ACM/IEEE 17th International Conference on
Model Driven Engineering Languages and Systems

September 28 – October 3, 2014 • Valencia (Spain)

MULTI 2014 – Multi-Level Modelling Workshop Proceedings

Colin Atkinson, Georg Grossmann, Thomas Kühne, Juan de Lara (Eds.)

Published on Sept 2014 v1.0

Editors' addresses:

Colin Atkinson
University of Mannheim (Germany)

Georg Grossmann
University of South Australia (Australia)

Thomas Kühne
Victoria University of Wellington (New Zealand)

Juan de Lara
Universidad Autónoma de Madrid (Spain)

Organizers

Colin Atkinson	University of Mannheim (Germany)
Georg Grossmann	University of South Australia (Australia)
Thomas Kühne	Victoria University of Wellington (New Zealand)
Juan de Lara	Universidad Autónoma de Madrid (Spain)

Steering Committee

Thomas Kühne	Victoria University of Wellington (New Zealand)
Juan de Lara	Universidad Autónoma de Madrid (Spain)

Program Committee

Ulrich Frank	Universität Duisburg-Essen (Germany)
Manfred Jeusfeld	University of Skövde (Sweden)
Ivan Kurtev	NSPYRE (Netherlands)
Alessandro Rossini	SINTEF (Norway)
Dirk Draheim	University of Mannheim (Germany)
Yngve Lamo	Bergen Univeristy College (Norway)
Ralph Gerbig	University of Mannheim (Germany)
Brian Henderson-Sellers	University of Technology, Sydney (Australia)
Esther Guerra	Universidad Autónoma de Madrid (Spain)
Tony Clark	Middlesex University (United Kingdom)
Cesar Gonzalez-Perez	Spanish National Research Council (CSIC)
Michael Schrefl	Johannes Kepler Universität Linz (Austria)
Daniel Varro	Budapest University (Hungary)
Martin Gogolla	University of Bremen (Germany)
Hans Vangheluwe	University of Antwerp (Belgium)
Juan de Lara	Universidad Autónoma de Madrid (Spain)
Alexander Egyed	Johannes Kepler Universität Linz (Austria)
Tomi Mnnist	University of Helsinki (Finland)
Colin Atkinson	University of Mannheim (Germany)
Jorn Bettin	S23M (Australia)
Samir Al-Hilank	develop group (Germany)
Wolfgang Pree	Universität Salzburg (Austria)
Joao-Paulo Almeida	Federal University of Espírito Santo (Brazil)
Stefan Jablonski	Universität Bayreuth (Germany)
Markus Stumptner	University of South Australia (Australia)
Manuel Wimmer	TU Wien (Austria)
Steffen Zschaler	King's College London (Germany)
Tihamer Levendovszky	Vanderbilt University (USA)
Georg Grossmann	University of South Australia (Australia)
Thomas Kühne	Victoria University of Wellington (New Zealand)

Table of Contents

Preface

In recent years there has been growing interest in the use of multi-level modelling approaches to better represent the multiple classification levels that are frequently found in the real world and are needed to effectively engineer languages. Multi-level modelling approaches have not only been successfully used in numerous industrial projects and standards definition initiatives they are now supported by an array of dedicated tools.

However there is still no clear consensus on what multi-level modelling actually is and what kinds of constructs and concepts provide the best support for it. For example, there are diverging views on whether it is sound to combine instance facets and type facets into so-called clabjects, whether strict metamodelling is too restrictive, and what principles should be used in establishing meta-level boundaries, etc.

The MULTI 2014 workshop was established to address this problem by bringing together researchers and practitioners with an interest in multi-level modelling to foster a fruitful cross-pollination of ideas and lay the foundation for a unified discipline. In particular, the workshop aimed to identify a set of criteria for judging the strengths and weaknesses of different multi-level modelling approaches and for defining possible benchmark case studies. To this end, the workshop encouraged submissions on new concepts, implementation approaches and formalisms as well as controversial positions, requirements for evaluation criteria or case-study scenarios. Contributions in the area of tool building, multi-level modelling applications, and educational material were also welcome.

From a total of 16 submissions, 12 papers were selected that addressed a range of topics related to multi-level modelling. In terms of technology papers, three papers presented proposals for enhancing existing multi-level modeling approaches, two papers presented alternative formal foundations for multi-level modeling, and two papers presented approaches for checking the consistency and integrity of multi-level models. A further four papers presented applications of multi-level modeling to different scenarios; two focusing on industrial applications and two focusing on the applications of multi-level modeling to the ubiquitous problems of interoperability and big, distributed data. Finally, the remaining paper presented ideas for improving the way in which multi-level modeling approaches can be evaluated and compared. In addition to two paper sessions, the workshop included an invited talk answering the question "What is Multi-Level Modeling?" and two plenary discussions focused on core multi-level themes and the future of the fledgling multi-level community.

September 2014 Colin Atkinson, Georg Grossmann, Thomas Kühne, Juan de Lara

Keynote

What is Multi-Level Modeling?

Cesar Gonzalez-Perez

Institute of Heritage Sciences (Incipit)
Spanish National Research Council (CSIC)

In line with the MULTI 2014 workshop theme, this talk aims to introduce and review the use of multi-level modelling approaches to represent multiple classification levels in language engineering and conceptual modelling communities. The talk will analyse the root motivation at the heart of multi-level modelling, namely the need for a type model to influence, to some extent, not only its instances but also the instances of these. Different approaches that have been proposed to tackle this issue will be examined, including strict metamodelling, orthogonal (linguistic plus ontological) approaches, deep instantiation, and powertype-based approaches. Finally, some challenges are briefly presented that haven't been solved yet, in anticipation of a deeper discussion by specific workshop papers.

Abstract vs Concrete Clabjects in Dual Deep Instantiation

Bernd Neumayr and Michael Schrefl

Department of Business Informatics – Data & Knowledge Engineering
Johannes Kepler University Linz, Austria
`firstname.lastname@jku.at`

Abstract. Deep Instantiation allows for a compact representation of models with multiple instantiation levels where clabjects combine object and class facets and allow to characterize the schema of model elements several instantiation levels below. Clabjects with common properties may be generalized to superclabjects. In order to clarify the exact nature of superclabjects, Dual Deep Instantiation, a variation of Deep Instantiation, distinguishes between abstract and concrete clabjects and demands that superclabjects are abstract. An abstract clabject combines the notion of abstract class, i.e., it may not be instantiated by concrete objects, and of abstract object, i.e., is does not represent a single concrete object but properties common to a set of concrete objects. This paper clarifies the distinction between abstract and concrete clabjects and discusses the role of concrete clabjects for mandatory constraints at multiple levels and for coping with dual inheritance introduced with the combination of generalization and deep instantiation. The reflections in this paper are formalized based on a simplified form of dual deep instantiation but should be relevant to deep characterization in general.

1 Introduction

What is represented as instance data in one application may be represented as schema data in another application. For example, in an application for managing a product catalog, product category *car* is represented by a class Car which is instantiated by objects representing particular car models, such as BMW Z4. In a customer service application, the same car model may be represented by a class which is instantiated by objects representing individual cars, such as PetersZ4.

Potency-based *Deep Instantiation* [1] allows for a compact and integrated representation of such scenarios. For example, clabject BMW Z4 in our running example (see Fig. 1), which makes use of a simplified and relaxed version of Dual Deep Instantiation [9], represents both: an object, namely the car model *BMW Z4*, and a class, namely the class of individual cars of model *BMW Z4*. Further up in the product hierarchy, the clabject Car with potency 2 represents product category *car* as well as the classes of individual cars and of car models. Finally, the whole product hierarchy is represented by clabject Product with potency 3.

Clabjects with common properties may be generalized to a superclabject. In Dual Deep Instantiation, superclabjects are abstract and are not considered as objects in their own right. For example, concrete clabjects Car and Motorcycle are generalized to an abstract superclabject Vehicle which defines properties shared by Car and Motorcycle, e.g., MsBlack is the category manager of the two product categories represented by Car and Motorcycle.

In the remainder of the paper we introduce, in Sect. 2, a simplified form of Dual Deep Instantiation along which we discuss, in Sect. 3, the distinction between abstract and concrete clabjects. In Sect. 4, we define an inheritance mechanism and describe the role of concrete clabjects in dealing with dual inheritance stemming from the combination of generalization and deep instantiation. Sect. 5 introduces support for defining mandatory constraints over concrete clabjects at multiple levels in order to control the stepwise instantiation process. Sect. 6 gives a brief overview of related work and Sect. 7 concludes the paper.

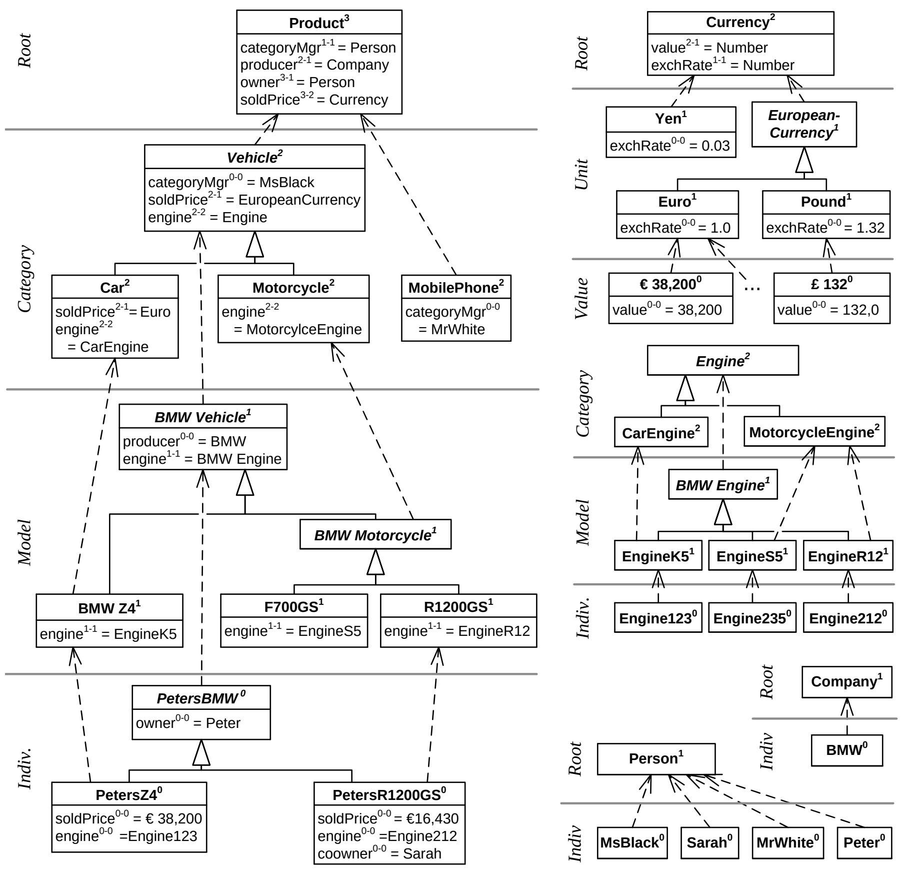


Fig. 1. Multi-level Hierarchies of Abstract and Concrete Clabjects

2 Dual Deep Instantiation and Generalization – Simplified

Dual Deep Instantiation (DDI) [9] allows to relate clabjects at different instantiation levels by *bi-directional and multi-valued relationships* and allows to separately indicate the depth of characterization for the source and target of a relationship (not assuming a strict separation of classification levels). By restricting clabjects to have only one parent, either related by instantiation or by generalization, it only allows *single inheritance*. For discussing the distinction between concrete and abstract clabjects and their role in multi-level models we introduce, in this section, a simplified and relaxed variant of DDI. It is *simpflified by only considering uni-directional and single-valued property references.* It is *relaxed by allowing dual inheritance*: a clabject may now have two parents, one at the same level and connected by a generalization relationship and the other at the next higher level and related by an instantiation relationship.

A DDI model contains a set of clabjects (see No. 1 in Table 2). Clabjects are organized in instantiation hierarchies with an arbitrary number of instantiation levels, where $in(x, y)$ expresses that clabject x is an instantiation of clabject y (No. 2); we also say y is the *class-clabject* of x. Clabjects at the same instantiation level may be organized in specialization hierarchies, where $spec(x, y)$ expresses that clabject x is a *direct specialization* of clabject y (No. 3); we also refer to y as *the superclabject* of x.

We use the term *clabject* in a wide sense. It covers what is traditionally modeled as individual objects (tokens, instance specifications), classes, simple values, and primitive datatypes. In DDI, even individuals may refine or extend their own schema, e.g., property coowner is introduced at individual PetersR1200GS, and we assume, in this regard, that every individual comes with its own class facet. So, in DDI *everything is a clabject.* Note, in the original DDI approach [9] we used the terms *object* or *DDI object* for what we now call *clabject.*

Each clabject comes with a clabject potency (No. 4). A potency of a clabject is given by a natural number (including 0) and indicates the number of instantiation levels of the clabject, where $ptcy(x) = n$ expresses that x has descendants at the next n instantiation levels beneath. Note, in our previous work [9], clabjects did not have an asserted potency.

In order to simplify discussion and formalization of the approach, we introduce auxiliary terms, predicates and shorthand notations. We say x *isa* y if x is either a specialization or an instantiation of y (No. 5), we also say y is a parent of x. We say x is a *member* of y if x relates to y by a chain of *isa* with exactly one instantiation step (No. 6). We say x is an n-member of y if x relates to y by a chain of *isa* with n instantiation steps (No. 7). We use $.^+$ and $.^*$ to denote the transitive and transitive-reflexive closure, respectively, of a binary relation. We say, x is a *descendant* of y and y is an *ancestor* of x if $isa^+(x, y)$. We say, x is a *specialization* of y and y is a *generalization* (or is *a superclabject*) of x if $spec^+(x, y)$.

When a clabject x instantiates a clabject c then the potency of x is the potency of c decremented by one (No. 8). Clabjects in generalization hierarchies all have the same clabject potency (No. 9).

Table 1. Instantiation and Generalization Hierarchies of Clabjects

Sorts & Asserted Predicates:
(1) C: *clabjects* (representing individuals, classes, datatypes and values)
(2) $in \subseteq C \times C$
(3) $spec \subseteq C \times C$
(4) $ptcy : C \rightarrow \mathbb{N}$ ($\mathbb{N}$ is the set of natural numbers including 0)
Auxiliary Predicates:
(5) $isa(x, y) :\Leftrightarrow in(x, y) \vee spec(x, y)$
(6) $member(x, c) :\Leftrightarrow \exists s \exists d : spec^*(x, s) \wedge in(s, d) \wedge spec^*(d, c)$
(7) $nmember(x, c, n) :\Leftrightarrow (n = 0 \wedge spec^*(x, c)) \vee$ $\exists m \exists d : ((n = m + 1) \wedge nmember(x, d, m) \wedge member(d, c))$
Well-formedness Criteria and Syntactic Restrictions:
(8) $in(x, y) \rightarrow ptcy(x) = ptcy(y) - 1$
(9) $spec(x, y) \rightarrow ptcy(x) = ptcy(y)$
(10) $isa^+(x, c) \rightarrow x \neq c$
(11) $in(x, c) \wedge in(x, d) \rightarrow c = d$
(12) $spec(x, s) \wedge spec(x, z) \rightarrow s = z$
(13) $spec(x, s) \wedge in(x, c) \rightarrow \exists y : isa^*(s, y) \wedge isa^*(c, y)$
(14) $spec^*(x, y) \wedge in(x, c) \wedge in(y, d) \rightarrow spec^*(c, d)$

In DDI every clabject hierarchy comes with its own set of instantiation levels which is introduced by the potency of the root clabject (a root clabject is a clabject without parent). For example, root clabject Product with potency 3 introduces a clabject hierarchy with three instantiation levels (not counting the instantiation level of the root clabject). These instantiation levels may be given labels. For example, root clabject Product has three instantiation levels, labelled Category, Model, and Individual. Instead of saying *"BMW Z4 is 2-member of Product"* one may now say *"BMW Z4 is a Product Model"*.

We assume acyclic clabject hierarchies (No. 10) with single classification, i.e., every clabject has at most one class-clabject (No. 11), and single generalization, i.e., every clabject has at most one direct superclabject (No. 12).

In the original formalization of DDI [9], in order to avoid multiple inheritance, every clabject had at most one parent, either a class-clabject or a superclabject. We now relax this global restriction and allow clabjects to have both a class-clabject and a superclabject. Class-clabject and superclabject of a clabject must have a common ancestor (No. 13). Further, if a clabject x, which is an instantiation of c, specializes a clabject y, which is an instantiation of d, then c needs to be a specialization of d (No. 14). In the forthcoming sections we will introduce further constraints on the combined use of generalization and instantiation.

Two clabjects may be related via property references. A quintuple $R(x, i, p, j, y)$ is an asserted property reference of source clabject x via a property p to target clabject y with source potency i and target potency j (No. 16), we also say there is a p^{i-j} reference from x to y. The source potency and the target potency

Table 2. Deep Characterization of Clabjects via Property References

Additional Sorts & Asserted Predicates:
(15) P: properties
(16) $R \subseteq C \times \mathbb{N} \times P \times \mathbb{N} \times C$
Additional Well-formedness Criteria and Syntactic Restrictions:
(17) $R(x,i,p,j,y) \wedge R(s,k,p,l,t) \rightarrow \exists c \exists m \exists n \exists d : R(c,m,p,n,d) \wedge isa^*(x,c) \wedge isa^*(s,c)$
(18) $R(x,i,p,j,y) \wedge R(x,k,p,l,d) \rightarrow i = k \wedge j = l \wedge y = d$
(19) $R(x,i,p,j,y) \rightarrow ptcy(x) \geq i \wedge ptcy(y) \geq j$
(20) $R(x,i,p,j,y) \wedge R(c,k,p,l,d) \wedge nmember(x,c,n) \rightarrow n = k - i$
(21) $R(x,i,p,j,y) \wedge R(c,k,p,l,d) \wedge nmember(y,d,n) \rightarrow n = l - j$
(22) $R(x,i,p,j,y) \wedge R(c,k,p,l,d) \wedge isa^+(x,c) \rightarrow isa^*(y,d)$

indicate how many instantiation levels below the source clabject and the target clabject, respectively, the property is to be ultimately instantiated. For example, the $\mathsf{soldPrice}^{2-3}$ reference from Product to Currency is ultimately instantiated by the $\mathsf{soldPrice}^{0-0}$ reference from PetersZ4 to 38,200.

Well-formed property references obey the following constraints. Every property is introduced with a single clabject, i.e., if two clabjects have the same property, then they must have a common ancestor which introduced that property (No. 17). For simplicity (and space limitations) we only consider single-valued properties, that is, there may only be a single property reference per source clabject and property (No. 18). The source and target potency must be lower or equal to the potency of the source and target clabject, respectively (No. 19). When instantiating and refining properties, source and target potencies must be reduced according to the number of instantiation steps between the source clabjects (No. 20) and target clabjects (No. 21), respectively. A clabject c with a reference to clabject d via property p with a target potency higher than 0 or if d is abstract introduces a range referential integrity constraint for all descendants of c: descendants of c may only refer via p to descendants of d (No. 22). This is akin to co-variant refinement and, in terms of the UML, to redefinition of association ends.

3 The Abstract Superclass Rule in the Context of Abstract and Concrete Clabjects

In this section we clarify the distinction between abstract and concrete clabject. We revisit the *abstract superclass rule* and adapt it to the setting of multi-level modeling with abstract and concrete clabjects.

Abstract clabjects combine aspects of abstract classes and abstract objects. *Concrete clabjects* combine aspects of concrete classes and concrete objects. The distinction between abstract and concrete class is described as: 'A class that has the ability to create instances is referred to as *instantiable* or *concrete*, otherwise it is called *abstract.*' [3] The distinction between *abstract objects* and *concrete objects* is heavily discussed in Philosophy [13] and there are many different ways to explain it. In this paper we follow 'the way of abstraction' which is also followed by Kühne [4]: 'An abstract object represents all instances that are considered to be equivalent to other for a certain purpose [...] An abstract object captures what is universal about a set of instances but resides at the same logical level as the instances'. In DDI, clabjects are either asserted as abstract (No. 23 in Table 3) or derived to be *concrete* (No. 24).

Table 3. Abstract and Concrete Clabjects and the Abstract Superclabject Rule

(23) $\mathit{abstract} \subseteq C$
(24) $\mathit{concrete}(x) :\Leftrightarrow x \in C \wedge \neg \mathit{abstract}(x)$
(25) $\mathit{spec}(x,y) \rightarrow \mathit{abstract}(y)$
(26) $\mathit{abstract}(c) \wedge \mathit{in}(x,c) \rightarrow \mathit{abstract}(x)$
(27) $\mathit{concrete}(x) \wedge \mathit{member}(x,y) \rightarrow \exists z : \mathit{concrete}(z) \wedge \mathit{member}(x,z)$

In the literature it has been proposed that only abstract classes may be specialized:

Abstract Superclass Rule: All superclasses are abstract [3] in that they have no direct instances.

Obeying this rule improves the clarity of object-oriented models, especially when the extension (set of instances) of classes is of interest. Despite the trade-off of additional classes to be modeled and maintained, we feel that obeying the abstract superclass rule in multi-level modeling is beneficial because of the increased clarity. That is why in the original DDI approach [9] only concrete clabjects could be instantiated. We now relax this restriction as follows:

Abstract Superclabject Rule: All superclabjects are abstract in that they have no direct concrete instances (but they may have abstract instances).

This is formalized as: All superclabjects are abstract (No. 25). If an abstract clabject acts as class in an instantiation relationship, then the clabject playing the instance role must be abstract as well (No. 26). Every concrete clabject that is member of some clabject must be member of a concrete clabject (No. 27).

The meaning of the allowed kinds of instantiation relationships depends on the abstractness of the related clabjects. An instantiation relationship between a clabject x in the role of the instance and a clabject c in the role of the class, denoted as $in(x,c)$, can be classified as one of the following:

– An *immediate concrete instantiation* relationship is between a concrete clabject in the instance role and a concrete clabject in the class role. For example, the relationship between BMW Z4 and Car is an immediate concrete instantiation relationship, meaning that BMW Z4 is an instance of Car, or, more exactly, that the object facet of BMW Z4 is an instance of a class-facet of Car.

- A *shared concrete instantiation* relationship is between an abstract clabject x in the instance role and a concrete class c in the class role. For example, the relationship between BMW Motorcycle and Motorcycle is a shared concrete instantiation, meaning that all concrete specializations of BMW Motorcycle, such as F700GS and R1200GS, are instances of Motorcycle.
- A *shared abstract instantiation* relationship is between an abstract clabject x in the instance role and an abstract clabject c in the class role. For example, the relationship between BMW Vehicle and Vehicle is a shared abstract instantiation relationship, meaning that all concrete specializations of BMW Vehicle are instances of a concrete specialization of Vehicle, e.g., BMW Z4 is an instance of Car.

4 Coping with Dual Inheritance

In this section we define the mechanism for inheritance of property references along generalization and instantiation relationships. A clabject inherits both from its class-clabject and from its superclabject. We refer to this specific form of multiple inheritance as *dual inheritance*. Dual inheritance leads to potential conflicts. We propose one way to guarantee that concrete clabjects are conflict-free and, thus, satisfiable.

Table 4. Dual Inheritance

(28)	$R^\circ(x,i,p,j,y) :\Leftrightarrow R(x,i,p,j,y) \vee (\exists s : \mathit{spec}(x,s) \wedge R^\star(s,i,p,j,y))$ $\vee (\exists c : \mathit{in}(x,c) \wedge R^\star(c,i+1,p,j,y) \wedge i \geq 0)$
(29)	$R^\star(x,i,p,j,y) :\Leftrightarrow R^\circ(x,i,p,j,y) \wedge \neg(\exists j' \exists y' : \mathit{isa}^+(y',y) \wedge R^\circ(x,i,p,j',y'))$
(30)	$\mathit{concrete}(x) \wedge \mathit{spec}(x,s) \wedge R^\star(s,i,p,j,y) \wedge \mathit{in}(x,c) \wedge R^\star(c,i+1,p,j',y')$ $\rightarrow \mathit{isa}^*(y,y') \vee \mathit{isa}^*(y',y) \vee (\exists j'' \exists y'' : R(x,i,p,j'',y''))$

Inheritance of property references is defined using the following predicates: predicate R (No. 16 in Table 2) holds *asserted property references* of all clabjects. Auxiliary predicate R° (No. 28 in Table 4) holds *asserted and inherited property references*. Derived predicate $R^\star$ (No. 29) holds *effective property references* which are the most-specific property references out of the asserted and inherited property references.

We first look at asserted and inherited property references (No. 28). From its superclabject a clabject inherits all effective property references. From its class-clabject it inherits all effective property references with a source potency of 1 or above. When inheriting property references from the class-clabject, the source potency is decremented by 1. For example, clabject BMW Z4 inherits from its superclabject BMW Vehicle a soldPrice reference to Euro and from its class-clabject Car a soldPrice reference to EuropeanCurrency.

An inherited or asserted property reference of a clabject x at source potency i for property p to target object y is effective if x has no inherited or asserted property reference for property p to a target object which is a descendant of y (No. 29). For example, the reference to Euro is the effective soldPrice reference for clabject BMW Z4 since it is more specific than the reference to EuropeanCurrency.

We now discuss the role of concrete clabjects in resolving or detecting conflicts introduced by dual inheritance along generalization and instantiation relationships. If a concrete clabject inherits from its superclabject and from its class-clabject references for property p to y and y', respectively, we demand that one of the references is a descendant of the other. If this is not the case the modeler needs to resolve the potential conflict by asserting a reference of property p to some clabject y'' (No. 30) which needs to be, due to the previously introduced constraints, a descendant of both y and y'. If this is not possible, the modeler detects a conflict. This guarantees that concrete clabjects that obey this constraint are satisfiable at the instantiation levels beneath. Conflicts are resolved or detected at concrete objects. For example, BMW Z4 inherits for property engine via specialization from BMW Vehicle and via instantiation from Car references to BMW Engine and CarEngine, respectively. To avoid a potential conflict, BMW Z4 asserts for property engine a reference to EngineK5.

Other approaches to detecting and resolving conflicts are (1) to ignore the problem and accept the possiblity of unsatisfiable properties, (2) to use more sophisticated techniques to decide whether two conflicting property references are satisfiable or not, or (3) to make the above check not only for concrete clabjects but also for abstract clabjects, for example it would make necessary to add a property reference from BMW Motorcycle to a to-be created clabject, e.g., called BMW MotorcycleEngine, that specializes BMW Engine and instantiates MotorcycleEngine and which is a generalization of BMW F700GS and BMW R1200GS. With regard to the effort associated with such an immediate conflict resolution, delaying conflict resolution to concrete clabjects, as introduced above, seems to be a good compromise.

5 Mandatory Constraints at Multiple Levels

By now, it is up to the modeler to decide whether properties are to be instantiated and at which levels they are to be instantiated. In this short section we introduce support for defining mandatory constraints at multiple levels. Mandatory constraints allow to control the stepwise instantiation process by demanding that concrete source clabjects at a given level of the domain of the property need to refer to a concrete clabject at a given level of the range of the property or to a clabject that is a descendant of a concrete clabject at the given level.

In more formal terms, a mandatory constraint $total(i, p, j)$ (No. 31 in Table 5) expresses that property p, which is introduced between clabject c and d with potencies n and m, is mandatory for potencies i and j, with potencies i and j being lower or equal to potencies n and m, respectively (No. 32). This means

Table 5. Mandatory constraints

(31)	$total \subseteq \mathbb{N} \times P \times \mathbb{N}$
(32)	$total(i,p,j) \rightarrow \exists c \exists n \exists m \exists d : R(c,n,p,m,d) \wedge i \leq n \wedge j \leq m$
(33)	$R(c,n,p,m,d) \wedge total(i,p,j) \wedge j \leq m \wedge nmember(x,c,n-i) \wedge concrete(x)$ $\rightarrow \exists j' \exists y \exists y' : R^{\star}(x,i,p,j',y') \wedge isa^{*}(y',y) \wedge concrete(y) \wedge nmember(y,d,m-j)$

that concrete $(n-i)$-members of c must refer via property p to a clabject that is a (descendant of a) $(m-j)$-member of d (No. 33).

For example, property engine is introduced at clabject Vehicle by a reference with source potency 2 and target potency 2 to clabject Engine. The multi-level domain of property engine is given by the 0-, 1-, and 2-members of Vehicle and its multi-level range is given by the 0-, 1-, and 2-members of Engine. Mandatory constraint $total(2, \mathsf{engine}, 2)$ demands that every concrete 0-member of Vehicle refers to some concrete 0-member of Engine or to a descendant of Engine.

6 Related Work

DDI is heavily influenced by the classical work on deep instantiation of Atkinson and Kühne [1]. Kühne and Schreiber [5] introduced the notion of *superclabject* and propose the use of metaclass compatibility rules and represent the abstractness of clabjects by giving them potency 0. De Lara et al. [7] propose to declare abstract clabjects as such. In both approaches, abstractness of clabjects only refers to the inability to create instances; abstract clabjects in the dual sense of abstract object and abstract class are not discussed. Kühne [4] provides an in-depth discussion of the distinction between generalization and classification together with a discussion of abstract objects.

From *M-Objects and M-Relationships* [8], DDI takes the idea that instantiation (then called concretization) levels have a label and that every instantiation (or concretization) hierarchy has its own set of instantiation levels. M-Objects do not come with the possibility to model generalization hierarchies of m-objects at one instantiation level. A comparison with different techniques for deep characterization, then called 'multi-level abstraction', is given in [10]. DDI is further influenced by Pirotte et al's work on Materialization [12]. Similar to M-Objects, materialization does not come with support for generalizing objects at the same abstraction level and does not come with the distinction between concrete and abstract classes. Many aspects of clabject hierarchies with deep instantiation may be alternatively modeled using powertypes [11] or the powertype pattern [2] (see [9]). It is, however, unclear how generalization of clabjects may be modeled using powertypes or the powertype pattern.

The most important related work is that of de Lara et al. [6] on the uniform handling of inheritance at every meta-level, which however does not come with the simplicity and conceptual clarity provided by the abstract superclabject rule. It is open to future work to analyze the trade-offs of both approaches and to combine the advantages of both approaches.

7 Conclusion

We have discussed the distinction between abstract and concrete clabjects as one way of clarifying the meaning of superclabjects in multi-level models. In a simplified setting (only considering single-valued and uni-directional property references) we have introduced and relaxed the abstract superclabject rule, showed how the distinction between abstract and concrete clabjects helps to cope with dual inheritance, and introduced support for mandatory constraints over concrete clabjects at multiple levels. We currently work on extending the full DDI approach (also considering bi-directional and multi-valued relationships) and its ConceptBase implementation [9] along the lines discussed in this paper, especially on extending DDI with full-fledged multi-level multiplicity constraints.

References

1. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: Gogolla, M., Kobryn, C. (eds.) Proceedings of the 4^{th} International Conference on the UML 2001, Toronto, Canada. LNCS, vol. 2185, pp. 19–33. Springer Verlag (Oct 2001)
2. Eriksson, O., Henderson-Sellers, B., Ågerfalk, P.J.: Ontological and linguistic metamodelling revisited: A language use approach. Information & Software Technology 55(12), 2099–2124 (2013)
3. Hürsch, W.L.: Should superclasses be abstract? In: Tokoro, M., Pareschi, R. (eds.) ECOOP. LNCS, vol. 821, pp. 12–31. Springer (1994)
4. Kühne, T.: Contrasting classification with generalisation. In: Kirchberg, M., Link, S. (eds.) APCCM. CRPIT, vol. 96, pp. 71–78. Australian Computer Society (2009)
5. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style: multi-level programming with deepjava. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) OOPSLA. pp. 229–244. ACM (2007)
6. de Lara, J., Guerra, E., Cobos, R., Moreno-Llorena, J.: Extending deep metamodelling for practical model-driven engineering. Comput. J. 57(1), 36–58 (2014)
7. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific meta-modelling languages. Software & Systems Modeling pp. 1–31 (2013)
8. Neumayr, B., Grün, K., Schrefl, M.: Multi-Level Domain Modeling with M-Objects and M-Relationships. In: Link, S., Kirchberg, M. (eds.) APCCM. CRPIT, vol. 96, pp. 107–116. ACS, Wellington, New Zealand (2009)
9. Neumayr, B., Jeusfeld, M.A., Schrefl, M., Schütz, C.: Dual deep instantiation and its conceptbase implementation. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE. Lecture Notes in Computer Science, vol. 8484, pp. 503–517. Springer (2014)
10. Neumayr, B., Schrefl, M., Thalheim, B.: Modeling techniques for multi-level abstraction. In: Kaschek, R., Delcambre, L.M.L. (eds.) The Evolution of Conceptual Modeling. LNCS, vol. 6520, pp. 68–92. Springer (2008)
11. Odell, J.: Power types. JOOP 7(2), 8–12 (1994)
12. Pirotte, A., Zimányi, E., Massart, D., Yakusheva, T.: Materialization: A Powerful and Ubiquitous Abstraction Pattern. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) VLDB. pp. 630–641. Morgan Kaufmann (1994), 0605
13. Rosen, G.: Abstract objects. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Fall 2014 edn. (2014)

On Metamodel Superstructures Employing UML Generalization Features

Martin Gogolla, Matthias Sedlmeier, Lars Hamann, Frank Hilken

Database Systems Group, University of Bremen, Germany
{gogolla|ms|lhamann|fhilken}@informatik.uni-bremen.de

Abstract. We employ UML generalization features in order to describe multi-level metamodels and their connections. The basic idea is to represent several metamodel levels in one UML and OCL model and to connect the metamodels with (what we call) a superstructure. The advantage of having various levels in one model lies in the uniform handling of all levels and in the availability of OCL for constraining models and navigating between them. We establish the connection between the metamodel levels by typing links that represent the instance-of relationship. These typing links are derived from associations that are defined on an abstraction of the metamodel classes and that are restricted by `redefines` and `union` constraints in order to achieve level-conformant typing. The abstraction of the metamodel classes together with the connecting associations and generalizations constitutes the superstructure.

Keywords. UML, OCL, Model, Metamodel, Metamodel constraint, Generalization, Redefines constraint, Union constraint.

1 Introduction

Software engineering research activities and results indicate that metamodeling is becoming more and more important [3, 4, 9]. However, there are a lot of discussions about notions in connection with metamodels like *potency* or *clabject* where no final conceptual definition has been achieved. On the other hand, software tools for metamodeling are beginning to be developed [5, 2].

Here, we propose to join the metamodels of several levels into one model (as in our previous work [7] without the use of `redefines` constraints) and to connect the levels with associations and generalizations. Typing conformance and strictness can be achieved through particular UML and OCL generalization constraints. General restrictions between the metamodel levels can be specified through the power of OCL. Restrictions can be built on metamodels and on the metamodel architecture. The metamodel architecture is the connection between (what we call) the metamodel superstructure and the contributing metamodels.

Our work has links to other metamodeling approaches. The tool Melanie [2] is designed as an Eclipse plug-in supporting strict multi-level metamodeling and support for general purpose as well as domain specific languages. Another tool is MetaDepth [5] allowing linguistic as well as ontological instantiation with an arbitrary number of metalevels supporting the potency concept. In [9] the authors describe an approach to flatten metalevel hierarchies and seek for a level-agnostic metamodeling style in contrast to the OMG four-layer architecture.

The structure of the rest of the paper is as follows. Section 2 gives a first, smaller example for a metamodel superstructure. Section 3 discusses a larger example. Section 4 shows other metamodel superstructures. The contribution is closed with a conclusion and future work in sect. 5.

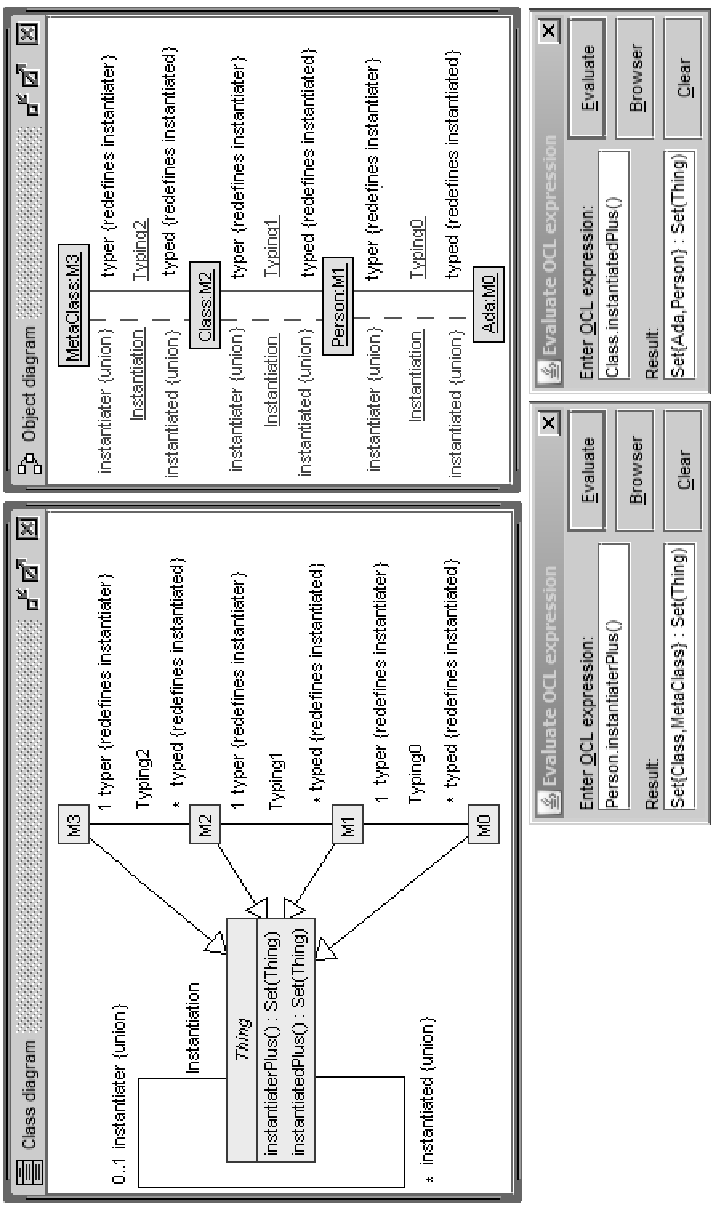
2 Superstructure Example with Ada, Person, Class, and MetaClass

The example in Fig. 1 shows a substantially reduced and abstracted version of the OMG four-level metamodel architecture with modeling levels M0, M1, M2, and M3. Roughly speaking, the figure states: `Ada` is a `Person`, `Person` is a `Class`, and `Class` is a `MetaClass`. The figure does so by formally building an object diagram for a precisely defined class diagram including an OCL invariant that requires cyclefreeness when constructing instance-of connections. The distinction between `MetaClass` and `Class` is that when `MetaClass` is instantiated something is created that can be instantiated on two lower levels whereas for `Class` instantiation can only be done on one lower level. The model has been formally checked with the tool USE [6]. In particular, we have employed the features supporting UML generalization constraints as discussed in [1, 8].

Concepts on a respective level M_x are represented in a simplified way as a class M_x. All classes M_x are specializations of the abstract class `Thing` whose objects cover all objects in the classes M_x. On that abstract class `Thing` one association `Instantiation` is defined that is intended to represent the instance-of connections between a higher level object and a lower level: an object of a lower level is intended to be an instance of an object on a higher level. The association `Instantiation` on `Thing` (with role names `instantiater` and `instantiated`) is employed for the definition of the associations `Typing0`, `Typing1`, and `Typing2` between M_x and M_{x+1} all having roles `typer` and `typed`. The role `typer` is a redefinition of `instantiater`, and `typed` is a redefinition of `instantiated`. The multiplicity `1` of `typer` narrows the multiplicity `0..1` of `instantiater`.

In the abstract class `Thing` the transitive closure `instantiatedPlus()` of `instantiated` is defined by means of OCL. Analogously, `instantiaterPlus()` is defined for `instantiater`. The closure operations are needed to define an invariant in class `Thing` requiring `Instantiation` links to be acyclic.

```
abstract class Thing
operations
  instantiatedPlus():Set(Thing)=
    self.instantiated->closure(t|t.instantiated)
  instantiaterPlus():Set(Thing)= ...
```


Class diagram
Thing
instantiaterPlus() : Set(Thing)
instantiatedPlus() : Set(Thing)
Instantiation
0..1 instantiater {union}
* instantiated {union}
M3
M2
M1
M0
1 typer {redefines instantiater}
Typing2
* typed {redefines instantiated}
1 typer {redefines instantiater}
Typing1
* typed {redefines instantiated}
1 typer {redefines instantiater}
Typing0
* typed {redefines instantiated}
Object diagram
MetaClass:M3
Class:M2
Person:M1
Ada:M0
instantiater {union}
Instantiation
instantiated {union}
typer {redefines instantiater}
Typing2
typed {redefines instantiated}
typer {redefines instantiater}
Typing1
typed {redefines instantiated}
typer {redefines instantiater}
Typing0
typed {redefines instantiated}
Evaluate OCL expression
Evaluate
Browser
Clear

Enter OCL expression:
Person.instantiaterPlus()
Result:
Set{Class,MetaClass} : Set(Thing)

Enter OCL expression:
Class.instantiatedPlus()
Result:
Set{Ada,Person} : Set(Thing)

Fig. 1. Ada, Person, Class, MetaClass within Single Object Diagram.

```
constraints
  inv acyclicInstantiation: self.instantiatedPlus()->excludes(self)
end
```

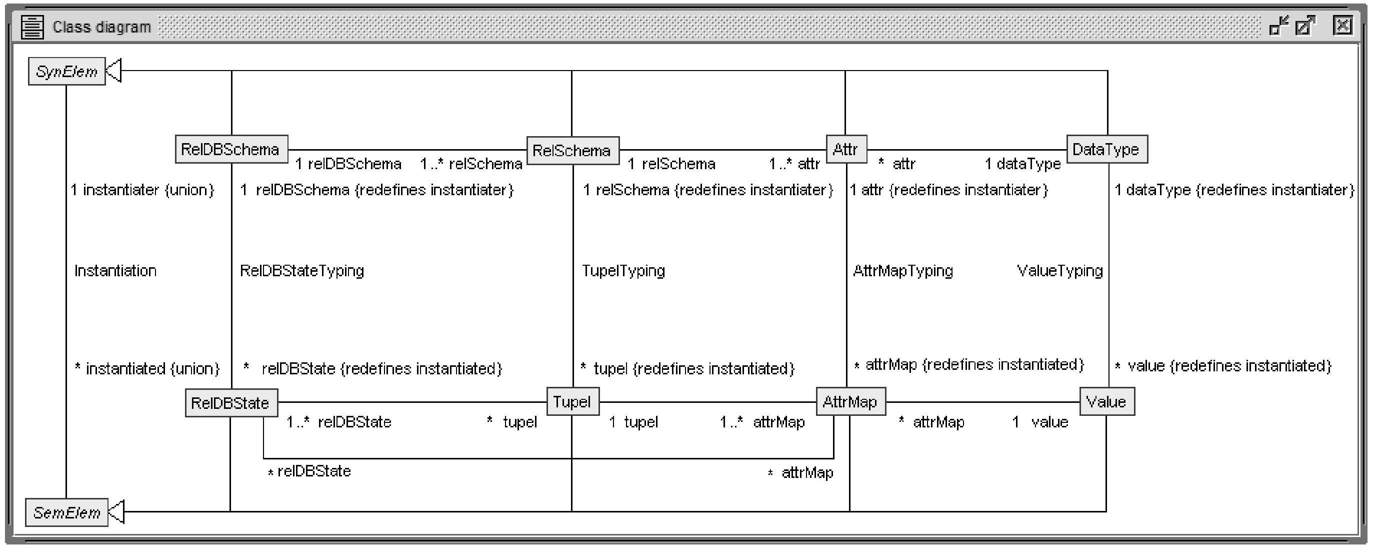
The class diagram from the left side of Fig. 1 is made concrete with an object diagram on the right side. The fact that the three associations `Typing0`, `Typing1`, and `Typing2` are all redefinitions of association `Instantiation` is reflected in the object diagram by the three dashed links for association `Instantiation` with common role names `instantiater` and `instantiated` (dashed links in contrast to continuous links for ordinary links). Viewing `Instantiation` as a generalization (in terms of redefinition) of all $\texttt{Typing}_x$ associations allows to use the closure operations from class `Thing` on objects from classes `M0`, `M1`, `M2` or `M3`. Thus the displayed OCL expressions and their results reflect the following facts: object `Person` is a (direct resp. indirect) instantiation of objects `Class` and `MetaClass`; objects `Ada` and `Person` are (direct resp. indirect) instantiations of object `Class`.

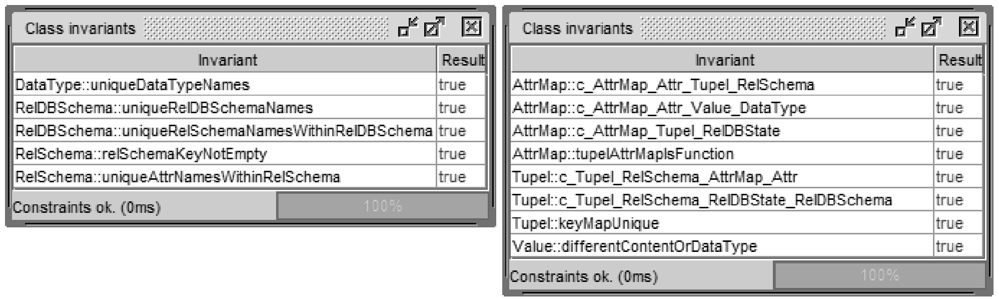
Summary: Metamodeling means to construct models for several levels. The metamodels on the respective level should be described and modeled independently (e.g., as M0, M1, M2, and M3). The connection between the models should be established in a formal way by a typing association (e.g., `Typing0` gives a type object from `M1` to a typed object from `M0`). The Typing associations are introduced as redefined versions of the association `Instantiation` from (what we call) a multi-level *superstructure*. This superstructure contains the abstract class `Thing` which is an abstraction of all metamodel elements across all levels and additionally contains the association `Instantiation` and accompanying constraints. Because `Instantiation` is defined as `union`, an `Instantiation` link can only connect elements of adjacent levels, i.e., the $\texttt{Typing}_x$ links are level-conformant and strict. The aim of the devices in the superstructure is to establish the connection between metamodel levels in a formal way and to provide support for formally restricting the connections.

3 Superstructure Example for Relational Database Model

In Fig. 2 we show two metamodels, one for the syntax and one for the semantics part of the relational database model. The upper part catches the syntax, i.e., relational database schemas, relational schemas, attributes, and data types. With regard to the class names, please recall that in the database field a *relational database schema* consists of possibly many *relational schemas*.[1] The lower part deals with the semantics (or runtime interpretation), i.e., database states, tuples, attribute mappings (for short attribute maps), and values. One also identifies two collections of invariant names, one collection for the syntax,

[1] A relational schema is called a table in SQL.




Class invariants

Invariant	Result
DataType::uniqueDataTypeNames	true
RelDBSchema::uniqueRelDBSchemaNames	true
RelDBSchema::uniqueRelSchemaNamesWithinRelDBSchema	true
RelSchema::relSchemaKeyNotEmpty	true
RelSchema::uniqueAttrNamesWithinRelSchema	true

Constraints ok. (0ms) 100%

Class invariants

Invariant	Result
AttrMap::c_AttrMap_Attr_Tupel_RelSchema	true
AttrMap::c_AttrMap_Attr_Value_DataType	true
AttrMap::c_AttrMap_Tupel_RelDBState	true
AttrMap::tupelAttrMapIsFunction	true
Tupel::c_Tupel_RelSchema_AttrMap_Attr	true
Tupel::c_Tupel_RelSchema_RelDBState_RelDBSchema	true
Tupel::keyMapUnique	true
Value::differentContentOrDataType	true

Constraints ok. (0ms) 100%

Fig. 2. Metamodels for Relational Database Schemas and States.

and one for the semantics. For example, `RelSchema::relSchemaKeyNotEmpty` requires that each relational schema must have at least one key attribute, and `Tupel::keyMapUnique` requires that two different tuples[2] must be distinguishable by at least one key attribute in each database state. The constraints starting with `c_` are (what we call) commutativity constraints which require that the evaluation of two different paths through the class diagram coincide. Both paths start in one class and typically end in one different class. For example, the constraint `AttrMap::c_AttrMap_Attr_Tupel_RelSchema` requires that for an object `am:AttrMap` the paths `am.attr.relSchema` and `am.tupel.relSchema` coincide.

In the left of the class diagram we identify the metamodel superstructure established by the abstract classes `SynElem`, `SemElem`, and the association `Instantiation`. `Instantiation` is specialized through redefinition to `RelDBStateTyping`, `TupelTyping`, `AttrMapTyping`, and `ValueTyping`. We regard the syntax model, i.e., `SynElem` and its specializations together with the associations, as a metamodel of the semantics model, i.e., `SemElem` and its specializations together with the associations. We take this view because each higher level class (in the syntax part) serves to instantiate a lower level class (in the semantics part), and thus each lower level object has exactly one type that is defined in the higher level. Another argument supporting the view that we here have two connected metamodels is the factor that the relationship between `RelSchema` and `Tupel` is the same as the relationship between `Class` and `Object` in the OMG four-level architecture. The same holds for the other (`SynElem`,`SemElem`) class pairs: (`RelDBSchema`,`RelDBState`), (`Attr`,`AttrMap`), and (`DataType`,`Value`).

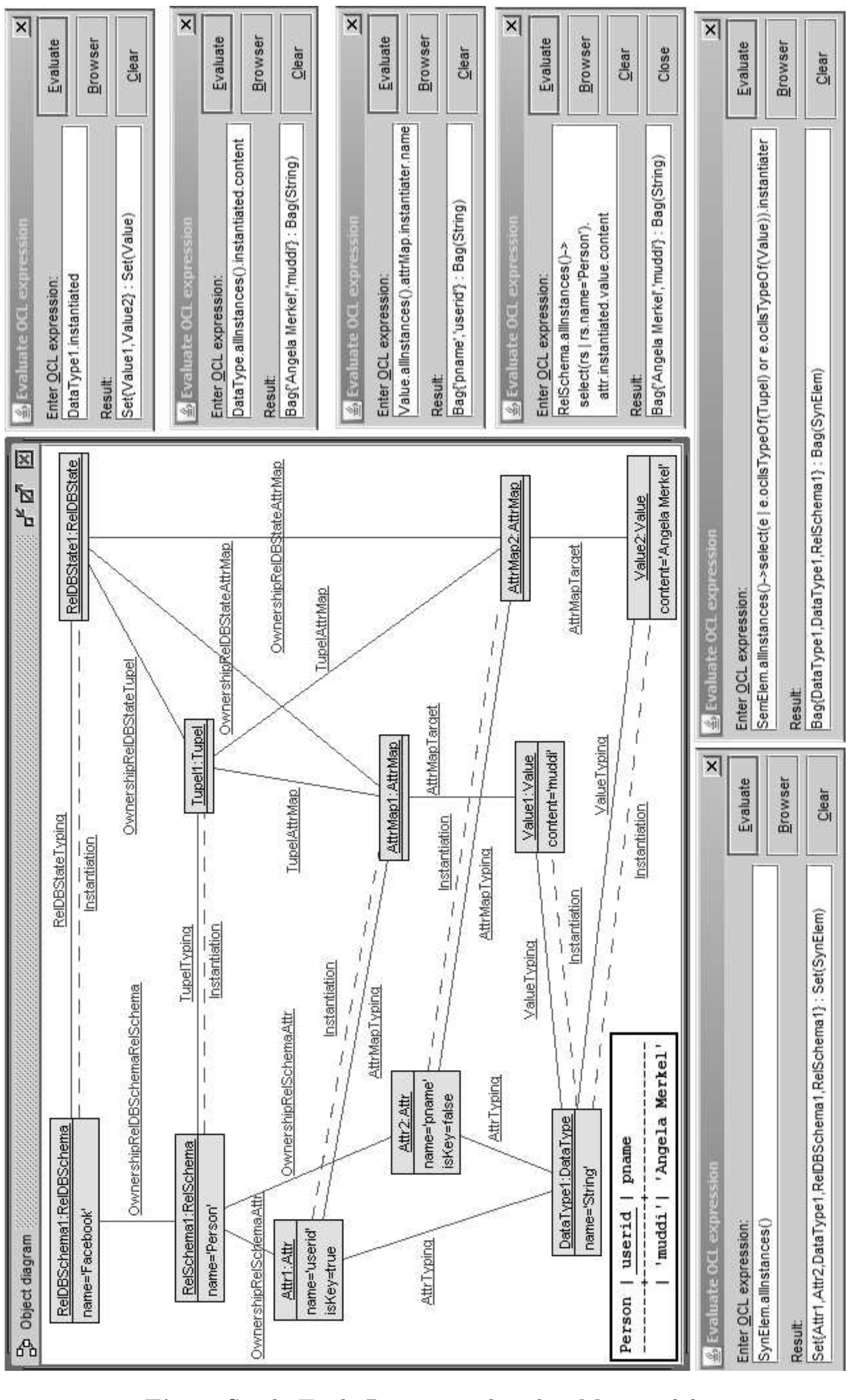
Interestingly, some invariants span across metamodel boundaries, i.e., an invariant from the semantics part sometimes uses elements from the syntax part. For example, the mentioned uniqueness requirement for tuples with regard to their key attributes is only required, if the tuples under consideration belong to the same relational schema. Thus the invariants of the semantics part rely on or use information from the syntax part.

In Fig. 3 we make the metamodels from Fig. 2 concrete by presenting a simple relational database schema consisting of one relational schema and a very simple accompanying database state with only one tuple. The presentation is done in form of an object diagram. The figure shows also OCL queries and their result that demonstrate how one can bridge the boundary between the metamodels. All queries either use the roles `instantiater` or `instantiated` which cross a metamodel boundary. For example, the fourth query from the top (`RelSchema.allInstances()->select(rs | rs.name='Person').attr.instantiated.value.content`) retrieves all values that are present in one of the attributes of the relational schema `Person`.

In Fig. 4 we show a larger object diagram[3] with two relational database states, two relational schemas and three tuples. The object diagram satisfies all invari-

[2] We have used the German spelling `Tupel` because `Tuple` is a keyword in OCL.

[3] In order to make the figure easier to grasp some links are hidden.



```
Person | userid  | pname
-------+---------+----------------
       | 'muddi' | 'Angela Merkel'
```

Evaluate OCL expression

Enter OCL expression:
DataType1.instantiated
Result:
Set{Value1,Value2} : Set(Value)

Enter OCL expression:
DataType.allInstances().instantiated.content
Result:
Bag{'Angela Merkel','muddi'} : Bag(String)

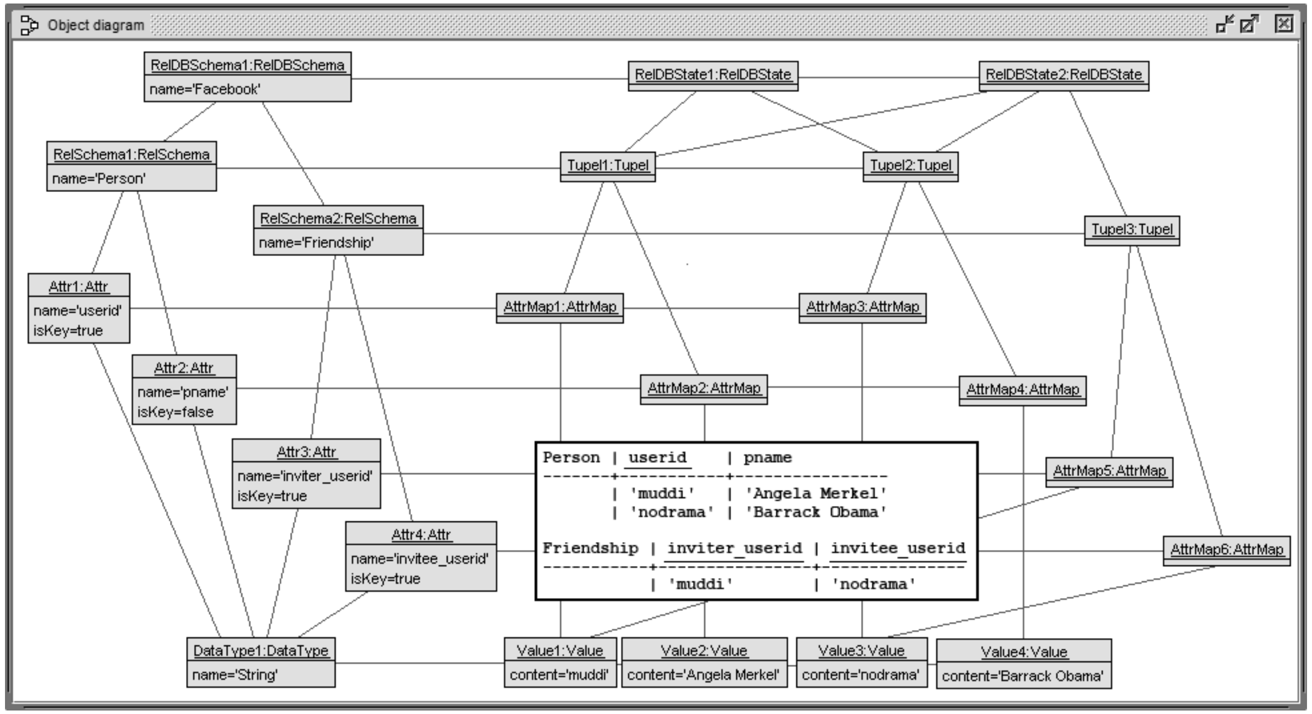
Enter OCL expression:
Value.allInstances().attrMap.instantiater.name
Result:
Bag{'pname','userid'} : Bag(String)

Enter OCL expression:
RelSchema.allInstances()->
  select(rs | rs.name='Person').
    attr.instantiated.value.content
Result:
Bag{'Angela Merkel','muddi'} : Bag(String)

Enter OCL expression:
SynElem.allInstances()
Result:
Set{Attr1,Attr2,DataType1,RelDBSchema1,RelSchema1} : Set(SynElem)

Enter OCL expression:
SemElem.allInstances()->select(e | e.oclIsTypeOf(Tupel) or e.oclIsTypeOf(Value)).instantiater
Result:
Bag{DataType1,DataType1,RelSchema1} : Bag(SynElem)

Fig. 3. Single Tuple Represented within Metamodel.



```
Person | userid    | pname
-------+-----------+-----------------
       | 'muddi'   | 'Angela Merkel'
       | 'nodrama' | 'Barrack Obama'

Friendship | inviter_userid | invitee_userid
-----------+----------------+---------------
           | 'muddi'        | 'nodrama'
```

Fig. 4. Three Tuples Represented within Metamodel.

ants. The metamodels reflect the syntactical and semantical requirements, in particular through the use of constraints. For example, if one changes in the object `Attr1:Attr` the `isKey` attribute value from `true` to `false`, the syntactical requirement that relational schemas must have at least one key attribute value would be violated and this would be indicated by a constraint violation for the respective constraint `RelSchema::relSchemaKeyNotEmpty`. As an example on the semantical side, if one changes in the object `Value1:Value` the `content` attribute value from `'muddi'` to `'nodrama'`, the semantical requirement that each two tuples must have at least one distinguishing key attribute would be violated and this would be indicated by a constraint violation for the respective constraint `Tupel::keyMapUnique`.

4 Other Metamodel Superstructures

In the two examples above we have employed different metamodel superstructures. The first example Ada-Person-Class-MetaClass used the superstructure displayed in the upper left part of Fig. 5. The second example for the relational

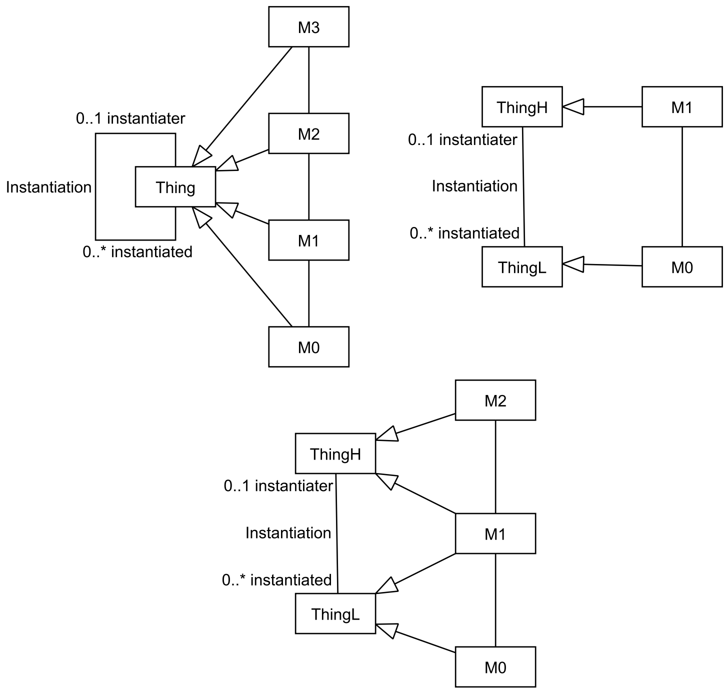


Fig. 5. Three Different Multi-Level Metamodel Superstructures.

database model used more or less the superstructure shown in the upper right part of the figure. However instead of the generic class names `ThingH[igh]` and `ThingL[ow]` the example used `SynElem` and `SemElem`, and instead of `M1` and `M0` the example used a bunch of connected classes like {`RelDBSchema`, `RelSchema`, `Attr`, `DataType`} and {`RelDBState`, `Tupel`, `AttrMap`, `Value`}. Other metamodeling superstructures could be used as well, for example the one displayed in the lower part of Fig. 5 utilizing multiple inheritance. Dependent on the actual needs for the metamodels at hand, a suitable superstructure with fitting classes, associations, and constraints can be chosen.

In our example superstructures we have been using the multiplicities `0..1` or `1` for the roles `instantiater` and `typer`. However, in principle other multiplicities like `1..*` could be used. It is an open question whether this could make sense, for example, in the context of multiple inheritance.

5 Conclusion

This paper proposed to describe different metamodels in one model and to connect the metamodels with a (so-called) superstructure consisting of generalizations and associations with appropriate UML and OCL constraints. We explained our ideas in particular with an example expressing the syntax and the semantics of the relational database model on different metamodel levels.

Future research includes the following topics. We would like to work out for our approach formal definitions for notions like potency or strictness. The notion of powertype will be given special attention in order to explore how far this concept can be integrated. Our tool USE could be extended to deal with different metamodel levels simultaneously. So far USE deals with class and object diagram. In essence, we think of at least a three-level USE (cubeUSE) where the middle level can be seen at the same time as an object and class diagram. Furthermore, larger examples and case studies must check the practicability of the proposal.

References

1. Alanen, M., Porres, I.: A Metamodeling Language Supporting Subset and Union Properties. Software and System Modeling **7**(1) (2008) 103–124
2. Atkinson, C.: Multi-Level Modeling with Melanie. Commit Workshop 2012 (2012) commit.wim.uni-mannheim.de/uploads/media/commitWorkshop_Atkinson.pdf.
3. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software **20**(5) (2003) 36–41
4. Bézivin, J.: On the Unification Power of Models. Software and System Modeling **4**(2) (2005) 171–188
5. de Lara, J., Guerra, E.: Deep Meta-Modelling with Metadepth. In Vitek, J., ed.: TOOLS (48). Volume 6141 of LNCS., Springer (2010) 1–20
6. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Comp. Prog. **69** (2007) 27–34
7. Gogolla, M., Favre, J.M., Büttner, F.: On Squeezing M0, M1, M2, and M3 into a Single Object Diagram. In Baar, T. et al., eds.: Proc. MoDELS'2005 Workshop Tool Support for OCL and Related Formalisms, EPFL (Switzerland), LGL-REPORT-2005-001 (2005)
8. Hamann, L., Gogolla, M.: Endogenous Metamodeling Semantics for Structural UML2 Concepts. In Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P.J., eds.: MoDELS. Volume 8107 of LNCS., Springer (2013) 488–504
9. Henderson-Sellers, B., Clark, T., Gonzalez-Perez, C.: On the Search for a Level-Agnostic Modelling Language. In Salinesi, C., Norrie, M.C., Pastor, O., eds.: CAiSE. Volume 7908 of LNCS., Springer (2013) 240–255

Instance Specialization – a Pattern for Multi-level Meta Modelling

Matthias Jahn, Bastian Roth and Stefan Jablonski

Chair for Applied Computer Science IV: Databases and Information Systems
University of Bayreuth, Universitätsstraße 30, 95447 Bayreuth, Germany
{Matthias.Jahn, Bastian.Roth, Stefan.Jablonski}@uni-bayreuth.de

Abstract. Conciseness is one major quality aspect for meta models. To keep them concise, language patterns like inheritance or powertypes can be used in an appropriated way. With instance specialization we present a further language pattern that rests on the idea of prototypal inheritance (e.g., known from Python or ECMAScript). Generally, it allows for a concept to specialize the instance facet of a particular instance and reuse its configuration. Thereby, all assignments of the latter are inherited by a specializing instance, which can be overwritten in different ways within this instance. Beyond describing the instance specialization pattern, we also introduce a semi-automatic, user-supporting mechanism for applying this pattern to existing meta models.

Keywords: meta modelling, meta model evolution, instance specialization, inheritance, prototypal inheritance

1 Motivation

In the field of software engineering meta models are often utilized to define the abstract and concrete syntax of domain specific modelling languages (DSMLs). Hence, the quality of such a DSML depends highly on the quality of those meta models describing it. The quality of a meta model is influenced by various aspects like conciseness, simplicity and extensibility [4]. For improving these aspects, in recent years several meta modelling patterns like clabjects [1, 2] or inheritance were discovered.

In general, multilevel meta modelling aims in contrast to common programming languages at defining more than two meta levels leveraging the modeller to create a higher degree of abstraction without manually (re-)implementing an instantiation mechanism [6, 17]. In modelling environments or programming languages supporting two meta levels the elements of the meta level are typically called classes or types whereas the elements of the instance level are called objects or instances. For multilevel environments elements can act as both [14]: as a type for an instance level's element and as an instance of another element of a higher meta level. Containing either an instance and a type facet, such elements are often called Clabjects (CLAss + obJECT) [1, 2] or concepts [12]. Hence, such concepts can define attributes in the type facet and assignments to attributes of the concept's type within the instance facet.

adfa, p. 1, 2011.

Beside instantiation, inheritance is another frequently used pattern to improve quality of meta models symbolizing the “is a” relationship between two different concepts [8]. Nevertheless, this relationship is limited to the type facet of both involved concepts whereas the instance facet is not affected. In modern programming languages like ECMAScript with prototypal inheritance a different pattern occurs, which expresses a specialization of the instance facet. Furthermore, this pattern can be observed in various domains like process modelling (type-usage) [11], car modelling (chapter 4.3) or graphical frameworks [18]. Bridging the described gap in meta modelling, in this paper we introduce the pattern of instance specialization for meta modelling and show how it can be integrated into a meta modelling platform. Furthermore, we present an operator that allows for applying the presented pattern to an existing meta model with full support of model migration.

2 Related Work

The pattern of instance specialization occurs in various domains. Also many programming languages use the paradigm of prototype-based inheritance, e.g. ECMAScript, Ruby, Python, Logtalk or OpenLaszlo. Beside programming languages the pattern of instance specialization is also used in the modelling domain. Lieberman [13] introduces the prototype pattern for object oriented systems and shows the advantages (flexibility) of delegation in contrast to inheritance. As mentioned in the paper [13], inheritance implements sets whereas delegation implements prototypes. Up to our knowledge, in the field of meta modelling there is only one approach that introduces the pattern of instance specialization. Volz presents the pattern of instance specialization in [20].

Other domains like process modelling (type-usage[11]) or graphical model environments [18] also use this pattern. Nevertheless, those approaches mainly need to implement the aimed behavior of the pattern manually since the according environments do not provide instance specialization support.

Additional to the lacking of out of the box support, none approach exists that supports applying instance specialization to an existing model. In the field of meta model evolution several operators were discovered [9, 10, 22] but none of them provide support for the presented pattern.

3 Inheritance and Instance Specialization

In this section we first take a look at traditional type specialization, which is typically called inheritance. Afterwards, we introduce the instance specialization pattern and explain how it can coexist and interact with inheritance.

3.1 Inheritance

The principle of inheritance is an often used pattern for object oriented software design (e.g. [15]). It is generally represented as an “is a” relationship between two concepts.

However, this is problematic, since instantiation is also used for that relationship [8]. The base class is called generalization and the other class specialization. Inheritance influences the structure of the according specialization concept. On the one hand attributes of the generalization are inherited and on the other hand the substitution principle is applied to the specialization, i.e., if an instance of the generalization is expected an instance of the specialization is valid as well.

In the multilevel meta modelling context this definition has to be more precise since each element has a type facet and an instance facet. According to the common semantic of inheritance, inheritance influences the type facets of both involved concepts whereas the instance facet is not affected. Hence, this relation links the type facet of the specialization to the type facet of the generalization with the effect that attributes declared at the generalization are inherited to the specialization.

3.2 Instance Specialization

As explained above, inheritance is a relationship that merely extends the type facet of a generalized concept. Nevertheless, in various use cases a specialization of the instance facet is needed (e.g. Process Modelling [11], Lieberman [13]). Similar to common programming languages, declaring a prototype (the base concept) of a specific concept (the instance specialization) enables inheriting concrete attribute values (assignments). Accordingly, the instance specialization is a relationship between two concepts linking their instance facets. The difference between inheritance and instance specialization was discussed in the programming language community for years (e.g. [13, 19]). Nevertheless, for multi-level meta modelling instance specialization is not limited to the instance level but can be applied to any concept.

To interact with inheritance the substitution principle needs to be extended for instance specialization. Hence, instance specialization defines the substitution principle for the instance facets of both concepts, i.e., if an instance of a concept is expected as an attribute value, an instance specialization of that instance is also a valid value. For example, if concept `A` declares an attribute `attr` with concept `B` as its attribute type. Furthermore, let `InstB` be an instance of `B` and `Special` an instance specialization of `InstB`. Then, each instance of `A` may assign `Special` (and `InstB`) for `attr`.

Instance specialization is hence a new relation between two concepts that does not exclude an inheritance between those two concepts. Since both relationships use different concept facets they can interact harmless with each other (on a conceptual point of view).

Overwrite Behavior.
The core idea of this pattern is an inheritance of assignments that are defined at the prototype of an instance specialization concept. However, these assignments may be overwritten by the instance specialization if needed. Since this behavior is not always suitable, the prototype can define whether and how an assignment can be overwritten by an instance specialization.

To configure this, each prototype can declare the overwrite behavior for every assignment. The possible strategies are:

- **Type 0 (forbidden):** An instance specialization is not allowed to overwrite the value of its prototype. The assigned value is always inherited from the prototype.
- **Type 1 (normal, default type):** The prototype's value for the specific attribute can be overwritten in any way by an instance specialization. If no type is specified explicitly type 1 is applied for the particular assignment. This type is also supported by languages like ECMAScript [7] that provide prototypal inheritance as an idiom.
- **Type 2 (limited):** The assignment at the prototype defines the domain of all values that are assigned at an instance specialization, i.e., the values of the instance specialization are a subset of the values defined at the prototype. A similar type is shown by Pirotte et.al. [16] with the difference that the type is not declared at the assignment but at the attribute and thus acts for all according assignments. This type is restricted to assignments which base upon a multi-valued attribute.
- **Type 3 (append):** The value of an instance specialization is appended to the value of the prototype for getting the concrete attribute's value.
- **Type 4 (prepend).** Similar to type 3 the concrete value of the instance specialization is a result of the assigned value together with the prototype's value. Instead, the instance specialization's value is prepended to the prototype's value.

The two types 3 and 4 are only applicable to assignments whose attribute is multi-valued or a string attribute. For strings, assignments within an instance specialization results in a concatenation. For collections, however, it leads to appending or prepending the value(s) of the specialization concept to the prototype's values. Apparently, the both types are equal if they are not ordered within the according collection (e.g., a set). A similar declaration (with some differences) of such types was introduced by Volz [20] but he limits the overwrite behavior types 3 and 4 to strings. The information about the overwrite behavior is stored within the linguistic meta model [21], which is an implementation of the orthogonal classification.

Example.
A typical scenario for instance specialization could be a model for cars. Often manufacturers offer their cars in a base series that can be specialized in various ways. In **Fig. 1** we give a possible example. At level `M1` we have modeled the concept `Car`, which is a representation of the real world counterpart and declares the attributes `typeName`, `manufacturer` and `releaseDate`. Each car may have some equipment (concept `Equipment` with a relation to `Car`).

At level `M0` an instance model is shown. Therein, a car `Ibiza` is modeled that has the name `"Ibiza"` (according assignment to `typeName`), produced by `Seat` (assignment to `manufacturer`), was released on the 1st of January 2009 (assignment to `releaseDate`) and may be equipped with the packages `ABS` and `ESC` (assignment to `equipment`). Each of these attributes defines a specific overwrite behavior. Since every instance specialization of the `Ibiza` base series will be produced by the same manufacturer (`Seat)`, the attribute `manufacturer` declares the overwrite type 0. In contrast to that `releaseDate` can be overwritten in any way. Owing to the fact that a new special car series may only have a subset of all possible equipment packages, the overwrite behavior of `equipment` is set to type 2 (limited). At last, `typeName` can be extended by any instance specialization. That is why the according assignment of `Ibiza` has the overwrite behavior type 3. Additionally, an instance specialization `IbizaReference` is available, which concretely uses the according prototype `Car`. The relationship between these two concepts is equipped with an arrow labeled with `<<concreteUseOf>>` to designate the instance specialization .

The instance specialization `IbizaReference` is a special series of Ibiza that has a special type name ("Ibiza Reference"), a different release date and the `ABS` equipment package as standard equipment. Because of that, `IbizaReference` overwrites `releaseDate` with the value `"2010-04-01"`, sets `typeName` to `" Reference"`, which implicitly results in `"Ibiza Reference"`, and finally chooses `ABS` for `equipment`. The attribute manufacturer cannot be overwritten and is hence inherited from the prototype `Ibiza`.

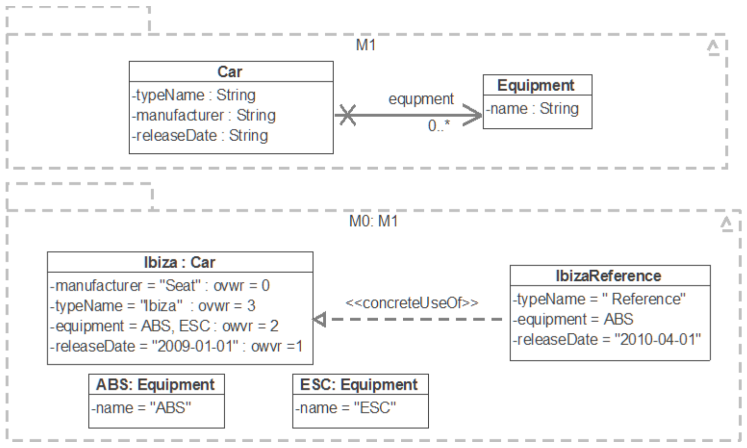


Fig. 1. Car model example

4 Extract Prototype

In this section we present a way for (semi-)automatically introducing an instance specialization into an existing model. This mechanism is supplied by a specific operator, which extracts a prototype out of instances that are similar. It can be seen as a counterpart of the extract super class refactoring method that is provided by many IDE or modelling systems [10, 22].

4.1 Overview

The *Extract Prototype* Operator creates a prototype out of similar instances of one type. Thereby, the operator sets the assignments at the prototype according to the chosen overwrite behavior and updates each instance specialization assignment if necessary.

4.2 Operator process

The operator process is shown by **Fig. 2**. Therein all steps or decisions that need user interaction are highlighted in black. At the beginning the operator is invoked on a concept `Base`, which instantiates another concept `Type`. In the first step this instantiated type is ascertained together with all instances of it. Out of this set a subset of all future instance specializations (including `Base` by default) is chosen. In the following we call this subset `Instances`. Afterwards, the operator fetches all attributes declared at `Type` that can be set at `Base`.
In the next step, a subset (`AttrsToSet`) out of these attributes have to be chosen, which will be set on the prototype. Of course, all attributes of `Type` that are mandatory (multiplicity `1` or `1..*`) have to be part of this subset. After that, for each attribute of `AttrsToSet` the according assignments that were declared at an element of `Instances` are ascertained since they influence the attributes value at the prototype. Subsequently, the prototype is created. Thereby, the prototype's name is defined and an instantiation to `Type` is created. After that, an assignment for each attribute of `AttrsToSet` is created at the new prototype and together with it, the overwrite behavior is defined by the user. In the last activity of the operator the value of each assignment is determined depending on the chosen overwrite behavior:

- **Type 0 (forbidden)**: If a change of the assignment's value is forbidden at an instance specialization, a specific value that was assigned at an element of `Instances` has to be chosen, which acts as new value for the prototype. Since the assignments of all elements of `Instances` are not valid anymore, they will be deleted afterwards.
- **Type 1 (normal)**: If this type is selected, each instance specialization may overwrite the attributes value in any way. Hence, just a selection of the new value out of those made at the elements of `Instances` for the prototype is needed. Then, all assignments that are equal to the chosen value and those that should be deleted (user selection) are removed from the according elements of `Instances`.
- **Type 2 (limited)**: For this type all assignment values of `Instances` are inserted into a set that acts as the resulting value for the prototype. Here, no further adaption is needed since all instance specializations values lie in the created domain.
- **Type 3 (append) and Type 4 (prepend)**: In this case a base value for the prototype has to be chosen. This value may consist of some values or a substring that was assigned at an element of `Instances`.

In the last step the relationship for the instance specialization is created. Thereby, all instantiations of `Instances` are deleted since an implicit instantiation exists via the instance specialization. Furthermore, all assignments of new instance specializations

(Instances) are deleted if overwriting is forbidden (type 0) or are adapted (respectively new chosen) if a base value was selected for the prototype assignment (type 3 and 4).

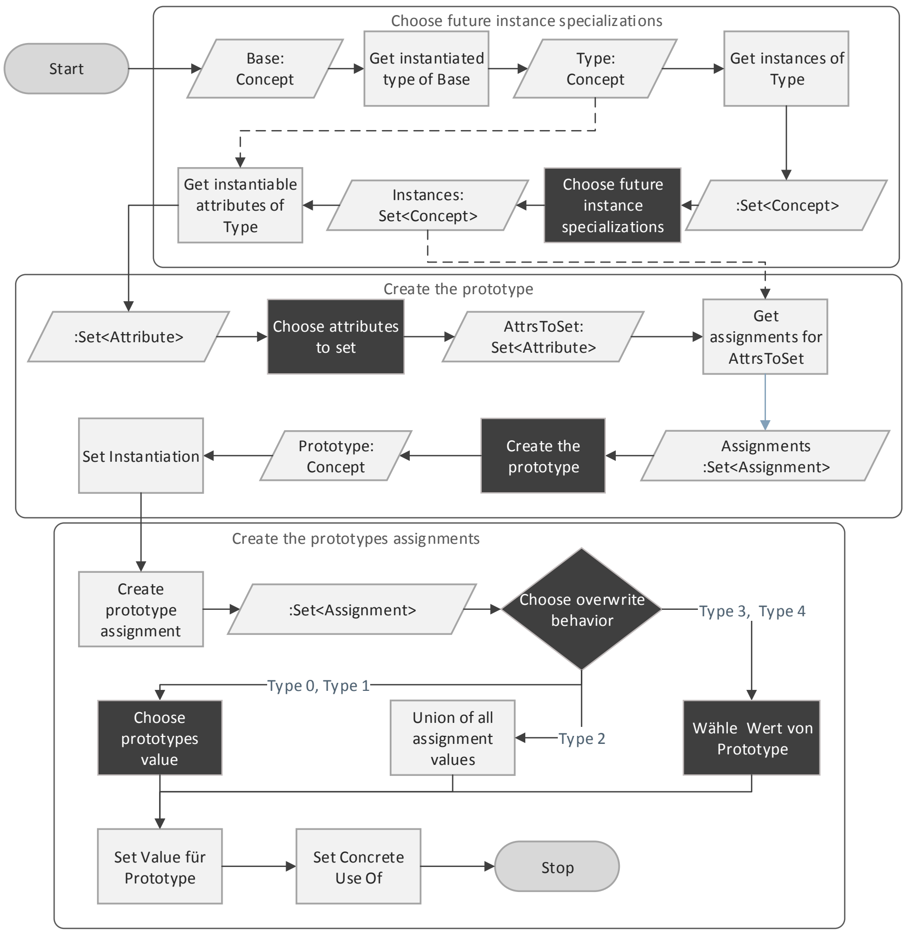


Fig. 2. Process of the Extract Prototype Operator

4.3 Example

In **Fig. 2** a model is shown on which we will demonstrate how the operator works. Therein a DSML for describing cars is presented. Hence, at the top level `M1` a concept `Car` is modeled representing the according real world element with a `manufacturer` attribute, a type (`typeName`), a release date (`releaseDate`) and a relationship to `Equipment` (attribute is called `equipment`). In general, a car may have various equipment parts.

One level below (`M0`) an instance model is given containing two instances of `Car` (`IbizaStyle`, `IbizaReference`). Both cars are produced by the manufacturer "`Seat`" and their type is almost equal to their concept's name. Additional to the both `Car` instances, the `M0` level contains two instances of `Equipment` (`ABS`, `ESC`) representing the anti-blocking system and the electronic stability control of a car. According to the modeled scenario (not the real life), `IbizaReference` provides only `ABS` whereas the `IbizaStyle` also has an `ESC`.

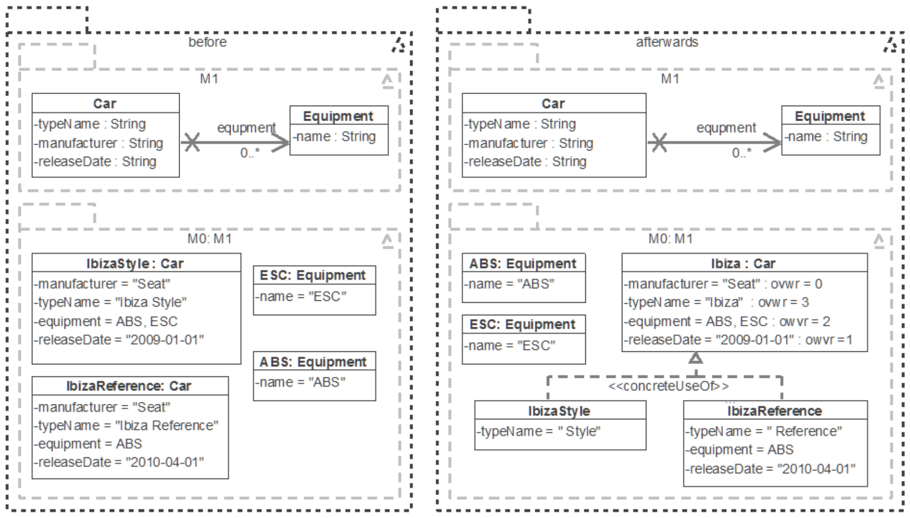


Fig. 3. Application of the operator to the car model example

Since both instances of `Car` can be seen as two different instance specializations of the car `Ibiza` we now invoke the *Extract Prototype* operator to create this prototype. That is why we select `IbizaReference` and call the operator on it. Afterwards, all instances of `Car` are gathered by the operator and we decide to add `IbizaStyle` to the set of future instance specializations. In the next step, all attributes of `Car` are calculated (`manufacturer`, `typeName` and `equipment`) that can be instantiated at `IbizaReference`. In our example we decide to set all of these attributes at the prototype and hence, all assignments relating to these three attributes of `IbizaStyle` and `IbizaReference` are collected. Subsequently, the new prototype can be created, which is called `Ibiza` and which becomes an instance of `Car`. Next, all assignments are created at the prototype with the following overwrite behaviors:

- `manufacturer` should not be overwritten by instance specializations and gets thus type 0 (forbidden)
- `typeName` can be extended by an instance specialization with any further string value and consequently gets type 3 (append)
- `equipment` gets type 2 (limited) because the prototype should declare all possible equipment parts

- `releaseDate` gets type 1 since the date can differ in each car series.

Owing to those overwrite behaviors, the assignments of `IbizaStyle` and `IbizaReference` for `manufacturer` are deleted and for `typeName` and `releaseDate` a value for `Ibiza` is selected (`"Ibiza"` and `"2009-01-01"`). According to that, the assignments of the two instance specializations for `typeName` are adapted to the new values `" Style"` or `" Reference"` respectively and thus the original value is retained virtually. The assignment for `releaseDate` of `IbizaReference` is not affected whereas the assignment of `IbizaStyle` is deleted because the value is equal to the prototype's value. For `equipment` the resulting value for `Ibiza` is the union of all values of the future instance specializations and hence `{ABS, ESC}`. The other assignments need not be adapted here since they are valid anymore. In the last step the instantiation of `IbizaStyle` and `IbizaReference` to `Car` is deleted and the instance specialization to `Ibiza` is created. Finally, the operator terminates. The resulting model is free of any redundant attribute values, which is a great benefit especially in case of models.

5 Conclusion and Outlook

Instance specialization as described in this paper is a language pattern that solely impacts on the instance facet of concepts. It enables users to easily define a default configuration regarding a certain use case, which then can be adapted through instance specialization for specific scenarios. We greatly utilize this patterns for the definition and usage of concrete syntaxes for DSMLs (similar to [18]). Thereby, a concrete syntax is determined as an instance of a given meta model designed for this particular purpose. Later on, the concepts of this syntax can be instance-specialized to shape the visual parts of concrete diagrams or documents. As a result, only one meta model is required to formulate diagrams and documents as well as the concrete syntax they base upon. Above all, the common model base extremely reduces the implementation effort for building a dedicated processing module, which can handle both, concrete syntax and all associated instance specializations, in an analogous manner. A suchlike DSML tool as well as the *Extract Prototype* operator introduced in section 4 is implemented on top of the Model Workbench [5], a web-based modelling platform. Since each atomic and complex model manipulation action has to be encapsulated by an operator, they constitute the core of this platform. Besides, the Model Workbench provides support for further multilevel meta modelling patterns (e.g., deep instantiation [3] and materialization [16]). For the future, we plan to offer user-guided support for the introduction of these patterns by means of suitable operators.

References

1. Atkinson, C.: Meta-modelling for distributed object environments. Proceedings of the 1st International Conference on Enterprise Distributed Object Computing (EDOC '97). pp. 90–101 IEEE (1997).

2. Atkinson, C., Kühne, T.: Meta-level independent modelling. Int. Work. Model Eng. 14th Eur. Conf. Object-Oriented Program. 12, 16 (2000).
3. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. «UML» 2001—The Unified Model. Lang. Model. Lang. Concepts, Tools, Lect. Notes Comput. Sci. 2185, 19–33 (2001).
4. Bertoa, M., Vallecillo, A.: Quality attributes for software metamodels. Proceedings of the In 13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2010) (2010). (2010).
5. Chair of Applied Computer Science IV - University of Bayreuth: Model Workbench, http://www.ai4.uni-bayreuth.de/de/research/projects/003_ModelWorkbench/index.html.
6. Demuth, A.: Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking. Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. 452–455 (2011).
7. Flanagan, D.: JavaScript: The Definitive Guide (Definitive Guides). O'Reilly Media, Inc, Sebastopol, CA (2011).
8. Frank, U.: Thoughts on classification/instantiation and generalisation/specialisation, http://www.econstor.eu/handle/10419/68462, (2012).
9. Herrmannsdoerfer, M. et al.: An extensive catalog of operators for the coupled evolution of metamodels and models. Softw. Lang. Eng. Lect. Notes Comput. Sci. 6563, 163–182 (2011).
10. Herrmannsdoerfer, M.: Evolutionary Metamodeling. PhD Thesis, Fakultät für Informatik, Technische Universität München (2011).
11. Jablonski, S., Bussler, C.: Workflow management: modeling concepts, architecture and implementation. International Thomson Computer Press (1996).
12. Jahn, M. et al.: Remodeling to Powertype Pattern. Proceedings of the Fifth International Conferences on Pervasive Patterns and Applications (PATTERNS 2013). pp. 59– 65 (2013).
13. Lieberman, H.: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Conference proceedings on Object-oriented programming systems, languages and applications (OOPLSA '86). pp. 214–223 (1986).
14. Odell, J.: Power types. J. Object-Oriented Program. 7, 2, 8–12 (1994).
15. OMG: Unified Modeling Language (OMG UML)-Infrastructure. Available http//www. omg. org/spec/UML/2.4.1. August, (2011).
16. Pirotte, A. et al.: Materialization : a powerful and ubiquitous pattern abstraction. Proc. 20th Int. Conf. Very Large Data Bases (VLDB '94). 630–641 (1994).
17. Roth, B. et al.: IT-as-a-Service for Building Virtual Research Environments. Proceedings of the 2rd International Conference on Cloud Computing and Service Science. , Porto, Portugal (2012).
18. Roth, B.: Konzeption und Implementierung eines generischen Modellierungswerkzeugs zur Unterstützung der domänenspezifischen Prozessmodellierung. (2010).
19. Stein, L.A.: Delegation Is Inheritance. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA). pp. 138–146 , Orlando, Florida, USA (1987).
20. Volz, B.: Werkzeugunterstützung für methodenneutrale Metamodellierung. PhD Thesis, Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth (2011).
21. Volz, B., Jablonski, S.: Towards an open meta modeling environment. Proceedings of the 10th Workshop on Domain-Specific Modeling - DSM '10. p. 1 ACM Press, New York, New York, USA (2010).
22. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. ECOOP 2007 – Object-Oriented Program. Lect. Notes Comput. Sci. 4609, 600–624 (2007).

An Implementation of Multi-Level Modelling in F-Logic

Muzaffar Igamberdiev, Georg Grossmann, and Markus Stumptner

Advanced Computing Research Centre
School of IT and Mathematical Sciences
University of South Australia, Mawson Lakes, SA 5095, Australia
{firstname.lastname}@unisa.edu.au

Abstract. Multi-level modelling is currently regaining attention in the database and software engineering community with different emerging proposals and implementations. One driver behind this trend is to reduce model complexity, a crucial aspect in a time of *big data* research in which more and more data from different sources are required to be integrated. From our experience, multi-level modelling also improves understanding of complex specifications, simplify their management and evolution, and facilitate interoperability between them. This paper focuses on the requirement of reasoning for interoperability. Although there exist formalisation approaches for multi-level modelling, only few have the implementation for three fundamental aspects: formalisation, querying and validation of multi-level models. We propose an F-Logic framework to implement these aspects. In addition, we believe this approach is more likely to be adapted in real-world use cases because of its simple object-oriented declarative nature.

Keywords: Multi-level modelling, interoperability, multi-level model reasoning, F-Logic, multi-level model querying, multi-level model validation

1 Introduction

Multi-level modelling (also called *deep meta-modelling/instantiation*) is currently regaining attention in the database and software engineering community with different emerging proposals and implementations. Recently there have been multiple works published which enrich multi-level modelling with new features [2,19], propose a formalisation for multi-level modelling [21] or demonstrate the practical application of it [8,13]. The most often mentioned arguments for multi-level modelling are *increased expressiveness*, by introducing multiple classifications [5], and *reduced complexity* [6,20]. This seems to be a contradiction, because one might expect increasing expressiveness may lead to increased complexity, but multiple classification allows to brake down a complex specification into smaller and simpler layers. Apart from the above mentioned advantages we have seen a further three advantages in a use case from the oil and gas industry [13]: (1) Simplification of the standards' specifications by classifying elements into ontological and linguistic elements, (2) simpler management and evolution of standards by structuring them into multiple ontological *instance-of* levels, and most importantly (3) checking specifications for consistency according to software engineering modelling principles.

The third advantage facilitates the interoperability between software systems. During the development life cycle of an interoperability solution, the matching, transformation, and synchronization of models and data rely on querying source and target specifications and checking for consistency to ensure a correct end result (i.e. validation). It only becomes possible with a formal specification that can be executed.

For multi-level modelling there exist some formalisation approaches. Neumayr et al. [19] proposed ConceptBase as the underlying formalisation framework. ConceptBase is a metamodeling system based on Datalog and the Telos data model [18]. Rossini et al. [21] implemented the semantics using the Diagram Predicate Framework (DPF) and Golra et al. [11] used a graph algebra.

We propose a novel approach, namely Multi-level Modelling in F-Logic (MiF), for the implementation, querying and validation of the multi-level models. We propose to use F-Logic as an alternative implementation for the following reasons: in comparison to DPF and graph-based approaches, F-Logic is object-oriented and represents an integrated framework which allows the specification of the semantics and ontological models as well as perform reasoning. ConceptBase also aims at an integrated approach but is not as widely accepted as F-Logic. In particular in the Ontology and Semantic Web community, e.g., there exist commercial and open-source implementations such as OntoBroker [1] and Flora [2], and it has been used for interoperability, such as the Rule Interchange Format [14][3] and model transformation [15].
In the next section we describe our motivating use case from the oil and gas industry, followed by a description of the implementation of multi-level modelling and related work.

2 Oil & Gas Interoperability Pilot

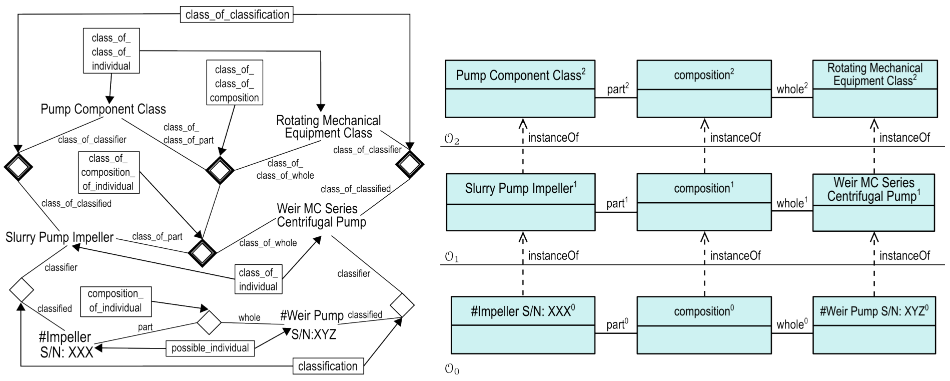
A large-scale standard-based interoperability is one of the main challenges in the oil and gas industry. Some comparable figures of how much an inadequate interoperability costs came from the US Capital Facilities Industry and the construction and engineering domain with an estimate of about $15.8 billion per year [9,10].

A lot of effort has been invested into data standards to overcome the interoperability issue in the oil and gas industry. One effort is the joint academic-industry project *Oil & Gas Interoperability (OGI) Pilot* hosted by MIMOSA[4] and supported by the ISO TC 184 OGI Technical Specification project. The

[1] OntoBroker: http://www.semafora-systems.com/
[2] Flora-2: http://flora.sourceforge.net/
[3] RIF: http://www.w3.org/2001/sw/wiki/RIF
[4] MIMOSA: http://www.mimosa.org

goal of the OGI Pilot is increased automation in the digital hand-over of design information of very large physical assets to the operation and maintenance side. This requires identifying commonalities and differences as well as open gaps in the specifications of the major standards in the area: ISO 15926 [12] and MIMOSA's Open Systems Architecture for Enterprise Application Integration (OSA-EAI) [17]. Within the OGI Pilot we have identified some challenges [16] of which we will focus on ISO 15926 standard in this paper. The ISO standard relies on a 4-dimensional information model specified in STEP/EXPRESS, RDF, OWL, and first order logic.

Example: Our motivating example is taken from an engineering diagram which specifies that "*The impeller with serial number XXX is part of the Weir Pump with serial number XYZ*". Figure 1a displays the flat model of this example using the *instance diagram* notation appearing throughout the ISO 15926 documentation [12].



(a) Instance diagram (b) Multi-level representation

Fig. 1: Two representations of the same example: *relating an impeller to a pump.*

In ISO 15926 terminology each box represents a *class* which is part of the specification and identified by its label. A diamond represents a *relationship* where a diamond with a thick line represents a *class of relationship.* A (class of) relationship has *roles* which are displayed by labelled arcs connected to the diamond, e.g., "part" and "whole" are the two roles of "composition_of_individual". A symbol with prefix # is a *possible individual with a temporal part*, e.g., "#Impeller S/N: XXX" is a possible temporal part with identifier "Impeller S/N: XXX". Remaining elements are *classes* identified by its label, e.g., "Pump Component Class".

Jordan et al. [13] applied rules on the ISO 15926 specification for the transformation of the flat model into a multi-level model. For example, one of those rules assigns a model level according to the prefix of a class label, e.g., "class_of_composition" is an instance of "class_of_class_of_composition" and removes the prefix "class_of_" to simplify the notation. Figure 1b shows the result of applying those rules on the example introduced above.

Some of the advantages of a multi-level model representation over the flat model are: (1) explicit *instance-of* relationships, (2) separation of concerns through multiple levels making it easier for users to focus on particular aspects of the model, (3) reduced complexity, and (4) clarified terminology which improves understandability.

In order to verify the multi-level models and perform queries, for example, to support matching with other standards for model transformation, we require a formal framework. We propose to use F-Logic because of its object-oriented semantics and its wide acceptance in the ontology, business rules and model transformation communities[1].

3 Implementation of Multi-level Modelling in F-Logic

This section introduces the semantics of multi-level modelling, its implementation, querying and validation in F-Logic.

3.1 Multi-level Modelling Semantics

The semantics of multi-level modelling involves the characterization of concepts and the definition of relationships between them. Similarly to semantics definition in two-level modelling, which is designated by its meta-model, the multi-level modelling semantics have been described by its meta-model that includes a *linguistic meta-model* and an *ontological stack* [3,21].

The fundamental concepts of multi-level modelling are characterized by linguistic and ontological perspectives. The linguistic meta-model deals with syntax and grammar, whereas the ontological stack addresses structural hierarchies and classification of an underlying domain. These perspectives make understanding of multi-level modelling easier and clearer. We also define the semantics in the light of the two perspectives.

The link between multi-level modelling perspectives is established by a *linguistic instance-of relationship.* It differs from the relationships within the linguistic meta-model and ontological stack in the sense that it connects the concepts across the perspectives. Every model element and relationship in the ontological stack is a linguistic instance-of the concepts from the linguistic meta-model. An ontological model element may or may not have an ontological type, but the linguistic type (e.g. clabject) is mandatory.

Meta-model for use case: The multi-level modelling meta-model for our use case is illustrated in Figure 2. It is organized in linguistic meta-model and ontological stack.

Some of the concepts are inspired by the work on multi-level modelling and the formalisation of deep meta-modelling [4,21]: The root element in the linguistic meta-model is called *instantiable element*, meaning all model elements are instantiable in the ontological stack. Depending on the representation of the

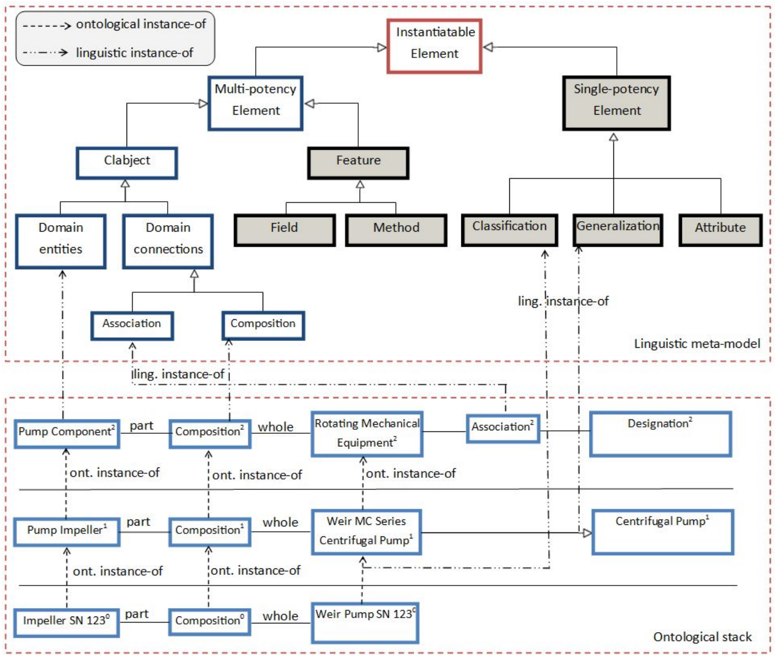


Fig. 2: Multi-level modelling meta-model with the motivational example

instantiated model element across ontological levels, instantiable elements are categorized as *multi-* and *single-potency elements* in Figure 2. While generalization and classification relationships cannot be instantiated across ontological levels, association and composition relationships can. The former ones are the examples for single-potency and latter ones for multi-potency elements. While the single potency elements can be instantiated in any ontological level and will not have further instances, the multi-potency elements can have further instances depending on the value of their potency[21]. The composition relationship is illustrated as a multi-potency relationship between *impeller* and *pump* elements in the example. The same multi-potency semantics is valid for clabject and association relationship.

Generalization relationship and attribute model elements are sub-class of a single-potency element. The difference between attribute and feature is that while the former one is considered as a single-potency, the latter one has multi-potency characteristics. Attribute is a property of the model element that can be instantiated in any ontological level and will not be instantiated in the next levels (single-potency). Alternatively, feature (field and method) is multi-potency element and can be instantiated across ontological levels. Clabject is sub-classed into *domain entity* and *domain connection*. Domain entities are the clabjects that characterize the domain concepts and domain connections address domain specific relationship, e.g., association and composition are domain connections.

The generalization is demonstrated with a relationship between *Weir MC Series Centrifugal Pump* and more general concept *Centrifugal Pump* in the ontological stack. The association is illustrated with a relationship between *Rotating Mechanical Equipment* and *Designation* model elements, that represents that rotating mechanical equipment has a designation.

3.2 Implementation in Flora-2

In this section, we first introduce F-Logic briefly, then discuss built-in features of F-Logic which directly support part of the meta-model and finally describe how we add new semantics to fully support the meta-model. Due to lack of space, we provide the excerpts of the semantics implementation.

F-Logic stands for Frame Logic, frame-based, object-oriented knowledge representation and reasoning language. It has a declarative, compact, simple and expressive syntax with well-defined semantics. These characteristics makes it attractive to apply on integration of information, semantic search, intelligent agents, semantic web and other areas.

In this paper, we use one of F-Logic implementations: Flora-2 [22]. Flora-2 is a dialect of F-Logic with numerous extensions and it supports extensibility, flexibility and modularity through dynamic modules. It is more suitable for a knowledge representation and reasoning in a way that multi-model semantics can be compactly expressed and the constrains can be checked, validation rules can be applied and more importantly the multi-model concepts and relationships can be easily queried. The source code presented in this paper is based on Flora-2 syntax. We now continue with the overlapping and distinct features of F-Logic and multi-level modelling.
Direct support: Some of the multi-level modelling concepts can be mapped directly to F-Logic. Concepts which are supported directly are represented with in grey in Figure 2. The mapping between modelling concepts and F-Logic elements are illustrated in Table 1.

MLM Concept	**F-Logic Concept**	**Flora-2 Example**
Generalization	Subclass	`A::B. subclass::class.`
Classification	"IS A" relationship	`A:B. object:class.`
Attribute	If M in `O[M->V]` is a constant it is dealt as an attribute	`Pump[component->'impeller'].`

Table 1: Multi-level modelling concepts and equivalent representation in F-Logic and its implementation in Flora-2.

Multi-level modelling aware extensions: The linguistic meta-model elements with white background in Figure 2 represent model elements and relationships that have not been addressed by F-Logic yet. Due to space limitation we provide its the implementation only for the linguistic and ontological instance-of relationship:

Linguistic instance-of relationship: It is similar to the IS-A(instance-of) relationship implemented by the colon (:) operator in F-Logic. We introduced a new operator, (<:), for the linguistic instance-of relationship with the following short excerpt of validation:

```
1. :- _op(400,xfx,<:).
2. linguistic_instance_of_validation(?X, ?Y) :-
3.    ?X <:: ?Y,
4.    ?Y \= clabject,
5.    writeln(?X, ' can only be an instance of CLABJECT') @ _plg.
```

The operator is defined by `_op(400,xfx,<:)` statement. The first argument (400) defines precedence order to follow when statement contains other elements. We define precedence as the same as F-logic's instance-of relationship precedence.
The Lines 2-5 represent a rule in F-Logic to validate the linguistic instance-of relationship. If the model element in ontological stack is not a linguistic instance of clabject (`?Y\=clabject`), then it prints a validation fail message on the screen (Line 5).

Ontological instance-of relationship: We introduced new operator '<::', which specifies the semantics of the ontological instance-of relationship. The rule (Lines 2-5) checks the potency before ontologically instantiating the model element.

```
1. :- _op(400,xfx,<::).
2. ?X <:: ?Y :-
3.      ?Y[potency -> ?_P], ?_P == 0,
4.      \+(?X[potency -> ?_P - 1]),
5.      writeln('cannot instantiate! It should be P > 0.') @ _plg.
```

The second argument of the `_op()` operator, `xfx` is used to define the type where `f` stands for the operator, and `x` and `y` stand for the arguments. The negation operator is denoted by `\+` symbol in Flora-2.

3.3 Querying and Validation

An essential feature of F-logic is reasoning. This paper focuses on the querying and validation aspects of reasoning. A knowledge base is built based on the facts (e.g. like the ones in the previous subsection) and can be easily queried. The following facts illustrate ontological instance-of relationships on the motivational example.

```
weir_mc_series_centrifugal_pump<::rotating_mechanical_equipment.
weir_pump_sn_123<::weir_mc_series_centrifugal_pump.
```

Validation rules can be introduced to check certain properties of the concepts and relationships. For example, the following predicate, validation rule checks for the condition that a potency of a property should be equal or less than the potency of an object.

```
validate_property_potency(?Prop):-
   ?X[property->?Prop],
   ?X[potency->?XPot],
   ?Prop[potency(?X)->?PropPot],
   ?PropPot=<?XPot.
```

Further, the knowledge base can be queried. For example, to determine the potency or all instances of a particular pump:

```
?- weir_pump_sn_123[potency->?Potency].
?- writeln('Give me all instances of the pump') @ _plg,
   instances(?X, weir_mc_series_centrifugal_pump).
```

4 Related work

Even though multi-level modelling was introduced more than ten years ago, the formalisation of its semantics has only recently been addressed. Research on multi-level modelling started to get momentum, and some formalization or implementation attempts were made. In this section we compare related work using six comparison criteria: *(1) Linguistic extension and open semantics*: support to extend the linguistic meta-model with new concepts and relationships. E.g. specifying semantics of "membership" relationship, *(2) object-oriented semantics*: the framework is based or supports object-oriented modelling principles, *(3) integrated framework*: a single framework for formalization, querying and validation, *(4) relationship across levels*: a support for the relationships across levels in the ontological stack, *(5) mediation of relationship*: a need of intermediate relationship to instantiate the relationship in not-immediate ontological/instantiation level, and *(6) single and multi-potency semantics*: a support of single and multi-potency concepts in the ontological stack (see Figure 2).

The evaluation of some multi-level modelling approaches is illustrated in Table 2.

The linguistic and semantic extension criteria are covered by almost all of approaches. The object-oriented criterion is supported by most of approaches including this paper. We benefit from object-orientation in two ways: (1) It already covers part of multi-level modelling and (2) it is built-in paradigm of F-Logic. In the context of the integrated framework criterion, ConceptBase [19] (based on Datalog and Telos) addresses formalism and validation, and METADEPTH [21] deals with all components of the integrated framework:

Criteria/Approaches	[4]	[11]	[19]	[21]	[7]	MiF
Linguistic extension and open semantics	–	✓	✓	✓	✓	✓
Object-oriented	✓	–	✓	–	✓	✓
Integrated framework	✓	✓	–	–	✓	✓
Relationship across levels	–	–	✓	✓	–	–
Mediation of relationship	–	–	✓	✓	–	✓
Single and multi-potency semantics	✓	–	✓	✓	✓	✓

Table 2: Evaluation of multi-level modelling approaches.

formalization(METADEPTH), querying(Epsilon Object Language (EOL)) and validation (Epsilon Validation Language (EVL)). The Multi-level modelling in F-Logic (MiF) approach uses F-Logic / Flora-2 to implement the formalism, querying and validation. The relationship across levels is used between models with different ontological structures and in different domains/spaces [19,21]. MiF approach could also support relationship across levels as well, however we did not come across a use case in the OGI Pilot so far. All DPF[21], DDI[19] and MiF support mediation of relationship, additionally MiF supports mediation of a composition relationship as well. Almost all approaches support the criterion of single and multi-potency semantics. F-OML [7] approach behaves as the same as MiF approach, except in mediation of relationship criterion.

5 Conclusion

In this paper we introduced an alternative implementation of multi-level modelling using F-Logic and Flora-2. We have applied the implementation on a subset of the OGI Pilot use case, which dealt with the specification of a pump using the ISO 15926 standard. The main benefit behind this proposal is the integrated approach of F-Logic which allows the specification of the semantics, ontological stack and reasoning capabilities (i.e. querying and validation) in a single framework, its object-oriented semantics and its wide acceptance in the ontology modelling community. In the future we plan to implement the semantics of another standard used in the OGI Pilot, the MIMOSA standard and use the reasoning capabilities of F-Logic to automate the matching and transformation between ISO 15926 and MIMOSA.

References

1. Juergen Angele, Michael Kifer, and Georg Lausen. Ontologies in F-Logic. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 45–70. Springer Berlin Heidelberg, 2009.
2. Colin Atkinson and Ralph Gerbig. Level-Agnostic Designation of Model Elements. In *Proc. of ECMFA 2014*, volume LNCS 8569, pages 18–34. Springer, 2014.
3. Colin Atkinson, Bastian Kennel, and Björn Goß. The level-agnostic modeling language. In *Software Language Engineering*, pages 266–275. Springer, 2011.

4. Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *UML 2001*, pages 19–33. Springer, 2001.
5. Colin Atkinson and Thomas Kühne. Rearchitecting the UML Infrastructure. *ACM TOMCATS*, 12(4):290–321, 2002.
6. Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software and System Modeling*, 7(3):345–359, 2008.
7. Mira Balaban and Michael Kifer. Logic-based model-level software development with F-OML. In *Model Driven Engineering Languages and Systems*, pages 517–532. Springer, 2011.
8. Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno Llorena. Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal*, 57(1):36–58, 2014.
9. Fiatech. Advancing Interoperability for the Capital Projects Industry: A Vision Paper. Technical report, Fiatech, February 2012.
10. M. P. Gallaher, A. C. O'Connor, Jr. Dettbarn, J. L., and L. T Gilday. Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry. Technical report, NIST, 2004.
11. Fahad R. Golra and Fabien Dagnat. The Lazy Initialization Multilayered Modeling Framework. In *Proc. of ICSE 2011*, pages 924–927. ACM, 2011.
12. ISO. ISO 15926: Industrial automation systems and integration Integration of life-cycle data for process plants including oil and gas production facilities. Technical report, ISO, 2004.
13. Andreas Jordan, Georg Grossmann, Wolfgang Mayer, Matt Selway, and Markus Stumptner. On the application of software modelling principles on ISO 15926. In *Proc. of the Modelling of the Physical World (MOTPW) Workshop at MODELS 2012*. ACM, 2012.
14. Michael Kifer. Rule Interchange Format: The Framework. In *Prof. of RR 2008*, LNCS 5341, pages 1–11. Springer, 2008.
15. Michael Lawley and Jim Steel. Practical Declarative Model Transformation with Tefkat. In *MoDELS Satellite Events*, LNCS 3844, pages 139–150. Springer, 2005.
16. Wolfgang Mayer, Markus Stumptner, Georg Grossmann, and Andreas Jordan. Semantic Interoperability in the Oil and Gas Industry: A ChallengingTestbed for Semantic Technologies. In *AAAI 2013 Fall Symposium on Semantics for Big Data*, 2013.
17. MIMOSA. Open systems architecture for enterprise application integration (osa-eai) 3.2.3. Technical report, MIMOSA, 2012.
18. John Mylopoulos, Alexander Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing Knowledge About Information Systems. *ACM TOIS*, 8(4):325–362, 1990.
19. Bernd Neumayr, Manfred A. Jeusfeld, Michael Schrefl, and Christoph Schätz. Dual Deep Instantiation and Its ConceptBase Implementation. In *Proc. of CAiSE 2014*, LNCS 8484, pages 503–517. Springer, 2014.
20. Bernd Neumayr, Michael Schrefl, and Bernhard Thalheim. Modeling techniques for multi-level abstraction. In *The Evolution of Conceptual Modeling*, pages 68–92. Springer, 2011.
21. Alessandro Rossini, Juan de Lara, Esther Guerra, Adrian Rutle, and Uwe Wolter. A formalisation of deep metamodelling. *Formal Aspects of Computing*, in press(in press):1–41, 2014.
22. Guizhen Yang, Michael Kifer, and Chang Zhao. Flora-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web. In *Proc. of OTM 2003.*

A Foundation for Multi-Level Modelling

Tony Clark[1], Cesar Gonzalez-Perez[2], Brian Henderson-Sellers[3]

[1] Middlesex University, London, UK. t.n.clark@mdx.ac.uk
[2] Institute of Heritage Sciences Santiago de Compostela, Spain
cesar.gonzalez-perez@incipit.csic.es
[3] University of Technology, Sydney, Australia
brian.henderson-sellers@uts.edu.au

Abstract. Multi-level modelling allows types and instances to be mixed in the same model, however there are several proposals for how meta-models can support this. This paper proposes a meta-circular basis for meta-modelling and shows how it supports two leading approaches to multi-level modelling.

1 Introduction

Contemporary and future engineering of information systems place an increasing emphasis on the use of models, either directly to aid design and implementation, in a more formal sense for code generation or as the backbone to model-driven engineering (MDE) [27]. Models must be described using a language that itself may be defined in many ways but typically using a meta-model *e.g.*, [26, 20]. That meta-model must itself be defined, by a meta-meta-model. Together with the instances conformant to the model, this leads to an identification of four abstraction levels of interest to the modeller and meta-modeller. Although in use for almost two decades, a four-layer architecture like that of the Object Management Group (OMG) raises some concerns both theoretically and pragmatically; a prime problem being the use of strict meta-modelling [5, 4] that constrains the instance-of relation to only be permitted between pairs of conterminous layers and never within a layer (see also [5]). This led several researchers (*e.g.*, [7, 6]) to seek a way of describing models and modelling languages without the use of this 'strict meta-modelling' hierarchy of the OMG.

A foundation for meta-modelling should be unifying and complete in the sense that it supports the development of both general-purpose and domain-specific languages and also integrates their representation so that tools can work across multiple languages. Leading approaches include: **strict meta-modelling**: The OMG strict meta-modelling architecture has been criticized, especially when applied to processes and methodologies (see summary in [18]) since the traditional strict meta-modelling approach is unable to support enactment *e.g.*, [2]; it defines attributes at level M2, thus giving them values at M1 by virtue of the prevailing type-instance semantics, when what is actually needed is values at M0. This enactment support is provided by the architecture used by ISO/IEC 24744 but at the expense of relying on power-type patterns, which do not accord with the philosophy of strict meta-modelling. **clabjects**: Potency is associated with the notion of deep instantiation, [8, 12, 13], and introduces the idea of an entity with both a class facet and an object fact, entity given the name *clabject* [5]. **OCA**: Two different kinds of meta-model structures have been identified: ontological meta-modelling in contrast to the linguistic meta-modelling utilized in a strict meta-modelling architecture. This was later called the Orthogonal Classification Architecture (OCA) [9]. In [22] we describe these ideas and relate them to some more recent concerns raised by the application of language use theory to this approach. More recently, Atkinson and colleagues have extended the OCA in their description of the Pan Level Model (PLM) and the Level-agnostic Modeling Language (LML) [7]. **powertypes**: The need to provide access to, and control over, the meta-types of elements in a model when designing languages led to proposals for *powertypes* [17, 23]. This is a methodological approach that uses standard classes both conventionally and as meta-classes by disciplined use of instance-of associations. The approach allows the modeller to control attribute definitions at M2 that affect the properties in model elements at M1.

Our claim is that none of the approaches above are complete as a basis for meta-modelling. In particular, such a basis must achieve the following features: **meta-circularity**: Self description is key to achieving virtually all of the desirable features for language engineering. Just as it is possible to embed a λ-calculus interpreter in itself and thereby characterize an infinite tower of operational languages, we seek to construct a self describing basis for an infinite tower of modelling languages. **uniformity**: Any basis for meta-modelling that is self-describing implies a precisely defined relation between representations for type and instance. A system that achieves the *conflation* of these representations, *i.e.*, uses the identity relationship, is *minimal* in the family of such relationships. Furthermore, a uniform representation is essential if we are not to encounter limitations on the type of languages that can be defined, for example where we need to mix instances and types. Therefore, we seek to provide a single representation for types and instances at any level. **extensibility**: We assume that any family of modelling languages will use type-based extension (sub-classes, inheritance, *etc.*), and that new languages are based on extending existing languages. Meta-circularity and extensibility implies that languages can be extended at both type and meta-type levels and therefore the question arises as to whether there is a limit to the levels over which extension can be applied. We seek a basis that places no restriction on the number of levels of both extensibility and instantiation. **views**: Languages should support multiple modes of interaction that are defined at the meta-level. Although we will use multiple language views, we will not consider this aspect further.

Our approach (subsuming those above) is to use simple *objects* together with two simple relations: **type** A relation that exists between every object and its class and can be applied an arbitrary number of times to define the meta-classifications of instance, class and meta-class; **extension** A relation that exists between classes that provides a minimal basis for incremental addition of features. The approach is based on existing proposals for meta-classes provided by languages such as Smalltalk [16] and ObjVLisp [10]. Although Smalltalk was the first language to introduce meta-classes (and thereby three-levels of meta-class, class and instance), each meta-class is restricted to having a single instance which severely limits its use as the basis for language engineering where meta-properties are reused across multiple languages.

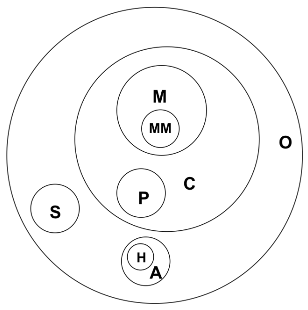


Fig. 1. Object Classifications As Sub-Sets of Object

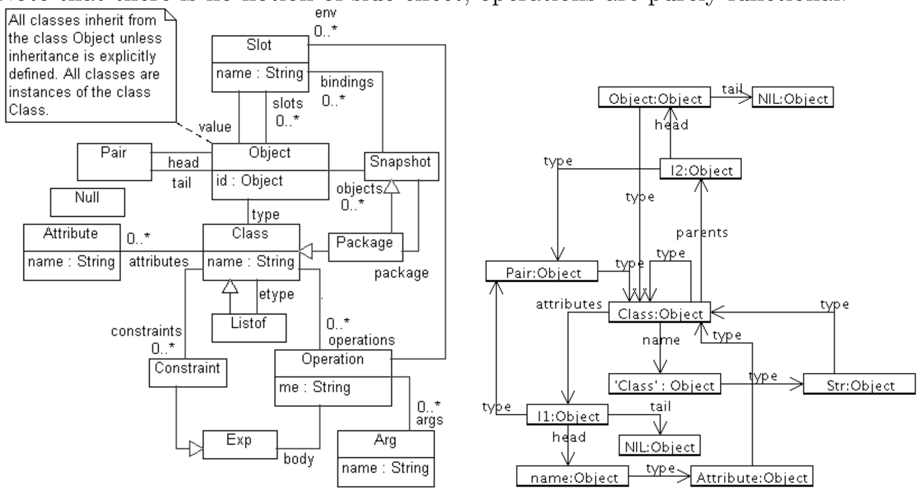
The approach to object classification and the instance-of relation is shown in Fig. 1 where circles represent sub-sets of the set **O** of objects. Consider the set **A** that denotes a set of objects representing animals. In order for an element of **A** to be well-formed, it must have an instance-of link to an object in the set **C** of all classes. Note that elements of **C** are objects (everything is an object), but they are objects that satisfy some criteria for class-hood. Since the element of **C** that represents the class Animal is itself an object, it must have an instance-of link to an object that represents its class. Such an object is a meta-class and is a member of the set of objects **M** (perhaps the class called Class). A meta-class is just an object that satisfies the constraint for membership of **M**. This means that it must have an instance-of link to a meta-meta-class in **MM**. It should be stressed at this point, that there is no limit to the instance-of regress. In addition to objects that satisfy Animal-hood. There are objects that are used to group objects: snapshots that are members of the set **S**. Snapshots contain objects that are all instances of related classes: packages that are members of the set **P**. Finally, classes can be related by extension so that there are two classes Animal and Herbivore in **C** that designate the rules for membership of the sets **A** and **H**. Of course, since every element of **M** is also in **C**, the extension relation can be defined between meta-classes that will designate different sub-sets of **C**.

Our basis for meta-modelling is defined as a self-describing object-oriented kernel. The Kernel is essentially a logic. However, unlike a traditional logic that consists of boolean valued formulas whose sub-expressions denote values drawn from a collection of predefined types, the Kernel can only denote objects. Some objects are designated *classes* because they conform to a particular object-interface that includes boolean valued expressions (or *constraints*) that characterize objects designated as well-formed instances of the class. Such a self-describing logic might lead to doubts related to Russell's Paradox, although the use of types and identities as described below, together with an implementation of the approach that supports a collection of real-world applications (including itself), gives us confidence that this is not a problem. Our claim is that this approach is novel and that it subsumes existing approaches to meta-modelling. Our contribution is the definition of a meta-circular foundation for model-based language engineering in the form of a kernel language that is validated in terms of an implementation as a toolkit that has been used for a variety of real-world applications. In addition we show that other approaches to multi-level modelling can be represented in the Kernel.

2 A Meta-Modelling Kernel

Our proposal is to set up a system whereby *everything is an object* [21] and where a simple set of rules governs the ability to construct configurations of objects that constitute self-describing languages. The system consists of an object-representation and then *sugarings* that are convenient language structures defined to *de-sugar* into the basic representation.

Figure 2(a) shows the proposed kernel language as a diagram. Fundamentally, everything is an object and a partial view of the Kernel as a collection of objects and slots is shown in figure 2(b). An object has a unique id, some slots, and a type. The type of an object is a class. Classes are organised into packages whose instances are snapshots that are assemblies of objects. Since classes are just objects that conform to some structural conditions, packages can be similarly viewed as snapshots with appropriate conditions. Collections of objects are organised as sequences in terms of pairs and `Null`. Since types are always implemented as classes, there is a special class called `Listof` whose instances are lists. There is no need to special types of atomic value such as integers and booleans because we can designate special objects via their identities as being members of these data types. Expressions are objects that can be asked to evaluate themselves in a supplied context. Constraints are special types of expressions that always return boolean values. Constraints are important because they are used in classes to *classify* objects that are considered to be *instances*. Classes have operations, that are objects used to handle messages sent to instances of the class. Note that there is no notion of side-effect, operations are purely functional.



(a) `Kernel` as a Class Diagram (b) `Kernel` as an Object Diagram

Fig. 2. Two Views of the Kernel

Fig. 3 shows the complete textual definition of the Kernel. It uses a number of external definitions and notational conventions that are outlined as follows: classes define a predicate ? that is used to determine instance-hood; operations use λ-notation where arguments are patterns; objects are `(C,i)[s ↦v]` where `C` is the class of the object, `i` is the id, `s` is a slot name and `v` is the corresponding value; `intern` maps a class and slots to an object; lists are `[v1,...,vn]` and can be appended using `+`; `::` is used to dereference names in a name-space; ⇑ is an inheritance relationship between classes.

Since the Kernel is essentially a logic we need something equivalent to OCL. We use the following shorthand where `l` is a list: `l.∀(p)` is `true` when the predicate `p` returns `true` for each element in the list `l`; `l.∃(p)` is `true` when the predicate `p` returns `true` for any element in the list `l`; `l.∋(x)` is true when the element `x` is contained in the list `l`; `l.⇐(p,a,y)` is the result of applying operation `a` to the first element `x` of `l` for which `p(x)` is `true` and `y` if no such element exists; `l.flatten()` expects `l` to be a list of lists and returns a list formed by appending all elements of `l` in order. # maps a list to its length. It is convenient to be able to construct and manipulate lists using *comprehension* expressions. For example, if `l` is the list `[2,3,4]` then `[x*2 | x ←l]` is the list `[4,6,8]`. Predicates may be used to filter lists as in `[x | x←l,?even(x)] = [2,4]`.

In order for this to be meta-circular, we require that and `Kernel.?(Kernel)` holds. This is difficult to establish without tooling since all the objects in the definition must be checked against their classes, and, since the classes themselves are part of the package, this requires the classes to be self-describing. The Kernel has been implemented as part of the XModeler toolkit and has been used to implement the rest of the tools including diagram tools, model browsers, model editors, model transformers and libraries. The XModeler Kernel contains many more classes than the language described in this article, but the essential features are the same. XModeler can be instructed to apply the Kernel-defined constraints to itself (over 100 classes) and to produce a report that shows that it is self-consistent.

3 Validation

Section 1 describes a list of features that we claim to be characteristic for any language that is used as a basis for meta-modelling. We have introduced such a language and used it to build a model of itself. This section analyses the Kernel language with respect to the characteristic features: **type**: In `Kernel` everything is an object and all objects have an intrinsic type property. **meta-circularity**: This property is essential for multi-level modelling and in order to be able develop tools (such as serializers) that are language-level agnostic [25]. The XModeler tool can be shown to establish that `Kernel.?(Kernel)`. **uniformity**: We have used a single representation (with a small number of externally defined conventions and rules) for all data in `Kernel`. **extensibility**: Extension is supported through class relationships that are then used by constraints in order to place conditions

```
class Object {
 id    : Object;
 type  : Class;
 slots : [Slot]
 constraints { type.?(self) }
 operations {
  dot(n) = slots.⇐(
   λ(n' ↦ _)n=n',λ(_ ↦ v) v,error)
  send(n,args) =
   type.ops().⇐(
    λ(n' ↦ (Operation)[args ↦ args'])
     n=n' and #args = #args',
    λ(_ ↦ f) f.invoke(self,args),
    error)
 }
}
class Slot {name:Str;value:Object}
class Operation {
 me   : Str;
 env  : [Slot];
 args : [Arg];
 body : Exp
 operations {
  invoke(target,values) =
   body.eval(env+['self' ↦ target] +
    [me ↦ self] + target.slots +
    target.type.ops()  +
    [a ↦ v | (a,v) ← args * values])
 }
}
class Listof extends Class {
 etype : Class;
 operations {
  ?(o) = list?(o) and
         o.∀(λ(x)etype.?(x))
 }
}
class Snapshot extends Object {
  package  : Package;
  objects  : [Object];
  bindings : [Slot]
  constraints {
   package.?(self);
   bindings.∀(λ(b)objects.∋(b.value))
  }
  operations {
   ::(k,d) = bindings.⇐(
    λ(s)s.name=k,λ(s)s.value,d)
  }
}
```

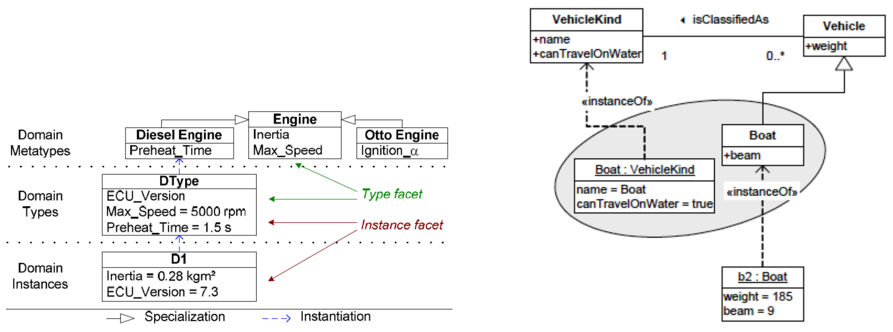
```
class Class {
 name         : Str;
 supers       : [Class];
 attributes   : [Attribute];
 operations   : [Binding];
 constraints  : [Constraint]
 operations {
  supers() = [self] +
   [c | p ← supers;
        c ← p.supers()].remDups()
  ⇑(c) = supers().∋(c)
  atts() =
   [a | c ← supers(),a ← c.attributes]
  ops()  =
   [b | c ← supers(),b ← c.operations]
  cond() =
    [a | c ← supers(),a ← c.constraints]
  ::(n,d) =
    atts().⇐(λ(n' ↦ a)n'=n,λ(n ↦ a)a,
      ops().⇐(λ(n' ↦ o)n'=n,λ(n ↦ o)o,d))
  ?(o) = o.type.⇑(self) and
   atts().∀(λ(a)o.slots.∃(λ(s)
    s.name = a.name and
    a.type.?(s.value))) and
    cond().∀(λ(c) c.eval([self ↦ o] +
     [s.name ↦ s.value | s ← o.slots]))
 }
}
class Package extends Snapshot,Class {
 constraints {
  objects.∀(Class?);
   attributes.∀(λ(a) objects.∋(a.atype));
   parents.∀(λ(p) p.type.⇑(Package))
 }
 operations {
  ::(n,d) = obj().⇐(λ(o)o.n=n,λ(o)o,d)
  obj() = objects +
   [p.objects | p ← parents].flatten()
  ⇑(p) = objects.∀(λ(c)obj().∃(λ(c')c.⇑(c')))
  ?(o) = o.type.⇑(Snapshot) and
   o.package.⇑(self) and
   o.objects.∀(λ(o)
    objects.∃(λ(c) c.?(o))) and
    Class::?(intern(self,o.slots))
 }
}
class Pair {head:Object;tail:Object}
class Nulll {}
class Constraint extends Exp {}
class Arg { name:Str }
```

Fig. 3. Definition of `Kernel`

on objects that are instances of a sub-class. The definitions are `Class::?` and `Package::?` in Fig. 3.

Our claim is that the Kernel is a suitable basis for multi-level modelling. In order to validate this claim we present the definition of two different languages, each based on independent approaches, both defined in the Kernel. Models written in the languages are shown in Fig. 4.

The model in figure 4(a) shows the use of *type facets* that allow classes to have properties. These can be implemented by including a *potency* as part of an attribute definition. The potency is an integer value indicating the number of type-levels (3 are shown in the model) spanned by the relationship between



(a) A Use of Potency (in [3])

(b) A use of Power-Types (in [23])

Fig. 4. Two Approaches to Multi-Level Modelling

an attribute and its corresponding slots. The model defines a language (Domain Metatypes) of engines. The class `Engine` defines a type facet called `max_speed` that results in a slot at the domain type (model) level, and an instance facet called `inertia` that becomes a slot at a remove of two type-levels.

The model in figure 4(b) shows the power-type pattern where a class (in this case `Vehicle`) is classified by another class (`VehicleKind`). Instances of `VehicleKind` are used to partition subclasses of `Vehicle` as shown in the ellipse, forming a *clabject*. The result is that an object is contributing to the type-level information in a class that will eventually affect instances of the class.

Each language definition takes the form of a package that is both an instance and an extension of `Kernel`. By the definition of `Package::?`, an instance of a package `P` should be a snapshot whose contents are all instances of classes in `P`. By the definition of `Package::extends?`, a package `P` extends a package `Q` when every class in `P` extends some class in (or inherited by) `Q`. Therefore, by extending and instantiating `Kernel` a package is a well-formed language definition in its own right, that can, by the definition of extension, modify the basic definition of `Class::?`. Such a modification might place extra conditions on instance-hood, or even relax existing conditions.

Fig. 5 contains the definition for the language and models shown in figure 4(a). The class `CAtt` extends `Attribute` with an attribute for potency-level. The class `CClass` modifies `atts` so that it gathers together all attributes that apply to this level. This is achieved using a counter that is incremented when the type-level is traversed. A concrete-syntax for potency-level in attributes is used in the definition of the package `DomainMetaTypes`, and slots are permitted in class definitions due to potency-levels becoming `0` in `DomainTypes`. The snapshot `DomainInstances` contains a single object whose slots correspond to attributes from different type-levels as defined by their respective potency-levels.

Fig. 6 contains the definition for the language and models shown in figure 4(b). The meta-class `PowClass` defines an attribute `classifier` and the constraint on `PartClass` requires that all its descriptor objects are instances of the classifier inherited by a parent power-class. The package `Vehicles` contains a single power-class `Vehicle` that is classified by `VehicleKind` and a partitioned-class `Boat`

```
package CKernel:Kernel extends Kernel {
 class CAtt extends Attribute {
  level:Integer;
 }
 class CClass extends Class {
  operations {
   atts() = catts(1,self)
   catts(n,c=(_,c)[]) = []
   catts(n,c) =
     [a | a ← c.atts(),
          ?a.type=CAtt,a.level=n] +
     catts(n+1,c.type)
  }
  constraints {atts.∀(λ(a)a.type=CAtt)}
 }
}
snapshot DomainInstances:DomainTypes {
 (DType)[inertia↦0.28;ECU_version↦7.3]
}
```

```
package DomainMetaTypes:CKernel  {
 class Engine:CClass extends CClass {
  inertia[2]:Float;
  max_speed[1]:Integer
 }
 class DieselEngine:CClass extends Engine {
  preheat_time[1]:Float
 }
 class OttoEngine:CClass extends Engine {
  ignition_alpha[1]:Float
 }
}
package DomainTypes:DomainMetaTypes {
 class DType:DieselEngine {
  ECU_version[1]:Float;
  max_speed=5000;
  preheat_time=1.5
 }
}
```

Fig. 5. Definition and use of `CKernel`

that includes an instance of `VehicleKind` as its descriptor. The snapshot `ABoat` is governed by the classes defined in the package `Vehicles` which in turn are governed by the language `PKernel` therefore, `ABoat` is constrained by the clabject `Boat` and `Boat.descr`.

```
package PKernel:Kernel extends Kernel {
  class PowClass extends Class {
   classifier:Class
  }
  class PartClass extends Class {
   descr:[Object]
   constraints {
    supers().∀(λ(c) PowClass?(c));
    descr.∀(λ(o)
     supers().∃(λ(c)
      c.classifier.?(o)))
   }
  }
}
snapshot ABoat:Vehicles {
 (Boat)[beam ↦ 9;weight ↦ 185]
}
```

```
package Vehicles:PKernel {
 class Vehicle:PowClass {
  classifier=VehicleKind
  weight:Int
 }
 class VehicleKind {
  name:Str;
  canTravelOnWater:Bool
 }
 class Boat:PartClass extends Vehicle {
  descr=[(VehicleKind)[
           name↦'Boat';
           canTravelOnWater ↦ true]]
  beam:Int
 }
}
```

Fig. 6. Definition and use of `PKernel`

The examples described above contribute evidence that `Kernel` can define different languages and is not restricted to a fixed number of type-levels, and that objects, classes and meta-classes can be mixed. This is possible because of the uniformity of representation, the unrestricted access to type-level information and meta-circularity. Although outside the scope of this paper, the formulation of `Kernel` makes it possible to write level-agnostic tools, such as those for model-management, that can be used on any type-level.

4 Conclusion

Our aim is to produce a meta-circular level-agnostic basis for model-based language engineering. We have reviewed the current proposals for such a basis and argued that they are not optimal by providing a new language definition that is self-describing and can be used to embed the competing approaches. The Kernel language is simple and can be implemented as demonstrated by the XMF and XModeler toolkit [11] that is capable of both describing and reasoning about itself. The toolkit was reported as a leading technology for Software Engineering [19] and has been used for a variety of applications including modelling languages for aerospace applications, telecoms applications [1], and is currently being used to implement aspects of the MEMO enterprise modelling language [24, 14].

In [15], the authors show how the OMG levels M0-M3 can be represented on a single object-diagram. This allows OCL constraints to range over all levels and thereby support clabjects and potency. This is consistent with our approach, although OCL is just one of the languages that could be used with our approach (as a view of models and constraints) and the authors of [15] do not claim to be a foundation for model-based language engineering.

Our intention is that the Kernel language defined in this article provides a basis for ourselves and others to experiment with language definitions. Because all such kernel-defined languages are based on a single object representation, it is feasible to build a collection of tools that work against well defined sub-sets of objects (as shown in figure 1) and thereby incrementally develop a shared library.

References

1. Achilleas Achilleos, Nektarios Georgalas, and Kun Yang. An open source domain-specific tools framework to support model driven development of oss. In *Model Driven Architecture-Foundations and Applications*, pages 1–16. Springer, 2007.
2. Anat Aharoni and Iris Reinhartz-Berger. A domain engineering approach for situational method engineering. In *Conceptual Modeling-ER 2008*, pages 455–468. Springer, 2008.
3. Thomas Aschauer, Gerd Dauenhauer, and Wolfgang Pree. Representation and traversal of large clabject models. In *Model Driven Engineering Languages and Systems*, pages 17–31. Springer, 2009.
4. Colin Atkinson. Meta-modelling for distributed object environments. In *Enterprise Distributed Object Computing Workshop [1997]. EDOC'97. Proceedings. First International*, pages 90–101. IEEE, 1997.
5. Colin Atkinson. Supporting and applying the UML conceptual framework. In *The Unified Modeling Language. UML 98: Beyond the Notation*, pages 21–36. Springer, 1999.
6. Colin Atkinson, Bastian Kennel, and Björn Goß. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *Procs. 7th Int. Workshop on Semantic Web Enabled Software Engineering, Bonn (October 24, 2011).*
7. Colin Atkinson, Bastian Kennel, and Björn Goß. The level-agnostic modeling language. In *Software Language Engineering*, pages 266–275. Springer, 2011.
8. Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 19–33. Springer, 2001.

9. Colin Atkinson and Thomas Kühne. Concepts for comparing modeling tool architectures. In *Model Driven Engineering Languages and Systems*, pages 398–413. Springer, 2005.
10. Jean-Pierre Briot and Pierre Cointe. The objvlisp model: Definition of a uniform, reflexive and extensible object oriented language. In *ECAI*, pages 225–232, 1986.
11. Tony Clark and James Willans. Software language engineering with xmf and xmodeler. *Formal and Practical Aspects of Domain Specific Languages: Recent Developments. IGI Global, USA*, 2012.
12. Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.
13. Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. Extending deep meta-modelling for practical model-driven engineering. *The Computer Journal*, page bxs144, 2012.
14. Ulrich Frank. Multi-perspective enterprise modeling: foundational concepts, prospects and future research challenges. *Software and System Modeling*, 13(3):941–962, 2014.
15. Martin Gogolla, Jean-Marie Favre, and Fabian Büttner. On squeezing m0, m1, m2, and m3 into a single object diagram. *Proceedings Tool-Support for OCL and Related Formalisms-Needs and Trends*, 2005.
16. Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.
17. Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software & Systems Modeling*, 5(1):72–90, 2006.
18. Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for software engineering.* Wiley Publishing, 2008.
19. Simon Helsen, Arthur Ryman, and Diomidis Spinellis. Where's my jetpack? *Software, IEEE*, 25(5):18–21, 2008.
20. Brian Henderson-Sellers. *On the mathematics of modelling, metamodelling, ontologies and modelling languages.* Springer, 2012.
21. Brian Henderson-Sellers, Tony Clark, and Cesar Gonzalez-Perez. On the search for a level-agnostic modelling language. In Camille Salinesi, Moira C. Norrie, and Oscar Pastor, editors, *CAiSE*, volume 7908 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2013.
22. Brian Henderson-Sellers, Owen Eriksson, Cesar Gonzalez-Perez, and Pär J Ågerfalk. Ptolemaic metamodelling? the need for a paradigm shift. *Cueva Lovelle JM, Pelayo García-Bustelo C, Sanjuán Martínez O (eds) Progressions and innovations in model-driven software engineering. IGI Global, Hershey, PA*, pages 90–146, 2013.
23. Brian Henderson-Sellers and Cesar Gonzalez-Perez. Connecting powertypes and stereotypes. *Journal of Object Technology*, 4(7):83–96, 2005.
24. Thomas Johanndeiter, Anat Goldstein, and Ulrich Frank. Towards business process models at runtime. In Nelly Bencomo, Robert B. France, Sebastian Götz, and Bernhard Rumpe, editors, *MoDELS@Run.time*, volume 1079 of *CEUR Workshop Proceedings*, pages 13–25. CEUR-WS.org, 2013.
25. Fabio Kon, Fabio Costa, Gordon Blair, and Roy H Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.
26. Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.
27. Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra. Bottom-up metamodelling: An interactive approach. In *Model Driven Engineering Languages and Systems*, pages 3–19. Springer, 2012.

Comparing Multi-Level Modeling Approaches

Colin Atkinson[1], Ralph Gerbig[1] and Thomas Kühne[2]

[1] University of Mannheim,
B6, C2.11, Mannheim, Germany
(atkinson, gerbig)@informatik.uni-mannheim.de
[2] Victoria University of Wellington,
P. O. Box 600, Wellington 6140, New Zealand
Thomas.Kuehne@ecs.victoria.ac.nz

Abstract. As the range of modelling approaches that claim to be "multi-level" diversifies, there is growing debate in the literature about what multi-level modelling actually is and what form supporting languages and infrastructures should take. However, there is no consensus yet on how this debate should be framed and what objective criteria should be used to evaluate different approaches. It is clear from the literature that proponents of different approaches base their arguments on fundamentally different assumptions about what multi-level modelling is and what benefits it should aim to provide. In this position paper we identify some of the core issues that currently hinder progress towards the required consensus and identify some of the terminological differences that have amplified confusion. Referencing various work that represents diverging viewpoints, our goal is to initiate a meta-discussion on what the open questions in multi-level modelling are, how respective proposals to answer them could be evaluated, and which kinds of discussions are expedient in this context.

Keywords: multi-level modelling, deep modelling, metamodelling, level-agnostic

1 Introduction

Multi-level modelling is gaining resonance. Many groups have applied the approach or created variants, and a community focusing on multi-level modelling is emerging. However, at the present time, there is little consensus in the literature on fundamental multi-level modelling concepts, and if anything proponents of multi-level modelling appear to be diverging rather converging in their understanding of the approach. In particular, in a series of recent papers [9,12,13] some authors have presented a long list of fundamental criticisms of one of the first proposed multi-level modelling approaches [6], based on the notions of the orthogonal classification architecture and deep instantiation. Many other authors have also identified issues in this and other approaches to multi-level modelling and have suggested their own solutions (e.g., [10,18]).

An analysis of these papers reveals that the criticism ranges from challenges to the fundamental validity of the approach from a set-theoretic point of view to objections to the use of particular terminology and disagreements about the basic goals and motivations underlying multi-level modelling.

We believe that some of the disagreements can be resolved by adopting a different style of debate. Hence, we argue that the main challenge facing the fledgling multi-level modelling community at the present time is to clarify the boundary conditions and assumptions within which the debate about multi-level modelling should take place. Without consensus on these meta-issues, the chances of forging a common understanding about fundamental concepts in multi-level modelling will be small.

In order to foster greater clarity in the debate between multi-level modelling approaches, in this paper we attempt to identify some of the main open questions in multi-level modelling and suggest potential ways of converging towards broadly accepted answers. In particular, we aim to characterize what kinds of discussions are most likely to increase convergence and suggest certain principles to be used in evaluating different proposals.

The rest of the paper is organised as follows. To help set the context for the discussions, the next section briefly outlines some core features we believe could be used to characterized multi-level and deep modeling approaches. Section 3 continues by describing five key issues with the potential to cause controversy in the context of multi-level modeling. Section 4 then provides a deeper consideration of one of the most subtle and sensitive of these issues – "terminology" – with a specific focus on the term "level-agnostic" which is the subject of some debate at the present time. Finally, section 5 concludes with some final observations and closing remarks.

2 Multi-Level Modelling

Perhaps the most fundamental and important question is what qualities an approach needs to posses in order to be characterized as multi-level. Without a consensus on how to recognize multi-level modeling approaches, it will be impossible to conduct a meaningful debate about their relative strengths and weaknesses. We suggest that minimal requirements for a multi-level modelling approach include:

– some fundamental notion of abstracting a multitude of model elements to a common classifier.
– the ability for model elements to form anti-transitive instantiation chains.
– a concept of level, formed by elements belonging to the same classification level, the latter being defined by both classification depth and level membership of the root of the instantiation chain.

The above minimal requirements are very inclusive and admit, for instance, traditional linguistic language definition stacks. We prefer to take a narrower view that furthermore:

– mirrors classification relationships in the domain with explicit relationships that are subject to well-formedness constraints [15].
– recognises type and instance facets of model elements and views them as inseparable [2].
– provides a mechanism for deep characterisation [5].

The approach containing the combination of all the above characteristics is often referred to as "deep (meta-)modelling". This is the particular flavour of multi-level modelling that has been the subject of the most debate in the recent literature [9,12,13] and we used some of this criticism to identify a number of potentially controversial issues that we discuss in the following section.

3 Potentially Controversial Issues

In order to move the field of multi-level modelling forward, it is of course desirable to have many alternative proposals and a healthy debate about their respective merits. There are a few pitfalls, however, that should be avoided in order to achieve progress as constructively as possible.

For example, we believe that subscribing to a particular "*school of thought*" and exclusively evaluating differing proposals from one subjective perspective can be problematic. Some examples of controversies that can emerge when issues have been evaluated with this mindset are discussed below.

3.1 Language Size

Always expecting a user to first define or choose their language [19] versus presenting a user with a rich library of modelling concepts to be adapted and used [6] represent the two different ends of a language engineering spectrum. No single point within this spectrum will be optimal for all types of users but we believe there is an interesting discussion to be conducted about the level of support tools aimed at the majority of modellers should provide with respect to language engineering.

Which modellers can be expected to be good language engineers and which library paradigms may turn out to be too narrow in the assumptions they make? A complete tool should recognise both language engineering and domain modelling as relevant tasks but there is certainly room for specialised tools that focus on one of these areas only. The potential pitfall to avoid is to assume that all user modelling is language engineering or that all language engineering can be subsumed under domain modelling.

3.2 Semantics

The clabject-based deep instantiation approach of Atkinson and Kühne has been criticised for lacking alignment with set theory and requiring the instantiation of elements that are not available for further instantiation [9]. While we do not dispute that there are formalisations and school of thoughts in which deep instantiation can be seen as "wrong", there is indeed a sound set-theoretic formalisation of deep instantiation [17] and in this framework – based on sets of sets – it is perfectly possible and natural to view elements as instances and types at the same time, with corresponding linear instantiation chains arising from this property.

We believe that instead of evaluating approaches according to whether or not they have compatible foundations, it is more helpful to examine whether approaches are internally consistent. If an approach is inconsistent to the effect of exhibiting contradictions or allowing unwanted paradoxes to occur then it could be viewed as “wrong”. Non-conformance to a particular semantic foundation, however, should not be held against an approach per se. Thus, while conformance to established disciplines can be a potential advantage, it cannot be the ultimate criterion for determining the adequacy of a proposal.

It would of course be desirable to obtain a common sound formal foundation for all multi-level modelling approaches in order to move the technology forward, but until consensus on such a formalism has been reached it is unhelpful to assume one particular approach as being a benchmark to evaluate other approaches against. In fact, judging an approach using an inappropriate perspective and formalism can lead to claims of unsoundness and inconsistencies which are not in fact valid [7].

3.3 Intended Target Audience

Whether consciously or unconsciously particular approaches target different user groups. For instance, while Henderson-Sellers et al. appear to focus on making it easy to build tools [12], the original multi-level approach by Atkinson and Kühne [4] focused on reducing accidental complexity for the domain modeller. On the other hand, Vangheluwe et al.’s AtomPM tool appears to be geared towards language engineering [19], etc.

A considerable amount of debate can be avoided if one takes the intended target audience of a particular approach into consideration. For example, the apparent difference in attributing significance to domain-motivated (ontological) instantiation relationships between Henderson-Sellers et al. and Atkinson et al. can be understood as reflecting the different target audiences. When focusing on the internals of a tool, i.e., addressing a tool builder audience, it is natural to regard user-defined relationships and associated modelling patterns as secondary [12]. In contrast, work on the orthogonal classification architecture [4] or in particular the “Unified Modelling Library” [6], targets the modeller and aims to provide a richer environment in order to reduce accidental complexity.

As an analogy, it is simpler to write a compiler for an assembly language compared to a software engineering language like Eiffel [20], however it can be argued that the investment necessary to develop the far more complex Eiffel compiler will pay dividends in subsequent safe language usage. We believe that “*everything is an object*” [12] is mainly a tool builder’s argument and is less helpful when aiming to support modellers.

3.4 Level of Modelling Discipline

Related to the above discussion regarding intended target audiences, there appear to be diverging views on what the intended target audience is. Some work appears to take a liberal approach, allowing concise and powerful solutions but also giving users rope to hang themselves [12], while other work attempts to enforce a discipline that is aimed at helping the user avoid inconsistent models [3].

We claim that is not ideal to point out perceived problems of disciplined approaches while not acknowledging the dangers that are implied by looser approaches. Strict metamodelling [3], for instance, has been criticised as being too restrictive and causing problems [10,12].

First, we believe that a lot of the criticism towards strict metamodelling is based on misunderstandings, e.g., the lack of awareness that new elements can always be introduced linguistically (i.e., without requiring an ontological type), that potency declarations are to be regarded as constraints that make certain guarantees but are not an enabler for deep instantiation, and that potency constraints do not cross instantiation dimensions, e.g., from the linguistic dimension to the ontological dimension, etc.

Second, we suggest that the debate is comparable to the debate about the value of static typing in programming languages. In the same way that some research in programming languages strives to make types and type checking available when needed [8] based on ideas of optional type inference, it seems promising to adopt a similar approach to modelling. In particular, since there is no extra effort involved in adhering to strict metamodelling (since levels can be inferred [15]) we suggest to support optional sanity checking on the basis of strict metamodelling principles.

3.5 Terminology

Terminology, i.e., the choice of words and their meaning, is always a potential source of confusion and unnecessary (or undiscovered) disagreement. As an example, the fact that the name of a particular, domain-oriented, instantiation type includes “ontological” should not be construed to mean that respective models in so-called ontological levels need to exhibit “*all the trappings of an ontological (meta)model such as the UFO*” [12].

The particular naming choice involving “ontological” was made because the original meaning of “ontology” relates to the things that exist (in the sense of a universe of discourse), in contrast to (linguistically classified) notation elements. There was no intention to invoke any of the features associated with contemporary ontology research.

We believe terminology can have various degrees of intuitiveness but effectively only the explicitly given definition of concepts and mechanisms should be decisive when it comes to criticising an approach.

A good example of an increasingly used term that can be interpreted in diametrically opposed ways is “level-agnostic”. In the following, we clarify these different interpretations.

4 Level-Agnostic Languages

The adjective “level-agnostic” implies that an approach does not make the treatment of an element dependent on its level in the ontological classification hierarchy. In this sense, the UML is definitely not a level-agnostic language since it uses a different notation for objects compared to classes, even though arguably all objects and classes are just modelling elements with the only difference being that classes have a type facet whereas object do not.

The idea of level-agnosticism for a modelling language as discussed in this context can be traced back to the proposal of using a uniform notation for UML elements, independently of which level they belong to, e.g., whether they are objects or classes [2]. This idea has later been expanded to include further notational elements [1].

A language can achieve level-agnosticism in one of two ways. One way is to essentially only recognise a single level [12] which internally can support arbitrary constellations of ontological classification relationships. We refer to such languages as "level-blind". The other way to support level-agnosticism is to use levels as a structuring and soundness-enforcing mechanism, without letting level membership impact on such things as element representation and rendering. We refer to such languages as "level-adjuvant".

4.1 Level-Blind Languages

Since the term "level-agnostic" implies the existence of levels, a "level-agnostic language" has to acknowledge them in some sense, even the approach put forward in [12]. Otherwise the described "search for a level-agnostic language" should have been characterised as a search for a "*level-less*" language. The characteristic feature of a level-blind language is that it essentially ignores the fact that there are levels.

An example for level boundaries that are considered insignificant in the level-blind approach of [12] are the ontological classification level boundaries implied by elements in a universe of discourse, such as "Lassie", "Collie", and "Breed" (c.f. section 3.3). In contrast, Atkinson and Kühne give the respective ontological classification hierarchy the same significance as the time-honoured linguistic classification hierarchy formed by language metamodelling [4].

The advantage of an approach that is blind to the ontological level boundaries is that it allows all elements to be treated the same, since membership to ontological levels is abstracted away from. The disadvantage, however, is that element membership to ontological levels can no longer be exploited to uncover unsound scenarios. Paradoxical situations like being one's own baby [14, p. 247] are hallmarks of single level, flat domain approaches where unification has been taken to the extreme. If, for instance, everything is a set, as in naïve set theory, then even the set of all sets that do not contain themselves is a set. Yet this construction establishes Russel's famous paradox, as it is not possible to find a single consistent answer to the question whether the said set is a member of itself.

An analogous construction is obviously possible with the "*everything is an object*"-approach put forward in [12]. Consider an object O whose instances are all those objects that are not instances of themselves. This leads to the paradoxical situation that O must be an instance of itself when it is not an instance of itself and must not be an instance of itself when it is an instance of itself. Such paradoxes are impossible in a language that uses levels to avoid self-reference. Stratification was one of the proposals to fix naïve set theory and works just as well when used in the form of ontological levels forming a domain-classification hierarchy.

4.2 Level-Adjuvant Languages

A level-adjuvant language is a level-agnostic language that recognises the utility of levels, without implying accidental differences in treatment. For instance, the orthogonal classification architecture [4] uses ontological level boundaries to avoid paradoxical modelling scenarios but does not require different treatments with respect to representation or rendering (c.f. [5, Fig. 16, p. 307]). The orthogonal classification architecture hence establishes level-agnosticism with respect to representation and notation but is not level-blind with respect to enforcing soundness for models of the universe of discourse.

The difference between using level-blind versus level-adjuvant languages is like the difference between untyped programming languages versus strongly typed programming languages. While the "anything goes" approach of untyped languages supports small but powerful solutions, the discipline attained by typing rules pays big dividends when unsound scenarios are rejected straight away as opposed to being discovered through testing, or worse, not uncovered at all (c.f., section 3.4).

For instance, the "everything (even a class) is an object" approach [12, Fig. 10] that was first used as the backbone of the Smalltalk metaclass hierarchy [11], allows an object o to be its own class or to be an instance of another object c and a subclass of c at the same time. Unless such scenarios are excluded by suitable well-formedness rules, it is possible to construct paradoxical situations and "sillygisms" [16]. The discipline afforded by requiring an ontological classification relationship to be acyclic and level-respecting [15] is exactly designed to exclude such problems.

It is ironic that the proponents of a single-level, flat domain approach [12] that takes unification to the extreme, criticise level-aware approaches for enabling paradoxes even though level-awareness is the key to avoiding a whole class of paradoxes that naturally occur in single-level, flat domain approaches.

5 Conclusion

As multi-level modelling grows as a research discipline, there is hope that all groups involved will be able to converge on some universally acceptable concepts and ideas. We are by no means suggesting that all current and future approaches should be brought into line with a single view, though. On the contrary, variety and lively discussions are obviously important to further any new discipline. Nevertheless, unless agreement can be found on a number of fundamental questions, the discipline and its community will struggle to grow cohesively.

In this paper we therefore suggested that future debates should be not be held from subjective standpoints using a particular school of thought as a frame of reference. Observing incompatibilities between different approaches can be a starting point for fruitful discussions, but cannot be used as a basis for claims of other approaches being "wrong".

Moreover, we propose that in the vast majority if cases different approaches should not be judged as "right" or "wrong" but as "better" or "worse". The intention behind this proposal is not just to use relative terminology versus absolute terminology, it more crucially also involves different evaluation criteria. In our view, the merits of an approach should be judged on what benefits and/or disadvantages it brings to the intended target audience. In other words, instead of using subjective "schools of thought" as a reference, we propose to use pragmatics as a deciding criterion. Who exactly is the target audience? How does the target audience benefit from a certain philosophical underpinning in a concrete manner? What are the extra tools and/or opportunities given to a user? How do different approaches compare in terms of model maintenance? We believe that these are some of the crucial questions that should be asked when debating divergent viewpoints.

Empirical evaluations are notoriously hard to conduct, but debates on the merits of a certain approach can also be resolved theoretically. As long as arguments and comparisons are made with an explicitly stated target audience in mind, we believe that the discipline of multi-level modelling can converge and future users can benefit from this new, emerging sub-discipline of model-driven development.

References

1. Atkinson, C., Kennel, B., Go, B.: The level-agnostic modeling language. In: Malloy, B., Staab, S., Brand, M. (eds.) Software Language Engineering, Lecture Notes in Computer Science, vol. 6563, pp. 266–275. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-19440-5_16
2. Atkinson, C., Kühne, T.: Meta-level independent modeling. In: International Workshop Model Engineering (in Conjunction with ECOOP'2000). Springer Verlag, Cannes, France (Jun 2000)
3. Atkinson, C., Kühne, T.: Profiles in a strict metamodeling framework. Journal of the Science of Computer Programming 44(1), 5–22 (Jul 2002)
4. Atkinson, C., Kühne, T.: Model-driven development: A metamodeling foundation. IEEE Software 20(5), 36–41 (Sep 2003)
5. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation 12(4), 290–321 (Oct 2003)
6. Atkinson, C., Kühne, T.: A tour of language customization concepts. In: Zelkowitz, M. (ed.) Advances in Computers, vol. 70, chap. 3, pp. 105–161. Academic Press, Elsevier (June 2007)
7. Atkinson, C., Kühne, T.: In defence of deep modelling. submitted for publication (2014)
8. Bracha, G., Griswold, D.: Strongtalk: Typechecking smalltalk in a production environment. In: Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (1993)
9. Eriksson, O., Henderson-Sellers, B., Ågerfalk, P.J.: Ontological and linguistic metamodelling revisited: A language use approach. Information & Software Technology 55(12), 2099–2124 (2013)
10. Gitzel, R., Merz, M.: How a relaxation of the strictness definition can benefit MDD approaches with meta model hierarchies. In: Proceedings of the 8^{th} World Multi-Conference on Systemics, Cybernetics and Informatics. vol. IV, pp. 62–67 (July 2004)
11. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, MA (1983)
12. Henderson-Sellers, B., Clark, T., Gonzalez-Perez, C.: On the search for a level-agnostic modelling language. In: Proceedings of the 25th International Conference on Advanced Information Systems Engineering. pp. 240–255. CAiSE'13, Springer-Verlag, Berlin, Heidelberg (2013)
13. Henderson-Sellers, B., Eriksson, O., Gonzalez-Perez, C., Ågerfalk, P.J.: Ptolemaic Metamodelling?: The Need for a Paradigm Shift, chap. 4, pp. 90–146. IGI Global (2013)
14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge, Mass. (Apr 2006)
15. Kühne, T.: Matters of (meta-) modeling. Software and Systems Modeling 5(4), 369–385 (2006)

16. Kühne, T.: Contrasting classification with generalisation. In: Sixth Asia-Pacific Conference on Conceptual Modelling, APCCM. pp. 71–78 (2009)
17. Kühne, T., Steimann, F.: Tiefe charakterisierung. In: Rumpe, B., Hesse, W. (eds.) Proceedings of Modellierung 2004. pp. 121–133. LNI (45), GI (Mar 2004)
18. de Lara, J., Guerra, E.: Deep meta-modelling with metadepth. In: TOOLS (48). pp. 1–20 (2010)
19. Mannadiar, R.: A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling. Ph.D. thesis, McGill University (2012)
20. Meyer, B.: EIFFEL the language. Prentice Hall, Object-Oriented Series (1992)

Towards Automating the Analysis of Integrity Constraints in Multi-level Models

Esther Guerra and Juan de Lara

Universidad Autónoma de Madrid (Spain)
{Esther.Guerra, Juan.deLara}@uam.es

Abstract. Multi-level modelling is a technology that promotes an incremental refinement of meta-models in successive meta-levels. This enables a flexible way of modelling, which results in simpler and more intensional models in some scenarios. In this context, integrity constraints can be placed at any meta-level, and need to indicate at which meta-level below they should hold. This requires a very careful design of constraints, as constraints defined at different meta-levels may interact in unexpected ways. Unfortunately, current techniques for the analysis of the satisfiability of constraints only work in two meta-levels.
In this paper, we give the first steps towards the automation of mechanisms to check the satisfiability of integrity constraints in a multi-level setting, leveraging on "off-the-shelf" model finders.

1 Introduction

Multi-level modelling [3] is a promising technology that enables a flexible way of modelling by allowing the use of an arbitrary number of meta-levels, instead of just two. This results in simpler models [4], typically in scenarios where the type-object pattern or some variant of it arises.

While multi-level modelling has benefits, it also poses some challenges that need to be addressed in order to foster a wider adoption of this technology [10]. One of these challenges is the definition and analysis of constraints in multi-level models. In a two-level setting, constraints are placed in the meta-models and evaluated in the models one meta-level below. This enables the use of "off-the-shelf" model finders [1, 6, 12, 13, 16] to reason about correctness properties, like satisfiability (*is there a valid model that satisfies all constraints?*). However, constraints in multi-level models can be placed at any meta-level and be evaluated any number of meta-levels below, which may cause unanticipated effects. This makes the design and reasoning on the validity of constraints more intricate.

In this paper, we present the first steps towards a systematic method for the analysis of a basic quality property in multi-level modelling: the satisfiability of integrity constraints. We base our approach on the use of "off-the-shelf" model finders, which are able to perform a bounded search of models conforming to a given meta-model and satisfying a set of OCL constraints. Since the state-of-the-art model finders only work in a two-level setting, we need to "flatten" the multiple levels in a multi-level model to be able to use the finders for our

purposes. This process has two orthogonal dimensions, which account for the number of meta-levels provided to, and searched by, the finder. Thus, we discuss alternative flattening algorithms for different analysis scenarios. As a proof of concept, we illustrate our method through the analysis of MetaDepth multi-level models [9] using the USE Validator [13] for model finding.
Paper organization. Section 2 introduces multi-level modelling as designed in the MetaDepth tool. Section 3 presents properties and scenarios in the analysis of multi-level models. Section 4 discusses strategies for flattening multi-level models for their analysis with standard model finders. Section 5 describes the use of a model finder to analyse MetaDepth models. Last, Section 6 reviews related research and Section 7 draws some conclusions and future works.

2 Multi-level modelling

We will illustrate our proposal using a running example in the area of domain-specific process modelling, while the introduced multi-level concepts are those provided by the MetaDepth tool [9]. The example is shown in Fig. 1 using MetaDepth syntax (left) and a graphical representation (right). For the textual syntax, we only show the two upper meta-levels of the solution.

The main elements in a multi-level solution are models, clabjects, fields, references and constraints. All of them have a potency, indicated with the @ symbol. The potency is a positive number (or zero) that specifies in how many meta-levels an element can be instantiated. It is automatically decremented at each deeper meta-level, and when it reaches zero, the element cannot be instantiated in the meta-levels below. If an element does not define a potency, it receives the potency from its enclosing container, and ultimately from the model. Hence, the potency of a model is similar to the notion of *level* in other multi-level approaches [3].

As an example, the upper model in Fig. 1 contains a clabject Task with potency 2, thus allowing the creation of types of tasks in the next meta-level (e.g., Coding), and their subsequent instantiation into concrete tasks in the bottom meta-level (e.g., c1). Task defines two fields: name has potency 1 and therefore it receives values in the intermediate level, while startDay has potency 2 and is used to set the start day of specific tasks in the lowest meta-level.

In MetaDepth, references with potency 2 (like next) need to be instantiated at potency 1 (e.g., nextPhase), to be able to instantiate these latter instances at level 0. The cardinality of a reference constrains the number of instances at the meta-level right below. Thus, the cardinality of next controls the instantiations at level 1, and the cardinality of nextPhase the ones at level 0.

Constraints can be declared at any meta-level. In this case, the potency states how many meta-levels below the constraint will be evaluated. Constraint C1, which ensures uniqueness of task names, has potency 1, and therefore it will be evaluated one meta-level below. Constraint C2 has potency 2, and hence it states that two meta-levels below, the start day of a task must be less than the start day of any task related to it by next references. C2 needs to refer to the instances of the instances of the reference next, two levels below, but the (direct)

```
Model ProcessModel@2 {
  Node Task {
    name@1 : String[0..1];
    startDay : int;
    next : Task[*];

    C1@1: $ Task.allInstances()->excluding(self)->
                forAll( t | t.name<>self.name ) $
    C2: $ self.references('next')->forAll( r |
              self.value(r)->forAll( n |
                self.startDay < n.startDay )) $
  }
}

ProcessModel SEProcessModel {
  abstract Task SoftwareEngineeringTask {
      final : boolean = false;
      C3: $ self.startDay>0 $
  }

  Task Coding : SoftwareEngineeringTask {
    name = 'Coding';
    nextPhase : SoftwareEngineeringTask[1..*]{next};
    C4: $ self.final = false $
  }

  Task Testing : SoftwareEngineeringTask {
    name = 'Testing';
    nextPhase : Testing[*]{next};
    C5: $ self.final implies self.nextPhase->size()=0 $
    C6: $ Coding.allInstances()->exists( c |
              c.startDay = self.startDay ) $
  }
}
```

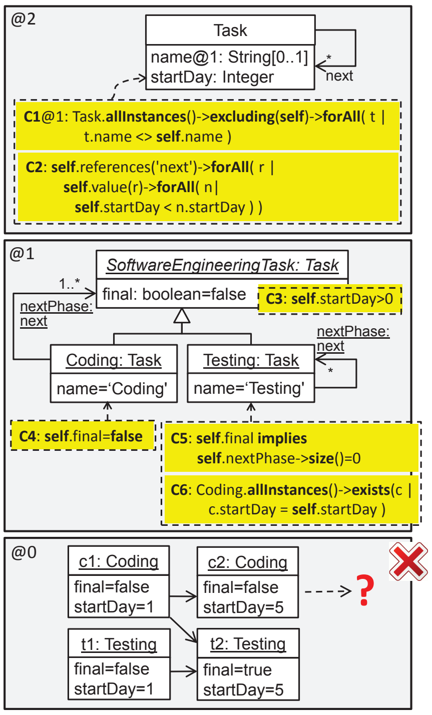


Fig. 1. Running example in MetaDepth syntax (left) and diagram (right).

type of the instances two levels below is unknown beforehand because it depends on the elements created at level 1. In the example, it means that C2 cannot make use of nextPhase to constrain the models at level 0. Instead, to allow the access to indirect instances of a given reference, MetaDepth offers two operations:

– references returns the name of the references that instantiate a given one. For example, self.references('next') evaluated at clabject c1 yields Set{'nextPhase'}, as the type of c1 defines the reference nextPhase as an instance of next.
– value returns the content of a reference with the given name. For example, self.value('nextPhase') evaluated in clabject c1 yields Set{c2,t2}.

In the intermediate meta-level, constraints C3, C4, C5 and C6 have potency 1, and thus they need to be satisfied by models at the subsequent meta-level. The purpose of C3 is to enforce positive starting days, where note that the feature startDay is defined one meta-level above. C4 ensures that Coding tasks are not final, while C5 requires final Testing tasks to have no subsequent tasks. Finally, C6 is an attempt to enable some degree of parallelization of tasks, where the modeller wanted to express that any coding task should have a testing task starting the same day.

The lower meta-level in the figure is an attempt (with no success) to instantiate the model with potency 1. The modeller started adding the coding tasks c1 and c2 at days 1 and 5. Then, to satisfy the constraint C6, he added two testing tasks starting at days 1 and 5 as well. Coding tasks must be followed by some other task to avoid they are left untested (controlled by the cardinality 1..* of nextPhase), and the start day of consecutive tasks must be increasing (controlled by constraint C2). Thus, the modeller connected the tasks as shown in the figure to satisfy these constraints. However, then, he realised that the coding task c2 needed to be followed by some other task. Connecting c2 with t2 is not a valid solution because this would violate constraint C2. Therefore, how can he connect the testing tasks to the coding tasks to satisfy all constraints? The next section explains how model finders can help in this situation.

3 Analysis of multi-level models: properties and scenarios

A meta-model should satisfy some basic properties, like the possibility of creating a non-empty instance that does not violate the integrity constraints. Several works [7] rely on model finding techniques to check correctness properties of meta-models in a standard two meta-level setting, like:

– **Strong satisfiability:** There is a meta-model instance that contains at least one instance of every class and association.
– **Weak satisfiability:** There exists a non-empty instance of the meta-model.
– **Liveliness of a class** c**:** There exists some instance of the meta-model that contains at least one instance of c.

Model finding techniques can also be helpful in a multi-level setting. If we consider level 0 in Fig. 1, a model finder can help model developers by providing a suitable model completion, or indicating that no such completion exists. At level 1, it can help to check the consistency of the integrity constraints at levels 1 and 2 through the analysis of the abovementioned correctness properties. At level 2, it can provide example instantiations for levels 1 and 0, to ensure that potencies of the different elements at the top-most level work as expected.

However, model finders work in a two-level setting (i.e., they receive a meta-model and produce an instance). To enable their use with several meta-levels, we need flattening operations that merge several meta-levels into one, which can then be the input to the finder. The flattening operations must take into account how many meta-levels are going to be used in the analysis (*depth of model*), as well as the number of meta-levels in the generated snapshot (*height of snapshot*).

Fig. 2 shows the different scenarios we need to solve for the analysis of multi-level models. Fig. 2(a) is the most usual case, where only the definition of the top-most model is available, and we want to check whether this can be instantiated at each possible meta-level below (2 in the case of having an upper model with potency 2). Thus, in the figure, the depth of the model to be used in the analysis is 1, while the height of the searched snapshot is 2. As standard model finders

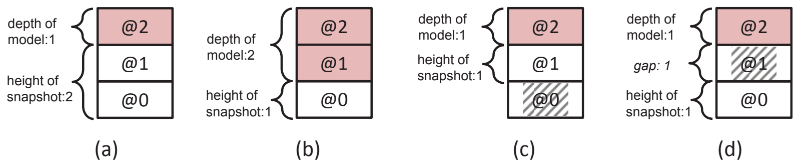


Fig. 2. Different scenarios in the analysis of a multi-level model.

only provide snapshots of models residing in one meta-level, we will need to emulate the generation of several meta-levels within one.

In Fig. 2(b), the models of several successive meta-levels are given, and the purpose is checking whether there is an instance at the next meta-level (with potency 0) satisfying all integrity constraints in the provided models. This situation arises when there is the need to check the correctness of the constraints introduced at a meta-level (e.g., `@1`) with respect to those in the meta-levels above (e.g., `@2`). This scenario would have helped in the analysis of the constraints at levels 2 and 1 in Fig. 1. In Fig. 2(b), the depth of the model to be used in the analysis is 2. Thus, we will need to flatten these two models into a single one, which can be fed into a model finder for standard snapshot generation.

Fig. 2(c) corresponds to the scenario that standard model finders are able to deal with, where a model is given, and its satisfiability is checked by generating an instance of it. However, the meta-model to be fed into the solver still needs to be adjusted, removing constraints with potency bigger than 1.

Finally, in Fig. 2(d), only the top-most model is available, and the designer is interested just in the analysis of the lowest meta-level. This can be seen as a particular case of scenario (a), where after the snapshot generation, the intermediate levels are removed. This scenario is of particular interest to verify the existence of instances at the bottom level with certain characteristics, like a given number of objects of a certain type, or to assess whether the designed potencies for attributes work as expected.

4 Flattening multi-level models for analysis

In this section, we use the running example to show how to flatten the depth of models to be used in the search, and how to deal with the height of the searched snapshot. Scenarios where both the height and depth are bigger than one are also possible, being resolved by combining the flattenings we present next.

4.1 Depth of analysed model

To analyse a model that is not at the top-most meta-level (like in Fig. 2(b), where the goal is analysing level 1), we need to merge it with all its type models at higher levels. This permits considering all constraints and attributes defined at such higher levels. For simplicity, we assume the merging of just two meta-levels. Fig. 3(a) shows this flattening applied to the running example: levels 1 and 2 are merged, as the purpose is creating a regular instance at level 0.

First, the flattening handles the top level. All its clabjects (Task in the example) are set to abstract to disable their instantiation. All references are deleted (next), as only the references defined at level 1 (nextPhase) can be instantiated at level 0. All attributes are kept, as if their potency is bigger or equal than the depth (2) plus the height (1), they still can receive a default value. Constraints with potency different from 2 (C1) are deleted as they do not constrain the level we want to instantiate (level 0). As the figure shows, the notion of potency does not appear in the flattened model. This can be interpreted as all elements having potency 1.

Then, the model at potency 1 is handled. For clabjects, the instantiation relation is changed by inheritance. In this way, SoftwareEngineeringTask is set to inherit from Task instead of being an instance of it. This is not required for Coding and Testing, as they already inherit from SoftwareEngineeringTask. This flattening strategy allows clabjects in level 1 to naturally define all attributes that were assigned potency 2 at level 2 (startDay), and receive a value at level 0. For attributes that receive a value at level 1, we need to emulate their value using constraints. Thus, in the example, we substitute the slot name from Coding and Testing, by constraints C7 and C8. The attributes, references, and constraints defined by the clabjects at level 1 are kept. Finally, we generate two operations references and value to emulate the homonym MetaDepth built-in operations by collecting the knowledge about reference instantiation statically. That is, they encode that nextPhase in Coding and Testing are instances of next. As a result of this flattening, we can use a model finder to check whether there is a valid instance at level 0.

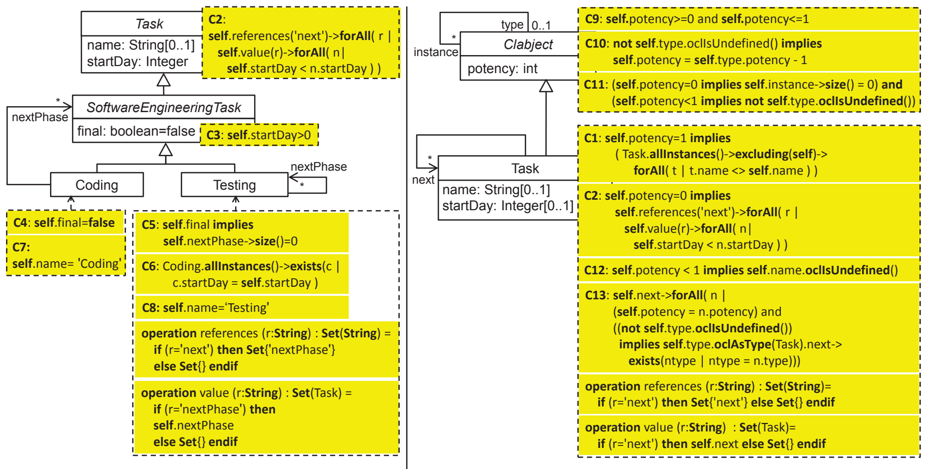
4.2 Height of snapshot generation

In this scenario, we need to emulate the search of a set of models spawning several meta-levels. For simplicity, we assume the scenario in Fig. 2(a), aimed at generating two models in consecutive levels from a meta-model with potency 2.

A possible strategy is to split each clabject C with potency 2 into two classes CType and CInstance holding the attributes, references and constraints with potency 1 and 2, respectively, and related by a reference type. However, this solution gets cumbersome if C is related or inherits from other clabjects, and requires rewriting the constraints in terms of the newly introduced types and relations.

Another possibility is to proceed in two steps: first a model of potency 1 is generated, which is promoted into a meta-model that can be instantiated into a model of potency 0. However, this solution may require rewriting the constraints with potency 2 in terms of the types generated at potency 1. Moreover, it does not consider all constraints at a time, which may result in different attempts before two valid models at potencies 1 and 0 are obtained.

Instead, we propose the flattening in Fig. 3(b), which adds a parent abstract class Clabject that makes explicit typical clabject features, like ontological typing and potency. All constraints are kept, but they need to be changed slightly to take into account their potency. Thus, C1 is added the premise self.potency=1 implies... so that it gets only applicable to tasks with potency 1, and similar for constraint C2 for tasks at potency 0. To emulate the potency of attributes, they are



```
C2:
self.references('next')->forAll( r |
  self.value(r)->forAll( n|
    self.startDay < n.startDay ) )

C3: self.startDay>0

C4: self.final=false

C7:
self.name= 'Coding'

C5: self.final implies
     self.nextPhase->size()=0

C6: Coding.allInstances()->exists(c |
       c.startDay = self.startDay )

C8: self.name='Testing'

operation references (r:String) : Set(String) =
   if (r='next') then Set{'nextPhase'}
   else Set{} endif

operation value (r:String) : Set(Task) =
   if (r='nextPhase') then
   self.nextPhase
   else Set{} endif
```

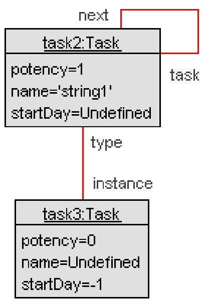
(a) Scenario with depth=2 and height=1.

```
C9: self.potency>=0 and self.potency<=1

C10: not self.type.oclIsUndefined() implies
       self.potency = self.type.potency - 1

C11: (self.potency=0 implies self.instance->size() = 0) and
       (self.potency<1 implies not self.type.oclIsUndefined())

C1: self.potency=1 implies
       ( Task.allInstances()->excluding(self)->
           forAll( t | t.name <> self.name ) )

C2: self.potency=0 implies
       self.references('next')->forAll( r |
         self.value(r)->forAll( n|
           self.startDay < n.startDay ) )

C12: self.potency < 1 implies self.name.oclIsUndefined()

C13: self.next->forAll( n |
       (self.potency = n.potency) and
       ((not self.type.oclIsUndefined())
        implies self.type.oclAsType(Task).next->
          exists(ntype | ntype = n.type)))

operation references (r:String) : Set(String)=
   if (r='next') then Set{'next'} else Set{} endif

operation value (r:String)  : Set(Task)=
   if (r='next') then self.next else Set{} endif
```

(b) Scenario with depth=1 and height=2.

Fig. 3. Flattenings for different depths and heights.

set to optional (cardinality [0..1]), and we add constraints ensuring that the attributes are undefined in the meta-levels where they cannot be instantiated. For example, name has originally potency 1, and hence it can only receive a value in tasks of potency 1 (constraint C12). Constraint C13 ensures that references (like next) do not cross meta-levels and are correctly instantiated at every meta-level. The latter means that, if two tasks at potency 0 are related by a next reference, then their types must be related via a next reference as well. While this does not fully captures the instantiation semantics as there is no explicit "instance-of" relation between references with different potencies, it suffices our purposes. Finally, constraints C9 to C11 ensure correct potency values for types and instances. As an example, the figure to the right shows a snapshot with height 2, generated by USE from the definition in Fig. 3(b).



5 Automating the analysis of constraints

Next, we illustrate our method by checking the satisfiability of MetaDepth multi-level models with the USE Validator [13] for model finding. The checking includes the following steps: (1) flattening of multi-level model according to the selected scenario; (2) translation of flattened model into the input format of USE; (3) generation of snapshot with the USE Validator tool; and (4) translation of the results back to MetaDepth. We will demonstrate these steps for the running example, considering the scenario in Fig. 2(b), i.e., we start from models at potency 2 and 1, and check their instantiability at potency 0.

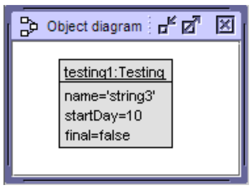
Fig. 3(a) shows the merging of levels 1 and 2 for the running example, while part of its translation into the input format of USE is listed below. The USE

Validator does not currently support solving with arbitrary strings, but they must adhere to the format 'string<number>'. Thus, we translate strings to this format, where 'next' is substituted by 'string0' (lines 12 and 27), 'nextPhase' by 'string1' (lines 28 and 31), and so on.

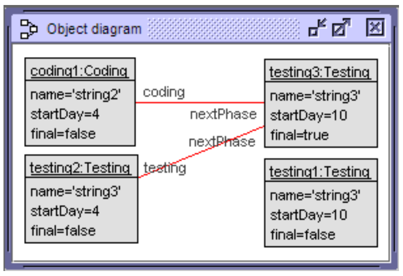
```
 1 model ProcessModel
 2
 3 abstract class Task
 4   attributes
 5     name : String
 6     startDay : Integer
 7   operations
 8     references(r:String) : Set(String) = Set{}
 9     value (r:String) : Set(Task) = Set{}
10   constraints
11     inv C2:
12       self.references('string0')−>forAll(r |
13         self.value(r)−>forAll(n |
14           self.startDay < n.startDay))
15 end
16
17 abstract class SoftwareEngineeringTask < Task
18   attributes
19     final : Boolean
20   constraints
21     inv C3: self.startDay > 0
22 end
23
24 class Coding < SoftwareEngineeringTask
25   operations
26     references(r:String) : Set(String) =
27       if (r='string0')
28       then Set{'string1'}
29       else Set{} endif
30     value(r:String) : Set(Task) =
31       if (r='string1')
32       then self.nextPhase
33       else Set{} endif
34   constraints
35     inv C4: self.final=false
36     inv C7: self.name = 'string2'
37 end
38
39 ...
40
41 association Coding_nextPhase between
42   Coding[*]
43   SoftwareEngineeringTask[1..*] role nextPhase
44 end
```

If we try to find a valid instance of this definition, we discover that the only model satisfying all constraints is the empty model. Thus, the model at level 1 is neither weak nor strong satisfiable. Revising the constraints at level 1, we realise that C6 does not express what the designer had in mind (that for any coding task, there should be a testing task starting the same day), but it expresses the converse (i.e., for each testing class, a coding class exists). One solution is moving constraint C6 from class Testing to Coding, modified to iterate on all instances of Testing (i.e., Testing.allInstances()...). If we perform this change, the resulting model becomes satisfiable, and the USE Validator generates the snapshots in Fig. 4.

Weak satisfiability is checked by finding a valid non-empty model. The USE Validator allows configuring the minimum and maximum number of objects and references of each type in the generated model. If we set a lower bound 1 for class Testing, we obtain the instance model shown in the left of Fig. 4. Strong



testing1:Testing
name='string3'
startDay=10
final=false



coding1:Coding
name='string2'
startDay=4
final=false

testing3:Testing
name='string3'
startDay=10
final=true

testing2:Testing
name='string3'
startDay=4
final=false

testing1:Testing
name='string3'
startDay=10
final=false

Fig. 4. Showing weak (left) and strong (right) satisfiability of the running example.

```
Coding.allInstances()−>exists(c1,c2|
  Testing.allInstances()−>exists(t1,t2|
    c1.final = false and
    c2.final = false and
    t1.final = false and
    t2.final = true and
    c1.startDay = 1 and
    c2.startDay = 5 and
    t1.startDay = 1 and
    t2.startDay = 5 and
    c1.nextPhase−>includes(c2,t2) and
    t1.nextPhase−>includes(t2) ))
```

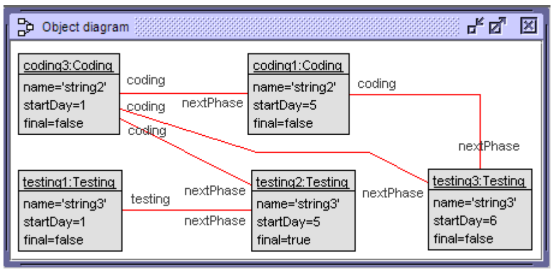


Fig. 5. Encoding of incomplete model at level 0 (left). Complete valid instance (right).

satisfiability is checked by finding a model that contains an instance of every class and reference. By assigning a lower bound 1 to all types, the USE Validator finds the model to the right of Fig. 4.

If the scenario to solve is completing a partial model, like the one at the bottom level of Fig. 1, we need to provide a seed model for the search. This can be emulated by an additional constraint demanding the existence of the starting model structure. Fig. 5 shows the OCL constraint representing our example model at level 0 (left), as well as the complete valid model found by USE (right).

6 Related work

Some multi-level approaches have an underlying semantics based on constraints, like Nivel [2], which is based on WCRL. This allows some decidable, automated reasoning procedures on Nivel models, but they lack support for integrity constraints beyond multiplicities.

There are several tools to validate the satisfiability of integrity constraints in a two-level setting. We have illustrated our method with the USE Validator [13], which translates a UML model and its OCL constraints into relational logic, and uses a SAT solver to check its satisfiability. UML2Alloy [1] follows a similar approach. Instead, UMLtoCSP [6] and EMFtoCSP [12] transform the model into a constraint satisfaction problem (CSP) to check its satisfiability, and ocl2smt [16] translates it into a set of operations on bit-vectors which can be solved by SMT solvers. The approach of Queralt [14] uses resolution and Clavel [8] maps a subset of OCL into first-order logic and employs SMT solvers to check unsatisfiability. HOL-OCL [5] is a theorem proving environment for OCL. In contrast to the enumerated tools, it does not rely on bounded model finding, but it is able of proving complex properties of UML/OCL specifications. All these works consider two meta-levels, and could be used to solve the multi-level scenarios in Section 3, once they have been translated into a two-level setting. Other works use constraint solving for model completion [15], also in a two-level setting.

Altogether, the use of model finders to verify properties in models is not novel. However, to the best of our knowledge, ours is the first work targeting the analysis of integrity constraints in multi-level models.

7 Conclusions and future work

In this paper, we have proposed a method to check the satisfiability of constraints in multi-level models using "off-the-shelf" model finders. To this aim, the method proposes flattenings that depend on the number of levels fed to the finder and the height of the generated snapshot. The method has been illustrated using MetaDepth and the USE Validator.

Currently, we are working towards a tighter integration of the USE validator with MetaDepth, providing commands to e.g., complete a given model. We also plan to analyse other properties, like independence of constraints [11].

Acknowledgements. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139).

References

1. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: a challenging model transformation. In *MoDELS*, volume 4735 of *LNCS*, pages 436–450. Springer, 2007.
2. T. Asikainen and T. Männistö. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling*, 8(4):521–549, 2009.
3. C. Atkinson and T. Kühne. The essence of multilevel metamodeling. In *UML*, volume 2185 of *LNCS*, pages 19–33. Springer, 2001.
4. C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7(3):345–359, 2008.
5. A. D. Brucker and B. Wolff. HOL-OCL: a formal proof environment for UML/OCL. In *FASE*, volume 4961 of *LNCS*, pages 97–100. Springer, 2008.
6. J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE*, pages 547–548, 2007.
7. J. Cabot, R. Clarisó, and D. Riera. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23, 2014.
8. M. Clavel, M. Egea, and M. A. G. de Dios. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST*, 24:1–13, 2009.
9. J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010.
10. J. de Lara, E. Guerra, R. Cobos, and J. Moreno-Llorena. Extending deep meta-modelling for practical model-driven engineering. *Comput. J.*, 57(1):36–58, 2014.
11. M. Gogolla, L. Hamann, and M. Kuhlmann. Proving and visualizing OCL invariant independence by automatically generated test cases. In *TAP*, volume 6143 of *LNCS*, pages 38–54. Springer, 2010.
12. C. A. González, F. Büttner, R. Clarisó, and J. Cabot. EMFtoCSP: a tool for the lightweight verification of EMF models. In *FormSERA*, 2012.
13. M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *TOOLS (49)*, volume 6705 of *LNCS*, pages 290–306. Springer, 2011.
14. A. Queralt and E. Teniente. Verification and validation of UML conceptual schemas with OCL constraints. *TOSEM*, 21(2):13, 2012.
15. S. Sen, B. Baudry, and H. Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109–126, 2010.
16. M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *DATE*, pages 1341–1344. IEEE, 2010.

Towards Flexible, Incremental, and Paradigm-agnostic Consistency Checking in Multi-level Modeling Environments

Andreas Demuth, Markus Riedl-Ehrenleitner, and Alexander Egyed

Institute for Software Systems Engineering,
Johannes Kepler University Linz, Austria
{firstname.lastname}@jku.at
http://www.jku.at/isse

Abstract. Multi-level modeling has become a popular paradigm as it allows for a natural and easy-to-understand representation of various real-word hierarchies. To date, several approaches have been proposed on how multi-level models should be represented and constructed – however, their continuous evolution and consistency has received considerably less attention. Consistency checking is critical to efficient and effective modeling—especially to understand the impact of model changes. Multi-level modeling adds another dimension because it allows for both model and metamodel changes over multiple levels. This paper discusses the key challenges for consistency checking in multi-level modeling environments and outlines an incremental and highly flexible approach for addressing these challenges effectively without being limited to a specific modeling paradigm. A prototype implementation of the approach has been developed; preliminary evaluation results suggest that the approach scales and provides instant consistency information during multi-level modeling.

1 Introduction

By applying model-driven engineering (MDE) approaches, practitioners raise the level of abstraction in software and systems engineering. This allows for easier communication and more efficient development processes as models become first class development artifacts that are used as blueprints for (semi-)automatic generation of the desired system. However, it has been shown that traditional two-layer approaches, in which a domain-specific language is used to model a specific instance of the domain, suffers from a lack of support for expressing the often complex hierarchies that occur in real-world domains. Even though there exist workarounds for handling such hierarchies in common two-layer modeling languages and tools (e.g., in UML), the solutions are usually not generic and often counter-intuitive. Multi-level modeling allows for modeling arbitrary deep hierarchies by allowing specific models to serve both as domain-specific instance models of a certain domain and the domain language of another instance model—a more specialized domain. Although it has been shown that multi-level modeling

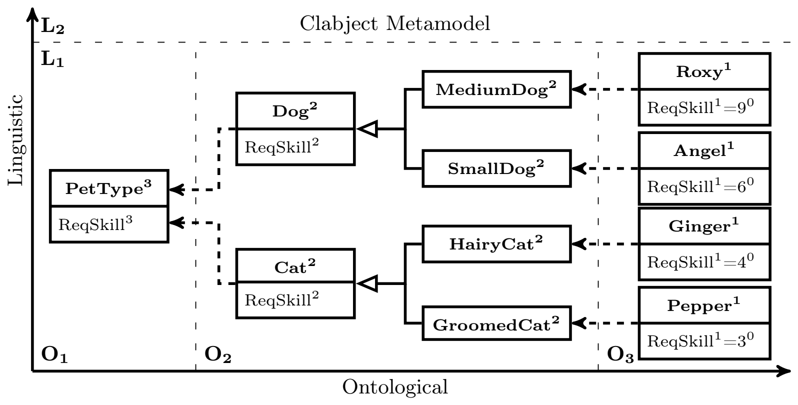


Fig. 1: Pet Store Web Shop Example (based on [4]).

makes models more intuitive to construct and read, it is still an open question in the research community how exactly model construction should be done and which methods should be used. Therefore, to date there exist various multi-level modeling approaches, paradigms, and tools (e.g., [1–3]). However, there is an important aspect that has been widely overlooked in multi-level modeling so far: the need for consistency checking in models. While it is well acknowledged that in MDE consistency checking is a crucial factor for building valid models efficiently, this has not been addressed sufficiently in multi-level modeling approaches. Although some approaches do define well-formedness constraints, these constraints typically define the semantics for a specific modeling paradigm only; there is usually no support for user-defined, domain-specific consistency rules.

In this paper, we outline the dimensions of consistency checking in multi-level modeling environments and present a generic and paradigm-agnostic approach that addresses these challenges, allowing modelers to easily write domain-specific consistency rules that check both syntax and semantics. While the approach is based on the general principles of incremental consistency checking for traditional two-level models, these concepts have been adapted and extended in order to handle multi-level models efficiently.

2 Multi-level Modeling Example

To illustrate the issues of consistency checking in multi-level modeling environments we use a standard example which is well known in the multi-level modeling community: the pet store web shop ontology [4]. As shown in Fig. 1, the example is modeled using the paradigm of *clabjects* [2] (the actual clabject metamodel is omitted for space reasons). Therefore, there are two levels in the linguistic dimension: the clabject metamodel at level $\mathtt{L_2}$ and the instance model at level $\mathtt{L_1}$ in which clabjects are used to model various ontological levels. In the ontological dimension, there are three levels. The level O_1 defines the concept of a `PetType`, which has an attribute named `ReqSkill` which indicates the required skill level a prospective owner of a specific pet should exhibit in order to take good care of it. Different types of pets are then defined at level O_2. Specifically, the two types `Dog` and `Cat` are defined. For each of these two types, two subtypes (or specializations) are defined through inheritance: `MediumDog` and `SmallDog` for the pet type `Dog` as well as `HairyCat` and `GroomedCat` for the pet type `Cat`. Finally, for each specialized kind of dog or cat, there is a single animal available: `Roxy`, `Angel`, `Ginger`, and `Pepper`. Since these animals are instances of the specialized kinds, they are modeled at level O_3.

3 Dimensions of Consistency Checking in Multi-level Modeling

Let us now discuss the different dimensions of consistency checking that are required in the example. Specifically, there are three major dimensions: i) linguistic conformance, ii) ontological conformance, and iii) evolution.

3.1 Checking Linguistic Conformance

As shown in Fig. 1, the example spans across two linguistic levels: L_1 and L_2. All model elements, regardless of the ontological level they reside on, must conform to the metamodel defined in L_2. In the example, the metamodel to which all model elements at L_1 must conform is that of the clabject modeling paradigm. Therefore, at L_1 it must be checked whether model elements are syntactically correct and whether they obey clabject semantics. For example, a syntactic consistency rule would be that every model element (e.g., the clabject `Dog`) must have a name and a potency assigned. This potency must be reduced by `1` with every instantiation (e.g., the clabject `Dog` must have a potency of `2` because it is an instance of `PetType`, which has a potency of `3`)—this is an example for a consistency rule checking modeling-paradigm-specific semantics. A formalization of these syntax and semantics rules that could be checked by a standard consistency checker, is depicted in Listing 1.1 (lines 1–4). The rule is written in OCL. However, note that checking semantics is necessary regardless of the used modeling paradigm; it would also be necessary if, for instance, the example was modeled with *powertypes* [1].

3.2 Checking Ontological Conformance

The second dimension we discuss is that of ontological conformance. In particular, this dimension has three major areas of interests: level-specific syntax and semantics, weak typing, and the handling of advanced modeling concepts such as inheritance.

```
1 context Clabject inv:
2 self.name<>null and
3 self.potency<>null and
4 self.potency = type.potency-1;
5
6 context PetType inv: self.ReqSkill>=0 and self.ReqSkill<=10;
7 context Dog inv: self.ReqSkill>=5;
8 context Cat inv: self.ReqSkill>=3;
9 context MediumDog inv: self.ReqSkill>=7;
```

Listing 1.1: Consistency Rules for Linguistic and Ontological Conformance.

Level-specific Syntax and Semantics. This involves checking whether a model at a given ontological level is semantically and syntactically conforming to its ontological metamodel (i.e., to its parent ontological level). Since domain-specific syntax rules do not differ significantly from linguistic syntax rules (e.g., lines 1–4 of Listing 1.1), we omit a detailed discussion here for space reasons. An example for domain-specific semantics could be the field `ReqSkill`, originally defined in `PetType` at level $\texttt{O}_1$, which must remain within a range of `0`–`10`. A value of `0` indicates that the animal does not need any care at all and `10` indicates that the animal requires extensive care on a daily basis. A corresponding OCL consistency rule is shown in 1.1 (line 6). It must be ensured that all model elements at level $\texttt{O}_3$ have an appropriate value set.

However, there might be additional semantic rules added at level $\texttt{O}_2$. For example, the range of skill levels required for handling a dog should be greater than or equal to `5` because dogs must be walked at least twice a day and they also tend to adopt undesired behavioral patterns if not handled correctly. For cats, on the other hand, the required skill level should be no smaller than `3` as they require, for instance, feeding at a regular basis. The corresponding OCL consistency rules are shown in Listing 1.1 (lines 7 and 8, respectively).

Moreover, there might be more specific requirements for certain kinds of dogs and cats. For instance, medium sized dogs may have a minimum skill level of `7` because they are harder to keep under control than small dogs due to their increased strength compared to small dogs. This is expressed in the rule shown in Listing 1.1 (line 9). Again, these domain-specific semantics have to be enforced at level $\texttt{O}_3$.

Therefore, different semantics are defined at different ontological levels. Depending on the followed modeling paradigm, it might be possible that each ontological level defines its own semantics, or each level might only refine the semantics defined at the levels above. Either way, it is required that at each ontological level conformance rules regarding syntax and semantics can be defined—which is typically not possible with existing consistency checking approaches.

Weak Typing. In multi-level modeling paradigms, type hierarchies across ontological levels are usually modeled by using concepts defined in the linguistic level $\texttt{L}_2$. Typically, references between model elements are used to model instantiations and similar relations. For instance, the type `Clabject` may have a reference `instanceof` that points to another `Clabject` and models instantiations. However, similar concepts exist in most—if not all—multi-level modeling paradigms. In order to write consistency rules for individual ontological levels, it is necessary to use these references to discover the ontological type of an element. Unfortunately, consistency checkers often work with linguistic types rather than with ontological types (i.e., they use runtime type information of objects). Thus, they are only capable of checking linguistic conformance (e.g., they may only check instances of `Clabject`, but not modeled instances of `Cat`).

Advanced Modeling Concepts. Similar to weak typing, there are advanced modeling concepts such as inheritance that are typically only handled at the linguistic level by consistency checkers. For example, above we discussed the semantics constraint that dogs require a minimum skill level of `5`. Indeed, this should be checked not only for direct instances of `Dog`, but also for instances of the defined subtypes `MediumDog` and `SmallDog` (i.e., `Roxy` and `Angel`). However, a standard consistency checker would not be able to handle such modeled inheritance (similar to modeled instantiation) as it would typically only consider inheritance at the linguistic level (e.g., if at the level $\mathtt{L}_2$ there was a specialization of `Clabject`, semantics defined for standard clabjects would also be checked for instances of the specialized clabjects). Moreover, at different ontological levels there might be different understandings of inheritance and thus different semantics attached, and at some levels it might be undesired to have available such modeling concepts at all. Thus, relying on a single understanding of inheritance that is defined at the top linguistic level—again, regardless of the actual paradigm used—is insufficient; it is required to support the explicit definition of concepts such as inheritance at ontological levels.

3.3 Handling Evolution

The third dimension that must be considered when checking consistency in multi-level modeling environments is evolution. While handling evolution is also necessary when checking traditional two-level models, multi-level modeling allows for evolution scenarios that are usually not present in two-level environments. Specifically, these scenarios are: i) dynamic type changes, and ii) dynamic changes of inheritance-relations. The scenarios can occur in multi-level modeling because of the weak typing and the flexible handling of concepts such as inheritance that is used for modeling ontological levels.

Let us consider a change in an inheritance-relation in our pet store example: the type `SmallDog` could be modeled as a specialization of `MediumDog` instead of `Dog` by simply changing the target of the inheritance reference. This would mean that model elements of the type `SmallDog` at $\mathtt{O}_3$ must not only conform to semantic rules defined for instances of the types `Dog` and `SmallDog`, but also to rules defined for instances of the type `MediumDog`. Moreover, note that in multi-level modeling the ontological type of a model element can be changed quite easily. The ontological type of `Angel`, for example, could be changed from

SmallDog to HairyCat by just changing the corresponding reference. This would mean that of the previously described rules only the allowed range of the required owner skill level (defined for PetType at level O_1) would be applicable to Angel.

Generally, for both change scenarios, syntax and semantics that a model element must conform to may change (i.e., a changed set of rules must be applied by a consistency checker). Such changes may be performed quite frequently due to the low cost and the typical way of how models are constructed by engineers. Therefore, it is crucial that these changes are handled efficiently—engineers typically expect modeling tools to immediately provide feedback after changes in a model have been performed.

3.4 Existing Support for Consistency Checking in Multi-level Modeling

To date, there exists a variety of consistency checking approaches (e.g., [5–11])—we now briefly summarize how they support the three dimensions of consistency checking in multi-level modeling environments. Generally, linguistic conformance can be checked sufficiently. Some approaches also handle evolution efficiently (e.g., [5]). However, checking ontological conformance and handling dynamic changes of types and inheritance at the ontological level is typically not supported by existing approaches. This is especially the case when requiring user-definable consistency rules.

4 Consistency Checking in Multi-level Modeling Environments

To address the issue of missing support for the dimension of ontological conformance checking, including the efficient handling of weak typing and dynamic changes of inheritance, we propose a novel approach to consistency checking in multi-level modeling environments that relies on a unification of linguistic and ontological levels. The approach allows for arbitrary and domain-specific consistency rules to be defined and applied at all modeled levels. It is paradigm-agnostic, thus supporting any multi-level modeling paradigm.

4.1 Linguistic and Ontological Dimension Unification

The cornerstone of our approach is the unification of ontological and linguistic levels. This allows a consistency checker to be employed for checking linguistic and ontological conformance alike. To achieve unification, the multi-level modeling paradigm, which is defined at level L_2, is transformed to a model at the newly added ontological level O_0. Thus, in our approach all levels of interest are of ontological nature, with the used multi-level modeling paradigm being the top-most ontological level. The result of this transformation for our running example is shown in Fig. 2. Again, please note that we use clabjects in the illustration but any modeling paradigm is supported at O_0. For modeling L_1 (i.e., O_0–O_3 in Fig. 2),

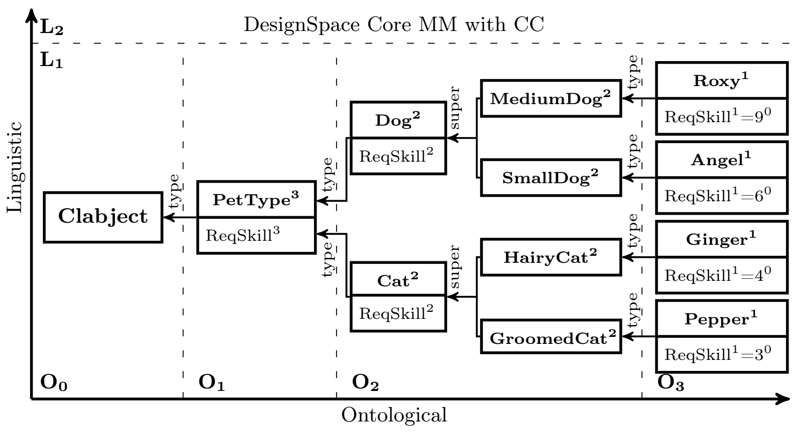


Fig. 2: Paradigm-agnostic Multi-level Modeling Scenario in the DesignSpace.

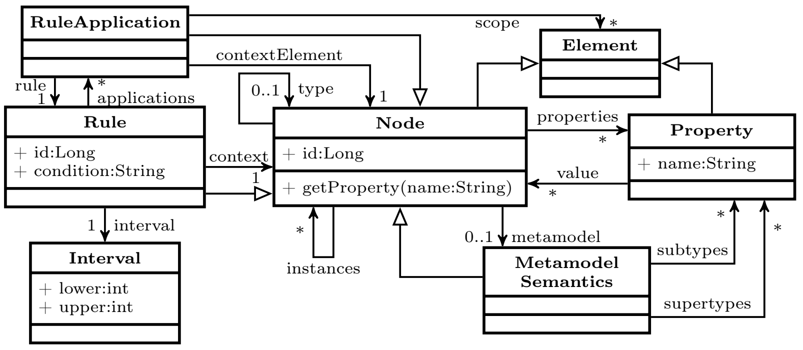


Fig. 3: DesignSpace Core Metamodel with Multi-level Consistency Checking.

which differs from `L₁` in Fig. 1, a new metamodel—called the *DesignSpace Core Metamodel (DSCM)*—is used that has been defined specifically for the purpose of flexible, multi-level modeling with consistency checking. This metamodel, which is depicted in Fig. 3, is based on a subset (hence “Core”) of the metamodel for flexible modeling used in our previous work on the *DesignSpace* [12] modeling environment that was enhanced with concepts to support multi-level consistency checking.

The DesignSpace Core Metamodel is sufficiently generic to model arbitrary data structures using the simple concept of nodes (type `Node`) that can provide named properties (type `Property`). Instantiation is modeled using the reference `type`. The modeled instances of a node can be retrieved through the reference `instances`. The DSCM can be used to model any multi-level modeling paradigm at the level `O₀`. In Fig. 2, the running example from Fig. 1 is modeled using DesignSpace concepts (simplified for readability reasons).

Note that consistency rules (type `Rule`), which can be used to check both syntax and semantics constraints (attribute `condition`), can be defined for any node (reference `context`), regardless of its ontological level. Our approach follows the principle of incremental consistency checking (e.g., [5]). There is a specific type (`RuleApplication`) that is used to model an individual application of a consistency rule. However, the way how rules are applied in a multi-level modeling environment differs significantly from two-level modeling approaches. Moreover, the metamodel must allow for the dynamic definition of (ontological) metamodel semantics. We will discuss these two aspects next, beginning with rule application strategies.

4.2 Rule Application Strategies

Applying consistency rules in multi-level models differs significantly from two-level models. In two-level models, rules are typically defined for a metamodel element and are then applied for all instances of that element. For example, a rule defined for the metamodel element `Clabject` in Fig. 1, such as the one defined on Listing 1.1, is applied to every single instance; i.e., every element at level $\mathtt{L}_1$. For multi-level modeling, this strategy is no longer sufficient due to the existence of multiple ontological levels and advanced rule application strategies are required.

Recall the semantics we discussed in Section 3 and the corresponding consistency rules shown in Listing 1.1 (lines 6–9), where the valid range of required owner skill levels was defined for different model elements (e.g., `PetType` at $\mathtt{O}_1$ and `Dog` at $\mathtt{O}_2$) and checked at level $\mathtt{O}_3$. Because of the used clabject modeling paradigm, checking rules that are defined at an arbitrary ontological level x at the level $x + 1$ does not make sense—the distance between the ontological level at which the rule is defined and at which it is applied can vary.

Moreover, note that there may be rules that should be checked not only at a single level, but at multiple levels—depending on the modeling paradigm. While the clabject paradigm requires attributes to have actual values assigned if, and only if, the attribute's potency is `1`, other paradigms may require that an attribute has a value assigned starting with a certain ontological level y. If the type containing the attribute is then, for example, refined at level $y + 1$, the attribute must still be set, yet it may have a different value assigned. Thus, the rule should be applied at the levels within the range of $[y; y + 1]$. Generally, it should therefore be possible to define for a rule a range of levels at which it should be checked (i.e., an interval $[a; b]$ where $a, b \in \mathbb{N}^+$).

Finally, there might be consistency rules that are defined at an ontological level z and that should be applied at all subsequent levels $z + i$ where $i \in \mathbb{N}^+$. Therefore, there should be the possibility of defining rule application ranges such as $[z + 1; \infty]$. Note that using such ranges semantically makes the ontological level at which the rule is defined (i.e., z) semantically to a linguistic level. In our example in Fig. 2, it is possible to check linguistic conformance to the modeling paradigm by defining consistency rules that express paradigm semantics at level $\mathtt{O}_0$ and using a rule application range of $[1; \infty]$.

Our approach supports all discussed rule application strategies. The strategy can be choosen for each consistency rule individually, typically depending on the used modeling paradigm. In the DSCM, rule application strategies are defined using the type `Interval`. Similar to cardinalities in UML, using `-1` instead of a positive integer allows the definition of an unbounded interval (e.g., $[1; -1]$).

4.3 Definition of Metamodel Semantics

As we have discussed above, advanced modeling concepts such as inheritance may be realized differently at different ontological levels. To support level-specific semantics, the DSCM included the type `MetamodelSemantics`. Specifically, for any node the metamodel semantics can be specified to define which properties are used to identify the node's respective super- and subtypes. For example, in Fig. 2 the property `super` models inheritance at $\mathtt{O}_2$. This allows consistency checkers to use a generic mechanism to dynamically discover metamodel semantics at any ontological level, and it enables dynamic changes of metamodel semantics at all times in flexible modeling tools. The latter is also beneficial for supporting incremental checking of constraints.

4.4 Efficient Evolution Handling

A key feature of multi-level modeling is the flexibility modelers have during modeling. Not only may modeled instances change, but also metamodels may change at all times. Therefore, it is of crucial importance that dynamic changes (e.g., type changes, changes in inheritance hierarchy) are processed efficiently so that modelers get immediate feedback. The DSCM in Fig. 3 includes the core concepts of incremental consistency checking [5] which were adapted for supporting multi-level modeling scenarios. In particular, notice the type `RuleApplication` which models a specific validation of a consistency rule on a specific instance (the `contextElement`) of the validated rule's `context`. During such a validation, a `scope` is built that contains all elements relevant by any means for the validation (i.e., any element that was accessed). This scope is used to find affected rule application whenever evolution takes place. A detailed explanation of the concept can be found in [5] and we omit a detailed discussion for space reasons. However, note that—in contrast to standarad consistency checking approaches—type-hierarchies and metamodel semantics are also `Element`s that are accessed dynamically when searching for locations to apply rules or when searching the rules to be applied to a specific model element. Thus, types and metamodel semantics can be part of rule application scopes. This means that dynamic type or inheritance hierarchies changes can also be handled efficiently (i.e., it can be determined easily which rule applications may be affected after such a change).

4.5 Prototype Implementation

A prototype implementation of the approach has been developed that is based on the DesignSpace [12] modeling framework and an adapted version of the Model/-Analyzer [5] consistency checker. A preliminary performance analysis suggests that required adaptations for multi-level modeling (e.g., rule application mechanism) do not impose performance drawbacks compared to the two-level version of the consistency checker. However, a detailed analysis of the performance effects is part of future work.

5 Conclusions and Future Work

In this paper we have discussed the dimensions of consistency checking in multi-level modeling and outlined a paradigm-agnostic approach that handles these dimensions and provides the well-known advantages of incremental consistency with domain-specific, user-definable rules. A prototype implementation of the approach demonstrated its feasibility. A complete validation of the approach, including scalability and applicability studies as well as a detailed description of key algorithms for handling changes unique to multi-level modeling, will be done in future work.

Acknowledgments

The research was funded by the Austrian Science Fund (FWF): P25289-N15 and P25513-N15, and the Austrian Center of Competence in Mechatronics (ACCM): Strategic Research Grant C210101

References

1. J. Odell, “Power types,” *JOOP*, vol. 7, no. 2, pp. 8–12, 1994.
2. C. Atkinson, “Meta-modeling for distributed object environments,” in *EDOC*, pp. 90–101, 1997.
3. C. Atkinson and T. Kühne, “Processes and products in a multi-level metamodeling architecture,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 6, pp. 761–783, 2001.
4. C. Atkinson, R. Gerbig, and B. Kennel, “On-the-fly emendation of multi-level models,” in *ECMFA*, pp. 194–209, 2012.
5. A. Egyed, “Automatically detecting and tracking inconsistencies in software design models,” *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 188–204, 2011.
6. C. Nentwich, W. Emmerich, and A. Finkelstein, “Consistency management with repair actions,” in *ICSE*, pp. 455–464, 2003.
7. C. K. F. Corrêa, “Towards automatic consistency preservation for model-driven software product lines,” in *SPLC Workshops*, p. 43, 2011.
8. M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, “Towards automated inconsistency handling in design models,” in *CAiSE*, pp. 348–362, 2010.
9. S. Easterbrook and B. Nuseibeh, “Using viewpoints for inconsistency management,” *Software Engineering Journal*, vol. 11, no. 1, pp. 31 –43, 1996.
10. C. Xu, S.-C. Cheung, and W. K. Chan, “Incremental consistency checking for pervasive context,” in *ICSE*, pp. 292–301, 2006.
11. L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack, “Enhanced automation for managing model and metamodel inconsistency,” in *ASE*, pp. 545–549, 2009.
12. M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed, “Towards model-and-code consistency checking,” in *COMPSAC*, pp. 85–90, 2014.

Multi-Level Modelling in the Modelverse

Simon Van Mierlo[1], Bruno Barroca[2], Hans Vangheluwe[1,2], Eugene Syriani[3], Thomas Kühne[4]

[1] University of Antwerp, Belgium
{simon.vanmierlo,hans.vangheluwe}@uantwerpen.be
[2] McGill University, Montréal, Canada
{bbarroca,hv}@cs.mcgill.ca
[3] Université de Montréal, Canada
syriani@iro.umontreal.ca
[4] Victoria University of Wellington, New Zealand
thomas.kuehne@ecs.vuw.ac.nz

Abstract. In this paper, we introduce the Modelverse, a metamodelling framework and model repository. It clearly distinguishes and supports physical and linguistic conformance relations and allows for deep characterization and deep instantiation using potency. We introduce language fragments, which are reusable pieces of a language definition, consisting of an abstract syntax definition, as well as the definition of concrete syntax, semantics, and a mapping onto physical (representational) concepts, as suitable concepts for modular language design and reuse. We focus on multi-level modelling, and use the Modelverse to model a four-level language hierarchy, demonstrating its deep instantiation and characterization capabilities, as well as the use of modelling language fragments.

Keywords: Multi-Level, Modelling Languages, Model-Driven Engineering

1 Introduction

Model-Driven Engineering (MDE) is a set of notations, methods, techniques and tools for designing, simulating, testing, and ultimately realizing so-called Software intensive Systems (SiS). MDE raises the level of abstraction compared to traditional software development techniques, which are mainly based on code.
The MDE approach can only be successful if there are tools supporting the various processes and methods used to develop these systems. Central to any modelling activity is the notion of a *modelling language*, defining the concepts a modeller can use, what their visual representation is (their *concrete syntax*), and their meaning, or semantics. Various modelling frameworks have been proposed, of which the Meta Object Facility (MOF) [1] is one of the most popular, and has been adopted as the standard by many metamodelling tools. The MOF uses a four-level language approach, of which two levels are user-accessible (the class and object level). Several articles have pointed out the limitations of this approach [2–5]. Most notably, the use of only one conformance dimension

(an object is an instance of exactly one class), and the fact that only two levels are user-accessible leads to inconsistencies, as strict metamodelling is made impossible by conformance links which cross multiple levels, and an increase in accidental complexity, as modellers have to resort to workarounds if they want to model types of types.
We contribute to this ongoing research by introducing a new framework and repository capable of modelling multi-level language hierarchies called the Modelverse. We also introduce language fragments, which allow for modular design of modelling languages. Section 2 provides background information for the rest of the paper. Section 3 presents the architecture of the Modelverse. In Section 4, the Modelverse is used to model a multi-level language hierarchy. Section 5 concludes the paper.

2 Background

In this section, the concepts of deep instantiation and deep characterization using potency is explained, and we take a look at the current tool support for multi-level language hierarchies.

2.1 Deep Instantiation and Deep Characterization

Traditional instantiation mechanisms consider only two levels: classes and their instances, objects. In case a modelling hierarchy requires types of types to be modelled, this approach falls short. For example, in a modelling system describing stores, it is necessary to model the types of objects which can appear in the store: books for a library, DVDs for a video store, or bread for a bakery. Instances of those types then describe actual products sold at those stores. It may be useful, however, to describe properties of products *in general*, in other words, to make statements about the *type of the product types*: for example, we might want to ensure that each product type has an attribute denoting its VAT. Current architectures do not have sufficient support for modelling these kinds of hierarchies, and the proposed solutions (for the MOF) are merely workarounds, not actual solutions to the inherent issue.
In a *deep instantiation* approach, a type model element can be instantiated more than one level down [4]. At level 0, the traditional object level, an element is *fully defined*, meaning that all of its attributes have received a value. Closely related is *deep characterization*: types can make statements about their indirect instances, two or more levels down in the modelling hierarchy. This is done through the use of *potency*. Each (deep) attribute (and modelling element) receives a potency number, signifying how many levels down it can be instantiated. Each element both has a type and an instance facet: an element with potency value 2 is an instance of an element with potency value 3, and is a type for elements with potency value 1. The top and the bottom level can be seen as exceptions: they only have a type or an instance facet, respectively. A special case are models with an undefined potency: for them, the number of levels down they can be instantiated is not known. For top-level type models, this is necessary, as the designer of such type models cannot know how

many levels will be introduced by users below it.
With *deep characterization*, a product type can ensure that actual instances of products (books, DVDs, bread) have a price, by declaring the price attribute with a potency value of 2. This can be seen as a constraint on instances of the product type: they all receive a potency 1 attribute with name 'price', and their instances have to provide a value for it.

2.2 Tool Support

As multi-level modelling is gaining importance, tools supporting multi-level modelling hierarchies and deep instantiation have been constructed, of which *metaDepth* [6], a modelling framework with built-in support for multi-level modelling, is an important example. The tool has a textual interface: models are constructed using a Human Usable Textual Notation (HUTN). *metaDepth* distinguishes two modelling dimensions: the linguistic dimension is static, and built into the tool. There is, however, support for linguistic extensions in the type models defined by modellers: attributes can be added, and there is support for inheritance on all levels of the modelling hierarchy. Deep instantiation and deep characterization are supported in the ontological dimension, with potency.
Melanie [7] is an Eclipse-based tool which allows multi-level modelling hierarchies in the ontological dimension. It allows to define domain-specific concrete syntax for languages, and as such it differs from the strictly textual approach of metaDepth. The linguistic dimension, however, is static and predefined, as is the case for metaDepth.
There is a need for a tool which allows language designers to define multi-level modelling language hierarchies, *i.e.*, to extend the linguistic dimension. Current tools either fail to distinguish clearly between the linguistic and physical (representational) type of model elements, or do not allow such extensions at all.

3 The Modelverse: Overview

In this section, we describe the architectural choices for the Modelverse, and how it supports multi-level modelling hierarchies. Languages are central concepts in the Modelverse: we explain how a language is modelled by a type model, and how we consistently adhere to the strict metamodelling approach, where each element of a model is an instance of an element in a type model, as well as the deep instantiation and deep characterization principles.

3.1 Architecure of the Modelverse

The Modelverse is a repository or database of models. The Modelverse stores any modelling artefact, including, but not limited to, type models, concrete syntax models, and rule-based model transformations. It is accessible through an interface, which exposes an Application Programming Interface (API). This API includes methods for Create, Read,

Update and Delete (CRUD) operations, as well as conformance checking, and the ability to execute models (such as constraint or action code). The API ensures a uniform, standardized access to the Modelverse, capturing all allowed operations. It will be referred to as the Modelverse Kernel (MvK) from now on. A user interacts with the Modelverse through the API exposed by the Modelverse. This user needs not be a human interacting through code with the API of the MvK: it can be a front-end, allowing a more user-friendly use of the Modelverse. A few examples of front-ends include a visual front-end, such as AToMPM [8], a human-usable textual notation, or any (formalism-specific) simulator, that interacts with the Modelverse to simulate the model.
Modelling languages are defined by a linguistic type model, which defines the concepts of the language and the valid ways in which they can be instantiated. A modeller, when performing a CRUD operation, always has to specify which linguistic type the element is an instance of. To make this possible, the Modelverse includes a number of built-in, predefined, type models used for modelling language engineering, model transformation, metamodelling, and model management.

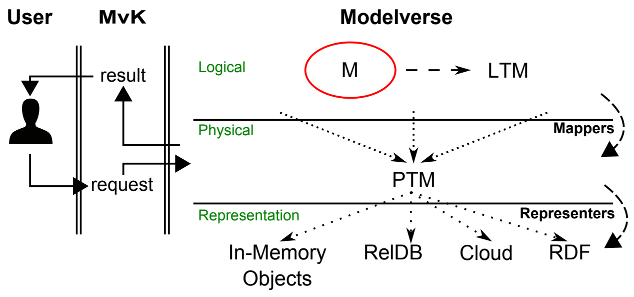


Fig. 1. The framework on which the development of the Modelverse is based.

To introduce the architecture of the Modelverse, Figure 1 shows the framework on which its development is based. There are two orthogonal dimensions: the logical and the physical, introduced in [4]. The logical level encompasses linguistic and ontological classification, but for the remainder of the paper, we only consider linguistic classification. In the figure, the central entity is a model *M*. It conforms linguistically to a linguistic type model *LTM*. In the physical dimension, one type model is defined. It defines the concepts the Modelverse needs to know about in order to function: clabjects, attributes, associations, primitive data types, action language, and so forth. It acts both as a type model (to which *all* models in the Modelverse conform), and an interface definition for the implementation, which defines the representation on a physical medium, of those structures. Although the Modelverse can be seen as a database of models, the *representation* of those models on physical media, such as a relational database or in-memory objects, is not known to the user. This knowledge is not necessary because of the uniform access through the MvK, as model management operations are performed on instances of the physical type model. The representation of physical type

model elements onto physical media is catered for by *representers*, one for each physical medium. For example, the default representer maps physical type model elements onto in-memory Python objects. Alternative representers would do the same for relational databases, RDF triple stores, and others.
With this level of indirection, we make sure that this representation only has to be defined once: we know how to represent physical type model elements, which means we can represent any linguistic concept in the Modelverse, as all elements by construction conform to the physical type model. To ensure this conformance relation is maintained at all times, *physical mappers* map linguistic elements onto physical elements. In these mappers, it is possible for a language engineer to encode custom *instantiation policies*. For the built-in formalisms, such as a *Class Diagrams* formalism, these policies are predefined.
Any element in the logical dimension can take the role of the model *M* — indeed, *everything* in the Modelverse is a model, and all models have a type model. This has certain benefits, one being the support for explicitly modelled model transformations [9] — often called the heart and soul of MDE [10]. If every model conforms to exactly one linguistic type model, it is possible to generate (automatically) transformation languages for each language, and transform every model using the same technique, which means higher-order transformations are enabled by default. A second important advantage of this approach is the ability to define a semantic mapping function. This function maps language elements onto concepts in a domain with known semantics, for example Petrinets [11]. This mapping needs to be unique, which can only be achieved when it is defined in the linguistic dimension — ontologies, for example, classify multiple models in different languages, and as such, have no unique semantic mapping.

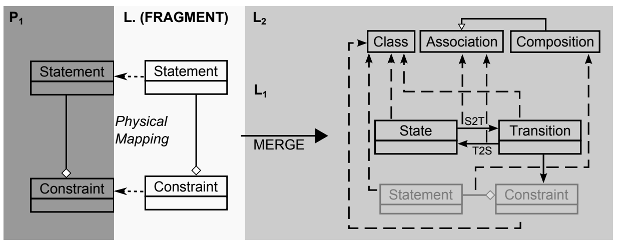


Fig. 2. An example of a modelling language fragment.

3.2 Modular Language Design

In recent literature, reuse and abstraction for modelling languages has received some attention. In [12], the authors explore a template-based approach for designing languages with similar characteristics. A language designer, however, might also want to add existing capabilities to a modelling language. An example is the action code used in Statechart transitions: while it is possible to define an action language from scratch and

include it in the type model of the Statechart language, chances are that an already existing formalism also has this feature.
A language designer needs to be able to reuse these modelling language concepts: ultimately, a tool can provide a library of reusable language *fragments*, which can be *merged* into the linguistic type model of any modelling language. Languages are more than abstract syntax alone, which describes the concepts of the language and the valid ways in which they can be combined. A language or formalism has one or more concrete syntax definition(s), a mapping of its concepts to the physical type model, a definition of its semantics, and a definition of the behaviour of its modelling environment. A modelling language fragment has to include these concepts, such that they can be reused when merging the fragment into a language definition. For now, we focus on the linguistic definition of the fragment, as well as the mapping of its concepts to the physical type model.
Figure 2 shows an example of such a fragment. The fragment contains the definition of a constraint, which contains a number of statements. These linguistic concepts are mapped onto physical entities that are predefined in the Modelverse. The most obvious mapping is to the concepts shown in the figure, as they have the expected semantics. Merging the fragment results in the *Statement* and *Constraint* concepts to be added to the Statecharts type model. Flexibility is achieved by leaving the potency value of a fragment undefined (denoted by *L.*), as well as the linguistic type of its elements: they are specified when merging the fragment with the linguistic type model.
Using this mechanism allows a language designer to modularly build modelling languages, and reuse concepts that are already defined. We envision this approach as an answer to the observation that 1) some concepts or structures of modelling languages are naturally reusable, including their concrete syntax and physical mapping, and 2) these concepts need to be linguistically available at the level of the type model, *i.e.*, instead of being part of all type models by default. In Section 4, we demonstrate how fragments may be used, by showing how, in a multi-level modelling hierarchy, new attributes can be introduced at any level.

4 Case Study

In this section, we model a multi-level language hierarchy in the Modelverse. The purpose of this section is to show the capabilities of the Modelverse, and the MvK, with respect to multi-level modelling. We focus on a linguistic hierarchy, with deep instantiation and deep characterization using potency.

4.1 A Visual Notation

In Figure 3, the example modelling hierarchy is visually represented. A model is represented by a coloured rectangle, where higher-level models are darker. All nodes of a model are represented by a rectangle (there is no language-specific concrete syntax). The name of an abstract class is shown in italics. Potency, if declared, is shown after the '@' sign. An

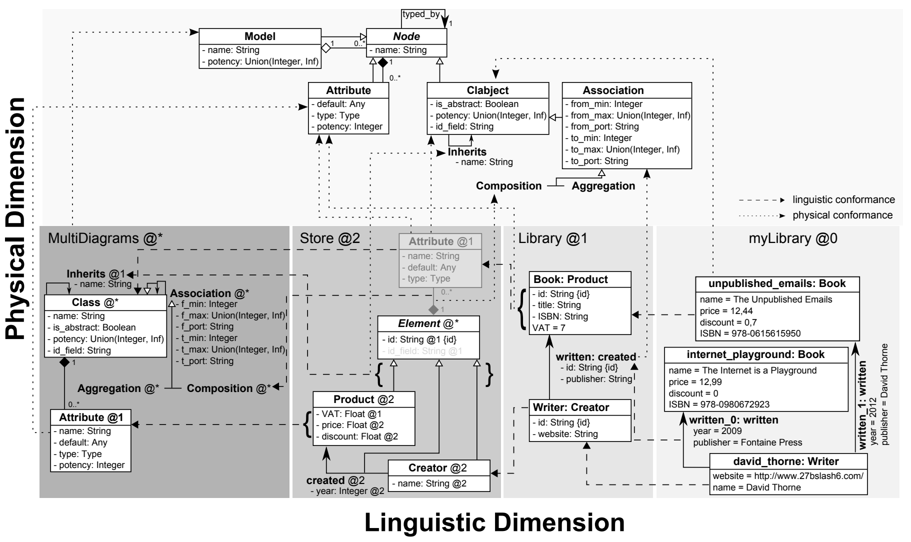


Fig. 3. The example modelling hierarchy.

undefined potency (meaning the element can be instantiated an undefined number of times) is denoted by an asterisk. The default potency value for Clabjects and models is undefined, for attributes it is 1. Conformance relations are shown for linguistic and physical conformance. Only a subset of the relations is shown, to avoid cluttering the figure.

4.2 Physical Representation

The physical dimension in the figure contains a relevant part of the physical type model, which is the built-in (static) type model of the Modelverse. Each element in the linguistic dimension is *mapped* onto these concepts.

At the top of the hierarchy, a language called *MultiDiagrams* is modelled. It can model classes, attributes, and associations, and allows potency to be specified. The *Attribute* class is special: the mapper for the *MultiDiagrams* formalism specifies that it is mapped onto the physical *Attribute* class, instead of the physical *Clabject* class, which is the default. Note also the *id_field* attribute of *Class*. The Modelverse uses a dot-separated notation to refer to elements, and elements are referred to by their name (for example, to refer to the *Class* concept, one would use *MultiDiagrams.Class*). The *id_field* attributes is used one level down to identify for each instance of *Class* what the identifying attribute will be. The physical mapper then maps this attribute onto the physical *name* attribute. In our notation, identifying fields are followed with {*id*}.

```
package MyFormalisms:
 Model:
  name = 'Store'
  potency = 2

  Class:
   name = 'Element'
   potency = *
   is_abstract = True
   id_field = 'id'

   Attribute:
    name = 'id'
    type = String

   Attribute:
    name = 'id_field'
    type = String

  Class:
   name = 'Product'

   Attribute:
    name = 'VAT'
    type = Float
    potency = 1

   Attribute:
    name = 'price'
    type = Float
```

```
   Attribute:
    name = 'discount'
    type = Float

  Inherits:
   name = 'product_i_element'
   from_clabject = 'Product'
   to_clabject = 'Element'

  Class:
   name = 'Creator'

   Attribute:
    name = 'name'
    type = String

  Inherits:
   name = 'creator_i_element'
   from_clabject = 'Creator'
   to_clabject = 'Element'

  Association:
   name = 'created'

   Attribute:
    name = 'year'
    type = Integer

  Inherits:
   name = 'created_i_element'
   from_clabject = 'created'
   to_clabject = 'Element'
```

Listing 1. Textual notation for Model Store

```
package MyFormalisms:
 Store:
  name =  'Library'
  potency = 1

  Product:
   name = 'Book'
   id_field = 'id'
   VAT = 7

   Attribute:
    name = 'id'
    type = String

   Attribute:
    name = 'title'
    type = String

   Attribute:
    name = 'ISBN'
    type = String
```

```
  Creator:
   name = 'Writer'
   id_field = 'id'

   Attribute:
    name = 'id'
    type = String

   Attribute:
    name = 'website'
    type = String

  created:
   name = 'written'
   id_field = 'id'

   Attribute:
    name = 'id'
    type = String

   Attribute:
    name = 'publisher'
    type = String
```

Listing 2. Textual notation for Store Library

```
package MyFormalisms:
 Library:
  name =  'myLibrary'
  potency = 0

  Book:
   id = 'internet_playground'
   name = 'The Internet is a Playground'
   price = 12.99
   discount = 0
   ISBN = '978-0980672923'

  Writer:
   id = 'david_thorne'
   name = 'David Thorne'
   website = 'http://www.27bslash6.com/'

  Book:
   id = 'unpublished_emails'
   name = 'The Unpublished Emails'
   price = 12.44
   discount = 0.7
   ISBN = '978-0615615950'
```

Listing 3. Textual notation for myLibrary

On the level below, a type model with potency 2 models a *Store* language. A store consists of *Products*, and are created by *Creators*. Certain attributes of the classes need to be defined on the level below (such as *VAT*), while others are left to be defined two levels down (such as *name*). This shows the deep characterization capabilities of the Modelverse.
While some attributes are already declared for the elements of the *Store* attribute, it might be that modellers making store instances want to add other attributes to *their* instances. While this seems reasonable, and is shown in the *Library* formalism below, this feature has to be modelled explicitly in the *Store* formalism, and, more importantly, proper semantics have to be given to it in its physical mapper. As can be seen from

the figure, the linguistic type of *Attribute* in the *Store* formalism is *MultiDiagrams.Class*. Its physical mapper, though, maps it onto the physical *Attribute* class. In this way, instances created of this class are seen as attributes of physical *Attributes* by the Modelverse, but at the same time, they are seen as linguistic instances of *Class*, which means they can, for example, be matched as such in transformation rules. This is an example of a language fragment, as explained in Section 3.2, although the merging is currently done manually. Another example is the *id_field* attribute.
The *Library* formalism has one specific product: books, which are created by writers. Any attributes introduced at this level have potency 1. Both classes have an attribute *id* as their identifying attribute (meaning that it is mapped onto the physical *name* attribute). The instances of *Library* are level-0 models, meaning they are fully defined: all attributes have values, and their potency level has decreased to 0.

4.3 A Textual Representation: the HUTN

Finally, we show in Listings 1, 2, and 3 what the example shown in Figure 3 looks like in the HUTN syntax, which is a possible front-end for the Modelverse. In Listing 1, the keywords 'package', 'Model', 'Class', 'Attribute', 'Inherits' and 'Association' are highlighted. Only 'package' is a reserved word in the HUTN, while the others are defined using an alias mechanism which refers to model elements belonging to the above mentioned 'MultiDiagrams' protected formalism. The model 'Store' defined in Listing 1 is a type model for the 'Library' model shown in Listing 2, which in turn is used as a type model for the 'myLibrary' model shown in Listing 3.
The verbosity of the HUTN is due to a rather conscious design choice for the concrete syntax: the main objective is to enable the modellers to seamlessly navigate between different modelling levels while maintaining the same concrete syntax look-and-feel.

5 Conclusions and Future Work

In this paper, we have introduced the Modelverse, a metamodelling framework which allows for multi-level linguistic modelling hierarchies, as well as modular language design. We have demonstrated its use with a representative case: a four-level modelling hierarchy for stores. We showed its use as a language workbench, allowing the definition of multi-level linguistic modelling hierarchies, which to our best knowledge, no other tool is capable of. We demonstrated the concept of "physical mappers" which allows to define custom instantiation policies. This allows to replace, for example, the top-level linguistic type model, which in most tools is static. In the example case we defined a type model for multi-level language hierarchies called MultiDiagrams, but it would be possible to replace this by a type model for modelling two-level language hierarchies.
In the future, we will continue enhancing the capabilities of the Modelverse, including:

– Introducing the ability to define ontological type models, and an ontological conformance check, which checks properties in the semantic domain of the model.
– Continue the work on language fragments, including the automation of the merge operation, and a definition of a library of fragments.
– The addition of representations on different physical media, to scale the Modelverse to a distributed environment.

Acknowledgement. This work was partly funded with a grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

References

1. "MOF 2.0." http://www.omg.org/spec/MOF/2.0/, 2006.
2. C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," *Software and Systems Modeling*, vol. 7, pp. 345–359, 2008.
3. C. Atkinson and T. Kühne, "Concepts for comparing modeling tool architectures," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3713 LNCS, pp. 398–413, 2005.
4. C. Atkinson and T. Kühne, "Rearchitecting the UML infrastructure," *ACM Transactions on Modeling and Computer Simulation*, vol. 12, pp. 290–321, 2002.
5. C. Atkinson and T. Kühne, "The Essence of Multilevel Metamodeling," *01 Proceedings of the 4th International Conference on The Unified Modeling Language Modeling Languages Concepts and Tools*, vol. 2185, pp. 19–33, 2001.
6. J. D. Lara and E. Guerra, "Deep Meta-Modelling with MetaDepth," in *Proceedings of TOOLS, Lecture Notes in Computes Science vol. 6141*, pp. 1–20, Springer, 20120.
7. C. Atkinson and R. Gerbig, "Melanie: Multi-level modeling and ontology engineering environment," in *Proceedings of the 2Nd International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, (New York, NY, USA), pp. 7:1–7:2, ACM, 2012.
8. E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, "AToMPM: A web-based modeling environment," in *MODELS'13 Demonstrations*, 2013.
9. T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Explicit Transformation Modeling," in *MoDELS Workshops, Lecture Notes in Computer Science vol. 6002*, pp. 240–255, Springer, 2009.
10. S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *Software, IEEE*, vol. 20, no. 5, pp. 42–45, 2003.
11. T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
12. J. D. Lara and E. Guerra, "Generic Meta-modelling with Concepts , Templates and Mixin Layers," in *Proceedings of MODELS, Lecture Notes in Computer Science vol. 6394*, pp. 16–30, 2010.

Multilevel Modelling for Interoperability

Andreas Jordan, Wolfgang Mayer, and Markus Stumptner

University of South Australia, Australia,
{andreas.jordan}@mymail.unisa.edu.au,
{wolfgang.mayer,mst}@cs.unisa.edu.au

Abstract. Model-driven approaches to establishing interoperability between information systems have recently embraced meta-modelling frameworks spanning multiple levels. However, no consensus has yet been established as to which techniques adequately support situations where heterogeneous domain-specific models must be linked within a common modelling approach. We introduce modelling primitives that support the multilevel modelling paradigm for information integration in heterogeneous information systems. We extend standard specialisation and instantiation mechanisms to enable the propagation of semantic and schema information across model levels and compare our approach using a suite of criteria to show that our approach improves modularity, redundancy, query complexity, and level stratification.

1 Introduction

The core idea of conceptual modelling since the ER-Model [1] is the notion of defining a particular language that can be used to effectively construct concise and clear domain models. Carrying over conceptual model characteristics into object-oriented software models (such as UML), led to rich design methods including meta-modelling frameworks: models that can define the languages used to produce the actual conceptual and design models for information systems. For example, Atkinson and Kühne [2] analyse the way in which design/product relationships cause inconsistencies with the fixed meta-model hierarchy of the UML standard. However, advanced application domains, such as those requiring the in-depth modelling of products, e.g. engineering standards, online catalogues, reference data libraries, continue to present a particular challenge to meta-modelling frameworks. This is further exacerbated when investigating appropriate multilevel modelling abstractions suitable for applications in an interoperability setting.

In this paper we discuss the multi-level modelling extensions that we believe will enable automated, model-driven transformation of data between two of the major data standards in the Oil & Gas industry [3,4]. A key concern is the notion that the same component in a real system can be subject to multiple classifications. This is to allow a multi-dimensional view of the modelled data, from recording specific technical solutions and constraints to be used for technical access as well as to serve business/ERP requirements.

The contribution of this paper is a set of modelling extensions to overcome limitations of existing approaches, such as the "heterogeneous level" issue of [5].

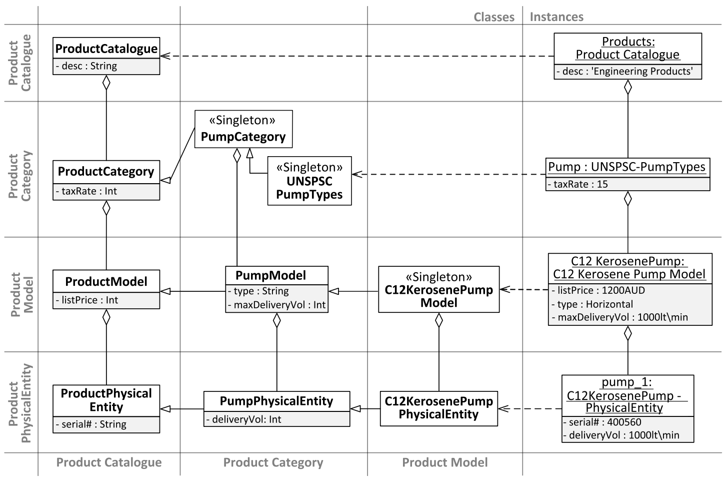


Fig. 1: Product catalogue modelled in plain UML, using generalisation, instantiation and aggregation (adapted from [5])

2 Motivating Example and Multi-Level Modelling

There are several important aspects and complexities to the domain that need to be modelled. We identify three levels of data: business level, specification level, and physical entity level. Firstly, both designs and the physical entities of a product catalogue must be represented and have their own life-cycles which forms the physical entity level. Secondly, a second level of classification provides the specifications of the physical entities. Thirdly, a third level of classification, or categorisation, must be imposed on the designs. This third level of classification also consists of complex taxonomies relevant from the business/ERP perspective. The modelling approaches taken by established related standards demand a flexible approach to support mappings and model transformations between them.

A UML-based approach to modelling this situation is shown in Figure 1, in which ProductCatalogue, ProductCategory, ProductModel, and ProductPhysicalEntity are modelled as an abstraction hierarchy using aggregation. Specialisation is used to distinguish categories (i.e. business classifications), models (i.e. designs), and physical entities of pumps and motors.

Modelling complex domains using UML suffers from what Henderson-Sellers et al. describe as *enactment* (see [6]). Moreover, the model in Figure 1 contains a number of redundant classes as discussed in [5,7]; the misuse of the aggregation relationship to represent a membership and/or classification relation; and the result that the physical entities are not intuitively instances of their product models, but rather are *part-of* their models. Basically, a domain model that seems to naturally have multiple levels of classification is forced into a 2-level modelling framework so that all aspects of the product data exist at the instance level. In an attempt to resolve such issues, multi-level modelling (MLM) techniques have been developed.

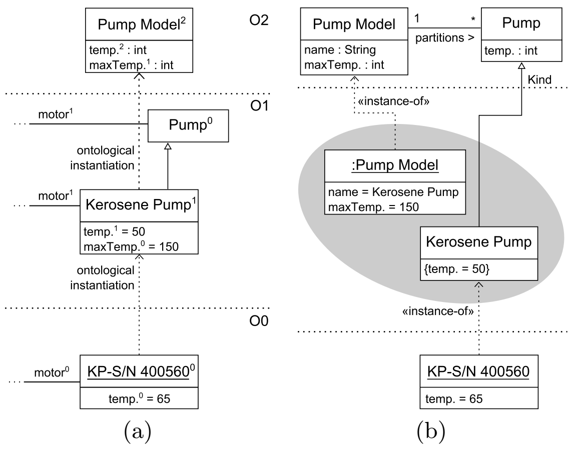


Fig. 2: Deep Instantiation (a) and Power Type (b) models of a product catalogue

The concept of *potency* [7] was originally introduced for *Deep Instantiation (DI)* to support the transfer of information across more than one level of instantiation. DI introduced the concept of *ontological* instantiation, which is a domain specific instantiation relationship (different from standard *linguistic* instantiation in UML meta-modelling).

Using DI techniques, which works within the boundaries of strict meta-modelling, invariably results in relationships (other than instantiation) crossing level boundaries, which is not permitted under *strict* meta-modelling. For example in Figure 2a, if the attributes $temp^2$ and $maxTemp^1$ were modelled as associations to values of the type DegreeCelsius (rather than just integers that must be interpreted as such), no matter at what level DegreeCelsius is placed it would result in an association crossing a level boundary at some point [8]. Moreover, if an additional level of instantiation was introduced into a DI-based model, global changes to the potency values of all concepts in the model (as opposed to just the subhierarchy directly affected) are required.

The concept of *power types* [9] has also been applied in the context of MLM frameworks [10]. Basically, for a power type t of another type u, the instances of t must be subtypes of u. In contrast to potency, power types do not necessarily provide deep instantiation semantics; they provide semantics closer to the conceptual situation being represented by clearly identifying the concepts involved, their relationships, and their properties.

For example, Figure 2b shows one instance of the Power Type pattern, where type and instance facets of a concept are depicted within the ellipse. The power type Pump Model for the type Pump is displayed; the concept ProductCategory would be represented as cascading uses of the power type pattern.

M-objects (multi-level objects) and m-relationships (multi-level relationships) were introduced in [11] along with the *concretization* relation which stratifies objects and relationships into multiple levels of abstraction. The m-objects

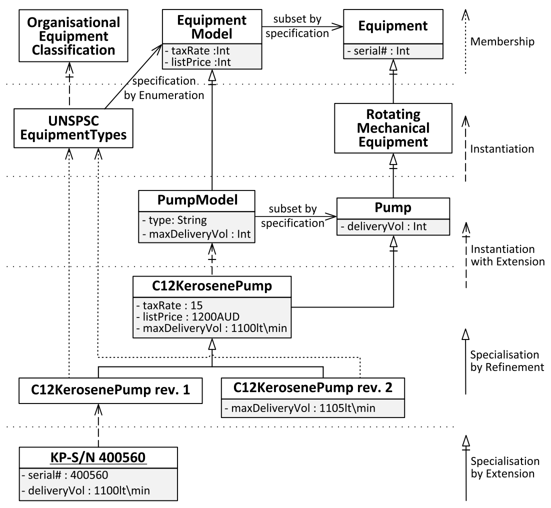


Fig. 3: Product catalogue example modelled in our framework

technique allows for the encapsulation of the different levels that relate to a specific domain concept in the m-object representing that concept. Furthermore, it combines the different abstraction hierarchies for specialisation, instantiation, and aggregation into a single concretization hierarchy. As such, the example situation would be modelled with a top-level concept ProductCatalogue containing the definition of its levels of abstraction: category, model, and physical entity. The lower levels would then include objects for the different PumpCategories, PumpModels, and PhysicalPumps, respectively.

While the m-objects technique produces concise models with a minimum number of relations, they hide complex semantics. This can lead to difficulties in interpreting the models as the concretization relation between two m-objects (or two m-relationships) must be interpreted in a multi-faceted way.

3 Modelling Extensions to Support Multilevel Modelling

Recently, the notion that every part of an object model is an object (embraced in certain OO programming and conceptual modelling [2] approaches) has regained popularity [8,6]. We follow this approach and treat all model elements as objects that can include both type and instance facets.

The MLM approaches summarised in the previous section place the emphasis on a clear and consistent layering of the levels in the model. However, when applied to ontological instantiation in information modelling for systems interoperability, the level construct actually becomes an artefact of the modelling outcome. Domain engineers do not think in levels; they work in terms of semantic relationships, and there can be arbitrary many levels for each of them. Capturing these in terms of potency is useful if the originating models are available for reorganisation according to the strictness criterion and if solid estimates exist on the expected number of levels that future developers may want to add in a particular domain. In the interoperability space this is generally not the case. We will now examine the specific relationships used in our framework.

3.1 Instantiation vs. specialisation relationship

In MLM, the existence of two basic relationship types for increasing specificity is commonly assumed: (1) instantiation which can be broadly defined by the conformance of instances to types in which specificity is increased by assigning distinct values to attributes (if they exist); and (2) specialisation (or generalisation), which makes a concept more specific by including finer distinction (based on the Liskov Substitution Principle (LSP) [12]). Intuitively our modelling extensions are compatible with the intuition of the LSP, however formal proof to verify such a claim is left for future work.

Viewed in terms of increasing specificity it becomes obvious that what we have generally referred to as specificity, and what MLM approaches measure by potency, actually represents *two different conceptual relations*: One is the reference of a specification to the specified item. This captures the powertype aspect of the relationship and is also at the core of the *materialization* relationship in the conceptual modelling literature. The other is the definition of the vocabulary used in the specification. It should be clear that this distinction is of fundamental importance for interoperability mappings, since they rely on being able to treat the specification of a system separately from its runtime state.

We characterise specialisation relationships along the lines of [13] to distinguish a relationship that *extends* a class (by adding attributes, associations, or behaviour) from one that *refines* a class (by adding granularity to the description). We call a specialisation relationship that extends the parent class a *Specialisation by Extension* (*SbE*) and adopt standard monotonic specialisation semantics. As such it can include but does not necessitate *refinement*. Most importantly, this form of specialisation introduces a new model level (as opposed to standard meta-modelling and MLM techniques where modelling levels are fixed a priori).

In contrast, a specialisation relationship that only refines the parent class is called *Specialisation by Refinement* (*SpecR*), which allows the introduction of subtypes that restrict the domain of the specialised class (e.g., by restricting the domains of properties and associations, or adding domain constraints on properties) but without introducing additional model levels. This allows for an arbitrary number of subtypes that simply refine the level of granularity.

We characterise instantiation as either *Instantiation with Extension* ($InstX$) or *Standard Instantiation* ($InstN$). Both forms of instantiation introduce additional model levels; however, standard instantiation means that all attributes of the type being instantiated must be assigned a value from their domain, while $InstX$ allows for additional attributes, behaviour, etc. to be added to the concept that can then be instantiated or inherited further to lower model levels.

The *Subset by Specification* (*SbS*) relationship represents the existence of a class of specification construct that identifies particular subtypes of another type. The specification (for example EquipmentModel) exists at the same level as the type it refers to, because the specification can only refer to properties of that type (and not to properties of individual subtypes). It is, however, possible to define subtypes of this type of specification to reference particular properties (so EquipmentModel can be specialised to PumpModel which can refer to properties of Pumps). Together with $InstX$, this relationship can be used to construct the powertype pattern [9]. In Figure 3, PumpModel could be modelled as the powertype of EquipmentModel since it specialises EquipmentModel and the instance of PumpModel, i.e. C12KerosenePump, is an indirect subtype (by extension) of Equipment.

Different to UML associations, most conceptual models permit general associations that represent domain specific relationships. We identify two particular such associations. The first we call $Member$. In contrast to the instantiation relation which it otherwise resembles, *member* does not have any constraints on the assignment of values to attributes as it is purely a basic set membership relation. However, this does not preclude the specification of membership criteria, or constraints, for allowing or disallowing the possible members of a set. $Member$ does require the existence of a "primary" instantiation relation. The second such association is *Specification by Enumeration SbE*, which represents a relationship between concepts A and B that describes how the extensions of the sets of entities that they represent are related. Specifically, it means that the *members* of A are *instances* of B. These relations permit us to emulate multiple inheritance by establishing statements about categories, without defining the attributes or associations of the included object or the category itself.

Below we present the formal properties of our model. The first argument denotes the instance or specialised concept, whereas the second argument represents the type or general concept. Relations $Member$ and *SbE* are jointly used to specify a classification scheme that subdivides the instances of a concept. Model elements are organised in levels numbered such that the type-level number is one less than the instance-level. Function *level* returns the level number of each element. Each element is described by a set of typed attributes, some of which may have assigned values, and a constraint expression stating necessary properties of the object's instances. Function *attr* maps each object to a set of attribute names, function *type* associates a data type to each object-attribute pair, and partial function *val* returns the value associated with a given object-attribute pair (if one has been assigned). We implicitly assume that primitive data types and their possible values are implicitly modelled as concepts and instances, respectively. Function *desc* maps each concept to a constraint expression capturing the properties that all instances of the object must satisfy, and *names* returns the attribute labels used in said description.

Domains:

O	Set of objects	L	Set of attribute labels
$\mathbb{N}$	Natural numbers	$\mathcal{S}$	Constraint language over attributes in L

Functions:

$level:$	$O \mapsto \mathbb{N}$	The level at which an object is defined (zero is top level)
$attr:$	$O \mapsto 2^L$	The set of attribute labels for an object
$type:$	$O \times L \mapsto O$	The type of an attribute of an object
$val:$	$O \times L \mapsto O$	The value of an attribute of an object
$desc:$	$O \mapsto \mathcal{S}$	Constraint expression instances of an object must satisfy
$names:$	$\mathcal{S} \mapsto 2^L$	The attribute labels used in a constraint expression

Relations:

InstN	$\subseteq O \times O$	$InstN(x,c)$: x is an instance of c
InstX	$\subseteq O \times O$	$InstX(x,c)$: x is an instance-with-extension of c
SpecR	$\subseteq O \times O$	$SpecR(c,c')$: c is a specialisation-by-refinement of c'
SpecX	$\subseteq O \times O$	$SpecX(c,c')$: c is a specialisation-by-extension of c'
Member	$\subseteq O \times O$	$Member(x,c)$: x is in a *Member* relation with c
SbE	$\subseteq O \times O$	$SbE(c,c')$: c is a *Specification-by-Enumeration* of c'
SbS	$\subseteq O \times O$	$SbS(c,c')$: c is a *Subset-by-Specification* of c'

We use (ρ^*) ρ^+ to denote the (reflexive-)transitive closure of relation ρ.

Definitions

The *Spec* relation generalises the two forms of specialisation
$Spec(c,c') \leftrightarrow (SpecR(c,c') \vee SpecX(c,c'))$

An object is a leaf iff it has no specialisations $Leaf(c) \leftrightarrow \nexists x : Spec(x,c)$

The *Inst* relation generalises the two forms of instantiation
$Inst(x,c) \leftrightarrow InstN(x,c) \vee InstX(x,c)$

Object x is a general instance of c iff it is instance of or specialises an instance of (a specialisation of) c
$GenInst(x,c) \leftrightarrow \exists x' \exists c' : Spec^*(x,x') \wedge Inst(x',c') \wedge Spec^*(c',c)$

Axioms for specialisation and instantiation

Inst, *Spec*, *Member* are jointly acyclic $(Inst \cup Spec \cup Member)^*(x,c) \rightarrow x \neq c$

InstN and *InstX*, *SpecR* and *SpecX* are mutually exclusive
$InstN(x,c) \rightarrow \nexists c' : InstX(x,c')$ $InstX(x,c) \rightarrow \nexists c' : InstN(x,c')$
$SpecR(x,c) \rightarrow \nexists c' : SpecX(x,c')$ $SpecX(x,c) \rightarrow \nexists c' : SpecR(x,c')$

Inst, *Spec* restricted to a unique parent object
$\rho(x,c) \wedge \rho(x,c') \rightarrow c = c'$ for $\rho \in \{Inst, Spec\}$

Only leaf objects can be instantiated $Inst(x,c) \rightarrow Leaf(c)$

Level consistent with *Inst*, *Spec*
$Inst(x,c) \rightarrow level(x) = level(c) + 1$
$SpecR(x,c) \rightarrow level(x) = level(c)$ $SpecX(x,c) \rightarrow level(c) \leq level(x)$

Axioms for schema consistency

Attribute type must be consistent with level order
$a \in attr(x) \wedge t = type(x,a) \rightarrow level(t) < level(x)$

SpecR does not change attribute set $SpecR(x,c) \rightarrow attr(c) = attr(x)$

SpecX extends attribute set $SpecX(x,c) \rightarrow attr(c) \subset attr(x)$

Spec may specialise attribute types
$Spec(x,c) \wedge a \in attr(x) \cap attr(c) \wedge t = type(x,a) \wedge t' = type(c,a) \rightarrow Spec^*(t,t')$

InstN does not change attribute set $InstN(x,c) \rightarrow attr(c) = attr(x)$

InstX extends attribute set $InstX(x,c) \rightarrow attr(c) \subset attr(x)$

Inst does not change attribute type
$Inst(x,c) \wedge a \in attr(x) \cap attr(c) \rightarrow type(x,a) = type(c,a)$

Inst instantiates all attributes of c
$Inst(x,c) \wedge a \in attr(c) \wedge t = type(x,a) \rightarrow \exists v : v = val(x,a) \wedge GenInst(v,t)$

Axioms for Member and Specification-by-Enumeration

Member relation must be consistent with level order
$Member(x,c) \rightarrow level(c) < level(x)$

Specification-by-Enumeration must be consistent with level order
$SbE(c,t) \rightarrow level(t) \leq level(c)$

Specification-by-Enumeration classifies general instance of the type
$SbE(c,t) \wedge Member(x,c) \rightarrow GenInst(x,t)$

Axioms for Subset-by-Specification

Subset-by-Specification must be consistent with level order
$SbS(c, c') \rightarrow level(c) = level(c')$
Instances of each subset specification must be a specialisation of the partitioned type
$SbS(c, c') \wedge Inst(x, c) \rightarrow Spec^{+}(x, c')$
The specification may refer only to the attributes of the partitioned type
$SbS(c, c') \wedge \phi = desc(c) \rightarrow names(\phi) \subseteq attr(c')$

Axioms for Descriptions
Constraints can use only attributes defined in its associated object
$\phi = desc(c) \rightarrow names(\phi) \subseteq attr(c)$
Constraints must respect the specialisation hierarchy
$Spec(c, c') \wedge \phi = desc(c) \wedge \phi' = desc(c') \rightarrow (\phi(x) \rightarrow \phi'(x))$
Instances of a object must satisfy its constraint
$GenInst(x, c) \wedge \phi = desc(c) \rightarrow \phi(x)$
Members in a *Member* relation must satisfy its classifier's constraint
$Member(x, c) \wedge \phi = desc(c) \rightarrow \phi(x)$

4 Comparison of MLM Techniques

A comparison of MLM approaches was performed in [5] based on a number of criteria from the perspective of reducing accidental complexity; that is, mismatches between what is being modelled and the modelling formalism being used. These criteria are: (1) Compactness, (2) Query Flexibility, (3) Heterogeneous Level-Hierarchies, and (4) Multiple Relationship-Abstractions. These criteria are important for modelling the domain in a concise, flexible, and simpler way. However, they do not cover certain aspects of particular importance to our domain and application in the OGI Pilot. We consider two additional criteria:

(a) ***Locality of Attributes & Relationships*** refers to what model elements attributes and relationships are defined on. Attributes/relationships are defined *locally* if they are defined on the model elements closest to where they are used. For example, an attribute relevant to product designs should be situated on the concept ProductModel rather than a related concept such as Product. This is in contrast to the modularity aspect, which attempts to minimise the different locations at which attributes and associations are located; however, it is particularly important in terms of interoperability as the different domain models exhibit different modelling approaches and scope, and attributes and associations may not have been grouped together consistently across the information system ecosystems. Having a flexible framework that can handle such a situation elegantly is important.
(b) ***Clarity of Relations' Semantics*** is concerned with whether the relations of the approach have clearly delineated semantics from other relations, or if they combine the semantics of multiple, commonly understood, relations together. For example, while a relationship that combines both the semantics of specialisation and instantiation may simplify the graphical representation of the model, the confounding of multiple relations in one could cause difficulties for constructing model transformations. Moreover, a number of issues can arise if the distinctions between relations are de-emphasised. Weakening the intrinsic differences between established relations comes at a significant cost such as "*sanity checks* regarding the integrity of metamodelling hierarchies that otherwise would not exist" [14].

In applying the criteria to our approach we conclude that it:

– is modular as it treats the class and object facets of a concept together allowing the specification of information regarding a concept in one place;
– allows redundancy-free modelling by using a range of relations that include various attribute propagation, inheritance, and assignment semantics;
– supports query flexibility through the different relationships and concepts in a domain model, e.g. different pump models can be accessed by retrieving the *instances* of PumpModel, the models in a particular category can be retrieved by accessing the *members* of the desired PumpCategory, and the physical pumps can be accessed through the instances of Pump;
– allows heterogeneous level-hierarchies through its flexible and dynamic nature of level stratification;
– supports specialisation and instantiation of domain and range of relationships;

While it is a balancing act to not produce too many relations, many of the relations in our approach are special cases of well known relations, alleviating possible issues with understandability and adding information of finer granularity. Moreover, existing domain models could be analysed to identify such distinctions, improving the identification of model transformations for interoperability.

There are three potency-based approaches in the comparison, the latter two of which are newly added to the comparison: Deep Instantiation [7], MetaDepth [15], and Dual-Deep Instantiation (DDI) [8]. MetaDepth allows the specification of different *views* (similar to the modelling spaces proposal of [2]), which allow for heterogeneous level-hierarchies. DDI extends DI with explicit levels based on “sort” hierarchies and attributes and associations with *2* potencies (source and target) rather than the single potency of DI and MetaDepth. DDI supports query flexibility (i.e. descendents of an object at a particular (sort) level can be queried in ConceptBase), heterogeneous level-hierarchies and multiple relationship-abstractions.

All three potency-based techniques can support locality of attributes and relations; however, it comes at the cost of not taking advantage of potency, i.e. restricting the possible potencies to zero or one. Similarly, under this restriction, standard DI and MetaDepth have clear cut relations, while using higher potency starts to mix instantiation with specialisation semantics. Finally, DDI more strongly combines the instantiation and specialisation relations and, hence, does not support the last criteria at all.

Assessing the power type approach (both simple and extended) with respect to the new criteria reveals that it supports locally specified attributes and relations. Whether the approach supports clarity of relation semantics is unclear as it depends on whether or not the partitions relation is provided with instantiation semantics [10]. However, the power type approaches only partly support compactness and only the extended power type approach fully supports query-flexibility while the simple approach only provides partial support. In terms of relationship abstraction, both approaches require OCL to provide this support.

The application of the additional criteria to m-objects show that the clarity of relation semantics is low, due to the combination of the relations aggregation, specialisation, and instantiation. In addition, as the intention of the M-Objects technique is to encapsulate all of the attributes and relationships of a concept on a single m-object, it does not support locally specified attributes and relationships.

5 Conclusion

Effective exchange of information about processes and industrial plants, their design, construction, operation, and maintenance requires sophisticated information modelling and exchange mechanisms that enable the transfer of semantically meaningful information between a vast pool of heterogeneous information systems. This need increases with the growing tendency for direct interaction of information systems from the sensor level to corporate boardroom level. One way to address this challenge is to provide more powerful means of information handling, including the definition of proper conceptual models for industry standards and their use in semantic information management. In this paper we have described our modelling framework for large scale ecosystem handling and the extended relationship types that help to succinctly express data models across a heterogeneous information system ecosystem.

References

1. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.
2. C. Atkinson and T. Kühne. Processes and products in a multi-level metamodeling architecture. *IJSEKE*, 11:761–783, 2001.
3. ISO. *ISO 15926 – Part 2: Data Model.* 2003.
4. *Open Systems Architecture for Enterprise Application Integration.* MIMOSA, 2012.
5. B. Neumayr, M. Schrefl, and B. Thalheim. Modeling techniques for multi-level abstraction. In *The Evolution of Conceptual Modeling.* Springer LNCS 6520, 2011.
6. B. Henderson-Sellers, T. Clark, and C. Gonzalez-Perez. On the search for a level-agnostic modelling language. In *Proc. CAISE'13*, pages 240–255, 2013.
7. C. Atkinson and T. Kühne. The Essence of Multilevel Metamodeling. In *Proc. of UML 2001*, LNCS 2185, pages 19–33. Springer, 2001.
8. B. Neumayr, M. A. Jeusfeld, M. Schrefl, and C. Schütz. Dual deep instantiation and its ConceptBase implementation. In *Proc. CAISE '14*. Springer, 2014.
9. J. J. Odell. Power types. *JOOP*, 7:8–12, 1994.
10. C. Gonzalez-Perez and B. Henderson-Sellers. A powertype-based metamodelling framework. *Software & Systems Modeling*, 5(1):72–90, 2006.
11. B. Neumayr, K. Grün, and M. Schrefl. Multi-level domain modeling with m-objects and m-relationships. In *Proceedings APCCM '09*, pages 107–116, 2009.
12. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
13. M. Schrefl and M. Stumptner. Behavior consistent specialization of object life cycles. *ACM TOSEM*, 11(1):92–148, 2002.
14. T. Kühne. Contrasting classification with generalisation. In *Proceedings APCCM '09*, pages 71–78, Australia, 2009.
15. J. de Lara, E. Guerra, R. Cobos, and J. Moreno-Llorena. Extending deep metamodelling for practical model-driven engineering. *Computer*, 57(1):36–58, 2014.

Using Multi Level-Modeling Techniques for Managing Mapping Information

Samir Al-Hilank[2], Martin Jung[1,2],
Detlef Kips[1,2], Dirk Husemann[2], and Michael Philippsen[1]

[1] Friedrich-Alexander University Erlangen-Nürnberg (FAU),
Programming Systems Group, Martensstr. 3, 91058 Erlangen, Germany
`philippsen@cs.fau.de`
[2] develop group Basys GmbH, Am Weichselgarten 4, 91058 Erlangen, Germany
`alhilank|husemann|jung|kips@develop-group.de`

Abstract. Traditional modeling approaches support a limited set of instantiation levels (typically one for classes and another adjacent one for objects). Multi-level modeling approaches on the other hand have no such limit to the number of levels. As a consequence, an arbitrary number of levels may be used to define models, and the distinction between class and instance is redefined.
The paper summarizes the experience gained from applying multi-level modeling techniques to a real application from the domain of development process improvement (DPI). The underlying case study has been conducted in cooperation with a large automotive supplier. We discuss the pros and cons of using multi-level modeling techniques and propose areas that we think would benefit from further research.

1 Introduction

Although problem domains often have a natural structure that spans more than two logical levels, traditional modeling languages like, for example, MOF [20] or UML [21] are limited to only two layers. As a consequence, using UML or MOF to capture such problem domains leads to squeezing several logical levels into two. This in turn causes accidental complexity [12] that does not originate from the problem domain itself. Consequently, concepts like powertypes [17], potency [12], dual classification [11], etc. have been developed to allow for the definition of an arbitrary set of classification levels.

This paper reports on experience with defining and using deep models in an industrial case study. It is organized as follows: Sec. 2 presents the case study's problem statement. Sec. 3 describes our experiences with using deep modeling techniques. Sec. 4 discusses related work. We conclude with a list of research areas that could improve the applicability of multi-level modeling.

2 Problem Statement

Most companies working in the automotive domain need to use development processes (DP) that comply with requirements defined by quality standards (QS) like CMMI [14] or ISO 26262 [18]. Company specific standards or laws enforced by governments may pose additional requirements. Moreover, those companies often have to provide evidence that their DP complies with all requirements, for example, due to economic reasons (precondition of contracts) or legal considerations (product liability). Hence, there is a strong need to collect this evidence information in a systematic way.

The basic approach is straightforward: Just collect and later analyze mappings between QS requirements and elements of the DPs. Here is a typical example mapping that we illustrate at the bottom of Fig. 1:

- `Integrate SW Component` is an item (e.g., a Task) defined by the company's DP.
- `SW Integration and Testing` is a QS requirement (e.g., a description of a phase) that has to be implemented by elements of the company's DP.
- `Mapping` connects both elements with the following semantics: The requirement `SW Integration and Testing` is fulfilled by the element `Integrate SW Components`.

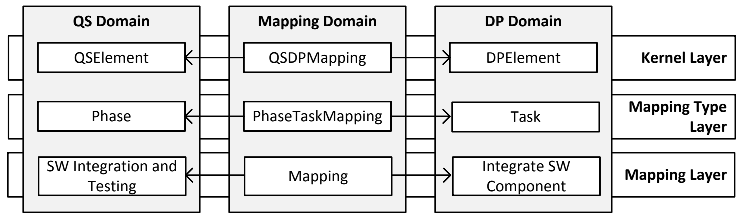


Fig. 1. Organization by responsibility (horizontal) and domain (vertical).

Note that these are items from three different domains. First, items from the **DP domain** need to be taken into account. There is no commonly used standard approach for describing DPs, but typical items found in DPs are Task, Step, Function or Template. In our case study, we use ARIS [6] to describe the company's DP. Second, we capture QS requirements in the **QS domain**. In our case, the company's process has to comply with three QSs simultaneously (CMMI, Automotive SPICE [13] and ISO 26262). Last, the **mapping domain** collects mappings between items of the DP and QS domain.

Fig. 1 also structures the problem into three orthogonal layers:

- The `Mapping Layer` collects mapping information.
- The `Mapping Type Layer` defines valid mappings.
- The `Kernel Layer` provides the foundation with respect to the terminology used within the DP, QS, and mapping domain.

The natural structure of the problem domain in Fig. 1 does not fit into two levels. Instead, Fig. 1 shows a scenario that is well suited for deep modeling techniques. Below we present such a deep model that we have implemented in an industrial application.

3 The Case Study

3.1 DeepML in a Nutshell

None of the research papers on concepts and features of deep modeling offers the flexibility and freedom needed for our case study (for details see Sec. 4). In general, existing approaches lack

- **freedom to combine concepts from several approaches.** There is no common set of deep modeling concepts, that would allow to cherry-pick suitable concepts from various approaches.
- **freedom to omit concepts.** Known approaches do not allow to omit concepts that are unnecessary for our solution.
- **freedom to experiment with new ideas.** None of the approaches we know of provides means to explore new ideas or to add new concepts.

As we needed this degree of freedom for our case study, we designed the DeepML language and infrastructure. DeepML supports the usual modeling concepts like inheritance, primitive data types, and enumerations. Furthermore, DeepML builds on the idea of dual classification and potency that is unified in the ontological classification architecture (OCA) [8]. OCA distinguishes between two kinds of instantiation relationships: The **linguistic instantiation** (lio) is a relation between elements of the model (that describes the core language capabilities, e.g., attributes, references, etc.) and elements of the problem domain. Linguistic instantiations therefore are not domain specific. Fig. 2 gives examples. On the other hand, **ontological instantiation** (oio) relations capture domain specific aspects. In Fig. 2, for example, `Book` is an ontological instance of `ProductType`.

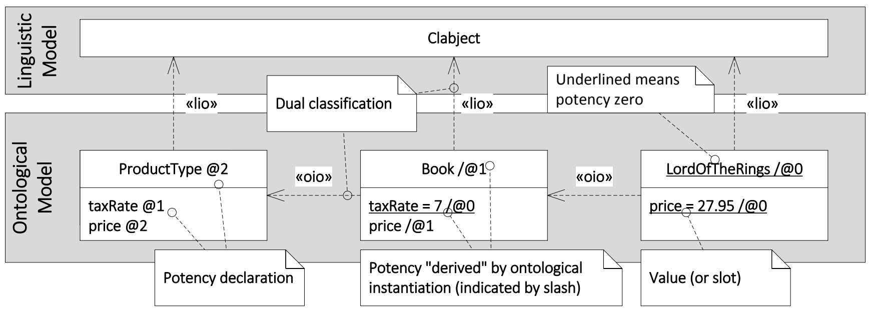


Fig. 2. DeepML in a Nutshell – Dual Classification.

Both elements (`Book`, `ProductType`) have a property called **potency** which is a positive integer value. Every ontological instantiation step decrements the potency by one. Model elements with a potency zero cannot be further instantiated. Consequently, the element `Book` has both an instance and a class facet. This is commonly called a **clabject** [10]. DeepML clabjects "derive" the potency in the same way as attributes. To express this inheritance, we use the "/" character as in `Book/@1`.

Due to space restrictions, instead of describing DeepML's language capabilities in full detail, the next sections highlight some of its key features. We also compare them to other deep modeling languages in Sec. 4.

3.2 Sample Mapping Scenario from the Case Study

As outlined in Sec. 2, collecting mappings is always done with a specific goal in mind. For example, in our case study, the DP needs to fulfil the requirements of a new and emerging QS (ISO 26262). However, the company's DP already fulfils a subset of the requirements as defined by the QS CMMI. One strategy to minimize the effort of gap analysis in such a situation is to map elements from both QSs onto each other. We consider those ISO 26262 requirements that we can map to CMMI requirements as already covered by the company's DP. ISO26262 requirements that cannot be mapped to CMMI necessitate further investigation.

The following subsections present parts of our Process Improvement and Quality Standard Harmonization Model (PIQSH-M), a deep model that underlies our mapping management application for the QS-DP-Mapping (Sec. 2) and QS-QS-Mapping (see below) scenario.

3.3 Ontological Containment

QSs are usually published as large text documents. These documents are typically organized with a specific ordering structure in mind. For example, each chapter at a specific level (e.g., "9.5.6 SW Integration and Testing") represents a phase. On its left side, Fig. 3 shows how to model the following clabjects for capturing QSs in DeepML:

– **QSDElement (Quality Standard Domain Element)**: Base clabject of all elements that belong to the QS domain.
– **QSTM (Quality Standard Type Model)**: Instances of this clabject are **top level** elements of models used for capturing the structure of one QS (e.g. entities, relations, etc.). An example is the clabject `ISO26262/@1`.
– **QSElement**: Instances of QSElements represent non top level elements of QSs (e.g., clabject `Phase/@1`).

To capture the relationships between elements of one QS, we introduce references. In DeepML, relationships among clabjects are expressed by references that are always unidirectional and owned by exactly one clabject. They are, thus,

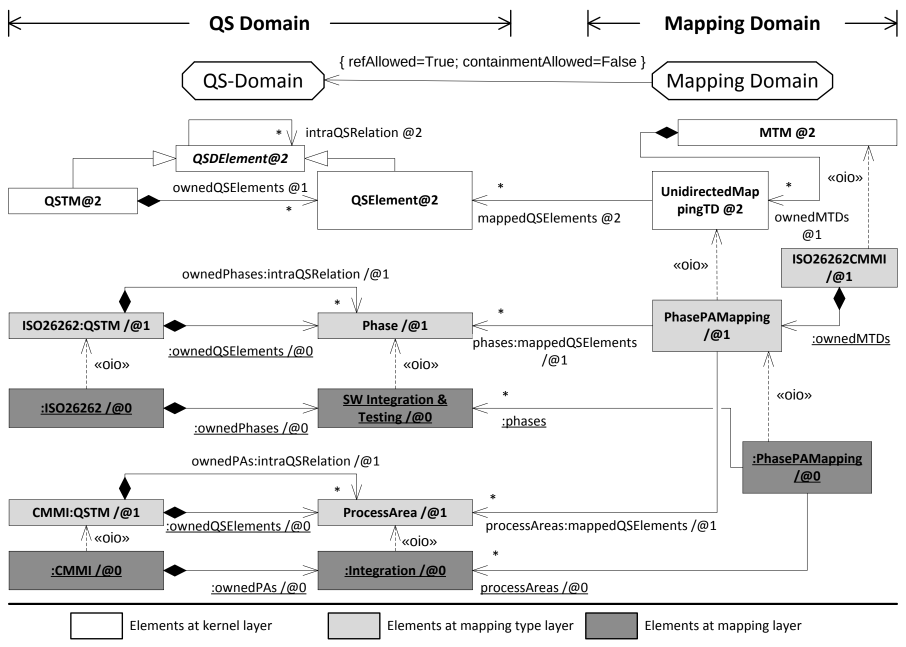


Fig. 3. An extract of the PIQSH-model.

more similar to references in MOF than to association classes in UML. References may participate in classification hierarchies in the same way as clabjects do. Consequently, the potency of a reference is either directly specified or derived from a classifying reference. For example, the reference `intraQSRelation@2` classifies the set of relationships (not values) between QS elements. Thus, instances of this reference, for example `ownedPhases:intraQSRelation/@1`, express that ISO 26262 is made up of a set of phases. A slot concept is used to store values and in DeepML underlined text is used as notation. For example, `:ownedPhases/@0` is a slot, and `SW Integration and Testing/@0` is owned by `:ISO26262/@0`.

A similar model structure captures mapping information: Instances of the clabject Mapping Type Model (`MTM`) are top level containers of models for specifying meaningful mappings. An example for such a mapping is `PhasePAMapping/@1` which links `Phases` (from ISO26262) and `ProcessAreas` (from CMMI). Concrete mappings can then be captured on the mapping layer (e.g., `:PhasePAMapping/@0`).

Although, references optionally may be containment references, most deep language specifications ignore this. Instead, DeepML adds an ontological containment concept because: Firstly, a containment is a common constraint that should be natively supported. Secondly, and more importantly, a containment hierarchy reflects the hierarchical structure that the domain expert had in mind. Finally, persistance layers typically use containments to find the one (or no) resource to store elements into. An example for a containment reference is ownedQSElements@1 (see Fig. 3). Given its definition, QSElements are part of exactly one QSTM.

3.4 Domain Stacks and Model Organization

One of the key requirements of the PIQSH-M is to support reuse. For example, other companies or organizational units may need to comply with a different set of QSs. The basic idea for addressing this problem is to develop a library of commonly used QSs that grows over time. Missing QSs can simply be transformed into deep models and added to that library for later reuse.

When managing such a library, it is important to define what kinds of models are part of it. Restricting and managing dependencies between models is equally important. We decided to use the matrix-like organization shown in Fig. 1 as a guideline because it separates model content with respect to the domain (QS, Mapping, DP) and to the role (kernel, mapping, and mapping type definition layer). For example, direct and indirect instances of QSTMs@2 (e.g., CMMI/@1 and :CMMI/@0, respectively) are models managed by the library.

As explained above, restrictions should apply to the set of permissible relations between models from different domains and layers. DeepML allows the definition of references without a classifying reference at any level. For example, we could define a direct reference between Phase and ProcessArea without defining the corresponding mapping type. However, such a direct reference would lead to a direct dependency between ISO26262/@1 and CMMI/@1 — thus breaking the intentional decoupling introduced by the mapping layer and leading to a pollution of model content.

To prevent pollution of this kind, the concept of **domain stacks** is added to DeepML, formalizing the separation into domains as illustrated in Fig. 1. Every clabject may be part of at most one domain stack. For example, all elements on the left side of Fig. 3 are members of the QS domain stack. References between elements within the same domain stack are allowed. References to elements of other domain stacks or to domainless elements are not allowed by default.

The situation is slightly different for members of the DP domain stack (right side of Fig. 3): They must refer to elements from other domain stacks, but must not contain them. To express these rules, a relationship between domain stacks is introduced that specifies which kind of references are allowed. An example is shown at the top of Fig. 3.

To summarize, by using top level containers, ontological containment and domain stacks, a clean separation of content is realized that spans multiple levels. However, a capability for managing mapping projects is still missing.

3.5 A Deep Model for DPI Project Management

With a deep model for organizing items of the QS and DP domains at hand, the practitioner also needs tool support to actually use this model for creating and managing mapping information. Let us now demonstrate how the DeepML model serves as a foundation of an application called PIQSH Support Center (PIQSH-SC) that covers the three main use cases of managing projects in the DPI domain:

(1) **Define Quality Standards**: Capture the structure and content of QSs.
(2) **Define Mapping Type Models**: Define the set of meaningful mappings between certain combinations of QSs (or QSs and DPs).
(3) **Collect Mapping Information**: Collect mapping information within a specific context (e.g. between QSs or between QSs and DPs).

Use case (1) is straightforward. As it is only concerned with creating models, we do not discuss it any further. Use case (2) involves at least three models, for example, two `QSTMs` (e.g., `CMMI/@1` and `ISO26262/@1`) and one `MTM` (e.g., `ISO26262CMMI/@1`) as shown in Fig. 3. The inter-model relationships can also be captured in DeepML as the PIQSH Project Management Model (PIQSH-PMM) and the clabject Model Type Definition Project (`MTDProject`). With this clabject, we string together all models in terms of an import relationship. The resulting model is shown on the left side of Fig. 4.

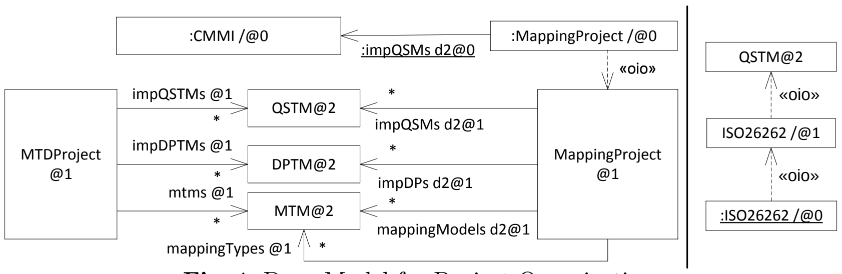


Fig. 4. Deep Model for Project Organization.

Use case (3) requires capturing mapping information. Within use case (2), `MTDProject` and the corresponding references had equal potency values, and both were reduced within an ontological instantiation step. In contrast, in use case (3) a project needs to store references to models on the mapping layer; that is, to instances of instances of `QSTMs` or `MTMSs`. Since references of this kind cannot be expressed with DeepML, we enhance references by the concept of distance.

The **distance** between two clabjects `A` and `B` is defined as the number of instantiation relationships that need to be traversed in order to reach A when starting from B. As an example, the distance between `QSTM@2` and `ISO26262/@0` is 2 (see right side of Fig. 4). The rule for assigning values to a reference depends on the distance: The distance between the reference's target type and the value to be assigned must be equal to the distance of the reference. Fig. 4 shows the clabject `MappingProject` and the notation (d<distance>@<potency >) that is used to specify both distance and potency. As a consequence, the semantics of the reference `impQSMs d2@1` in the context of a concrete mapping management project is to store instances of instances of QSTMs. There is an example for reference values (i.e., instances of MappingProject) shown at the top of Fig. 4.

3.6 DeepML's Framework Architecture

Fig. 5 gives an architectural overview of our framework. DeepMLCore is the core language implemented with the Eclipse Modeling Framework (EMF) [2]. On top of it there is a set of supporting libraries: The DeepML Editor is a generic component that provides views and editors for visualizing and modifying DeepML models from both perspectives, the ontological and the linguistic one. The Epsilon Adapter is a bridge for using the language family (M2M, M2T, etc.) provided by the Epsilon framework [3]. The Epsilon Adapter is also used to realize a bridge to BIRT (Business Intelligence and Reporting Tool) [1]. The PIQSH-SC in turn uses BIRT for generating reports (e.g., gap analysis).

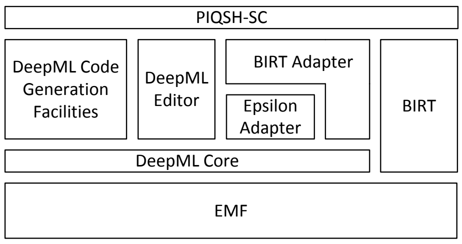


Fig. 5. DeepML's Architecture and Framework.

4 Related Work

The need to define efficient strategies for managing mappings in scenarios as outlined in Sec. 2 has been identified before, for example, in [16] and [22]. However, this paper focuses on experience made by using deep modeling techniques. So we will concentrate on related work in that area of research.

Melanee [4] provides a deep modeling framework with graphical syntax and editors. It is built on top of EMF and also incorporates ideas from the domain of ontologies. It therefore supports an additional so-called exploratory mode [9]. In exploratory mode, a reasoning engine establishes ontological instantiation relationships. However, our goal was to use a deep modeling language with a minimal set of language constructs that is optimized for building models in a constructive way, that is, by using explicit instantiation relationships. Also, Melanee does not define a concept similar to domain stacks for managing model partitioning. There is also no concept available in Melanee for realizing layer spanning references as defined in Sec. 3. However, some framework capabilities like for example emendation support [7] are neither provided by DeepML nor by any other deep modeling framework we are aware of.

Another deep modeling language is MetaDepth [5]. MetaDepth provides a convenient textual syntax for creating deep models and has some unique language features, for example, the *-potency. A clabject with a star potency has an unlimited potency. With each instantiation step, that potency may either remain unlimited or a concrete potency value may be assigned. Yet, MetaDepth lacks support for ontological containment and domain stacks that greatly simplified the architecture needed for our case study 3.

5 Conclusion and Future Work

In this paper we report on our experience gained by applying deep modeling techniques to a real life industrial use case. The corresponding application has been successfully used by our industrial partner for approximately one year, and we are currently discussing further extensions. The underlying deep model is very well suited for this kind of problem: The concept of layers and stacks provides a clean separation of concerns, and the model is very flexible. We continue to develop both, the application and the DeepML framework, mainly focussing on the following topics:

First, one of the main problems with traditional programming languages, such as Java, is, that they only support two levels of instantiation (class and object). We are currently investigating solutions to this problem either by using appropriate patterns to emulate multiple levels or by enhancing the programming language (like e.g. done in DeepJava [19]). **Second**, a lot of enhancements are planned with respect to the infrastructure. For example, diagram based editors, additional code generators for automatically generating code to ease the development of user interfaces, and so forth. **Third**, we are looking for other domains that might benefit from using deep modeling techniques. A promising candidate is the domain of variant management in software development and systems engineering. **Fourth**, we intend to explore the combination of PIQSH-M with executable process models such as eSPEM [15]. One possibility is to introduce a process execution layer below the mapping layer, but limited to the DP domain. This leads to a four-level model architecture.

To summarize, we think multi-level modeling techniques are very well suited for the kinds of problem outlined in Sec. 2. However, there remains a lot of research to be done. For example, we did not find any solutions for handling model migration in a multi-level aware way. Yet, in our view, the major stumbling block is the lack of an agreed and publicly available specification of the core concepts and terminology of multi-level modeling (a standard deep meta model).

Finally, we hope our experience report encourages other people to use deep modeling techniques — at least in areas similar to the one outlined in Sec. 2.

References

1. BIRT. `http://www.eclipse.org/birt/` (July 2014)

2. Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/ (July 2014)
3. Epsilon. http://www.eclipse.org/epsilon/ (July 2014)
4. Melanee. `http://melanee.org/` (July 2014)
5. MetaDepth. `http://astreo.ii.uam.es/ jlara/metaDepth/` (July 2014)
6. software AG: ARIS. `http://www.softwareag.com/aris` (July 2014)
7. Atkinson, C., Gerbig, R., Kennel, B.: On-the-Fly Emendation of Multi-Level Models. In: Modelling Foundations and Applications. LNCS, vol. 7349, pp. 194–209. Springer (2012)
8. Atkinson, C., Kennel, B., Go, B.: The Level-Agnostic Modeling Language. In: Software Language Engineering, LNCS, vol. 6563, pp. 266–275. Springer (2011)
9. Atkinson, C., Kennel, B., Go, B.: Supporting Constructive and Exploratory Modes of Modeling in Multi-Level Ontologies. 7th Intl. Workshop on Semantic Web Enabled Softw. Eng. (2011)
10. Atkinson, C., Kühne, T.: Rearchitecting the UML Infrastructure. ACM Trans. Model. Comput. Simul. 12(4), 290–321 (Oct 2002)
11. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software 20(5), 36–41 (2003)
12. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software and System Modeling 7(3), 345–359 (2008)
13. Automotive SIG: Automotive SPICE Process Reference Model, Ver. 4.5 (May 2010)
14. CMMI Product Team: CMMI for Development, Ver. 1.3. Tech. Rep. CMU/SEI-2010-TR-033, Carnegie Mellon Univ. – Software Eng. Inst. (Nov 2010)
15. Ellner, R., Al-Hilank, S., Drexler, J., Jung, M., Kips, D., Philippsen, M.: A FUML-Based Distributed Execution Machine for Enacting software process models. In: ECMFA. LNCS, vol. 6698, pp. 19–34 (2011)
16. Ferreira, A.L., Machado, R.J., Paulk, M.C.: Supporting audits and assessments in multi-model environments. In: PROFES. Lecture Notes in Business Information Processing, vol. 6759, pp. 73–87 (2011)
17. Gonzalez-Perez, C., Henderson-Sellers, B.: A powertype-based metamodelling framework. Software and System Modeling 5(1), 72–90 (2006)
18. Intl. Org. for Standardization: ISO 26262: Road vehicles – Functional safety (Nov 2011)
19. Kühne, T., Schreiber, D.: Can Programming be Liberated from the Two-Level style? Multi-Level Programming with DeepJava. In: OOPSLA. pp. 229–244 (2007)
20. OMG: Meta Object Facilities. `http://www.omg.org/mof/` (July 2014)
21. OMG: Unified Modeling Language. `http://www.uml.org/` (July 2014)
22. Siviy, G. et al.: Maximizing your Process Improvement ROI through Harmonization. Tech. rep., Carnegie Mellon Univ. – Software Eng. Inst. (March 2008)

Modeling Techniques for Enterprise Architecture Documentation: Experiences from Practice

Thomas Trojer, Matthias Farwick, and Martin Haeusler

University of Innsbruck, Innsbruck, Austria
`firstname.lastname@uibk.ac.at`

Abstract. Enterprise Architecture Management (EAM) is an IT-management process in which the relationships of business services, applications and the underlying IT-infrastructure is modeled. With dedicated EA models, activities such as architectural consolidation, planning and risk analysis are facilitated. A key factor in modeling and documenting enterprise architectures is the the underlying meta-model that enables to capture the information demand of an organization. Current EA tools provide no or only inflexible mechanisms to create and evolve such organization-specific meta-models. Therefore we present a novel modeling framework that has been established from a consulting and a research project with two data centers. It makes use of both, concepts from multi-level modeling and classical three-level modeling and separates structural and ontological model ingredients. A key aspect of the presented approach is the separation of modeling tasks across different stakeholders and modeling levels and the favoring of practical usability over language feature richness.

1 Introduction

In the context of *Enterprise Architecture Management* (EAM) and *(IT-)systems operation management*, specialized modeling tools are typically used to model the dependencies between the IT-infrastructure, deployed applications and the business functions they support [14]. These models are then used to analyze the current architecture, assess risks and plan changes to it.

In our previous work [7] we showed that keeping such a model in-sync with reality is a major problem in practice. In line with Schweda [16], we argue that a key aspect of an organization's ability to effectively utilize and update the model, is to create an evolvable organization-specific meta-model that matches the stakeholders current information demand. In the context of EAM we call this meta-model the *information model* (i.e. *M1*).

An information model is defined to only capture organization relevant data and specifies the architectural patterns that occur in the organization's business and IT. However, the proper definition of such a model is hard to achieve in practice: Typically multiple types of modeling artifacts need to be maintained and different stakeholders use and adjust them. This poses challenges to both, the underlying modeling framework itself as well as the user interfaces that manage certain stakeholder groups to only model those parts for which they have expertise.

In this paper we briefly present the EA and IT-modeling tool *Txture*[1] that, among other features, is capable of a flexible creation of information models and the maintenance of instances thereof. The main purpose of *Txture* is to support the architecture documentation efforts of stakeholders in an enterprise. It separates the modeling concerns at different modeling layers via dedicated user interfaces and thus links the different stakeholder groups to their individual area of expertise and responsibility. The tool is the result of ongoing consulting and research work in collaboration with two data centers that helped us to identify modeling challenges in practice.

A central part of this paper is to show how we tackled modeling challenges with a combination of a classical three-level modeling approach, a type – instance based modeling approach and flexible extensions for individual model elements. The goal of work is to share our experiences from practice and to engage in discussions with the multi-level modeling research community on our approach and potential alternative methods.

The remainder of this paper is structured as follows: We first provide general background information for the *Txture*-tool. We then continue by presenting IT-architecture modeling requirements and challenges in Section 3. In Section 4, we describe how our modeling framework tackles these. Finally, we discuss related work and end with concluding remarks.

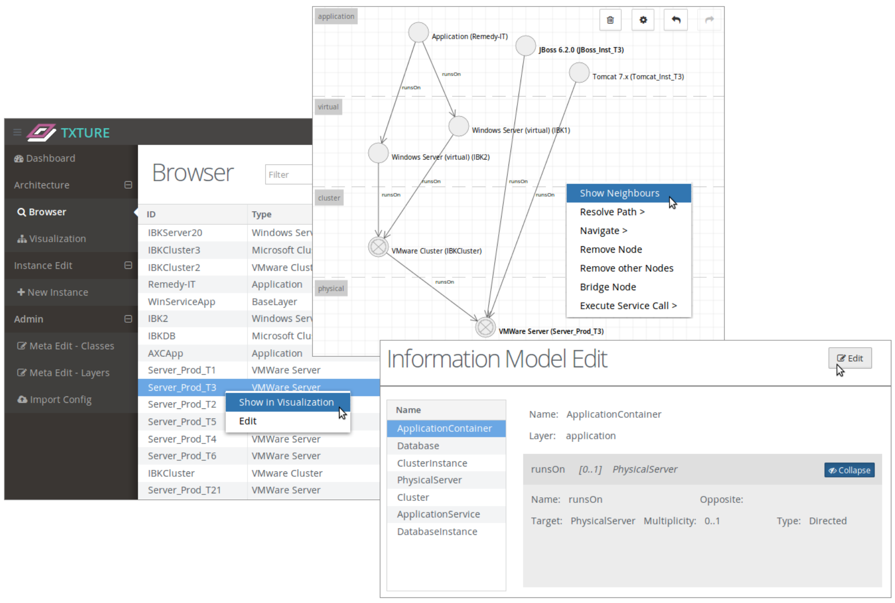
2 Background of the IT-Modeling Tool Txture

In 2011 we started a consulting project with a banking data center and subsequently a research project with the data center of a large semiconductor manufacturer. The overall goal of both projects was to make IT-infrastructure documentation more efficient and effective. Enhanced usability features and stakeholder-orientation of the implemented tool were generally considered important. Besides, requirements of common work activities on top of an IT-documentation, such as flexible visualizations of the architecture for planning and risk analysis, have been elicited.

The key features of the resulting *Txture* modeling tool are:

– Dynamic and flexible visualizations of IT-architectures via graphs.
– Configurable import mechanisms to automatically use architectural data contained in external sources such as in *Configuration Management Databases* (CMDB), *Excel* spreadsheets or relational databases.
– Modeling of the architecture via a form-based web-client to support less technically skilled users.
– Textual architecture modeling via an information-model aware *Eclipse*-based text editor [8].

[1] `http://www.txture.org`



ID	Type
IBKServer20	Windows Serv
IBKCluster3	Microsoft Clu
IBKCluster2	VMware Clust
Remedy-IT	Application
WinServiceApp	BaseLayer
IBK2	Windows Serv
IBKDB	Microsoft Clu
AXCApp	Application
Server_Prod_T1	VMWare Server
Server_Prod_T3	VMWare Server
Server_Prod_T2	
Server_Prod_T5	
Server_Prod_T4	VMWare Server
Server_Prod_T6	VMWare Server
IBKCluster	VMware Cluster
Server_Prod_T21	VMWare Server

Name
ApplicationContainer
Database
ClusterInstance
PhysicalServer
Cluster
ApplicationService
DatabaseInstance

Name: ApplicationContainer
Layer: application

runsOn [0..1] PhysicalServer

Name: runsOn Opposite:
Target: PhysicalServer Multiplicity: 0..1 Type: Directed

Fig. 1. The *Txture* environment showing the architecture browser (left screenshot), navigable visualizations (top-most) and the ability to view and change the information model (bottom-most). The tool is fully functional.

- High-performance model queries via optimized persistence of models in a graph database.
- The ability to define and change the information model at runtime.

In order to exemplify the motivation for such a tool, Figure 1 depicts a graph-based architecture visualization in *Txture*. Here, relationships between *application containers*, an *application* and the underlying (clustered) hardware infrastructure are shown. Such a visualization is used in practice e.g., to perform impact and risk analysis of application deployments.

Several other key visualization features can be seen in this figure:

- Architectural elements are assigned to configurable layers, hence the visualization automatically shows an intuitive architectural stack.
- The visualization is navigable via a set of traversal, deletion and grouping operations for depicted documentation nodes (see context menu in Figure 1).
- Nodes are styled based on their type or other attributes, like mission-criticality.

Furthermore, Figure 1 also shows the meta-modeling capabilities for defining information models via a form-based editor.

In the following section we outline the main modeling challenges that led to *Txture*'s underlying modeling framework.

3 Modeling Challenges

Related work in the domains of IT-systems modeling as well as in the multi-level modeling community has mostly focused on modeling software systems (refer to Section 5). Compared to the modeling of software systems, IT-infrastructure documentation has unique modeling requirements and its very own challenges. We can summarize the most important challenges as follows:

CH1: *Separation of stakeholders defining the information model and the ones who are actually responsible for documenting IT-systems:* Two main modeling stakeholder groups were determined in both projects: One group consists of *Expert Architects* and the other group are *Element Responsibles*. Expert architects have an interest in overseeing the entire IT-architecture of a given organization. This group of stakeholders usually works together in order to develop the organization-specific information model that forms the basis for the actual IT-systems documentation. Architects are not necessarily the persons who document the architecture. This is the task of *Element Responsibles*, who maintain dedicated parts of the architecture documentation. Opposed to the architects, they are mostly experts in a very narrow field that revolves around the items and technologies they work with. *Server* responsibles are likely to be experts in very specific types of virtualizations or hardware. *Application* responsibles, on the other hand, often only roughly know the hardware their applications are deployed on. As one can see, distinct user groups perform modeling on different levels. This challenge needs to be tackled by a proper IT-documentation tool.

CH2: *Documentation and information models both need to be evolvable:* In cases where the general architectural structure of an organization shifts, it becomes necessary to adapt the information model (think e.g. introduction of cloud computing), or an architectural pattern was discovered that can not be documented with the current information model. These types of changes should be applicable while the tool is running and without the help of a modeling expert (i.e. no recompilation, complex configuration and adherence to certain modeling patterns is required).

CH3: *A documentation tool needs to expose familiar terminology and enterprise-aligned concepts to stakeholders:* Especially in the more technology-related parts of an architecture documentation, changes occur frequently and require updates to modeling artifacts. A documentation tool needs to enable stakeholders to quickly re-establish a useful documentation that is in-sync with the real world and reflects enterprise-specific terminology.

CH4: *Documentations need to be extensible according to individual stakeholder's documentation requirements:* Different stakeholders typically have different documentation demands. We found out e.g. that components which are central to an IT-architecture are likely to be documented in a more detailed way than others. Therefore a documentation tool needs to flexibly cater for documentation intents of stakeholders that are beyond a defined common model.

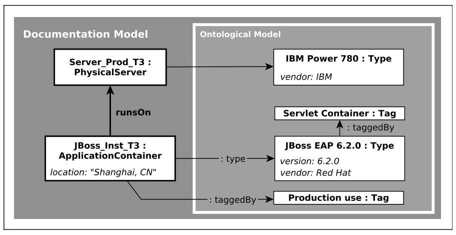


Fig. 2. A simple IT-infrastructure documentation model.

4 Modeling Solutions

In this section we further outline the *Txture* modeling framework, first, by providing an example model on which we base our discussions and second, by presenting our solution approach to tackle the abovementioned challenges.

The *documentation model* (i.e. *M0*) in Figure 2 shows instances of IT-system components that are documented. The specific example describes an application container instance "*JBoss_Inst_T3*" which "*runs on*" a physical server named "*Server_Prod_T3*". As we have described in the previous section, such a documentation model can be used e.g., to perform impact analysis ("What happens if the specific server crashes?") or to do infrastructure planning ("Is the specific server appropriately dimensioned to run such software?"). Additional to modeling IT-component instances and their structural dependencies, a simple notion of *ontology* can be seen on the right side of the figure. Such ontological classifications are modeled as part of the documentation activity (also on *M0*) and allow *Element responsibles* to further describe and categorize their documented instances.

The descriptive concepts which are available are *types* and *tags*. Types are used to provide a (domain-related) classification of instances and may also define additional attributes for them. In our example case, the application container instance is of type *"JBoss EAP 6.2.0"*. Additionally, a set of tags can be assigned to instances and types. They provide a simple means to further classify elements via keywords. In our example the server type is tagged *"Servlet Container"* to indicate its relatedness to *Java servlet* technology. The typing and tagging mechanisms are also used in *Txture* to allow browsing, search and filter functionality across the IT-systems documentation.

Figure 3 provides an extended picture of our example model by including its meta-model hierarchy. On the information model level, the expressiveness of the underlying documentation model is set. At this level the *linguistic structure* that architects agreed upon is modeled. Opposed to this, the *ontological structure*, which reflects domain expert knowledge, is modeled as part of the documentation process. This reflects the separation that is demanded in challenge **CH1**.

The top-level artifact, the *meta-meta model* (i.e. *M2*), defines all concepts that are used in *Txture* and, in line with requirements of our industry partners, are needed to properly describe their IT-infrastructures. It defines the concepts

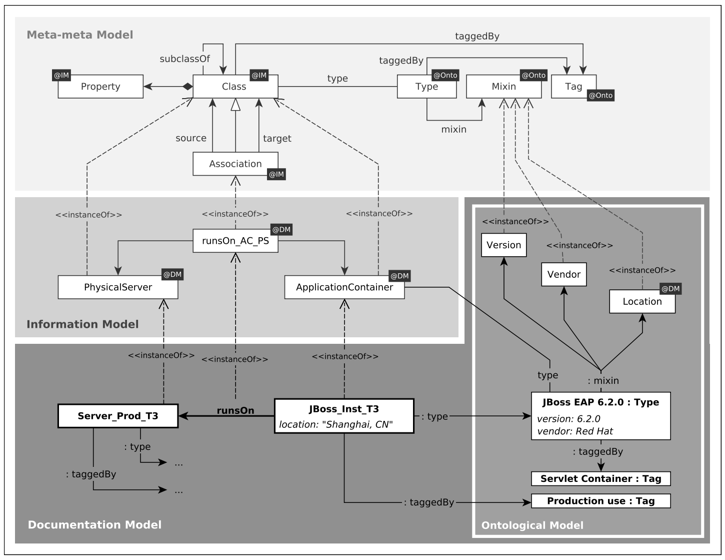


Fig. 3. The *Txture* modeling infrastructure. Annotation boxes (black) reflect where a model element gets instantiated (@IM = Information Model, @Onto = Ontological model and @DM = Documentation Model).

class, *association* (i.e. association classes) and *property* to develop the structure of an organization-specific architecture modeling language (i.e. the linguistic model) and the concepts *type*, *tag* and *mixin* that shape the ontological model.

4.1 Classical Hierarchies to separate Modeling Activities

One of the experiences we gained from modeling workshops with our industry partners is that modeling novices or software developers understand modeling best when using strict and limited hierarchies in which modeling concepts and their instantiations are described. In our case the modeling levels that users have to interact with are manifested by the information model and the documentation model as its instantiation.

Besides understandability of concepts, having a clear cut between modeling levels also supports a permission and concern-oriented separation for managing the IT-documentation and the information model it relies on. This separation is important as different modeling activities are performed by individual stakeholders with potentially diverse domain expertise. In one of our projects, stakeholder roles like employees from operations, database administrators, software developers and also project managers were involved in the documentation process and a few selected stakeholders of these groups together with the IT-architects managed the information model.

In our tool the modeling of each level is separated by different user interface and therefore tackles challenge **CH1**.

4.2 Types to mitigate Invasive Information Model Changes

Another experience we made was that adapting the information model is typically a recurring activity, triggered by frequent change requests from industry partners and driven by adjustments, extensions and simplifications to modeled concepts. It is common to any modeling activity, that changes to models may involve corresponding changes on dependent models, as part of re-establishing conformance in the model hierarchy. To minimize the efforts and consequences of such changes, either well-defined automated model refactoring procedures are required or an information model needs to be realized in a way so that the most-common changes to it only minimally interfere. For our industry partners a manual refactoring after information model changes was out of question. This is why we settled on a modeling pattern similar to the one of *power types* [15] that allows for creating types at the documentation model level and therefore reduces the need to actually adapt the related information model.

Our original modeling approach made heavy use of inheritance on the information model level. For example we applied a deep inheritance structure to model different *application containers* according to their *vendor*, *software version* or required *runtime platform*. This rendered the information model both, large in size (i.e. number of model elements) and prone to frequent changes (e.g. on software version changes).

Using *types* greatly helped to reduce the size of the information model and therefore maintaining comprehensibility and lowering the frequency in which changes to it needed to be applied. Based on using types, our new modeling approach consists of only including generic information model elements like *physical server* or *application container*. The goal was to provide basic, but stable modeling concepts that are *invariant* to an organization. These concepts span the structure of the model, i.e. the permissible nodes and relations between them. This reflects the generic structure that all involved stakeholders can relate to. E.g. no highly-specific vendor-based product terminology is used that would only be understood by a minority of stakeholders. Accordingly, in our ontological model we allow to extend information model elements with the help of *types*. Types are part of the documentation model, but reference elements of the information model. In the example of Figure 2 and 3 *JBoss EAP 6.2.0* extends the meaning of the documented instance *JBoss_Inst_T3* beyond that of being an *application container*. While *application container* can be considered a stable information model concept, *JBoss*-specific server software will likely change from time to time and by our understanding of types can be easily adjusted within the documentation model. This is in line with Atkinson and Kühne [2], who describe the need for changes and newly added types that are possible while the system is running. Our type concept delivers a light-weight way for dynamic additions and proved to be intuitively usable in IT-infrastructure documentation practice.

In addition to types, we use *tags* to further categorize documentation model elements. Tags are comparable to *UML stereotypes*[2] and can be applied to types

[2] cf. *UML 2.4.1* infrastructure specification, `http://www.omg.org/spec/UML/2.4.1/`

and individual instances. In *Txture* both, type and tag elements are modeled by element responsibles and as part of the documentation model.

Our intention with types is to reduce the amount of changes on the information model, tackling challenge **CH2**. Using types together with tags as a means to categorize and describe documented instances contributes a solution to challenge **CH3**.

4.3 Multi-level Instantiation to support Dynamic Extensions

With the introduction of types on the documentation model level, we are able to limit the amount of changes that otherwise are applied to the information model. While this is beneficial, maintaining an information model of only generic concepts bares issues regarding the expressiveness of the documentation: Generic information model concepts leave out detail and shift the specification of properties of documentation elements onto types. Our documentation activities require that types and instances can be managed by the same stakeholders within the documentation model. For proper infrastructure documentation, types not only define properties to be instantiated by their related instances, but need to specify values for certain properties themselves.

Figure 3 shows that the *JBoss*-example type defines values for the properties *version* and *vendor*, whereas our example application container defines a text value reflecting its deployment *location* to be "Shanghai". In our exemplary documentation model we assume this property to be dependent on the actual type, as e.g., not for all application containers the location is known or relevant to be documented. Because of this, we needed to realize a property-like concept, so called *mixin*s [5], that can be instantiated on both, the level of types and documented instances. This is comparable to the concept of *deep instantiation* [1] or that of *intrinsic attributes* in the *MEMO* meta-modelling language [10].

The mixin concept aligns well with the flexible nature of our type concept and allows the documenting stakeholders to adapt the documentation model to cater their particular documentation needs. With mixins we provide a potential solution to challenge **CH4**.

5 Related Work

In the context of EAM it is common that tools provide predefined information models that can often only be adapted in a very limited way. For example, the EAM tool *iteraplan*[3] only allows for the extension of existing classes via attributes. No additional classes or relationships can be added. As shown in the EAM tool survey by Matthes et al.[14] there exist some configurable tools, their technical foundation, however, is not clear. Other tools work with fixed information models based on EA modeling standards such as *The Open Group Architecture Framework* [11] or *Archimate* [13]. We argue that these standards

[3] http://www.iteraplan.de/en

are inflexible as it is difficult to adapt them to the terminology used in an organization or to evolve. Schweda presents a sophisticated approach for pattern-based creation of organization-specific information models [16] and shares our modeling requirements in his research. However, other than the scope of our work, its practical applicability was not shown so far. With the *MEMO* meta-modeling language, Frank et al. [10] present a language and a tool suite for building modeling languages in the enterprise context. The tool is Eclipse-based and needs code generation steps in order to react on a changed information model. The proposed language for IT-infrastructure modeling, ITML [9], provides fixed concepts and can not support organization-specific information models. In line with Kattenstroth [12], we conclude that although the need for organization-specific and evolvable EA information models has been identified in literature [7, 16], related work mostly focuses on formulating generic and fixed information models that cannot be adapted to the requirements of a given organization.

In the general modeling research much related literature can be identified. Still, modeling in this area mainly discusses requirements from software engineering and does not necessarily consider modeling techniques from other domains. For *Txture* we mainly built on top of known modeling paradigms, but unified them in a novel way to contribute a usable EA documentation method. This includes ideas from *UML stereotypes*, *power types*, the proposed separation of *linguistic* and *ontological* models (and instantiations) [2]. Implementation-wise we rely on *Ecore*[4], an object-oriented meta-modeling framework with reflective capabilities and a programming interface for *Java*, which proved to be stable and reliable. For persisting models we used a custom hybrid repository approach including a relational database together with a graph database for permanent storage and fast model query capabilities respectively. These technologies were chosen due to prior experience; another promising alternative to be evaluated for our use case is *MetaDepth* [6], a framework for supporint arbitrary numbers of meta levels and advanced modeling concepts. As we implemented custom form-based modeling user interfaces to ease the IT documentation for non-modelers, we were unable to rely solely on UML and its provided graphical notations; despite many UML concepts are used in our work. *Txture* also consists of a number of different editors for different stakeholder groups and purposes (cf. e.g., the approaches described by Atkinson et al. [3, 4]).

6 Conclusion & Outlook

In this paper we presented the modeling framework *Txture* that offers a flexible mechanism to create organization-specific architecture models. In the main section we described IT-architecture documentation challenges and presented our solutions. These solutions are derived from practical experiences from two industry projects. Based on these experiences we claim that proper architecture modeling requires a hybrid approach, consisting of a classical meta-model hierarchy and multi-level modeling methods. The classical modeling part is important

[4] `http://www.eclipse.org/modeling/emf/?project=emf`

to make documentation capabilities comprehensible to a wide range of different stakeholders in an enterprise. Multi-level modeling, like we employ via types and mixins renders the overall documentation process flexible and extensible. Thus, many of our design decisions were influenced by the overriding principles of practical tool usability and intuitiveness. For example, we favored a free tagging mechanism over type-inheritance.

While we strongly believe that many application domains would benefit from multi-level modeling concepts, however, corresponding modeling frameworks that have proven maturity are rare. This is why we built *Txture* on top of *Ecore*, although we needed to heavily deviate from its intended usage (regarding power types and mixins) to build the here-described framework.

In the future we want to gather additional practical experiences, in order to further evaluate the flexibility of *Txture*'s modeling capabilities.

References

1. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. The Unified Modeling Language. Modeling Languages, Concepts, and Tools (2001)
2. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. IEEE Software 20(5) (Sep 2003)
3. Atkinson, C., Gerbig, R.: Harmonizing Textual and Graphical Visualizations of Domain Specific Models Categories and Subject Descriptors. In: Proceedings of the 2nd Workshop on Graphical Modeling Language Development. ACM (2013)
4. Atkinson, C., Gerbig, R., Tunjic, C.: A multi-level modeling environment for SUM-based software engineering. Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '13 (2013)
5. Bracha, G., Cook, W.: Mixin-based inheritance. ACM SIGPLAN Notices 25(10), 303–311 (1990)
6. De Lara, J., Guerra, E.: Deep meta-modelling with metadepth. In: Objects, Models, Components, Patterns, pp. 1–20. Springer (2010)
7. Farwick, M., Schweda, C.M., Breu, R., Hanschke, I.: A situational method for semi-automated Enterprise Architecture Documentation. SOSYM (Apr 2014)
8. Farwick, M., Trojer, T., Breu, M., Ginther, S., Kleinlercher, J., Doblander, A.: A Case Study on Textual Enterprise Architecture Modeling. In: Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2013 17th IEEE International. IEEE (2013)
9. Frank, U., Heise, D., Kattenstroth, H., Fergusona, D., Hadarb, E., Waschkec, M.: ITML: A Domain-Specific Modeling Language for Supporting Business Driven IT Management. In: Proceedings of the 9th workshop on domain-specific modeling (DSM). ACM (2009)
10. Frank, U.: The MEMO meta modelling language (MML) and language architecture. 2nd Edition. Tech. rep., Institut für Informatik und Wirtschaftsinformatik (ICB) Universität Duisburg-Essen (2011)
11. Haren, V.: TOGAF Version 9.1. Van Haren Publishing (2011)
12. Kattenstroth, H.: DSMLs for enterprise architecture management. In: Proceedings of the 2012 workshop on Domain-specific modeling (DSM). ACM Press (Oct 2012)
13. Lankhorst, M.: Enterprise Architecture at Work, vol. 36. Springer Berlin Heidelberg, 3rd editio edn. (Jan 2012)
14. Matthes, F., Buckl, S., Leitel, J., Schweda, C.M.: Enterprise Architecture Management Tool Survey 2008. Tech. rep., Technische Universität München,Chair for Informatics 19 (sebis) (2008)
15. Odell, J.J.: Power Types. Journal of OO Programming (1994)
16. Schweda, C.M.: Development of Organization-Specific Enterprise Architecture Modeling Languages Using Building Blocks. Ph.D. thesis, TU Munich (2011)