. This pattern also arises in programming, where it is called *type-object* [Martin et al. 1997], *item descriptor* [Coad 1992] or *metaobject* [Kiczales and Rivieres 1991] pattern.

Motivation. In the running example, we have the need to define different kinds of optimization problems and solution methods dynamically, and then create instances of them. Another example is the classical problem of modelling product types – like books or CDs – which need to be added to the system on-demand, as well as their instances [Atkinson and Kühne 2002]. Product types have a value-added tax (*vat*), while instances have a *price*.

Solutions. We next describe seven possible solutions for the "product types" example.

Static types. The first two-level solution, shown in Figure 5(a), models product types as static entities at the meta-level. They are defined as subclasses of `Product`, from

which they inherit both the `price` (instance level property) and the `vat` (type level property). The `vat` is given a value at the meta-level via redefinition.
The advantage of this solution is that it uses a native meta-model facility (the instantiation) to relate product instances (`GoF`) with their type (`Book`). The drawback is that this solution emulates values at the meta-level by means of redefinition, which may be difficult for references/associations as some meta-modelling frameworks (e.g., EMF) do not support their redefinition. Moreover, this solution does not meet the intent of the pattern, as the product types (like `Book`) are fixed a priori; if a new kind of `Product` is to be added, the meta-model has to be changed to add a new subclass of `Product`.

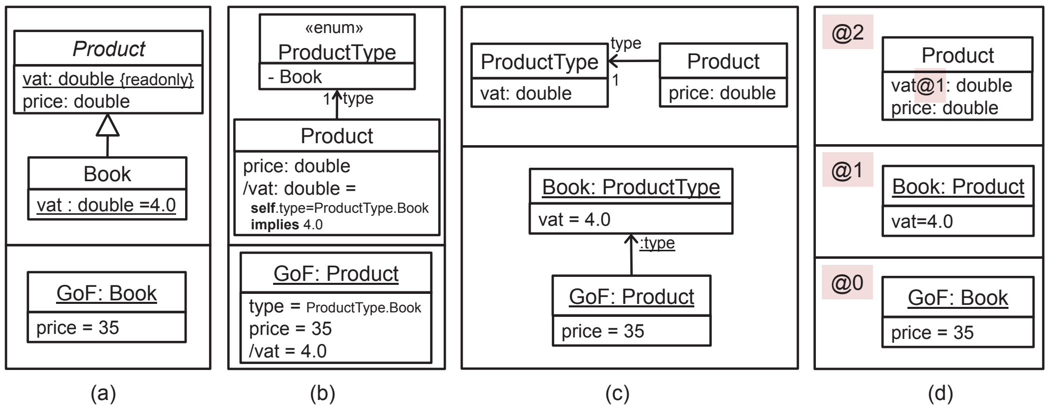


Fig. 5. Solutions for the *type-object* pattern. (a) Static types. (b) Enumerated types. (c) Explicit dynamic types. (d) Multi-level.

Enumerated types. Figure 5(b) shows another two-level solution where an enumerated type (`ProductType`) defines all possible types of product (`Book` in this case). Since the types defined in the enumeration (`Book`) cannot declare attributes[5], we define *vat* as a derived attribute in `Product`, and calculate its value depending on the selected type for the product.
Albeit simple, this solution does not meet the intent of the pattern because, similar to the *static types* solution, the product types are fixed a priori in the enumeration. Another issue is that not all meta-modelling frameworks support the definition of derived attributes or automate their computation. On the other hand, the advantage of using an enumerated type is that it allows instances to have zero or several types just by changing the cardinality of the reference `type`, and it would be possible to change the type of an instance dynamically at the model level [Riehle et al. 2000].

Explicit dynamic types. Figure 5(c) shows a two-level solution consisting of the explicit modelling of types (`ProductType`) and instances (`Product`). Thus, product types and products are both objects, fully satisfying the requirements of the scenario.
The drawback of this solution is that it requires adding an extra class to enable the instantiation of both product types and instances, and the instantiation relation needs to be explicitly modelled (reference `type`). As we will see, this solution can get quite complicated if sophisticated features like inheritance or dynamic features are also

[5]Some programming languages like Java support the definition of attributes in enumerated types, but most meta-modelling frameworks lack this feature.

needed. Nonetheless, the explicit modelling of meta-modelling facilities also has advantages, as it provides greater flexibility to define domain-specific typings. Similar to the *enumerated types* solution, it allows a product to have several types or no type at all by changing the cardinality of reference `type`, and the dynamic retyping of instances. This solution also enables *non-uniform* instantiation, where types and instances have different structure. We will explore this possibility in Section 4.4.

Multi-level. Figure 5(d) shows the multi-level solution. The feature for product types (`vat`) is assigned potency 1, while the feature for instances (`price`) has potency 2 inherited from the container model. Compared to (a), this solution allows the dynamic addition of new product types. Compared to (b), this solution permits assigning instance facets (e.g., `vat`) to types. Compared to (c), this solution does not need to explicitly model the instantiation relation, as it uses the native meta-modelling facilities of the framework. However, the drawback is that the instantiation semantics is fixed, as given by the meta-modelling framework. Moreover, instantiation is always mediated, that is, in order to be able to create `GoF` at level 0, first it is necessary to create its type `Book` at level 1.

Promotion. A way to solve the drawbacks of the *static types* and the *explicit dynamic types* approaches using a two-level architecture is to model product types as objects, and then promote these objects into instantiable meta-classes. This can be done through a so-called *promotion* model-to-model transformation which receives a model as input and produces a meta-model as output, as shown in Figure 6(a). In this way, we are conceptually emulating the multiple meta-levels of the *multi-level* solution, but here the instance and type facet of elements are separated. The instance facet is defined by instantiating the meta-model with label (1) in the figure, and then a transformation creates the type facet (meta-model with label (3)). Figure 6(b) shows a variant of this solution, where the meta-classes generated by the transformation inherit from some base class which provides them with common features (e.g., the `price`). Altogether, the generated meta-model is similar to the *static types* solution of Figure 5(a), but product types can be created dynamically as objects.

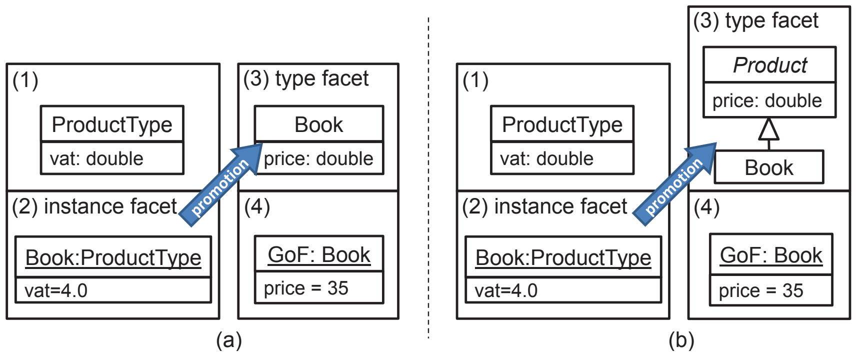


Fig. 6. Solutions for the *type-object* pattern. (a) Promotion. (b) Promotion with base classes.

The main advantage of this design is its flexibility, as one can create types at will using the promotion transformation, enabling non-uniform instantiation. This benefit was also present in the *explicit dynamic types* solution. However, there are a number of drawbacks. First, there is the need to add an additional operational element when modelling: a promotion transformation. This transformation can “hide” important information about the system. For example, the fact that every `Book` has a price in solution (a), or that any meta-class coming from a product type must inherit from

Product in solution (b), is hard-coded in the transformation. Moreover, there is a disconnection between the instance facet and the type facet of a same element (e.g., accessing the vat value of Books from the GoF object may become cumbersome).

Powertypes. A powertype [Odell 1994] is a type, the instances of which are subtypes of another type. Conceptually, this is what the promotion transformation with base classes shown in Figure 6(b) yields, as the instances of ProductType – like Book – become subtypes of Product. Thus, following the same idea, we can use powertypes to model the system as depicted in Figure 7(a). The powertype (ProductType) holds the attributes for types (vat), while another type (Product) holds the attributes for instances (price). Then, we can create Book as an instance of ProductType, which by definition becomes a subclass of Product and therefore can be instantiated to create GoF. In this way, using powertypes is a multi-level solution where the subtyping relation can cross meta-levels[6].

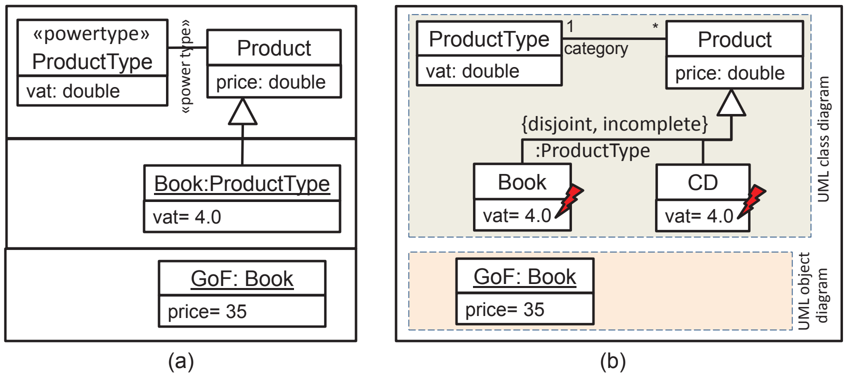


Fig. 7. Solutions for the *type-object* pattern. (a) Powertypes. (b) UML powertypes.

The UML realizes the powertype concept using the notation shown in Figure 7(b). The figure contains a class diagram and an object diagram. Both diagrams are instances of the UML meta-model, which is defined one meta-level above. We use dash lines to separate both diagrams because the instantiation relation between class and object diagrams is modelled within the UML (i.e., it could be represented as an association named *type*, similar to the *explicit dynamic types* approach).

This solution is problematic because UML classes do not have an instance facet (i.e., Classifier, which is the meta-class for UML classes, is separated from InstanceSpecification, which is the meta-class to represent UML objects); hence, it is not possible to assign a value to the slot vat. Actually, the UML standard states that "*Power types are a conceptual, or analysis, notion. [...] In object-oriented implementations, the instances of a class could also be classes*".

Altogether, powertypes separate the type and instance facet of elements in two meta-classes. This provides clarity to the design, but at the expense of a bigger number of meta-classes and associations. While potency-based multi-level modelling allows the characterization of features at any meta-level below by using potency, powertypes are only able to characterize features of the next two meta-levels. Moreover, clabjects can define slots due to their instance facet; hence, they do not suffer from the problem in UML powertypes shown in Figure 7(b). However, subtyping across meta-levels as required by the powertype concept is not supported in potency-based

[6]Note that [Eriksson et al. 2013] propose a more compact notation where inheritance is hidden.

multi-level modelling and would be forbidden in strict meta-modelling frameworks (like the EMF), where only the instantiation relation can cross meta-levels and is not allowed between elements at the same meta-level [Atkinson and Kühne 2002]. Alternatively, one could use a promotion transformation with base classes to implement powertypes. Very few tools natively support powertypes [Volz and Jablonski 2010].

Stereotypes. Stereotypes [OMG 2011g] are an extension mechanism proposed by the OMG for MOF-based meta-models. They are quite popular among engineers to define extensions for UML (called UML profiles) for different domains, like real-time [OMG 2011f] or business process modelling [OMG 2013b; 2008a].

A stereotype represents a domain concept. It is defined as an extension of some meta-class of the UML meta-model, and can have attached tagged values (meta-attributes) defining specific features, as well as OCL constraints. Figure 8 shows a solution of the example using stereotypes. We need to define the stereotype *ProductType* with the tagged value `vat` as an extension of the UML meta-class `Class`. This way, in UML models, we can assign a vat to a class stereotyped *ProductType*. Moreover, since all product instances have a price, we define the stereotype *Product* as an extension of `InstanceSpecification`, and attach to it the feature `price`. Finally, we need an OCL constraint to check that any UML object stereotyped *Product* must be an instance of a UML class stereotyped *ProductType*.

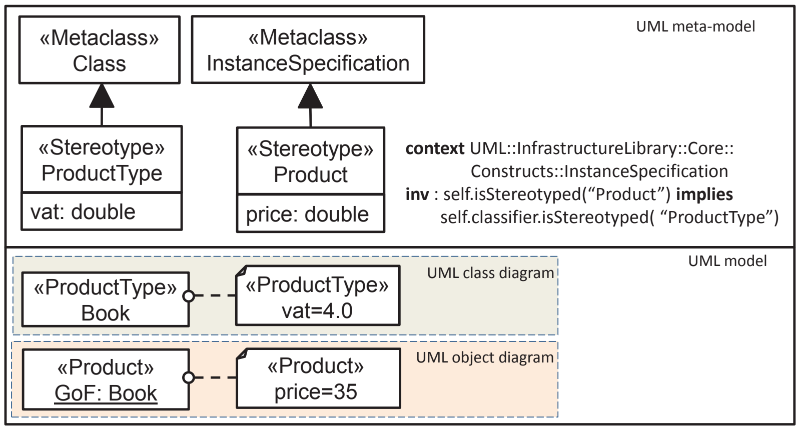


Fig. 8. Solution for the *type-object* pattern using stereotypes.

This solution is simple and builds on MOF, which is a standard. As in multi-level modelling, there is no need to explicitly model the instantiation relation between domain types and instances, but the one defined in the UML meta-model is reused; however, note that this instantiation relation is different from the one used between meta-levels, and its semantics is fixed, with objects being an instance of a type at a time. Among the disadvantages, this solution requires from the definition of two stereotypes, while just one clabject is needed in the multi-level solution. Moreover, the definition of profiles requires knowledge of the UML meta-model, and the result is an extension of the (rather large) UML, which sometimes may be useful but other times not. A small meta-model will normally be easier to instantiate for users than a large meta-model with many concepts. Stereotypes are limited to model domain types (e.g., `Book`) and instances (e.g., `Product`), but are unable to model more than two levels, because the UML provides linguistic support for classes (meta-class `Class`) and instances (meta-class `InstanceSpecification`) only. Moreover, this solution relies on the degree of coverage of the *profiling* mechanism by the used UML tool, which may vary between different tools. In particular, the definition of arbitrary OCL constraints in profiles is only supported by some tools. Finally, the fact that an instance of `Book` (like `GoF`) should be stereotyped with `Product` is not explicit, but it needs to be encoded as constraints. Some OMG specifications, like MARTE [OMG 2011f], informally document this fact through dependency relations.

Structure. Figure 9 shows the structure of the *type-object* pattern using multi-level modelling, which is used as a basis to compare with the structure of the other solutions. The *multi-level* solution declares the domain type with potency 2, the features of the type with potency 1, and the features of the instances with potency 2. In this way, the dynamic types can be created in the intermediate level. Such dynamic types have a dual type/instance facet, as they can be instantiated in the lower meta-level.

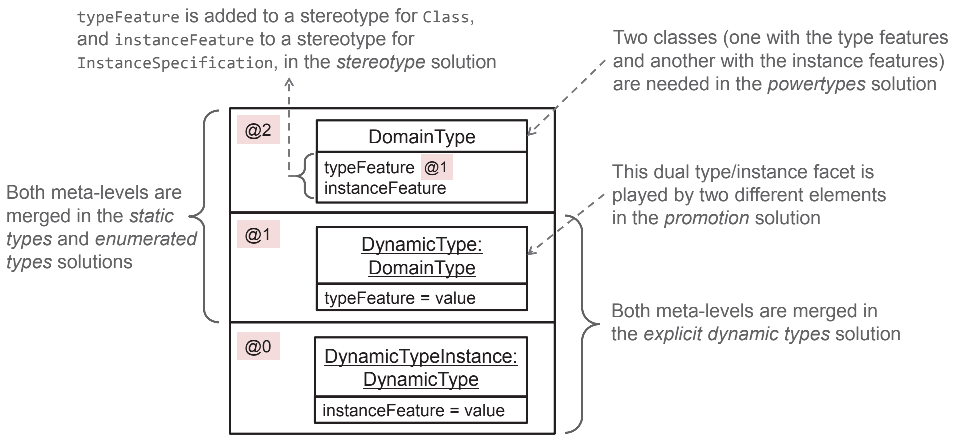

typeFeature is added to a stereotype for Class, and instanceFeature to a stereotype for InstanceSpecification, in the *stereotype* solution
Two classes (one with the type features and another with the instance features) are needed in the *powertypes* solution
@2
DomainType
typeFeature @1
instanceFeature
Both meta-levels are merged in the *static types* and *enumerated types* solutions
This dual type/instance facet is played by two different elements in the *promotion* solution
@1
DynamicType: DomainType
typeFeature = value
Both meta-levels are merged in the *explicit dynamic types* solution
@0
DynamicTypeInstance: DynamicType
instanceFeature = value

Fig. 9. Structure of the *type-object* pattern using multi-level modelling.

Other solutions emulate the instantiation relation and the dual type/instance facet of `DynamicType` in different ways. The *static types* solution declares both `DomainType` and `DynamicType` at the top-most meta-level using subclassing, while the *enumerated types* solution declares the dynamic types as literals in an enumeration. Hence, these two solutions "merge" the meta-levels with potency 2 and 1 in Figure 9, so that dynamic types are not really dynamic, but they are statically defined. Instead, the *explicit dynamic types* solution supports the creation of truly dynamic types and their instances at the model level by defining meta-classes for them. Hence, this solution merges the meta-levels with potency 1 and 0 in Figure 9. The *promotion* solution emulates the dual type/instance facet of `DynamicType` using two separate elements: first, one has to create an instance of `DomainType` (i.e., it works with the models at potency 2 and 1 in Figure 9); and then, it promotes that instance into a new type that can be instantiated (i.e., it works with the models at potency 1 and 0). *Powertypes* use two meta-classes to represent `DomainType`: one contains the features with potency 1, and the other the features with potency 2. In this way, the dynamic types are created by both instantiating the first meta-class and subclassing the second. Finally, *stereotypes* allow extending UML with domain types (stereotype for `Class` defining the type features) and instances (stereotype for `InstanceSpecification` defining the instance features). Then, dynamic types can be created in class diagrams and instantiated in object diagrams.

Conclusions. From the previous solutions, the ones that do not solve the scenario are the *static types* and the *enumerated types* solutions, because adding new dynamic types implies modifying the meta-model. While this may not break the conformance of existing models, it may imply a recompilation step. Moreover, it might not be adequate to permit a direct manipulation of the meta-model to the users, as they would be able to make arbitrary changes which could break conformance of existing models. *Explicit dynamic types* and *promotion* are the only solutions which may enable non-uniform instantiation. *Explicit dynamic types* provide extra flexibility, enabling instances with several types, as the instantiation relation is hand-made and can be customized. If such facilities are not needed, either *multi-level* or *stereotypes* are simpler to implement, as the instantiation relation is built-in. Whereas *multi-level* modelling allows the construction of tailor-made DSMLs containing only concepts of the domain, *stereotypes* extend the UML with domain concepts (i.e., the DSML is embedded in the UML). Altogether, the potency-based *multi-level* solution is simpler (it requires less modelling elements), and should be preferred to *stereotypes* when reusing the whole UML is not desired.

4.2. Dynamic features

Intent. This pattern permits the dynamic addition of new features to a type, and the corresponding slots to the instances of those types.

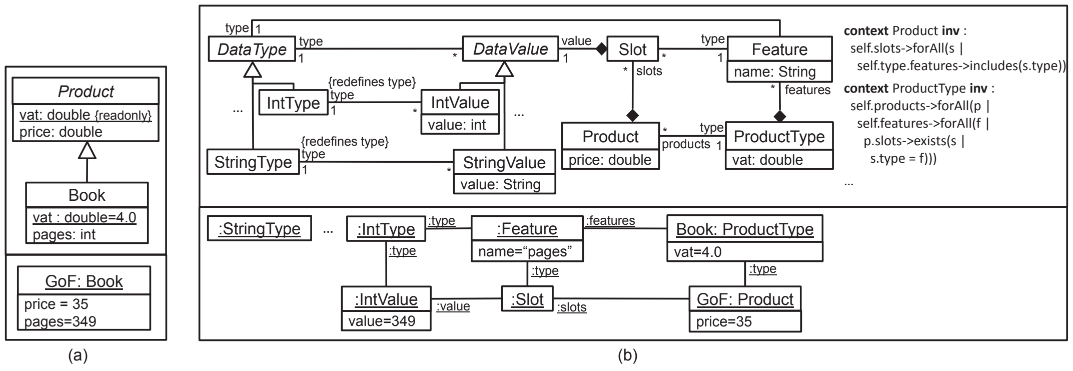
Also known as. This pattern arises in data modelling, where it is called *entity attribute value* [Sanders 1995] model, and in programming, under the name of *object attribute value*, *dynamic object model* [Riehle et al. 2000], and *adaptive object-model* [Yoder and Johnson 2002].

Motivation. Sometimes, it is not possible to foresee the features needed by a certain type, or extensibility by the end-user is required in that aspect. In such cases, a mechanism is needed to add and instantiate features on demand. This is in contrast with the *type-object* pattern, where features for types and instances are fixed a priori. *Dynamic features* appear together with the *type-object* pattern and it can be seen as a special case of it, where the feature is the type, and the slot is the instance; however, dynamic features frequently need to impose additional constraints regarding the compatibility of the elements defining and instantiating the features, and the adequacy of data types and values. A "*shallow*" version of this pattern appears when there is the need to define features for some element, but such features do not get instantiated.

The motivating example of Section 2 contains two instances of this pattern, as we need to define features for the created problem and solution types, and assign them a value in the instances. Coming back to the products example, we can extend it to require the definition of features for product types. For example, we may add the number of pages to the product type `Book`, or the RAM memory size to `Computer`.

Solutions. We next describe six solutions for the "product types" example. We omit the *enumerated types* solution because it does not allow a direct declaration of features in the enumeration, but features should be modelled using a similar approach to the one we will explain for *explicit dynamic types*. Unless explicitly stated, the same advantages and disadvantages discussed in Section 4.1 for each approach remain, so next we only comment on new issues.

Static features. Figure 10(a) shows an inadequate solution combining static types and features defined at the meta-model level. For instance, the product type `Book` is assigned the book-specific feature `pages` at the meta-level. Adding features dynamically at the model level is not possible.



```
context Product inv :
  self.slots->forAll(s |
    self.type.features->includes(s.type))
context ProductType inv :
  self.products->forAll(p |
    self.features->forAll(f |
      p.slots->exists(s |
        s.type = f)))
...
```

Fig. 10. Solutions for the *dynamic features* pattern. (a) Static features. (b) Explicit dynamic features.

Explicit dynamic features. Figure 10(b) shows a general two-level solution solving the scenario, where features, data types, slots and data values are explicitly modelled. As an example, the lower model shows the addition of the feature `pages` to the product type `Book`, as well as the assignment of the value 349 to the corresponding slot in the `GoF` book instance. Actually, one can interpret that the meta-model entails three *nested* occurrences of the *type-object* pattern: for product types and their instances (`ProductType` / `Product`), for features and their instances (`Feature` / `Slot`), and for data types and their instances (`DataType` / `DataValue`)[7]. However, we need to define additional OCL constraints ensuring the compatibility of the instantiated elements. The figure shows a pair of such constraints, which check whether a product has slots for all features in its product type; further constraints should take care of the correct assignment of data values to the slots according to the data type of the feature. Although data types could be modelled with an enumerated type, the proposed solution is more general as it enables the definition of attributes and associations on data types, being easier to extend if dynamic references are also needed.

This solution has the same pros and cons as the dynamic types version of the *type-object* pattern, but now a few more disadvantages arise. First, the meta-model introduces quite a number of extra meta-classes and OCL constraints, in particular to model data types and instances and to ensure compatible instantiation. Moreover, this solution does not model uniformly "fixed" features (`price`) and dynamic features (`pages`), which contributes to further complexity (e.g., when writing a model-to-model transformation).

Multi-level. Figure 11(a) shows a multi-level solution. It makes use of linguistic extensions at the intermediate meta-level (i.e., definition of elements without an ontological type) in order to add new features, like `pages`, dynamically. The added features can be assigned a value in the meta-level below. The advantage of the solution is its simplicity, and the fact that both dynamic features (`pages`) and "fixed" features (`price`) are handled uniformly.

Promotion. Figure 11(b) shows a promotion-based solution. Only product types, data types and features are explicitly modelled (meta-model 1). A transformation promotes their instances (label 2) into meta-classes with an attribute coming from each feature (meta-model 3). Then, instantiation can be used to natively provide values for these attributes (model 4). Although dynamic and "fixed" features are used uniformly in

[7]Actually, one can also recognise the use of a static approach to define a fixed set of data types (like `IntType`) and data values (like `IntValue`).

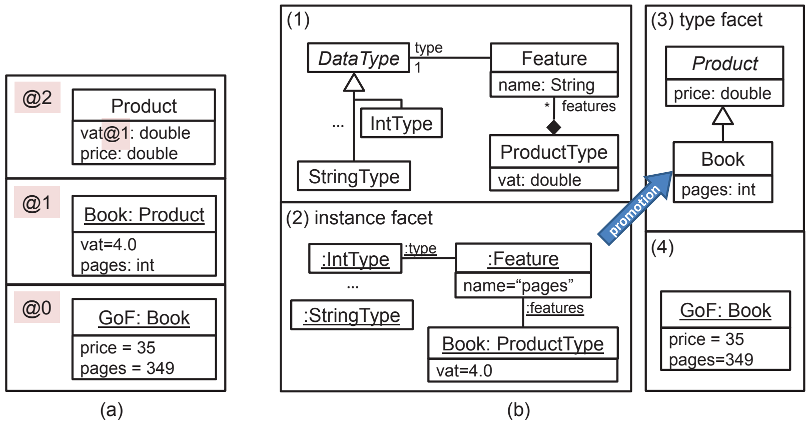


Fig. 11. Solutions for the *dynamic features* pattern. (a) Multi-level. (b) Promotion.

model 4, they are defined using different techniques (dynamic ones are modelled, but the generation of the "fixed" ones must be encoded in the promotion transformation).

Powertypes. Figure 12(a) shows a solution using UML powertypes, in which we simply use the native UML modelling facilities to add features to the UML class `Book` and instantiate them in the UML object `GoF`. However, as we explained previously, UML classes do not have an instance facet, and therefore, even though we could define the attribute `vat` in the meta-class `ProductType`, it would not be possible to assign it a value in `Book` (see Figure 7(b)).

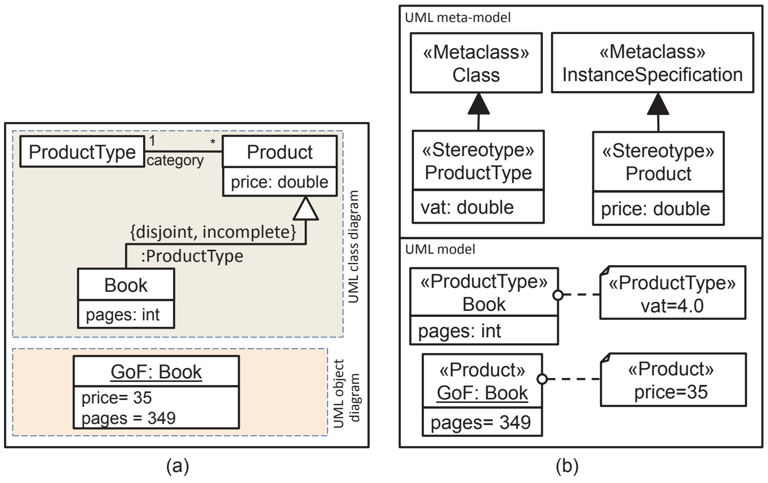


Fig. 12. Solutions for the *dynamic features* pattern. (a) UML powertypes (b) Stereotypes.

Stereotypes. Figure 12(b) shows a solution using stereotypes. As before, we can use the native UML modelling facilities to define features in `Book` and instantiate those in the `GoF` object. However, a disadvantage is that the features coming from the stereotype (`price`) and those coming from the type (`pages`) are not handled uniformly, adding accidental complexity to the solution.

Structure. Figure 13 shows the structure of the pattern using multi-level modelling. The dynamic feature is defined at level 1 as a regular feature because `DynamicType` has a type facet.

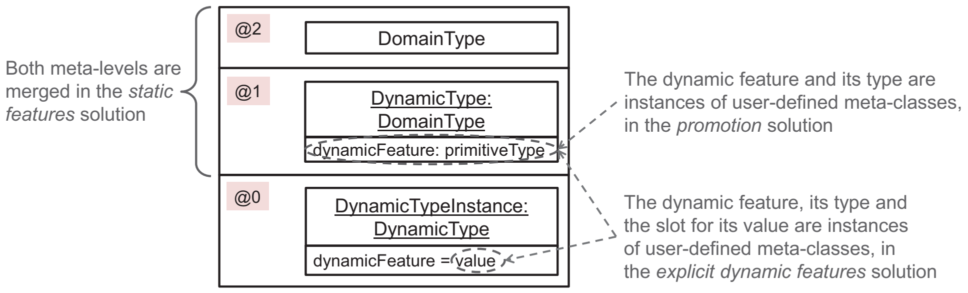

@2
DomainType
@1
DynamicType: DomainType
dynamicFeature: primitiveType
@0
DynamicTypeInstance: DynamicType
dynamicFeature = value
Both meta-levels are merged in the *static features* solution
The dynamic feature and its type are instances of user-defined meta-classes, in the *promotion* solution
The dynamic feature, its type and the slot for its value are instances of user-defined meta-classes, in the *explicit dynamic features* solution

Fig. 13. Structure of the *dynamic features* pattern using multi-level modelling.

In contrast, the *static features* solution uses inheritance to relate `DynamicType` and `DomainType` in the same meta-level, while `dynamicFeature` is declared in `DynamicType`. In the *explicit dynamic features* solution, the meta-model needs to include facilities to both declare and instantiate dynamic features, while the *promotion* solution only requires defining facilities for feature declaration. Powertypes can use the standard facilities to declare and instantiate features, and similar for stereotypes, as the UML provides support for feature definition (in UML classes) and feature instantiation (in UML objects).

Conclusions. The only approach that covers all requirements of the scenario and handles uniformly all types of features (i.e., type features like `vat`, "fixed" instance features like `price`, and dynamic instance features like `pages`) is the *multi-level* solution. The explicit modelling of dynamic features results in much complexity, though this solution allows for the definition of domain-specific data types that are not natively provided by the meta-modelling framework.

4.3. Dynamic auxiliary domain concepts

Intent. The pattern helps with the dynamic addition of new entities related to a type, as well as the instantiation of those entities, which should be correctly related to instances of the type. Entities may define their own features, in which case, their instances need to define slots for the feature values. This pattern can be seen as a variant of *dynamic features*, but instead of defining features for dynamic types, here the aim is defining dynamic entities in relation to dynamic types.

Also known as. None.

Motivation. Sometimes, it is not possible to foresee the entities needed to describe all properties of a certain type, or want to achieve extensibility in that aspect. In such cases, it is necessary to dynamically create and instantiate new entities, being able to relate these entities to the type they are describing. For example, resuming the products example, we may want to be able to dynamically add an entity `Author` and use it to describe the authors of the product type `Book`.

Solutions. We next describe six solutions for the "product types" example.

Static entities. Figure 14(a) shows how to define new entities (`Author`) related to particular types (`Book`) at the meta-model level. However, this solution using static types and static relations is invalid because entities cannot be created dynamically.

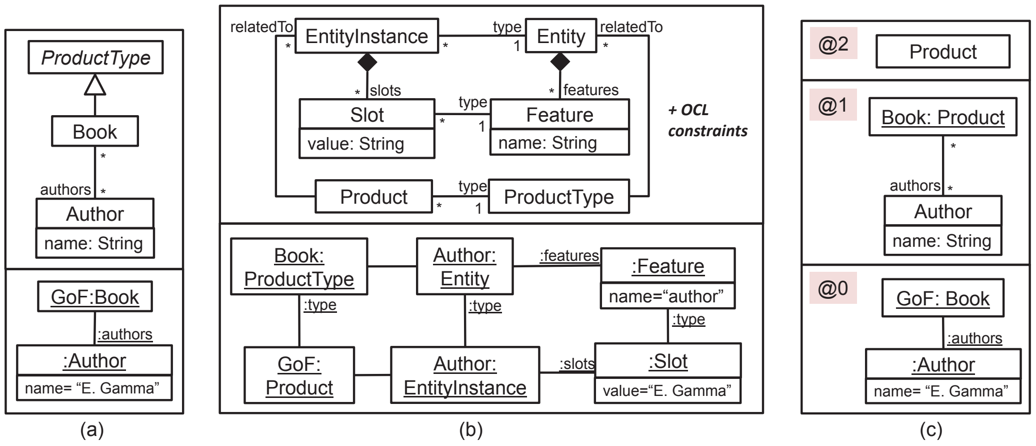


Fig. 14. Solutions to the *dynamic auxiliary domain concepts* pattern. (a) Static entities. (b) Explicit modelling of domain concepts. (c) Multi-level.

Explicit modelling of domain concepts. Figure 14(b) shows a solution where entities and their features are explicitly modelled in the meta-model (we omit the OCL constraints needed to ensure their correct instantiation, as they are similar to those in Figure 10(b)). This solution introduces a number of additional meta-classes and constraints, which makes it complex to define (at the meta-model level) and use (at the model level). Moreover, it is not possible to configure the relations defined between the instances of `ProductType` and `Entity`, for instance, to enforce `Books` to have more than one `Author`. The proposed solution allows any number of them. In section 4.4 we will show how to deal with this problem using the *relation-configurator* pattern.

Multi-level. Figure 14(c) corresponds to the multi-level solution for this pattern. It makes use of linguistic extensions at the intermediate meta-level to define a new entity `Author` related to the product type `Book`. Relation `authors` is a proper relation, and therefore, it is possible to configure its cardinality, e.g., `1..*`.

Promotion. A solution based on promotion would require explicitly modelling product types, entities and their features. Then, the promotion transformation should create meta-classes from the instances of the product types and entities, and attributes from the features. This is similar to the solution provided for *dynamic features* in Figure 11(b), having the same benefits (flexibility) and drawbacks (heterogeneity of specification mechanisms).

Powertypes. The solution for this pattern using powertypes is similar to the one shown for *dynamic features* in Figure 12(a). Since `Book` is a UML class, it can define relations to other UML classes by means of UML properties. This solution has the same advantages and disadvantages as the use of powertypes for modelling *dynamic features*.

Stereotypes. The solution for this pattern using stereotypes is similar to the one shown for *dynamic features* in Figure 12(b). Since `Book` is a UML class, it can define relations to other UML classes by means of UML properties. This solution has the same advantages and disadvantages as the use of stereotypes for modelling *dynamic features*.

Structure. Figure 15 shows the structure of the pattern using a multi-level approach. The new dynamic auxiliary domain entity, as well as its features and relations, are defined at the meta-level with potency 1, and can be instantiated at the meta-level below in a straight-forward way.

In comparison, the two upper meta-levels in Figure 15 are merged in the *static entities* solution, so that the class `DynamicAuxiliaryDomainConcept` and its relation with

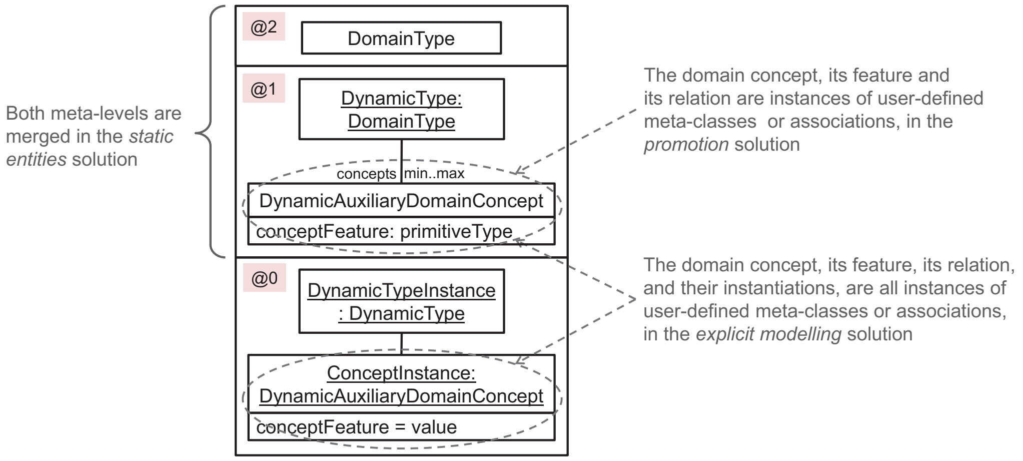

@2
DomainType
@1
DynamicType: DomainType
concepts min..max
DynamicAuxiliaryDomainConcept
conceptFeature: primitiveType
@0
DynamicTypeInstance : DynamicType
ConceptInstance: DynamicAuxiliaryDomainConcept
conceptFeature = value
Both meta-levels are merged in the *static entities* solution
The domain concept, its feature and its relation are instances of user-defined meta-classes or associations, in the *promotion* solution
The domain concept, its feature, its relation, and their instantiations, are all instances of user-defined meta-classes or associations, in the *explicit modelling* solution

Fig. 15. Structure of the *dynamic auxiliary domain concepts* pattern using multi-level modelling.

`DynamicType` can be defined in the same meta-level as `DomainType`; however, this solution does not allow defining new concepts dynamically. In the *explicit modelling of domain concepts*, the meta-model needs to include facilities to both declare and instantiate the auxiliary domain concept, its feature and its reference. In the *promotion* solution, it is enough to include facilities for the declaration of these elements, but not for their instantiation. Finally, both powertypes and stereotypes can use the standard facilities to declare and instantiate dynamic entities.

Conclusions. Multi-level modelling, powertypes and stereotypes natively support the dynamic creation of entities and their features. Moreover, it is straightforward to constrain the cardinality of the relations defined between the primary type (e.g., product type) and its constituent entities (e.g., author). The latter becomes more complex for the solutions based on the explicit modelling of domain concepts and promotion.

4.4. Relation configurator pattern

Intent. It allows the configuration of a reference type dynamically created, and the instantiation of the reference type according to that configuration.

Also known as. None.

Motivation. Sometimes, it is needed to define reference types on demand (e.g., inside new dynamic types) which can be instantiated (e.g., in the instances of the dynamic type). In this scenario, it may become necessary to configure the association ends of reference types created on demand, for instance to determine their cardinality or the allowed class of objects to which the reference or association instances can be connected.

Solutions. We next describe six solutions for an extended version of the “product types” example, where we consider that product types are made by zero or more manufacturer types, and we want to configure the relation between concrete products and manufacturer types in a dynamic way. For example, we may want to specify that books are published by exactly one editorial.

Static references. Figure 16(a) shows a static solution where we redefine the relation `madeBy` for the `Book` product type. At the instance level, we can specify that the `GoF` book is published by `AdWesley`.

Although this is a neat solution, it does not allow creating product types and references to manufacturer types dynamically, but the meta-model needs to be changed, which can break the conformance of existing models. Moreover, as previously stated, association redefinition is not supported in many meta-modelling frameworks. In addition, the solution does not adequately capture the requirements: the `madeBy` relation is aimed at specifying that a product type (a `Book`) can be made by zero or more `Manufacturer` types (e.g., published by one `Manufacturer` and written by one `Author`). Hence, even if the * cardinality of `madeBy` actually constrains the lower meta-level, our aim here is to constrain how many times the reference can be redefined, and then let the cardinality of the redefined associations constrain the lower meta-level. However, this is not possible with the static solution.

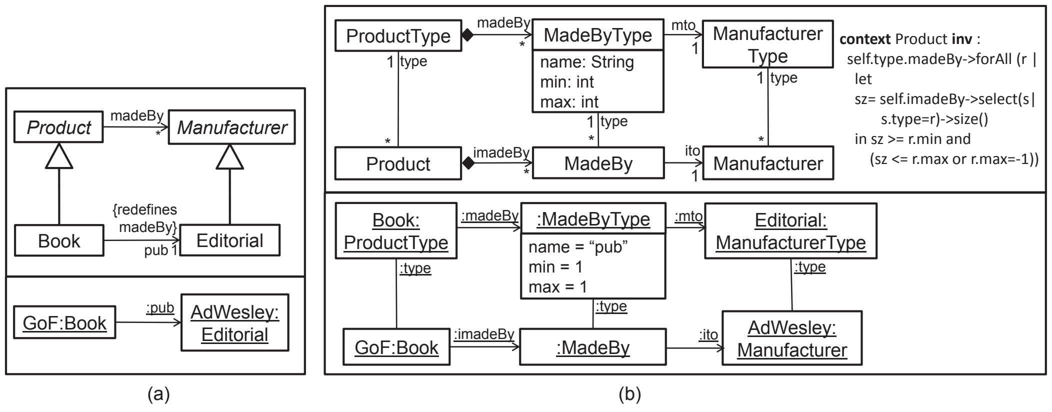


Fig. 16. Solutions to *relation configurator* pattern. (a) Static references. (b) Explicit reference modelling.

Explicit reference modelling. Figure 16(b) shows a solution based on the explicit modelling of reference types and their instances at the meta-level. In this solution, we need to create an intermediate class `MadeByType` to store the configuration (name and cardinality) of relation types. The OCL constraint ensures an appropriate number of instances of each relation type, according to the cardinality defined in `MadeByType`. This solution improves the previous one because it allows the creation of new relation types dynamically, and the cardinality constraint * on relation `madeBy` does constrain the number of relation types that can be defined on specific product types. However, it introduces complexity because the solution involves a relation configurator class (`MadeByType`) and another class for their instances (`MadeBy`). At the model level, this solution is not optimal because we would prefer to use references instead of an intermediate `MadeBy` object. This object is needed to distinguish between different relation types, which in the solution is done by the reference `type` from the `MadeBy` object to the `MadeByType` object.

Multi-level. Figure 17(a) shows the multi-level solution. Instead of modelling the relation configurator as a meta-class at the top meta-level, it is defined as a reference (`madeBy`) which can be instantiated at level 1 (`pub`). Since the instantiated reference `pub` has a type facet, we can assign it a cardinality (`1`) and instantiate it at the lower meta-level. This solution is simpler than the *explicit reference modelling*, and more flexible than the *static references* solution.

In the running example, the *relation configurator* customises typical features of references, like their name and cardinality. If we need to customise features beyond those offered by the meta-modelling framework, like a feature dictating whether it is

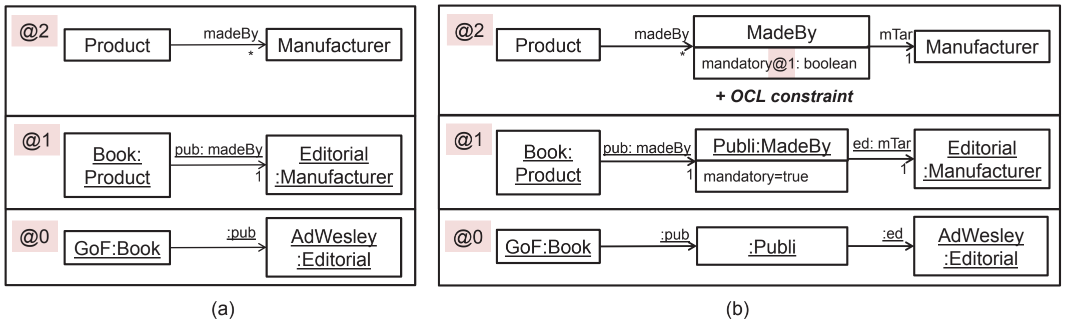


Fig. 17. (a) Multi-level solution to the *relation configurator* pattern. (b) Multi-level solution to the extended *relation configurator* pattern.

mandatory for the target objects to be pointed by the reference, then the multi-level solution needs to explicitly model these features in a meta-class, as shown in Figure 17(b) (clabject `MadeBy`). In addition, we need an OCL constraint at the top-most meta-level to take care of the semantics of the `mandatory` attribute. Having the constraint potency 2, it gets evaluated two levels below (see [de Lara et al. 2014a] for details on the working scheme of constraints in multi-level modelling). Thus, in general, multi-level modelling is most suitable when the features to configure are native for references (like cardinality), but otherwise, it requires their explicit modelling. Nonetheless, some multi-level meta-modelling frameworks (like [de Lara and Guerra 2010]) skip this problem by supporting attributed associations (though not attributed references).

Promotion. Figure 18 shows a solution using promotion. Similar to the *explicit reference modelling* approach, the meta-model (1) contains meta-classes that enable the definition of new `Product` and `Manufacturer` types. From the instances of this meta-model (model 2), a promotion transformation creates a meta-model (3) that contains a meta-class for each `ProductType` and `ManufacturerType`, and a reference for each `MadeByType` instance. The created references have the configured name, type and cardinality.

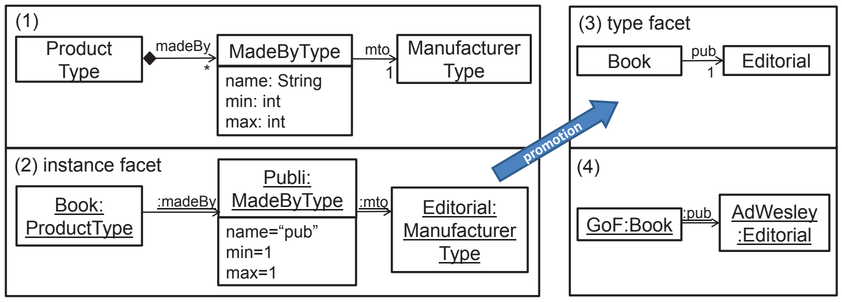
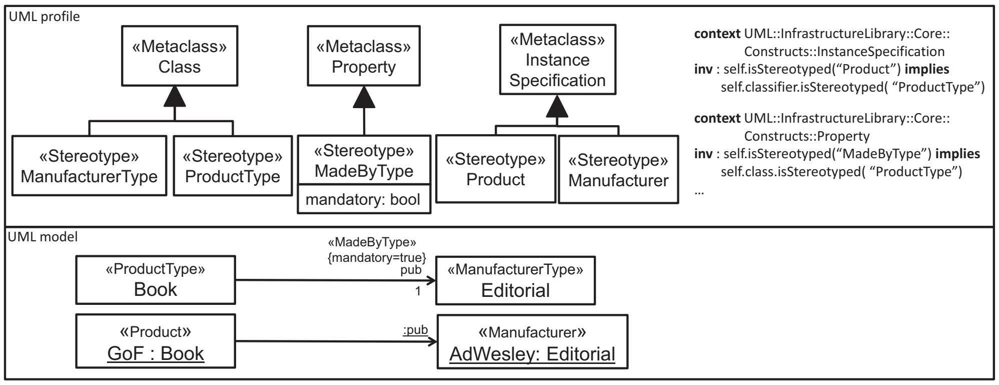


Fig. 18. Promotion solution to the *relation configurator* pattern.

The advantage of using promotion is that it removes complexity from the model (4), as there is no need to explicitly instantiate the `MadeByType` meta-class, but a native reference (`pub`) is used instead. This is an example of what we call *non-uniform* instantiation: if we see model (4) as an instance of model (2), we notice that while the instances of `Book` are objects, the instances of `Publi` are references. This flexibility in instantiation is gained by an appropriate encoding of the promotion transformation. In order to handle the extended *relation configurator* scenario (Figure 17(b)) using promotion, we should add the feature `mandatory` to the meta-class `MadeByType`, and modify the promotion transformation to generate OCL constraints in the meta-model (3) interpreting the value of the feature.

Powertypes. A solution based on powertypes would use the native instantiation capabilities of UML properties, having the same advantages and limitations as a pure multi-level solution (i.e., non-standard features of UML properties, like `mandatory`, must be explicitly modelled).

Stereotypes. As Figure 19 shows, we can create stereotypes for `Manufacturer` types and instances, `Product` types and instances and `MadeByType` links. The stereotype for `MadeByType` extends the UML `Property` meta-class; hence, we can configure the cardinality of any property stereotyped `MadeByType` (like `pub` in the figure) and use the native instantiation semantics of UML. In case we need to define a property `mandatory` to configure the relation, then it can be added to the definition of stereotype `MadeBy`, together with additional OCL constraints to check its semantics.



```
context UML::InfrastructureLibrary::Core::
        Constructs::InstanceSpecification
inv : self.isStereotyped(“Product”) implies
     self.classifier.isStereotyped( “ProductType”)

context UML::InfrastructureLibrary::Core::
        Constructs::Property
inv : self.isStereotyped(“MadeByType”) implies
     self.class.isStereotyped( “ProductType”)
...
```

Fig. 19. Solution to the *relation configurator* pattern using stereotypes.

Structure. Figure 20 shows the structure of the pattern using a multi-level approach. The relation is defined at the upper meta-level, and its cardinality indicates how many of their instances are allowed in the next meta-level. Then, when the relation is instantiated at level 1, each instance can define its own name and cardinality, used to constrain the lowest meta-level.

Instead, the *static references* solution merges the two upper meta-levels, and hence the definition and configuration of reference `dynRel` is performed as a regular reference; however, this solution does not allow creating new references dynamically. In the *explicit reference modelling* solution, the two lower meta-levels are merged, and so the upper meta-model needs to define meta-classes to allow declaring the relation configuration information and to instantiate the relation according to that configuration. This solution implies representing the relation instances as objects at the model level. The *promotion* solution also requires defining meta-classes to explicitly declare the relation configuration; however, in contrast to the *explicit reference modelling* approach, relations are represented as links (and not as objects) at the model level.

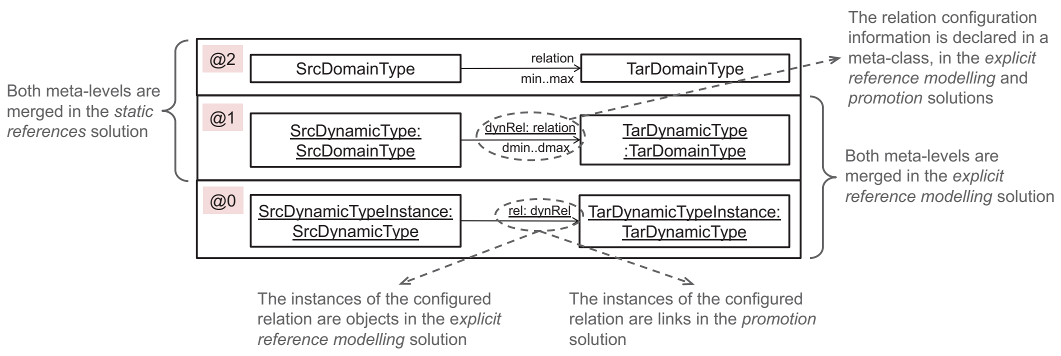

@2
SrcDomainType
relation
min..max
TarDomainType
@1
SrcDynamicType: SrcDomainType
dynRel: relation
dmin..dmax
TarDynamicType :TarDomainType
@0
SrcDynamicTypeInstance: SrcDynamicType
rel: dynRel
TarDynamicTypeInstance: TarDynamicType
Both meta-levels are merged in the static references solution
The relation configuration information is declared in a meta-class, in the explicit reference modelling and promotion solutions
Both meta-levels are merged in the explicit reference modelling solution
The instances of the configured relation are objects in the explicit reference modelling solution
The instances of the configured relation are links in the promotion solution

Fig. 20. Structure of the *relation configurator* pattern using multi-level modelling.

Conclusions. If the configurator is being used to customise typical features for relations, like cardinality, the simplest solutions (i.e., with less classes and not requiring coding) are multi-level modelling and stereotypes, this latter restricted to UML-based systems. However, if arbitrary features need to be configured, the multi-level solution may require to explicitly modelling association ends, adding some complexity. This is not the case for stereotypes, or if the multi-level framework supports attributed associations. Finally, promotion is the only solution enabling *non-uniform* instantiation.

4.5. Element classification

Intent. This pattern allows the organization of dynamically created elements along subtyping and inheritance hierarchies. Features defined in a dynamically created type are inherited by the child types, so that the features will be allocated in the instances of both the parent and the child types. Additionally, some types in the hierarchy may be abstract, meaning that there cannot be instances of such types.

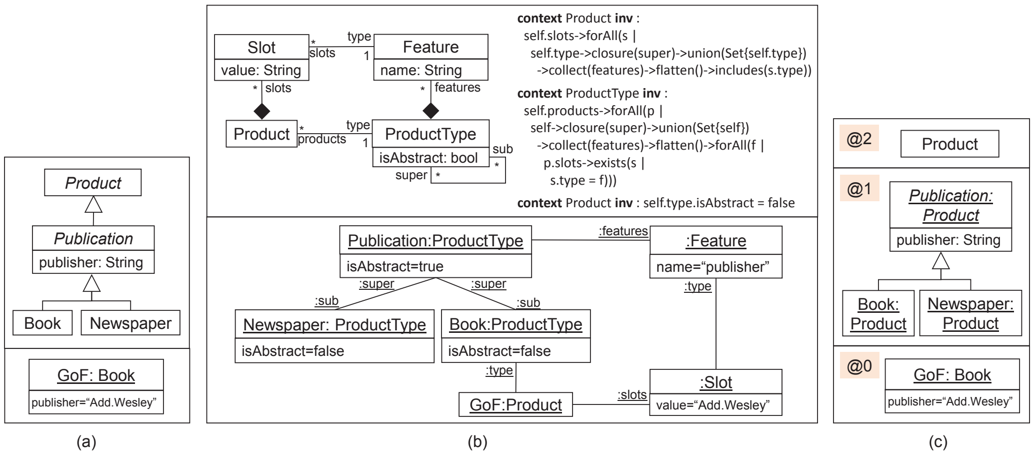
Also known as. None.

Motivation. Organizing elements in hierarchies allows the factorization of common features which can be reused by children. This also applies to hierarchies of types created dynamically (see *type-object* pattern). The optimization problem presented in Section 2 includes two instances of this pattern, as both dynamically created problem types and solution methods can be arranged hierarchically, as well as be abstract if they act just as holders of common attributes. Another example is the "product types" if we want to enable the definition of hierarchies of products with different characteristics, like publications, electronic devices, etc.

Solutions. We next describe six solutions for the "product types" example extended with hierarchies of product types.

Static inheritance. Figure 21(a) shows how to organize hierarchies of product types using static types and static inheritance relations at the meta-level. However, this is not a valid solution for the pattern as new product types and inheritance hierarchies cannot be created dynamically.

Explicit modelling of inheritance. Figure 21(b) explicitly models the inheritance relation as an association. The first two OCL constraints ensure that any product has slots for all features directly or indirectly defined in its type. The last OCL constraint checks that the type of products is not abstract. Additional constraints would be needed to ensure acyclic inheritance. As in the previous patterns, such an explicit



```
context Product inv :
  self.slots->forAll(s |
    self.type->closure(super)->union(Set{self.type})
      ->collect(features)->flatten()->includes(s.type))
context ProductType inv :
  self.products->forAll(p |
    self->closure(super)->union(Set{self})
      ->collect(features)->flatten()->forAll(f |
        p.slots->exists(s |
          s.type = f)))
context Product inv : self.type.isAbstract = false
```

Fig. 21. Solutions to *element classification* pattern. (a) Static inheritance. (b) Explicit modelling of inheritance. (c) Multi-level.

modelling results in more complex systems. However, the semantics of inheritance can be customized for the specific domain.

Multi-level. Figure 21(c) shows the multi-level solution. Due to the type facet of elements at the intermediate meta-level, it is possible to define inheritance relations between them, as well as to define abstract clabjects (like `Publication`). Thus, support for *element classification* is natively available at every meta-level. Moreover, this solution ensures that the type of the elements in the inheritance hierarchies is compatible [de Lara et al. 2014b]. That is, `Book` can inherit from `Publication` because both are instances of `Product` (indeed, the type of `Book` could also be a subtype of `Product`); otherwise, such an inheritance relation would be forbidden. The benefit is that hierarchies of incompatible types, which could give rise to safety problems, cannot be constructed.

Promotion. In this case, the starting meta-model (1) should explicitly model the inheritance relation between product types as an association (see Figure 11(b) for a reference), and the promotion transformation would generate proper inheritance relations from the instances of this association.

Powertypes. The solution for the *dynamic features* pattern using powertypes shown in Figure 12(a), is a valid solution for *element classification* as well. This is because UML natively supports the definition of UML class hierarchies and the correct allocation of slots in the class instances according to the defined hierarchies. However, it is not possible to constrain the type of the objects in these hierarchies. Hence, instances of the powertype `ProductType` can inherit from, and be child of, any arbitrary class, not necessarily other product types. As a consequence, safety-related problems may arise.

Stereotypes. As for powertypes, the solution previously proposed for *dynamic features* covers the *element classification* pattern (see Figure 12(b)). This is because inheritance is native between UML classes, so that it is possible to define inheritance relations between UML classes having the `ProductType` stereotype. Nonetheless, as a difference from powertypes, here we can attach OCL constraints to the stereotype definition in order to constrain the types appearing in inheritance hierarchies to be all (or none) product types.

Structure. Figure 22 shows the structure of the pattern using multi-level modelling. In the intermediate meta-level, the instances of `DomainType` have a type facet, and therefore, they can participate in inheritance relations or be abstract. The support for this is native in multi-level frameworks.

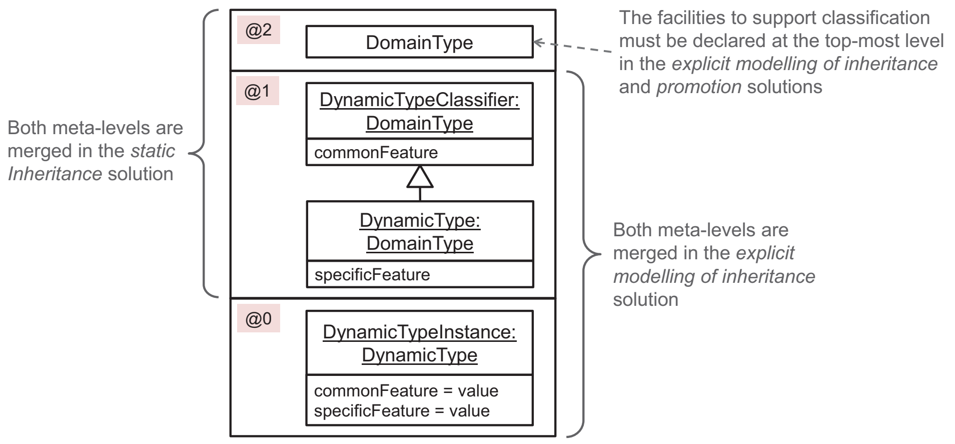

@2
DomainType
The facilities to support classification must be declared at the top-most level in the explicit modelling of inheritance and promotion solutions
@1
DynamicTypeClassifier: DomainType
commonFeature
Both meta-levels are merged in the static Inheritance solution
DynamicType: DomainType
specificFeature
Both meta-levels are merged in the explicit modelling of inheritance solution
@0
DynamicTypeInstance: DynamicType
commonFeature = value
specificFeature = value

Fig. 22. Structure of the *element classification* pattern using multi-level modelling.

In comparison, the *static inheritance* solution merges the two upper meta-levels, and classification is established by using the regular inheritance relation; however, new inheritance hierarchies cannot be created dynamically. The solution based on the *explicit modelling of inheritance* merges the two lower meta-levels instead, and so, the meta-model must explicitly define facilities to declare inheritance relationships and check that the instances contain slots respecting the inheritance semantics. In the case of *promotion*, the meta-model also needs to provide facilities to declare inheritance, but then, the promotion transformation can create proper inheritance relations in the resulting meta-model. *Powertypes* natively support classification, but it is not possible to restrict which types of elements can inherit from each other. *Stereotypes* support the definition of such restrictions by means of constraints.

Conclusions. The typesafe solutions for *element classification* use either explicit modelling of inheritance, promotion, multi-level modelling or stereotypes. Among them, the two latter are simpler because they do not require modelling the inheritance relation explicitly. However, explicit modelling and promotion allows to customize inheritance (e.g., to restrict it to single inheritance) and its semantics.

5. ASSESSING THE APPLICABILITY OF MULTI-LEVEL MODELLING: A FIELD STUDY

In this section, we present a field study with the aim of collecting existing meta-models that contain occurrences of the identified patterns, and therefore could have been modelled using the OCA and multi-level technology. The purpose is to analyse how frequently these patterns arise in practice, and which are the approaches most frequently adopted. We have examined more than 400 meta-models and UML profiles, gathered from four sources:

— The ATL meta-model zoo [AtlanEcore 2014], a collection of 305 meta-models contributed by the EMF meta-modelling community.
— The OMG [OMG 2014] specifications, which amount to 116 specifications from areas like modelling, finance, manufacturing and electronic commerce, among many others. These specifications are proposed by committees of professionals and researchers.

— The ReMoDD repository [ReMoDD 2014], which gathers heterogeneous MDE artefacts contributed by the modelling community, like model transformations, UML models, domain-specific models, and meta-models. In particular, it contains 15 Ecore meta-models.
— Scientific papers in renowned conferences and journals in the modelling area, including MODELS, ECMFA, SOSYM, ACM TOSEM, and IEEE TSE. Whenever possible, we have looked at the meta-model implementation, normally in Ecore format.

In this study, we only report on occurrences of at least the *type-object* pattern. That is, we do not report the occurrence of "shallow" versions of the *dynamic features* or *element classification* patterns. Moreover, we do not report on usages of potency-based multi-level solutions, as we did not find any repository of multi-level models beyond our own models built using METADEPTH [de Lara and Guerra 2010]. Our purpose is to assess the opportunities for using multi-level technologies in practice.

Interestingly, we have discovered that most occurrences of the patterns under study are in the following application areas:

— **Software architecture, components and services**. The *type-object* pattern is pervasive in this domain, as many modelling languages in this area model component types and component instances, together with component ports and port instances. In most cases, the approach followed is explicit modelling of types and instances. Prototypical examples include MetaH [Barbacci and Weinstock 1998] and CloudML [CloudML 2014]. Sometimes, components are instantiated, but not ports, like in UML Components. In any case, inheritance is seldom supported.
— **Business process/enterprise modelling**. Enterprise modelling languages often provide a stratified modelling approach, where a base meta-model is customized for different enterprises or project types. In this way, languages like DoDAF [DoDAF 2010] contain type-object pairs like OrganizationType/Organization, ProjectType-/Project, ActivityType/Activity and so on. The same approach is also common in process modelling languages, like ISO/IEC 24744 [González-Pérez and Henderson-Sellers 2007], where there is the need to define task types (e.g., Testing) which are used (i.e., instantiated) in specific project plans. Conceptually, a common solution in this area is the use of powertypes, though for implementation they resort to explicit modelling (like in UEML [Bergholtz et al. 2005]) or stereotypes (like in the UPDM OMG profile [OMG 2013j]). We have observed an intense occurrence of patterns in this domain, i.e., many different occurrences of the patterns in the same meta-model, which suggests that many problems in this domain are intrinsically multi-level.
— **Software and systems modelling**. In software design, it is often necessary to describe both types and sets of interacting instances. A prominent example is the UML [OMG 2011g] and its class/object diagrams. The most usual approach in these cases is their explicit modelling.
— **Data modelling and data management**. Some languages allow describing the structure of data and build data samples. The prototypical case is the Common Warehouse Meta-Model (CWM) [OMG 2003], but other similar cases exist, like the Historical Data Access from Industrial Systems Specification (HDAIS) [OMG 2005e], or the Records Management Service (RMS) for government data management [OMG 2011c].

Other areas where we have found occurrences of the analysed patterns are reverse reengineering (in the Knowledge Discovery Metamodel [OMG 2011a] and the Abstract Syntax Tree Metamodel [OMG 2011e] OMG standards), requirements engineering (most notably in the Requirements Interchange Format OMG specification [OMG

2013c]), bibliographic data (most notably in the Bibliographic Query Service OMG specification [OMG 2002a]), metrics (in the Structured Metrics Meta-Model OMG specification [OMG 2012e]), software testing (in the UML testing profile [OMG 2013k]), user interfaces (UsiXML [UsiXML 2014]), traceability (TML [Drivalos et al. 2008] and the Traceability Metamodel TmM [Espinoza and Garbajosa 2011]) and feature-oriented development (fmp [Czarnecki et al. 2005]).

Tables I, II, III, and IV detail the analysed meta-models and profiles where we have found occurrences of the patterns described in Section 4. The first two columns of the tables show the name of the meta-model or profile and the adopted approach (Static, Explicit, ENumerated types, Promotion, PowerType, StereoType, or meta-model EXtension[8]). Columns 3–7 summarize the number of occurrences of each pattern. Note that, for columns 4–7, we only indicate the occurrences where the *type-object* pattern occurs as well. While this is not strictly necessary for some of the patterns (e.g., one can define data structures using *element classification* without explicitly instantiating the structures), our goal is to assess the applicability of multi-level modelling in practice, which revolves around the *type-object* pattern. The last column of the tables shows the following meta-model size metrics: number of classes (*c*), number of references (*r*, for Ecore-based meta-models), number of associations (*a*, for UML or CMOF-based meta-models), and number of stereotypes (*s*, for UML profiles). In most cases, the size was automatically calculated from the *xmi* file with the meta-model; however, such a computer-processable file was not available in a few cases, where we had to do the counting by hand (these cases are marked with "*m*"). Finally, in two cases [Ceri et al. 2009; González-Pérez and Henderson-Sellers 2007], the metrics were taken on the meta-models as published in research papers, even if they were fragments of a bigger meta-model, which we could not access.

While we resorted to automatic means for computing metrics, the assessment of pattern occurrences was done manually. We used a systematic approach, where each pattern occurrence found by one author was reassessed by at least another author. This leads to a conservative reporting of occurrences. In the case of OMG standards, we went through the specifications accessing the profiles or meta-models in *xmi* format whenever possible. The assessment of other meta-models (e.g., from the ATL zoo) was sometimes complemented by reading published papers with the description of the meta-models.

More detailed tables, which include the main classes participating in the pattern occurrences and pointers to the meta-models, can be found at `http://miso.es/multilevel/multi-level-patterns.htm`.

From the analysed meta-models, 84 contain at least one occurrence of the *type-object* pattern. Out of these cases, roughly 69% use the explicit modelling approach, 10% static inheritance, 8% stereotypes, 8% enumerated types, 3% powertypes, and 2% promotion. Solutions based on powertypes were only found in the traceability and process/enterprise modelling domains.

Regarding the rest of the patterns, 30 meta-models contain at least one occurrence of the *dynamic features* pattern, 5 meta-models contain the *dynamic auxiliary domain concepts* pattern, 14 the *relation configurator* pattern, and 11 the *element classification* pattern. The areas where we found more pattern instances are software architecture (17 meta-models), enterprise/process modelling (19 meta-models) and systems/-software modelling (13 meta-models).

Concerning the total number of occurrences of patterns, we found 459 occurrences in total, distributed among 84 meta-models, with an average of 5.5 pattern occurrences per meta-model or profile. More specifically, we found 363 occurrences of the *type-object*

[8]This approach involves direct extension of the Ecore meta-meta-model. We only found this technique once.

Table I. Meta-Models in the Software Architecture Domain. The second column indicates the followed approach: S (Static Inheritance), E (Explicit Modelling), EN (Enumerated Types), ST (Stereotypes).

Name	**App.**	**Type Object**	**Dyn. Feats.**	**Domain Concepts**	**Relation Config.**	**Element Classif.**	**Meta-Model Size**
ACME [AtlanEcore 2014]	E	**1**	**1**	–	–	–	16c, 13r
ARCAS [Google Code]	E	**2**	**2**	–	–	–	40c, 71r
ArchiMeDeS [Sanz and Marcos 2012]	E	**1**	–	–	–	–	25c, 34r (m)
CCMP [OMG 2005d]	E	**5**	**3**	–	–	–	20c, 19r (m)
	ST	**2**	**1**	–	–	–	19s (m)
CloudML [CloudML 2014]	E	**6**	**1**	–	**1**	–	33c, 36r
EDOC [OMG 2004a]	E	**2**	**2**	–	–	**2**	75c, 53a (m)
MARTE [OMG 2011f]	ST	**4**	**1**	–	–	–	158s, 56a
	EN	**5**	–	–	–	–	
MetaH [AtlanEcore 2014]	E	**4**	**2**	–	–	–	14c, 8r
ProMARTE [AtlanEcore 2014]	E	**4**	**1**	–	–	–	108c, 142r
QFTP [OMG 2008c]	E	**5**	**1**	–	–	–	29c, 50a
	ST	**2**	–	–	–	–	67s
SAM [ReMoDD 2014]	E	**4**	–	–	–	–	48c, 56r
SoaML [OMG 2012b]	ST	**1**	**1**	–	–	–	27s
SPTP [OMG 2005c]	E	**3**	–	–	–	–	88c, 121r (m)
SysML 1.2 [OMG 2012c]	S	**2**	–	–	–	–	47s, 6a
	E	**2**	**1**	–	–	–	66c, 27r
UML2 Components [OMG 2011g]	E	**1**	**1**	–	**1**	**1**	143c, 180a
UML2 Deployment [OMG 2011g]	E	**2**	**1**	–	–	–	157c, 190a
Wright [Fourati 2010]	E	**2**	–	–	–	–	19c, 24r (m)

Table II. Meta-Models in the Business Process/Enterprise Modelling Domains. The second column indicates the followed approach: E (Explicit Modelling), EN (Enumerated Types), PT (Powertypes), ST (Stereotypes).

Name	**App.**	**Type Object**	**Dyn. Feats.**	**Domain Concepts**	**Relation Config.**	**Element Classif.**	**Meta-Model size**
Agate [AtlanEcore 2014]	E	**3**	–	–	–	–	69c, 123r
Ant [AtlanEcore 2014]	E	**1**	**2**	–	–	–	48c, 28r
BPDM [OMG 2008b]	E	**4**	**1**	–	**1**	**1**	152c, 139a
BPMN 2.0.1 [OMG 2013f]	E	**12**	**3**	**1**	–	–	137c, 193a
Intalio BPMN 1.1 [AtlanEcore 2014]	EN	**1**	–	–	–	–	18c, 31r
BPMNProfile [OMG 2013b]	ST	**12**	**1**	**1**	**1**	–	130s, 179a
CMMN [OMG 2013g]	E	**2**	**1**	–	–	–	30c, 65a
DeclarativeWorkflow [ReMoDD 2014]	E	**4**	–	–	–	**1**	39c, 31r
DoDaF 2.02 [DoDAF 2010]	PT	**35**	–	–	–	–	130c, 135a (m)
DT4BP [ReMoDD 2014]	E	**2**	**1**	–	–	–	86c, 73r
ISO/IEC 24744 [González-Pérez and Henderson-Sellers 2007]	PT	**5**	–	–	–	–	19c, 5r (m)
ITPMF [OMG 2007]	E	**13**	–	–	–	–	49c, 61a
Promenade [AtlanEcore 2014]	E	**1**	–	–	–	–	18c, 22r
REA (Resource-Event-Agent) [Google code]	E	**6**	–	–	–	–	23c, 57r
SPEM [OMG 2008a; AtlanEcore 2014]	E	**5**	–	–	–	–	302c, 380a
UEML [AtlanEcore 2014; Bergholtz et al. 2005]	E,PT	**3**	–	–	–	–	23c, 27r
UML2 Activities [OMG 2011g]	E	**1**	–	–	–	–	228c, 274a
UPDM 2.1 [OMG 2013j]	ST	**25**	**3**	–	–	–	226s
XPDL [AtlanEcore 2014]	EN	**1**	–	–	–	–	52c, 66r

pattern (average of 4.3 occurrences per meta-model), 59 of the *dynamic features* (0.7 occurrences per meta-model), 5 of the *dynamic auxiliary domain concepts* (0.06 occurrences per meta-model), 14 of the *relation configurator* (0.17 occurrences per meta-model) and 18 of the *element classification* pattern (0.21 occurrences per meta-model).

If we classify the pattern occurrences by application domain, we have 153 occurrences in the enterprise/process modelling domain, 85 in the software architecture domain, and 48 in the systems/software modelling domains. Hence, our patterns are frequent in software architecture and enterprise/process modelling, and in particular, in this latter domain they occur intensely. Thus, the greatest benefits from the use of

Table III. Meta-Models in the Software/Systems Modelling Domain. The second column indicates the followed approach: S (Static Inheritance), E (Explicit Modelling), EN (Enumerated Types), P (Promotion), EX (Meta-Model Extension).

Name	**App.**	**Type Object**	**Dynamic Features**	**Domain Concepts**	**Relation Config.**	**Element Classif.**	**MM size**
AbstractSyntaxStereotypes [AtlanEcore 2014]	E	**1**	–	–	**1**	–	9c, 16r
Domain meta-model [Gallardo et al. 2012]	P	**1**	**1**	–	**1**	–	11c, 16r
DSLModel [AtlanEcore 2014]	E	**2**	**1**	–	–	–	14c, 15r
EMF Profiles [Langer et al. 2012]	EX	**1**	–	–	**1**	–	9c, 7r (m)
FUML [OMG 2013h]	E	**3**	**2**	–	–	**1**	223c, 199a
LQN 1.0 [AtlanEcore 2014]	EN	**2**	–	–	–	–	12c, 40r
Matlab/ Simulink [AtlanEcore 2014]	E	**1**	**1**	–	–	–	45c, 75r
MOF 2.4.1 [OMG 2013i]	E	**3**	**1**	–	**1**	**1**	16c, 7a
OCL [OMG 2012d]	S	**1**	–	–	–	–	48c, 37a
SCADE [AtlanEcore 2014]	E	**1**	**1**	–	–	–	106c, 231r
SMOF 1.0 [OMG 2013e]	E	**2**	**1**	–	**1**	**1**	19c, 16r
UML2 Classes [OMG 2011g]	E	**3**	**3**	–	**1**	**1**	118c, 147a
UML2 Composite Structures [OMG 2011g]	E	**1**	**3**	–	**1**	**1**	137c, 171a

multi-level technology are expected in these domains. We will illustrate how to rearchitect existing solutions into multi-level solutions in Section 6.

Looking at individual repositories, the ATL zoo contains 305 meta-models, out of which 9 are Ecore versions of OMG standards (e.g., CWM, SPEM and MARTE). In the latter case, the tables show a reference to the standard, and we count them as OMG standards. Out of the 296 remaining meta-models, 25 (8.4%) include at least one occurrence of the *type-object* pattern. Most occurrences (70.4%) use explicit modelling, some use enumerated types (22.2%), and only a few (7.4%) use a static approach. In every case, a potency-based multi-level approach could have been used instead. Static approaches are the most difficult to detect, as one has to discern whether a certain inheritance hierarchy is incomplete and whether it makes sense to have an extensible number of instances for the leaf classes. We were conservative in this respect. For example, a prototypical case is SWRC, which introduces a set of incomplete hierarchies for publications, events, people, topics and products. However, recall that the static approach may lead to the redefinition of associations, and this is not supported by Ecore. A similar criterion was taken to signal the occurrences of the enumerated types.

The second analysed repository, ReMoDD, is considerably smaller than the ATL zoo. ReMoDD contains 15 meta-models, out of which 3 have some pattern occurrence (20%), all of them in the software architecture and enterprise/process modelling domains.

Since these two repositories contain representative samples of the meta-models used in practice, we can conclude that, even if multi-level scenarios are less common than a pure two meta-level setting, they need to be considered for a number of particular domains. This is especially acute in the software architecture and process modelling domains.

We have also reviewed 116 OMG specifications, out of which 41 (35%) contain some occurrence of the *type-object* pattern. This percentage is even higher if we discard those specifications (10) that only contain CORBA interfaces but no meta-model or profile, in which case we obtain 41 out of 106 (38.7%). Regarding the approach, 69.4% use explicit modelling, 16.3% static inheritance, 14.3% stereotypes, and 4% enumerated types. In this case, the identification of static approaches was easier because many meta-models are based on UML (which supports association redefinition, a "smell" of this situation) and sometimes they are defined as base languages expected to be subclassified. We will further discuss this issue in Section 6.

Table IV. Meta-Models in the Traceability, Data Management, Manufacturing, Software Development and other Domains. The second column indicates the followed approach: S (Static Inheritance), E (Explicit Modelling), EN (Enumerated Types), P (Promotion), PT (Powertypes), ST (Stereotypes).

Name	**App.**	**Type Object**	**Dyn. Feats.**	**Doma. Conce.**	**Relation Config.**	**Elem. Classif.**	**MM size**
Traceability							
TML [Drivalos et al. 2008]	P	**1**	–	**1**	**1**	–	5c, 7r (m)
TmM [Espinoza and Garbajosa 2011]	PT	**5**	–	–	–	–	24c, 14r (m)
Data modelling and data management							
CWM [OMG 2003]	E	**6**	**1**	**1**	–	**1**	183c, 289r
Mining Mart (Case representation)[AtlanEcore 2014]	EN	**1**	–	–	–	**1**	31c, 21r
DAIS [OMG 2005a]	E	**3**	–	–	–	–	29c, 75a (m)
HDAIS [OMG 2005e]	E	**2**	–	–	–	–	13c, 11a (m)
RMS [OMG 2011c] (governamental)	E	**7**	**1**	–	**1**	–	137c, 128a
Manufacturing and construction							
Express/STEP [AtlanEcore 2014; OMG 2010]	E	**1**	**2**	–	–	**3**	185c, 299r
ifc2x3 [AtlanEcore 2014]	E EN	**1** **34**	– –	– –	– –	**1** –	699c, 592r
PLM [OMG 2011b] (manufacturing)	E	**1**	**1**	–	–	**1**	148c, 53a
Software reengineering and metrics							
ASTM [OMG 2011e] (reverse engineering)	S	**1**	–	–	**1**	–	193c, 127r
KDM [OMG 2011a] (reverse engineering)	S E EN	**10** **1** **5**	– **2** –	– – –	**1** **1** –	– – –	300c, 256r
Measure 2.0 [AtlanEcore 2014] (software metrics)	E	**1**	–	–	**1**	–	8c, 11r
Metrics [AtlanEcore 2014] (software metrics)	E	**1**	–	–	**1**	–	6c, 1r
SMM [OMG 2012e] (software metrics)	E	**1**	**1**	–	–	–	57c, 100r
Software development							
EAI [OMG 2004b] (enterprise application integration)	E S	**2** **1**	– –	– –	– –	– –	91c, 63a (m)
Mantis [AtlanEcore 2014] (bug tracking)	EN	**1**	–	–	–	–	10c, 16r
RAS [OMG 2005b] (reusable software assets)	E	**1**	–	–	–	–	36c, 57r (m)
ReqIF [OMG 2013c] (requirements)	E	**4**	**1**	–	**1**	–	48c, 51a
Reqtify [AtlanEcore 2014] (requirements)	E	**1**	**1**	–	–	–	11c, 13r
RequisitePro 0.1 [AtlanEcore 2014] (requirements)	E	**1**	**1**	–	–	–	10c, 9r
Software Quality Control 1.1 [AtlanEcore 2014]	E	**1**	–	–	–	–	6c, 7r
UML Testing Profile [OMG 2013k] (testing)	E,ST	**3**	**1**	–	–	–	31s, 13a
UsiXML [AtlanEcore 2014] (user interfaces)	E	**2**	**1**	–	–	–	73c, 36r
Other Domains							
Bibliographic Query Service (BQS 1.0) [OMG 2002a]	S	**2**	**1**	–	–	–	20c, 12r (m)
Diagram Definition [OMG 2012a]	S	**5**	–	–	–	–	26c, 19r
Feature Modelling Plugin [Czarnecki et al. 2005]	E	**2**	**1**	–	–	–	8c, 16r
HAL [AtlanEcore 2014] (bibliographic data)	E	**2**	–	–	–	–	42c, 16r
NEG [OMG 2002b] (electronic commerce)	S	**6**	–	–	–	–	81c, 45a (m)
openEHR [openEHR 2014] (health care)	E	**1**	–	**1**	–	–	219c, 105a
SACM [OMG 2013d] (systems assurance)	S E	**4** **3**	– –	– –	– –	– –	125c, 66r
Simple Web Services Connection [AtlanEcore 2014]	S	**1**	–	–	–	–	5c, 2r
SOPES [OMG 2011d] (C4/i)	E	**14**	–	–	–	–	740c, 1440a
SWRC 1 [AtlanEcore 2014] (bibliographic data)	S	**5**	–	–	–	–	55c, 68r
WebML [Ceri et al. 2009] (web engineering)	E	**2**	–	–	–	–	6c, 9a (m)

Compared to the ATL zoo, we observe a much higher number of occurrences of multi-level patterns in the OMG specifications. This is mainly due to three factors: customizability, generality and size of the OMG specifications. Customizability refers to the need to have user-defined types in models, in addition to the predefined ones: when the user-defined types need to be instantiated, the *type-object* pattern comes into play. By generality we mean that many specifications are to be used by a wide community of interest, which leads to including extensibility elements, frequently realised using the *type-object* pattern. Finally, meta-models proposed by the OMG tend to be large. In contrast, some meta-models in the ATL zoo were toy examples. Altogether, we can conclude that multi-level patterns are more likely to occur in large, general, extensible modelling languages, covering many facets of a domain.

6. REARCHITECTING INTO MULTI-LEVEL SOLUTIONS, AND OPEN CHALLENGES

The previous field study makes evident that multi-level patterns occur in practice. In this section, we illustrate how a multi-level modelling framework can be used to address the different solutions found, and discuss the benefits this provides. In this process, we also identify a number of challenges that multi-level modelling tools should address in order to provide even further flexibility to the proposed solutions.

6.1. Explicit modelling approach

Explicit modelling is by far the most common approach, likely because apart from promotion, it is the only one available in widely used meta-modelling frameworks like EMF. Promotion-based solutions are more elaborate but seldom used. Usually, the relation between a type and its objects is explicitly modelled using a reference, but in some cases, identifiers (e.g., `String` attributes) are used instead. The next subsection shows how to migrate a meta-model with occurrences of the *type-object* pattern implemented using explicit modelling, into a multi-level solution.

6.1.1. Rearchitecting to multi-level. Figure 23 depicts the strategy to rearchitect a meta-model with occurrences of the *type-object* pattern expressed using explicit modelling (the original two-level solution is shown to the left, and the resulting multi-level solution to the right). The original meta-model is converted into a meta-model with potency 2, and a class with potency 2 is added (`DomainType`) for each pair of classes (`DomainType` and `DomainInstance`) in each type/object pattern occurrence. The class will define the features of the type-class with potency 1 (`typeFeature`), and the features of the instance-class with potency 2 (`instanceFeature`).

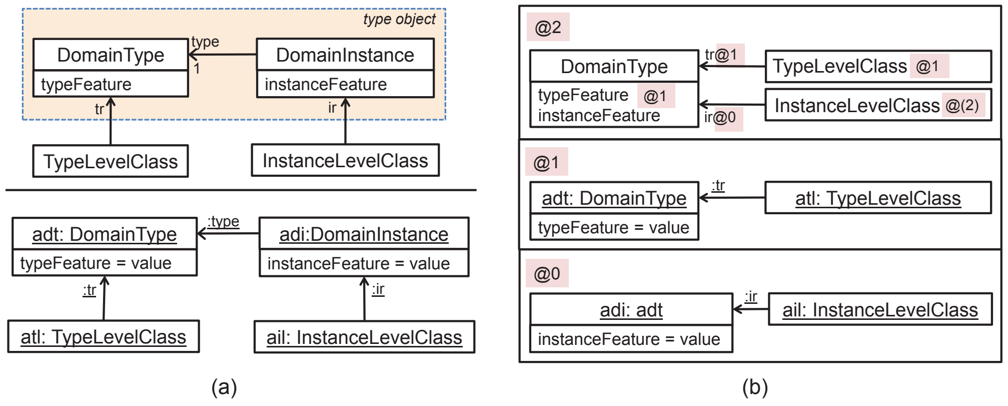


Fig. 23. Rearchitecting an explicit modelling approach into a multi-level solution. (a) Scheme of the type-object pattern in the explicit modelling approach. (b) Refactored multi-level model.

Then, the rest of classes of the original meta-model need to be organized depending on their relation with `DomainType` and `DomainInstance`. The classes connected to `DomainType` are added with potency 1 to the meta-model, as they only need to be used in the intermediate meta-level. The classes connected to `DomainInstance` are added with potency 2 to the meta-model, to allow their instantiation at the lowest meta-level.

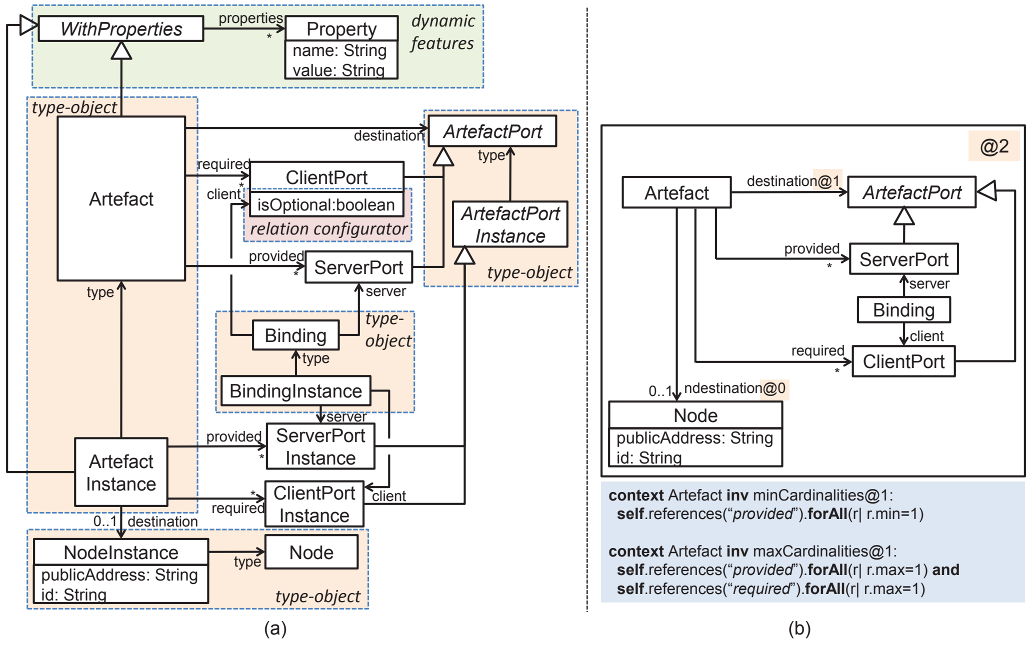
There are two important additional details. First, `InstanceLevelClass` objects are not needed at the intermediate meta-level, but only at potency 0. To this aim, the class is assigned a so-called *leap potency* [de Lara et al. 2014b] (shown between parenthesis), which is a shortcut that permits instantiating the class directly two meta-levels below, without instantiating it at the intermediate level. Second, the suffix `@0` in the reference `ir` indicates that it is a deep reference [de Lara et al. 2014b] that can connect instances of `InstanceLevelClass` to instances of `DomainType` at level 0.

In case a class is connected to both `DomainType` and `DomainInstance` in the original meta-model, there are several possible migration strategies. One possibility is to assign potency 2 to such a class. Another option is to define the class in a separate meta-model with potency 1, and declare references to `DomainType` and deep references to its instances.

Altogether, rearchitecting to multi-level yields a simpler meta-model (one class and one reference less for each occurrence of the *type-object* pattern), and it pays off if there are several interconnected occurrences of the pattern, or they occur together with other patterns, especially the *dynamic features* one, which is obtained "for free" in the multi-level approach, but requires complex structures in explicit modelling. The next section applies this strategy to an example found in our field study, and comments on other mechanisms found in the study.

6.1.2. Examples from the field study. In the software/system architecture domain, the *type-object* pattern is pervasive, typically implemented using explicit modelling. CloudML is a prototypical example of this situation, as its meta-model in Figure 24(a) shows. This language allows defining `Artefact` types equipped with `ClientPort` and `ServerPort` types, all of which can be instantiated and deployed in `NodeInstances`, which are also typed. All meta-classes inherit from `WithProperties` (though we only show some inheritance relations for clarity). This allows the addition of *dynamic features* to every element. At the type level, `ClientPorts` can be declared optional, which is an incarnation of the *relation configurator* pattern.

As we discussed in Section 4.1, using a multi-level approach avoids replication of concepts for types and instances, leading to simpler meta-models. Figure 24(b) shows a multi-level definition for CloudML, obtained using the strategy presented in the previous subsection. Thus, the pair of classes in each occurrence of the *type-object* pattern has been merged into a class with potency 2. The attributes `publicAddress` and `id` have potency 2 because they were defined in `NodeInstance`. Reference `destination` from `ArtefactInstance` to `NodeInstance` in the left meta-model, is converted into the reference `ndestination` from `Artefact` to `Node` in the multi-level meta-model. The suffix `@0` indicates that `ndestination` is a deep reference, which can connect instances of `Artefact` and `Node` at level 0, but not at level 1, hence preserving the semantics given by the left meta-model. The *dynamic features* pattern is natively supported by the multi-level solution: properties for types can be defined at level 1 and receive a default value, whereas properties for instances can be defined in the instances' types and receive a value at level 0. Figure 25(b) shows an example for the definition of the dynamic feature `count` in the artefact `cons`. Finally, the *relation configurator* pattern in `ClientPort` allows setting a port to optional. This is achieved in the multi-level solution by assigning a minimum cardinality 0 to the incoming `required` relations of optional ports at level 1. In this way, the client port `cp` in Figure 25(b) is optional while `c` is mandatory. This optionality is not allowed for `provided` ports, which we control using the OCL constraint `minCardinalities`. The constraint uses METADEPTH's syntax, where the keyword *references* returns the instances of a given reference [de Lara et al. 2014a]. The second OCL constraint enforces all port types declared by an artefact type to be instantiated at most once in each artefact instance. Please note that we have



```
context Artefact inv minCardinalities@1:
  self.references("provided").forAll(r| r.min=1)

context Artefact inv maxCardinalities@1:
  self.references("provided").forAll(r| r.max=1) and
  self.references("required").forAll(r| r.max=1)
```

Fig. 24. (a) Excerpt of the CloudML meta-model. (b) Multi-level meta-model for CloudML.

omitted all integrity constraints from the original meta-model in Figure 24(a), which control the consistency of artefact ports and port instances and their optionality. It can be observed that the resulting meta-model is much simpler than the original one (6 clabjects vs. 14 classes).

Figure 25 shows the instantiation of the meta-models in Figure 24 for the same example, the left one being an instance of the original definition of CloudML, and the right one of the multi-level definition. The latter solution separates types and instances in two different meta-levels. This neat separation facilitates the reuse of the models built at the intermediate meta-level. Moreover, it reduces the complexity the designer has to face when building a model, as in each level he is only concerned with the instantiation of elements of the upper meta-level: either types (level 1) or instances (level 0). In contrast, this distinction is unclear when building the solution in Figure 25(a). As explained before, the optionality of `ClientPorts` is controlled through cardinalities in the multi-level solution. In addition, our intention was to declare the property `count` in the artefact `cons` to be instantiated in all `cons` instances. While this cannot be achieved in solution (a) because there is no relation between the properties declared by types and their instances, it is natively supported in solution (b), where in addition a native data type can be assigned to the property.

Finally, it is worth noting that, in several cases in the field study, we observed the need for model libraries and cloning mechanisms, combined with the explicit approach. Libraries offer a way of cloning a certain set of elements (which conceptually "belong together") into the model. For its effective use in tools, advanced mechanisms for cloning, referencing the original objects of the library, are required. For example, in the OMG's Structured Metrics Meta-model (SMM) [OMG 2012e], it is possible to define libraries of measures like module count, lines of code, McCabe or cyclomatic complexity. Actually, the Automated Function Points (AFP) specification [OMG 2013a] is an instance of SMM (a library). In the case of CloudML, libraries could be used to emulate the

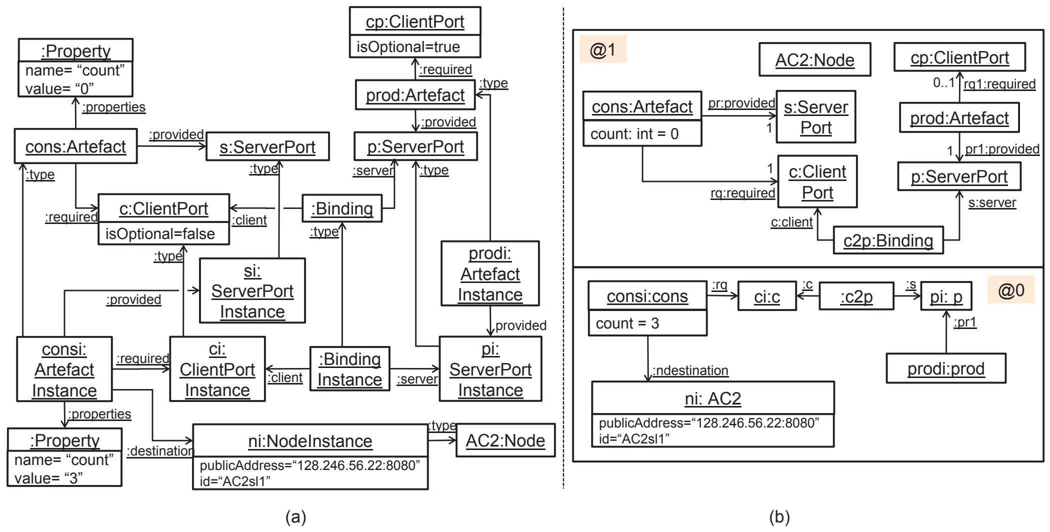


Fig. 25. (a) CloudML model. (b) Multi-level CloudML model.

instantiation of predefined artefact types with their port configurations. In the case of multi-level solutions, a similar mechanism for the instantiation of several elements at a time would be needed instead.

6.2. Static types approach

The static types approach yields static inheritance hierarchies. By rearchitecting into multi-level, the static subclasses can become dynamic objects and be created at run-time. Moreover, workarounds like reference and attribute redefinitions, which are typically unsupported by meta-modelling frameworks, can be avoided due to the dual type/object facet of elements.

6.2.1. Rearchitecting to multi-level. Figure 26 shows how to rearchitect a static implementation of the *type-object* and *relation configurator* patterns, into a multi-level solution. A meta-model with potency 2 is created, containing the base abstract classes (e.g., `DomainType1`), though they are defined concrete. The classes declare the type-level features with potency 1, and the instance-level features with potency 2. In this way, the instance classes in the original meta-model (e.g., `DomainInstance1`) can be created in a model with potency 1 as instances of the classes in the top-most meta-level. Finally, the lower models remain equal in the original and the refactored systems.

6.2.2. Examples from the field study. Some OMG specifications are intended to be used by extending from certain base meta-classes, following a static approach. This is the case of the Diagram Definition (DD), the Semantics of Business Vocabulary and Rules (SBVR), the Knowledge Discovery Meta-model (KDM) and the Abstract Syntax Tree Meta-model (ASTM). All of these cases could be refactored into multi-level solutions.

The DD specification is used to define the concrete syntax of languages by static extension of a base meta-model. The specification includes an example of extension for the case of class diagrams. The Date-Time Vocabulary (DTV) extends SBVR to define concepts related to date and time. In both cases, it is necessary to redefine associations and the workarounds described in Section 4.

ASTM includes a core set of modelling elements – called the Generic Abstract Syntax Meta-Model (GASTM) – common to many programming languages. Extensions

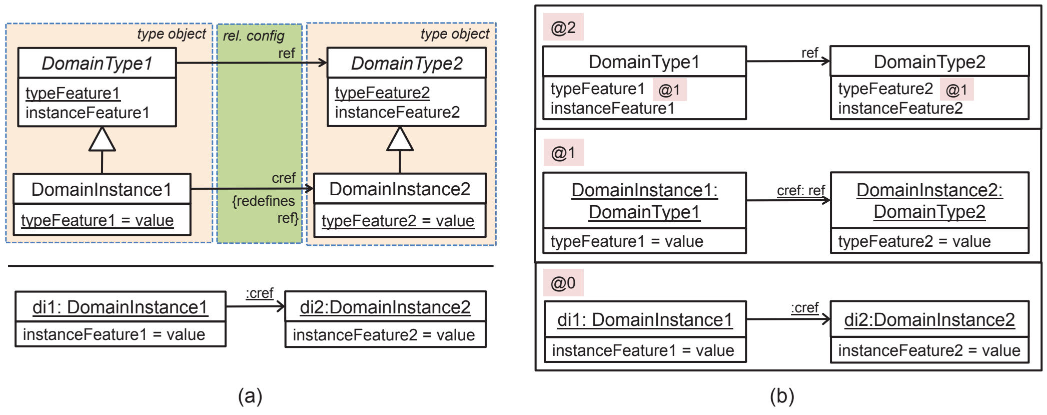


Fig. 26. Rearchitecting a static approach into a multi-level solution. (a) Scheme of the type-object and relation configurator patterns in the static approach. (b) Refactored multi-level model.

to GASTM – called Specialized Abstract Syntax Meta-Models (SASTMs) – can be defined for particular programming languages by extending classes of the GASTM, like `Definition`, `DataType` and `Statement`. Similarly, in reverse engineering, KDM is defined atop a core set of meta-classes: `KDMEntity`, `KDMRelationship` and `KDMModel`. Then, a number of packages are defined by extending those classes: `Inventory`, `Code`, `Action`, `Platform`, `UI`, `Event`, `Data`, `Structure`, `Conceptual` and `Build`. In order to define other extensions, the standard prescribes a way (called the Framework Extension meta-model pattern, see page 35 of [OMG 2011a]) based on naming conventions and on a specific structure: the existence of certain base classes that extend `KDMModel`, `KDMEntity` and `KDMRelationship`, and subsetting the associations among them. Figure 27(a) shows a very simplified example for the definition of the `Code` package through extension. It can be observed that extending KDM requires the redefinition of associations (see `from` and `to`), which is not possible in EMF.

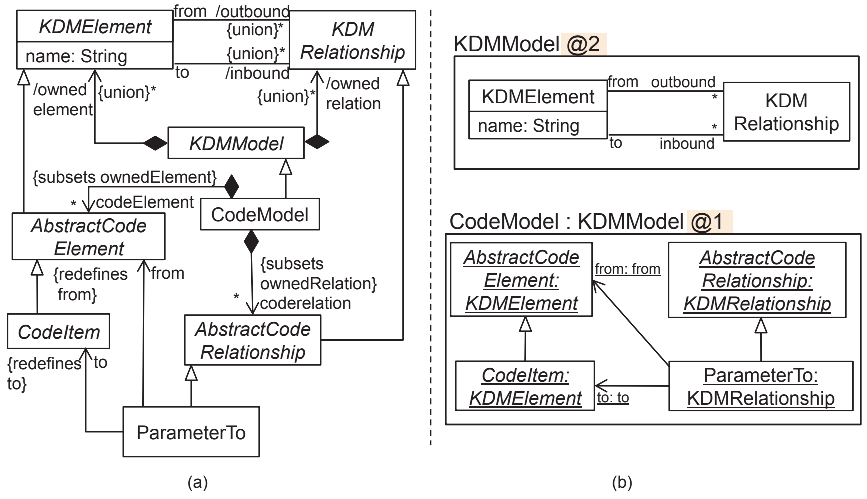


Fig. 27. (a) A small excerpt of KDM (core and code packages). (b) Multi-level solution.

Figure 27(b) shows an alternative solution using multi-level modelling, built using the rearchitecting guidelines given in the previous subsection. In this solution, the core meta-classes are defined at the higher meta-level, and the different packages are instances of these. In this way, there is no need to resort to association redefinition or association unions as in the static approach. The use of full-fledged models at every meta-level makes also unnecessary the use of container classes like `KDMModel` and `CodeModel`. Moreover, this solution does not rely on naming conventions like the Framework Extension meta-model pattern, but instantiation provides a more guided and dynamic way to specialize the KDM core. Finally, another advantage is that the resulting model at level 1 is simpler than the one in Figure 27(a), and therefore more likely to be simpler to understand and instantiate by end users.

In our study, we detected meta-models with rigid inheritance hierarchies which could be rearchitected in a multi-level setting to gain flexibility and extensibility capabilities. This is the case of markup languages which contain a fixed set of markings (e.g., KML or HTML), and meta-models for bibliographic data with fixed sets of document kinds, publication venue kinds, and so on. The multi-level solution would refactor the language definition in two levels, and provision a library with standard concepts as instances of the top-level definition. This library would be extensible by instantiation of the top meta-level types. In the case of meta-models for markup languages, this would be similar to the relation between the Standard Generalized Markup Language (SGML [Goldfarb 1991]) and its applications, like the HTML.

6.3. Enumerated types approach.

The enumerated types approach is similar to the static one, but it uses an enumeration instead of static subclasses to define the possible types. Rearchitecting to a multi-level solution enables the dynamic creation of types, which would not need to be predefined in the enumeration literals.

6.3.1. Rearchitecting to multi-level. The strategy to migrate into a multi-level solution is similar to the static type approach. In this way, the enumerated type is eliminated, and its literals are dynamically created at the meta-level below. Enumerated types do not define features, and hence, less gain is obtained by their refactoring into a multi-level solution. Nonetheless, the benefit is the possibility to have extensibility without changing the meta-model, as well as richer descriptions of types.

6.3.2. Examples from the field study. We found occurrences of the enumerated types approach both in the ATL zoo and the OMG specifications. An indication for the need of extensibility was having in the enumerate literals like “Other” or “Unknown”. A typical example is the `ifc2x3` meta-model in the ATL zoo, which contains 34 occurrences of the *type-object* pattern using enumerated types. This meta-model implements the *Industry Foundation Classes* standard, which is heavily used in the architecture, engineering and construction industry. In this particular case, the meta-model would be simpler using a multi-level approach because each enumeration is wrapped into a container class (e.g., the enumerate `IfcBeamTypeEnum` is wrapped into the container class `IfcBeamType`, the type for class `IfcBeam`). In this way, the equivalent multi-level model would only contain a clabject at the top meta-level, while one clabject would be created at level 1 for each enumerate literal.

6.4. Stereotype approach

Stereotypes is a concept mostly used in UML, and hence frequently found in OMG specifications. Rearchitecting to multi-level yields simpler specifications, and it is especially useful when the whole UML language is not needed.

6.4.1. Rearchitecting to multi-level. In this approach, the *type-object* pattern typically provides stereotypes to `Class` and `InstanceSpecification`. In the simplest case, rearchitecting to multi-level consists on merging both stereotypes into a clabject with potency 2, obtaining a simpler specification. However, as stereotypes extend the UML with domain-specific concepts, if such concepts do not have an equivalent in the multi-level meta-modelling framework (i.e., they go beyond class/object diagrams, like elements belonging to State machine diagrams), then such concepts should be modelled in the multi-level solution.

6.4.2. Examples from the field study. We found several occurrences of the *type-object* pattern using stereotypes, mostly in OMG specifications. This is natural, as the OMG promotes the use of UML profiles in the Model-Driven Architecture. The stereotype approach is suitable when the UML is to be reused and extended. The main drawback is the difficulty of documenting the *type-object* pattern in profiles. We identified two ways for documentation. The first one is by providing a conceptual meta-model that contains the explicit version of the *type-object*, to be used as documentation but not for instantiation. The second one is declaring a dependency relation between the stereotype acting as "type" and the stereotype acting as "instance".

As an example, the Unified Profile for DoDAF and MODAF (UPDM) heavily relies on the *type-object* pattern. Figure 28(a) shows a small excerpt of its definition, which includes an occurrence of the *type-object* pattern for `ProjectType` and `ActualProject`. A stereotyped dependency relation documents this fact. Figure 28(b) shows the equivalent multi-level solution. While using stereotypes is the only alternative if extension of UML is desired, it leads to more complex solutions than potency-based multi-level solutions. In Figure 28(a), two stereotypes and a reference need to be explicitly declared, and appropriate meta-classes of the UML need to be selected, while the multi-level solution consists of one clabject.

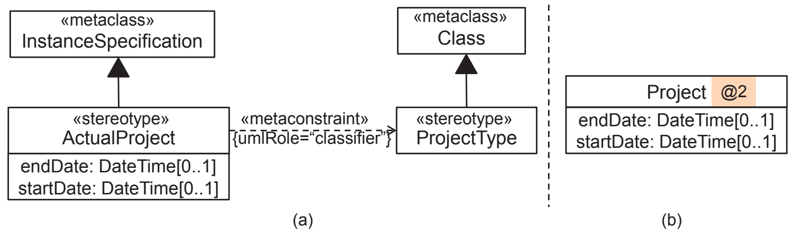


Fig. 28. (a) Excerpt of the UPDM specification. (b) Multi-level solution for the excerpt.

6.5. Powertype approach

The powertype approach was mostly found in documents, but not in the analysed repositories of artefacts. This is perhaps a sign of a still emerging tool support. Rearchitecting into a potency-based multi-level approach yields simpler models and flexibility is gained regarding deep characterization.

6.5.1. Rearchitecting to multi-level. This is done by merging both classes in the powertype into a single class with potency 2, and assigning a suitable potency to its attributes (1 for type-level attributes, and 2 for instance-level attributes). The resulting meta-model is simpler as it has less classes.

6.5.2. Examples from the field study. In our field study, we only found powertype-based approaches in the traceability and business/process modelling domains. Being a conceptual solution, these approaches are actually implemented using some other technique. For example, the DoDaF description [DoDAF 2010] makes heavy use of the powertype pattern, but it is implemented using stereotypes in the Unified Profile for DoDaF and MoDaF (UPDM) OMG specification.

6.6. Further requirements for multi-level modelling tools

In this paper, we have shown that potency-based multi-level modelling provides a flexible framework to build solutions that involve the *type-object* and related patterns, and in this section, we have illustrated how to reorganize existing solutions into multiple levels. Since these patterns appear frequently, this should be considered a relevant, useful technology. However, from the discussion in our pattern proposals and the findings in our field study, we have identified some capabilities that still need to be addressed by multi-level modelling tools in order to be able to tackle a wider range of problems. As the reader may notice next, these capabilities are missing in many general two-level modelling tools as well.

The first capability is being able to define multiple types for an element. This is a requirement for some languages, like the UML Profile for Schedulability, Performance and Time [OMG 2005c], where an instance can be typed by one or more types. Another example is the Ontology Definition Metamodel (ODM) [OMG 2009], where two different classes may have the same set of instances, and a class may even be an instance of itself. Some multi-level tools, like METADEPTH and Melanie [Atkinson et al. 2012b], allow linguistic extensions, that is, elements with no ontological typing. However, the ability of a clabject to have several ontological types is not natively supported by any of them, having to resort to explicit modelling. This multi-type capability has also been proposed in the context of the MOF by the “MOF support for semantic structures” (SMOF) OMG specification [OMG 2013e]. In this case, the set of possible types is statically predefined a-priori so that they can be combined arbitrarily.

Most flexibility in multi-level modelling is obtained by the availability of built-in meta-modelling facilities at every meta-level, like inheritance, typing or feature definition. This avoids their explicit modelling. However, some particular problems may require the use of customised, domain-specific meta-modelling facilities. For example, one might need to restrict to single inheritance, consider domain-specific semantics for inheritance, give a special semantics for association ends, or use a domain-specific set of data types. Actually, having zero or multiple types for instances can be seen as a customised meta-modelling facility. While some preliminary steps in this direction are proposed in [Laarman and Kurtev 2009], much work is still necessary, perhaps adapting ideas from Meta-Object Protocols [Kiczales and Rivieres 1991]. Thus, we claim that the customization of meta-modelling facilities is one of the main challenges to be solved by multi-level modelling tool builders. This challenge is applicable to modelling tools in general, not necessarily multi-level, though it is in this latter case where one can obtain more benefits.

Finally, in order to improve the interoperability between two-level and multi-level technologies, import and export facilities would be required to bridge these two technological spaces. Moreover, general mechanisms – similar to libraries – to ease instantiation of multiple related elements (like a component and its defined ports), would be useful to improve productivity in modelling.

7. RELATED WORK

This section reviews some lines of related works. First, we review works comparing multi-level modelling to other meta-modelling approaches. Then, we look at proposals to tackle multi-level problems besides those we have studied so far. Next, we comment on meta-modelling languages supporting explicit modelling of meta-modelling features. Last, we review model libraries and meta-modelling patterns.

7.1. Comparing multi-level and two-level modelling

Potency-based multi-level modelling was proposed in [Atkinson and Kühne 2002] as a means to simplify system modelling, especially when this implies implementing some variant of the *type-object* pattern. In our paper, we have identified further patterns (based on the type-object) where multi-level modelling may be convenient, and analyse their potential applicability in practice.

There are few works comparing multi-level modelling and standard 2-level modelling. In [Atkinson and Kühne 2002], the authors identify some disadvantages of the use of UML stereotypes, like the fact that it requires "meta" capabilities in current UML CASE tools, and it introduces an additional classification mechanism (stereotyping) which contributes to vagueness of the instantiation semantics and confusion about its interpretation. In [Atkinson and Kühne 2008], the authors compare a potency-based multi-level solution for the *type-object* pattern with respect to a solution based on powertypes and explicit modelling, for a synthetic example. They conclude that the potency-based multi-level solution yields simpler results with less modelling elements. Powertypes and potency-based multi-level modelling are also compared in [Eriksson et al. 2013], where a compact representation of powertypes is proposed, analysing the construction of modelling languages from the perspective of speech act theory.

In summary, we are not aware of existing literature investigating the occurrence of the *type-object* and related patterns in practice and assessing the value of (potency-based) multi-level modelling for MDE.

7.2. Other multi-level approaches

Approaches to multi-level meta-modelling have existed since the 1980s in knowledge-based systems like Telos [Mylopoulos et al. 1990] and deductive object base managers like ConceptBase [Jarke et al. 1995]. The latter implements the object model of a Datalog-based variant of Telos, supporting instantiation chains of arbitrary depth but not the organization of elements in models and meta-levels. Telos was used in [Dahchour 1998] to formalize the *materialization relation*, a relation between two entities similar to the one arising in the type-object pattern. Multi-level modelling can also be recasted in other technological spaces, like ontologies, with languages like OWL [W3C 2012].

The VPM framework [Varró and Pataricza 2003] formalizes multi-level meta-modelling by a refinement relation. Entities are viewed both as sets (a type that defines the set of its instances) and elements of a set (an instance in the set of instances defined by its type). Thus, as in multi-level meta-modelling, an element retains both class and instance facet. While VPM is realized in the VIATRA tool, it lacks deep characterization and does not consider attributes or constraints.

In [Álvarez et al. 2001], the authors propose a nonlinear framework where the nature of elements at the instance level depends on the viewpoint: they are instances of a domain class from the domain modeller viewpoint, and instances of a meta-class `Object` from the tool perspective, being possible to transform between both views. As discussed in [Atkinson and Kühne 2002], this solution makes the instantiation relation context-dependant, and it requires replicating meta-classes `Class` and `Object` in all meta-levels bigger than 1. In contrast, the nature of elements in multi-level modelling is context-independent (i.e., clabjects have simultaneously both ontological and linguistic types [Atkinson and Kühne 2002]) and all meta-levels share the same linguistic meta-model.

7.3. Multi-level modelling tools

The modelling community has made available several tools enabling multi-level modelling. Next, we analyse their support for the requirements identified in Section 6.6.

METADEPTH [de Lara and Guerra 2010] is a potency-based multi-level meta-modelling tool developed by our group. It uses a textual syntax and is integrated with the Epsilon languages [Paige et al. 2009] for model management. Models in METADEPTH are first-class entities and can define linguistic extensions (elements without ontological typing) and deep references (as used in Section 6.1). We have used METADEPTH as the technological basis of some projects. In [de Lara et al. 2014b], we reported on its use for web engineering, for which we included in the language code generation capabilities, and advanced meta-modelling facilities like deep references, or leap potency. However, multiple ontological types are not allowed in METADEPTH. Simple customizations of meta-modelling facilities, like restriction to single inheritance, are possible using constraints [de Lara and Guerra 2012], but sophisticated customizations are not supported. Compared to our previous works, the contributions of the present paper are the following: we have identified and discussed different patterns where using a multi-level technology may have benefits, we have performed an exhaustive field study to evaluate the frequency of occurrence of such patterns in practice, and we have outlined guidelines to migrate to multi-level solutions.

MelAniE [Atkinson et al. 2012b; Atkinson et al. 2012c; Atkinson et al. 2009] is a graphical, potency-based multi-level modelling tool based on EMF and GMF. In MelAniE, fields can define so-called traits, like *value mutability*, which specifies over how many levels the value may be changed from the default. Modelling elements must define a *level*, which in METADEPTH is indicated through the *potency* of the model. MelAniE supports linguistic extensions, but not multiple ontological types or customization of meta-modelling facilities.

DeepJava [Kühne and Schreiber 2007] extends Java with the possibility to assign a potency to Java classes, attributes and methods. It is embedded in Java and, therefore, each element has exactly one type. Neither multiple ontological types nor customization of meta-modelling facilities are possible.

The cross-layer modeller (XLM) [Demuth et al. 2011] supports an arbitrary number of modelling layers by using an embedding in UML and giving semantics to instantiation as OCL constraints (templatized OCL constraints for the case of the instantiation of associations). However, XLM does not provide advanced instantiation mechanisms like potency, optional or multiple ontological types, meta-modelling features like attributes/links, inheritance, or customization of meta-modelling facilities.

Other multi-level frameworks are Nivel [Asikainen and Männistö 2009], which is based on the weighted constraint rule language (WCRL); OMME [Volz and Jablonski 2010], which implements dual ontological/linguistic typing, deep characterization and powertypes; the DPF workbench [Lamo et al. 2013], backed by a strong theoretical basis but without support for attributes; the VMTS tool [Levendovszky et al. 2005], which offers support for model transformations; or the approach in [Aschauer et al. 2009], whose main concern is the efficient navigation between meta-levels. However, none of these frameworks support customization of meta-modelling facilities or the definition of multiple or optional ontological types. Hence, these remain as challenges for the multi-level modelling community.

7.4. Modelling patterns, software design patterns and libraries

This paper has proposed several modelling patterns. In general, modelling patterns are used to document and gather knowledge about good modelling practices and to guide the construction and refactoring of models. The MDE community is actively working on developing new patterns and pattern languages for modelling and meta-modelling. For instance, some design patterns for meta-models are described in [Cho and Gray 2011], while in [Schäfer et al. 2011], the requirements for meta-models are represented as use case diagrams and the meta-models are evolved by applying patterns. Patterns for domain-specific programming languages, mostly applicable to DSMLs as well, are proposed in [Spinellis 2001]. In our case, we are interested in patterns related to multi-level modelling and on the availability of meta-modelling facilities at the model level.

In software design, patterns related to the *type-object* have been proposed since the 1990s [Coad 1992; Martin et al. 1997; Yoder and Johnson 2002]. As solutions are based on mainstream object-oriented languages, they rely on explicit modelling, hence causing replication, as we have observed in Section 4.1. Having available more advanced infrastructures would enable simpler solutions, as in the case of the potency-based multi-level approach. One related pattern is the *dynamic template pattern*, which decouples instances from their classes by allowing the creation of new object templates dynamically, where both template and instance attributes may be added at runtime [Lyardet 1997]. While modifying clabjects at run-time is possible, being one of the advantages of multi-level modelling, care should be taken to keep consistent the instances of the modified clabjects. Some techniques to solve this issue are proposed in [Atkinson et al. 2012a].

On the other hand, model libraries enable the reuse of modelling solutions and implementations. As acknowledged in [Herrmannsdörfer and Hummel 2010], support for reuse in the modelling world is poor, especially for DSMLs. In [Herrmannsdörfer and Hummel 2010], the authors propose using library concepts in order to make reuse of models more systematic and simplify dealing with model changes. Their libraries contain model types reusable through a cloning mechanism, following the prototype pattern [Gamma et al. 1994]. Thus, models can import libraries and instantiate their types, which are full copies of the type. The instantiation relation is then resolved by name. This is an application of the *type-object* pattern, where the proposed solution emulates two meta-levels (for types and instances) within one. As a consequence, the authors have to manually encode support for the definition of classes/features/data types and their instantiation, the emulation of inheritance within a single meta-level, and the propagation of changes from types to instances. As previously stated, we found several uses of libraries in our study: in the OMG's Structured Metrics Meta-model (SMM) [OMG 2012e], it is possible to define libraries of measures, and the Automated Function Points (AFP) specification [OMG 2013a] is a library of SMM. However, for the effective use of libraries in tools, advanced mechanisms for cloning, referencing the original objects of the library, are required. Cloning is also used in [Karsai et al. 2004] to enable type reuse, atop the GME tool. In general, cloning-based approaches make it difficult to distinguish between type level and instance level features, and moreover, few meta-modelling frameworks support (configurable) cloning natively.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a catalogue of patterns where using multi-level technology is advantageous. These include the *type-object*, *dynamic features*, *dynamic auxiliary domain concepts*, *relation configurator* and *element classification* patterns. For each one of them, we have discussed alternative solutions and benefits of using a potency-based multi-level approach.

In order to assess the relevance of multi-level technology for real MDE projects, we have studied the occurrence of these patterns in practice by analysing more than 400 meta-models, including those in the ATL zoo, the ReMoDD repository, and the OMG. Every such occurrence indicates a possible application of multi-level technology. The analysis has exposed some domains, like software architecture, or enterprise/process modelling, where these patterns are pervasive, and their occurrences particularly intense (many different occurrences of the patterns in the same meta-model). This suggests that many problems in these domains are intrinsically multi-level. Moreover, the high occurrence of these patterns in OMG specifications suggests the relevance of multi-level technology for modelling significant domains. We have also presented guidelines to rearchitect different solutions to the identified patterns into multi-level solutions, illustrated with real examples from the field study. To conclude, we have identified some requirements for state-of-the-art potency-based multi-level tools concerning meta-model facilities.

In the future, we plan to derive semi-automatic means for refactoring plain meta-models containing some occurrences of our patterns using explicit modelling, into potency-based multi-level solutions. To this aim, we plan to use our METADEPTH tool, as it provides support for model transformation. This would help in performing an empirical study of the real gain obtained by refactoring existing meta-models into multi-level solutions. Moreover, in order to improve the interoperability with widely used frameworks, like the EMF, we will also provide flattenings of multi-level models into two meta-levels.

Acknowledgements. We are grateful to the reviewers for their accurate and useful suggestions, which helped to improve a previous version of the paper.

REFERENCES

José M. Álvarez, Andy Evans, and Paul Sammut. 2001. Mapping between levels in the metamodel architecture. In *UML'01 (LNCS)*, Vol. 2185. Springer, 34–46.

Thomas Aschauer, Gerd Dauenhauer, and Wolfgang Pree. 2009. Representation and traversal of large clabject models. In *MoDELS'09 (LNCS)*, Vol. 5795. Springer, 17–31.

Timo Asikainen and Tomi Männistö. 2009. Nivel: A metamodelling language with a formal semantics. *Software and System Modeling* 8, 4 (2009), 521–549.

Colin Atkinson. 1997. Meta-modeling for distributed object environments. In *EDOC*. IEEE Computer Society, 90–101.

Colin Atkinson, Ralph Gerbig, and Bastian Kennel. 2012a. On-the-fly emendation of multi-level models. In *ECMFA'12 (LNCS)*, Vol. 7349. Springer, 194–209.

Colin Atkinson, Ralph Gerbig, and Bastian Kennel. 2012b. Symbiotic general-purpose and domain-specific languages. In *ICSE'12 (New Ideas and Emerging Results track)*. IEEE, 1269–1272.

Colin Atkinson, Ralph Gerbig, and Christian Tunjic. 2012c. Towards multi-level aware model transformations. In *ICMT'12 (LNCS)*, Vol. 7307. Springer, 208–223.

Colin Atkinson, Matthias Gutheil, and Bastian Kennel. 2009. A flexible infrastructure for multilevel language engineering. *IEEE Trans. Soft. Eng.* 35, 6 (2009), 742–755.

Colin Atkinson, Bastian Kennel, and Björn Goß. 2010. The level-agnostic modeling language. In *SLE'10 (LNCS)*, Vol. 6563. Springer, 266–275.

Colin Atkinson and Thomas Kühne. 2001. The essence of multilevel metamodeling. In *UML'01 (LNCS)*, Vol. 2185. Springer, 19–33.

Colin Atkinson and Thomas Kühne. 2002. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.* 12, 4 (2002), 290–321.

Colin Atkinson and Thomas Kühne. 2003. Model-driven development: A metamodeling foundation. *IEEE Software* 20, 5 (2003), 36–41.

Colin Atkinson and Thomas Kühne. 2008. Reducing accidental complexity in domain models. *Software and System Modeling* 7, 3 (2008), 345–359.

AtlanEcore. 2014. AtlanEcore metamodel zoo. `http://www.emn.fr/z-info/atlanmod/index.php/Ecore`. (2014).

Mario R. Barbacci and Charles B. Weinstock. 1998. *Mapping MetaH into ACME*. Technical Report. Software Engineering Institute, Carnegie Mellon University, CMU/SEI-98-SR-006.

Maria Bergholtz, Paul Johannesson, and Petia Wohed. 2005. UEML: Providing requirements and extensions for interoperability challenges. In *INTEROP-ESA*. 89–102.

Stefano Ceri, Marco Brambilla, and Piero Fraternali. 2009. The history of WebML. Lessons learned from 10 years of model-driven development of web applications. In *Conceptual Modeling: Foundations and Applications (LNCS)*, Vol. 5600. Springer, 273–292.

Hyun Cho and Jeff Gray. 2011. Design patterns for metamodels. In *SPLASH Workshops*. ACM, 25–32.

CloudML. 2014. `http://cloudml.org/`. (2014).

Peter Coad. 1992. Object-oriented patterns. *Commun. ACM* 35, 9 (1992), 152–159.

Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice* 10, 1 (2005), 7–29.

Mohamed Dahchour. 1998. Formalizing Materialization Using a Metaclass Approach. In *CAiSE (LNCS)*, Vol. 1413. Springer, 401–421.

Juan de Lara and Esther Guerra. 2010. Deep meta-modelling with METADEPTH. In *TOOLS'10 (LNCS)*, Vol. 6141. Springer, 1–20. See also `http://astreo.ii.uam.es/∼jlara/metaDepth`.

Juan de Lara and Esther Guerra. 2012. Domain-specific textual meta-modelling languages for model driven engineering. In *ECMFA'12 (LNCS)*, Vol. 7349. Springer, 259–274.

Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. 2014b. Extending deep meta-modelling for practical model-driven engineering. *Comput. J.* 57, 1 (2014), 36–58.

Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014a. Model-driven engineering with domain-specific meta-modelling languages. *Software and System Modeling* To appear (2014), 1–31.

Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2011. Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking. In *SIGSOFT FSE*. ACM, 452–455.

DoDAF. 2010. The DoDAF architecture framework version 2.02. http://dodcio.defense.gov/TodayinCIO/DoDArchitectureFramework.aspx. (2010).

Marco Dorigo and Luca Maria Gambardella. 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Trans. Evolutionary Computation* 1, 1 (1997), 53–66.

Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran Jude Fernandes. 2008. Engineering a DSL for software traceability. In *SLE'08 (LNCS)*, Vol. 5452. Springer, 151–167.

Owen Eriksson, Brian Henderson-Sellers, and Pär J. Ågerfalk. 2013. Ontological and linguistic metamodelling revisited: A language use approach. *Information and Software Technology* 55, 12 (2013), 2099 – 2124.

Angelina Espinoza and Juan Garbajosa. 2011. A study to support agile methods more effectively through traceability. *ISSE* 7, 1 (2011), 53–69.

Farah Fourati. 2010. *Une approche IDM de transformation exogéne de Wright vers Ada*. Master's thesis. Ecole Nationale d'Ingenieurs de Sfax.

Jesús Gallardo, Crescencio Bravo, and Miguel A. Redondo. 2012. A model-driven development method for collaborative modeling tools. *J. Network and Computer Applications* 35, 3 (2012), 1086–1105.

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design patterns. Elements of reusable object-oriented software*. Addison Wesley.

Charles F. Goldfarb. 1991. *The SGML handbook*. Oxford University Press.

César González-Pérez and Brian Henderson-Sellers. 2006. A powertype-based metamodelling framework. *Software and System Modeling* 5, 1 (2006), 72–90.

César González-Pérez and Brian Henderson-Sellers. 2007. Modelling software development methodologies: A conceptual foundation. *Journal of Systems and Software* 80, 11 (2007), 1778–1796.

Gregory Gutin, Abraham Punnen, Alexander Barvinok, Edward Kh. Gimadi, and Anatoliy I. Serdyukov. 2002. The traveling salesman problem and its variations. (2002).

Markus Herrmannsdörfer and Benjamin Hummel. 2010. Library concepts for model reuse. *Electron. Notes Theor. Comput. Sci.* 253 (September 2010), 121–134. Issue 7.

Matthias Jarke, Rainer Gallersdörfer, Manfred A. Jeusfeld, and Martin Staudt. 1995. ConceptBase - A deductive object base for meta data management. *J. Intell. Inf. Syst.* 4, 2 (1995), 167–192.

Frédéric Jouault and Jean Bézivin. 2006. KM3: A DSL for metamodel specification. In *FMOODS'06 (LNCS)*, Vol. 4037. Springer, 171–185.

G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits. 2004. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology* 12, 2 (2004), 263 – 278.

Gregor Kiczales and Jim Des Rivieres. 1991. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA.

Thomas Kühne and Daniel Schreiber. 2007. Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. In *OOPSLA'07*. ACM, 229–244.

Alfons Laarman and Ivan Kurtev. 2009. Ontological metamodeling with explicit instantiation. In *SLE'09 (LNCS)*, Vol. 5969. Springer, 174–183.

Yngve Lamo, Xiaoliang Wang, Florian Mantz, Øyvind Bech, Anders Sandven, and Adrian Rutle. 2013. DPF workbench: a multi-level language workbench for MDE. *Proceedings of the Estonian Academy of Sciences* 62, 1 (2013), 3–15.

Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. 2012. EMF Profiles: A lightweight extension approach for EMF models. *Journal of Object Technology* 11, 1 (2012), 1–29.

Gilbert Laporte. 1997. Modeling and solving several classes of arc routing problems as traveling salesman problems. *Computers & Operations Research* 24, 11 (1997), 1057 – 1061.

Tihamer Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. 2005. A systematic approach to metamodeling environments and model transformation systems in VMTS. *Electr. Notes Theor. Comput. Sci.* 127, 1 (2005), 65–75.

Fernando D. Lyardet. 1997. The dynamic template pattern. In *PLOP'97*. Washington University.

Robert C. Martin, Dirk Riehle, and Frank Buschmann. 1997. *Pattern languages of program design 3*. Addison-Wesley.

John Mylopoulos, Alexander Borgida, Matthias Jarke, and Manolis Koubarakis. 1990. Telos: Representing knowledge about information systems. *ACM Trans. Inf. Syst.* 8, 4 (1990), 325–362.

James Odell. 1994. Power types. *JOOP* 7, 2 (1994), 8–12.

OMG. 2002a. Bibliographic Query Service Specification 1.0. `http://www.omg.org/spec/BQS/1.0/`. (2002).

OMG. 2002b. Negotiation Facility 1.0. `http://www.omg.org/spec/NEG/1.0/`. (2002).

OMG. 2003. CWM 1.1. `http://www.omg.org/spec/CWM/1.1/`. (2003).

OMG. 2004a. EDOC 1.0. `http://www.omg.org/spec/EDOC/1.0/`. (2004).

OMG. 2004b. UML Profile for EAI 1.0. `http://www.omg.org/spec/EAI/1.0/`. (2004).

OMG. 2005a. DAIS 1.1. `http://www.omg.org/spec/DAIS/1.1/`. (2005).

OMG. 2005b. RAS 2.2. `http://www.omg.org/spec/RAS/2.2/`. (2005).

OMG. 2005c. SPTP 1.1. `http://www.omg.org/spec/SPTP/1.1/`. (2005).

OMG. 2005d. UML Profile for CORBA Components 1.0. `http://www.omg.org/spec/CCMP/1.0/`. (2005).

OMG. 2005e. HDAIS 1.0. `http://www.omg.org/spec/HDAIS/1.0/`. (2005).

OMG. 2007. ITPMF 1.0. `http://www.omg.org/spec/ITPMF/1.0/`. (2007).

OMG. 2008a. SPEM 2.0. `http://www.omg.org/spec/SPEM/2.0/`. (2008).

OMG. 2008b. BPDM 1.0. `http://www.omg.org/spec/BPDM/1.0/`. (2008).

OMG. 2008c. QFTP 1.1. `http://www.omg.org/spec/QFTP/1.1/`. (2008).

OMG. 2009. ODM 1.0. `http://www.omg.org/spec/ODM/1.0/`. (2009).

OMG. 2010. EXPRESS 1.0. `http://www.omg.org/spec/EXPRESS/1.0/`. (2010).

OMG. 2011a. KDM 1.3. `http://www.omg.org/spec/KDM/1.3/`. (2011).

OMG. 2011b. PLM Services 2.1. `http://www.omg.org/spec/PLM/2.1/`. (2011).

OMG. 2011c. RMS 1.0. `http://www.omg.org/spec/RMS/1.0/`. (2011).

OMG. 2011d. SOPES 1.0. `http://www.omg.org/spec/SOPES/1.0/`. (2011).

OMG. 2011e. ASTM 1.0. `http://www.omg.org/spec/ASTM/1.0/`. (2011).

OMG. 2011f. MARTE 1.1. `http://www.omg.org/spec/MARTE/1.1/`. (2011).

OMG. 2011g. UML 2.4.1. `http://www.omg.org/spec/UML/2.4.1/`. (2011).

OMG. 2012a. DD 1.0. `http://www.omg.org/spec/DD/1.0/`. (2012).

OMG. 2012b. SoaML 1.0.1. `http://www.omg.org/spec/SoaML/1.0.1/`. (2012).

OMG. 2012c. SySML 1.3. `http://www.omg.org/spec/SysML/1.3/`. (2012).

OMG. 2012d. OCL 2.3.1. `http://www.omg.org/spec/OCL/2.3.1/`. (2012).

OMG. 2012e. SMM 1.0. `http://www.omg.org/spec/SMM/1.0/`. (2012).

OMG. 2013a. AFP 1.0 beta 2. `http://www.omg.org/spec/AFP/1.0/Beta2/`. (2013).

OMG. 2013b. BPMNProfile 1.0 Beta 1. `http://www.omg.org/spec/BPMNProfile/1.0/Beta1/`. (2013).

OMG. 2013c. ReqIF 1.1. `http://www.omg.org/spec/ReqIF/1.1/`. (2013).

OMG. 2013d. SACM 1.0. `http://www.omg.org/spec/SACM/1.0/`. (2013).

OMG. 2013e. SMOF 1.0. `http://www.omg.org/spec/SMOF/1.0/`. (2013).

OMG. 2013f. BPMN 2.0.1. `http://www.omg.org/spec/BPMN/2.0.1/`. (2013).

OMG. 2013g. CMMN 1.0 - Beta 1. `http://www.omg.org/spec/CMMN/1.0/Beta1/`. (2013).

OMG. 2013h. FUML 1.1. http://www.omg.org/spec/FUML/1.1/. (2013).

OMG. 2013i. MOF 2.4.1. http://www.omg.org/spec/MOF/2.4.1/. (2013).

OMG. 2013j. UPDM 2.1. http://www.omg.org/spec/UPDM/2.1/. (2013).

OMG. 2013k. UTP 1.2. http://www.omg.org/spec/UTP/1.2/. (2013).

OMG. 2014. Summary of OMG specifications. http://www.omg.org/spec/. (2014).

openEHR. 2014. http://www.openehr.org/. (2014).

Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fiona A. C. Polack. 2009. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS '09*. IEEE Computer Society, Washington, DC, USA, 162–171.

Christos H. Papadimitriou and Kenneth Steiglitz. 1998. *Combinatorial optimization: Algorithms and complexity*. Dover Publications.

ReMoDD. 2014. The repository for model-driven development. http://www.cs.colostate.edu/remodd. (2014).

Dirk Riehle, Michel Tilman, and Ralph Johnson. 2000. Dynamic object model. In *PLOP'00*. Washington University Technical Report number: wucs-00-29.

G. Lawrence Sanders. 1995. *Data modeling*. Course Technology Ptr.

Marcos López Sanz and Esperanza Marcos. 2012. ArchiMeDeS: A model-driven framework for the specification of service-oriented architectures. *Inf. Syst.* 37, 3 (2012), 257–268.

Christian Schäfer, Thomas Kuhn, and Mario Trapp. 2011. A pattern-based approach to DSL development. In *DSM'11*. ACM, 39–46.

Diomidis Spinellis. 2001. Notable design patterns for domain-specific languages. *Journal of Systems and Software* 56, 1 (2001), 91–99.

Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd edition*. Addison-Wesley Professional, Upper Saddle River, NJ.

UsiXML. 2014. UsiXML 1.0: USer Interface eXtended Markup Language. http://www.usixml.org. (2014).

Dániel Varró and András Pataricza. 2003. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Software and System Modeling* 2, 3 (October 2003), 187–210.

Markus Völter. 2013. *DSL engineering - Designing, implementing and using domain-specific languages*. dslbook.org. 1–558 pages.

Markus Völter and Thomas Stahl. 2006. *Model-driven software development*. John Wiley & Sons.

Bernhard Volz and Stefan Jablonski. 2010. Towards an open meta modeling environment. In *10th Workshop on Domain-Specific Modeling*. ACM, New York, NY, USA, Article 17, 6 pages.

W3C. 2012. OWL 2 Web Ontology Language Document Overview. http://www.w3.org/TR/owl2-overview/. (2012).

Michael J. Wooldridge. 2009. *An introduction to multiAgent systems (2nd edition)*. Wiley. I–XXII, 1–461 pages.

Joseph W. Yoder and Ralph E. Johnson. 2002. The adaptive object-model architectural style. In *WICSA'02 (IFIP Conference Proceedings)*, Vol. 224. Kluwer, 3–27.