ACM/IEEE 18th International Conference on
Model Driven Engineering Languages and Systems

September 27 – October 2, 2015 • Ottawa (Canada)

MULTI 2015 – Multi-Level Modelling Workshop Proceedings

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

Published on Sept 2015 v1.0

© 2015 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners.

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

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

Table of Contents

Preface 1

Experimenting with Multi-Level Models in a Two-Level Modeling Tool 3
Martin Gogolla

Aspect-oriented Concrete Syntax Definition for Deep Modeling Languages 13
Colin Atkinson and Ralph Gerbig

Exploring Multi-Level Modeling Relations Using Variability Mechanisms 23
Iris Reinhartz-Berger, Arnon Sturm, and Tony Clark

Multi-Language Modelling with Second Order Intensions 33
Vadim Zaytsev

Practical Multi-level Modeling on MOF-compliant Modeling Frameworks 43
Kosaku Kimura, Yoshihide Nomura, Yuka Tanaka, Hidetoshi Kurihara, and Rieko Yamamoto

An algebraic instantiation technique illustrated by multilevel design patterns ... 53
Zoltan Theisz and Gergely Mezei

Preface

MULTI 2015 was the second installment of the MULTI workshop series focusing on multi-level modeling. It was held as a satellite event of the ACM/IEEE sponsored conference MODELS 2015, in Ottawa (Canada). The goal of this workshop was to continue the community building initiated in the first edition held in conjunction with MODELS 2014 in Valencia (Spain). In order to increase the level of dissemination of work in multi-level modeling and to provide more opportunities for plenary discussions and/or group work, MULTI 2015 was organized as a two-day workshop rather than the typical one-day workshop.

The aim of the first day of the workshop was to set the scene for discussions by means of paper presentations that included an invited paper and two regular paper sessions. The invited paper by Professor Martin Gogolla of the University of Bremen, entitled "Experimenting with Multi-Level Models in a Two-Level Modeling Tool", explored the extent to which multi-level models can be simulated in two-level modeling environments by explicitly modeling different underlying linguistic (meta) models at the class level, and simulating ontological classification relationships at the instance level in the form of links.

The first paper in the first regular paper session by Colin Atkinson and Ralph Gerbig, entitled "Aspect-oriented Concrete Syntax Definition for Deep Modeling Languages", described how to support a deep, context sensitive visualization of multi-level models using concepts from aspect-orientation to merge concrete syntax elements across instantiation chains. The second paper by Iris Reinhartz-Berger, Arnon Sturm and Tony Clark, entitled "Exploring Multi-Level Modeling Relations Using Variability Mechanisms", explored the relationship between the instantiation forms found in multi-level modelling and those used in product line engineering.

The first paper of the afternoon session by Vadim Zaytsev, entitled "Multi-Language Modelling with Second Order Intensions", proposed the use of second-order intensions and extensions to more closely model linguistic and ontological conformance. The second paper by Kosaku Kimura, Yoshihide Nomura, Yuka Tanaka, Hidetoshi Kurihara and Rieko Yamamoto, entitled "Practical Multi-level Modeling on MOF-compliant Modeling Frameworks" explored how multi-level modeling can be supported using existing modeling frameworks based on the MOF. The third paper by Zoltan Theisz and Gergely Mezei, entitled "An algebraic instantiation technique illustrated by multilevel design patterns" proposed a new algebraic instantiation approach aiming to provide a solid, algebraic foundation for multi-level meta-modelling which is easily customizable. The first day concluded with a plenary session that allowed participants to discuss issues raised in the presentations and find consensus on how to best structure the second afternoon.

The morning of the second day of the workshop was devoted to providing an insight into the range of tools currently available to support multi-level modeling. Six different tools with various kinds of multi-level modeling capabilities were demonstrated over two sessions: the DPF workbench from Bergen University College, Norway (presented by Xiaoliang Wang and Yngve Lamo), WebDPF from Bergen University College, Norway (presented by Fazle Rabbi and Yngve Lamo, Melanee – The Deep Modeling Domain-specific Language Workbench from the University of Mannheim, Germany (presented by Ralph Gerbig), MetaDepth – a tool for multi-level model-driven engineering, from the Universidad Autnoma de Madrid, Spain (presented by Juan de Lara and Esther Guerra), a tool for Multilevel Modelling and Reasoning with FOML from Ben-Gurion University, Israel and SUNY Stony Brook, USA (presented by Mira Balaban, Igal Khitron and Michael Kifer), and TouchCORE from McGill University, Montreal (presented by Jörg Kienzle).

The final afternoon of the workshop was structured by two plenary sessions discussing the state of multi-level modelling technology and identifying areas where further clarification would be helpful to the community. The debate included arguments on the nature of the distinction between linguistic and ontological classification and the roles these different forms of classification should play within classification architectures. One of the intensely discussed questions was whether ontological classification relationships should require certain syntactic conformance rules in addition to being based on (either explicitly or implicitly represented) mappings to the meaning of the related elements. Due to a lack of time, the idea of agreeing on a definition of "multi-level modeling" could only be partially pursued by collecting a set of candidate definitions. Likewise, a concrete identification of use cases, scenarios, reference architectures, etc. had to be deferred.

In order to further support the dissemination of new information and provide a common repository for definitions, tool descriptions, modeling artifacts, etc., the participants agreed to establish a forum for online interaction for the multi-level modelling community in the form of a wiki web. The respective wiki can be found at:
http://homepages.ecs.vuw.ac.nz/Groups/MultiLevelModeling/ and everyone is invited to update contents. The website devoted to the MULTI workshop series is still hosted at: http://www.miso.es/multi/.

We are grateful to all authors of submitted papers, to the reviewers for their constructive criticism, to all participants of the workshop for the interesting and lively discussions, and to the organizers of MODELS 2015 for their support.

October 2015 Colin Atkinson, Georg Grossmann, Thomas Kühne, Juan de Lara

Experimenting with Multi-Level Models in a Two-Level Modeling Tool

Martin Gogolla

Database Systems Group, University of Bremen, Germany
gogolla@informatik.uni-bremen.de

Abstract. This paper discusses two ways to establish the connection between two levels in a multi-level model. The first approach uses normal associations and generalizations under the assumption that the multi levels are represented in one UML and OCL model. The second approach views a middle level model both as a type model and as an instance model and introduces operations to navigate between the type and instance level. The paper reports on some experiments that have been carried out.
Keywords. UML, OCL, Model, Multi-level model, Metamodel, Level connection.

1 Introduction

Metamodeling has become a major topic in software engineering research [6, 7, 16]. There are, however, a lot of discussions about central 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 [9, 2].

Metamodeling is closely connected to multi-levels models because the instantiation of a metamodel is a model. That model, when viewed as a type model, may be instantiated again and can then be viewed as an instance model. Proceeding this way, at least three model levels arise.

This paper discusses two ways to establish the connection between two levels in multi-level models. The first approach uses ordinary associations and generalizations under the assumption that the multi levels are represented in one UML and OCL model. The second approach views a middle level model both as a type model and as an instance model and introduces operations in order to navigate between the type and instance level. The paper reports on some experiments that have been carried out. The first approach joins the metamodels of several levels into one model as in our previous work [12]. Details about the first approach can be found in [13]. The second approach has not been put forward in a paper yet.

Our work has links to other related or similar 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 [9] allowing linguistic as well as ontological instantiation with an arbitrary number of metalevels supporting the potency concept. In [16] 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 approach in [14] also transforms multi-level models into a two-level model and focusses on constraints. Similar to our approach employing UML and OCL, the work in [17] uses F-Logic as an implementation basis for multi-level models including constraints. A conceptual clean foundation for multi-level modeling is discussed in [8]. [4] studies model constraints and [3] discusses multi-level connections in presence of a multi-level architecture distinguishing between a linguistic and an ontologic view on modeling.

The structure of the rest of the paper is as follows. Section 2 gives a small example for establishing the multi-level connection with associations and generalizations. Section 3 discusses the second approach that uses operations to connect the multi-levels. The contribution is closed with a conclusion and future work in sect. 4.

2 Connecting Multi-Levels with Associations and Generalizations

The first approach connects multi-levels with usual model elements: associations and generalizations. 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 [11, 10]. In particular, we have employed the features supporting UML generalization constraints as discussed in [1, 15].

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`.

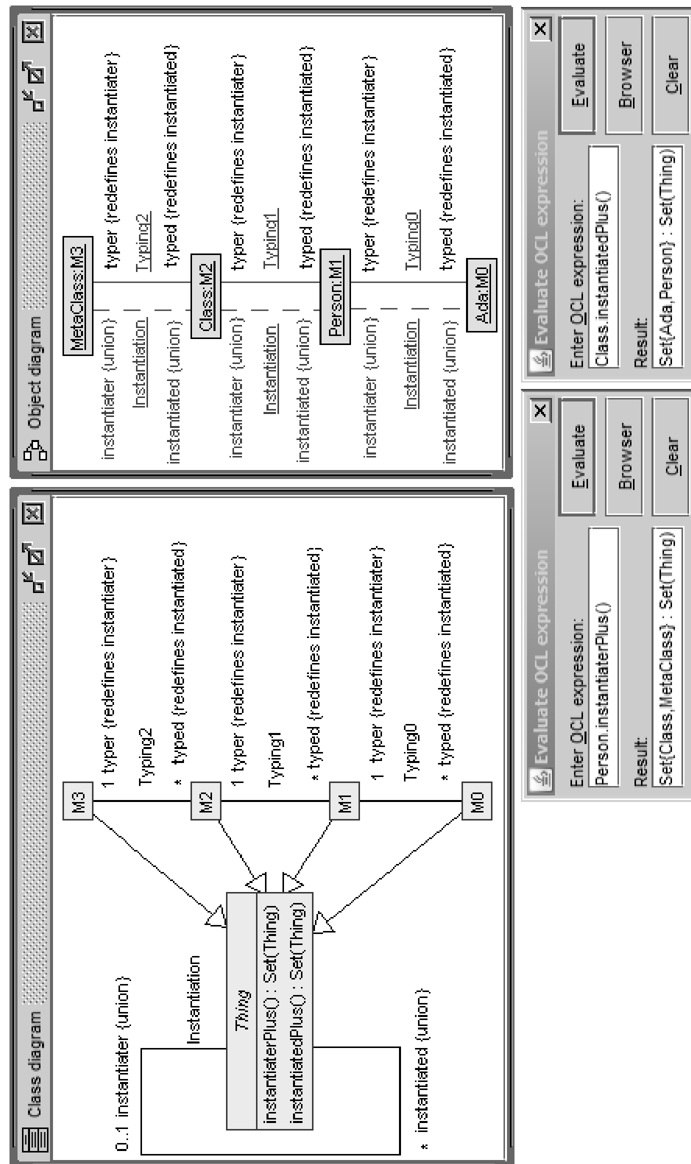


Fig. 1. Ada, Person, Class, MetaClass within Single Object Diagram.

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)= ...
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.

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 Connecting Multi-Levels with Operations

The second approach for connecting multi-levels is based on viewing one model both as a type model and as an instance model. In Fig. 2 an example as elaborated with USE [11, 10] is shown. The object diagram on the left side is essentially

equivalent to the class diagram on the right side. The displayed example can be realized in USE in this way, however, there is currently no option to connect the two equivalent representations in form of the class diagram and the object diagram. Additional features would be needed.

In Fig. 4 such a connection is indicated with (a) one operation accessing a model element through its String-valued name `$_$ : String -> ModelElement` and (b) one operation returning the String-valued name of a model element `#_# : ModelElement -> String`.

Apart from the two new operations dollar `$` and sharp `#` some new modeling features for UML and OCL would be needed as well: (a) OCL clauses (for elements such as invariants or pre- and postconditions) should allow parameters that represent variables for model elements and (b) the new operations dollar and sharp should be allowed in expressions for model elements in clauses (e.g., for a class or an attribute). The following examples try to give an impression of the aimed functionality.

```
parameter[rs:RelSchema]
let relSchemaClass = $rs.name$ in -- in this example: $rs.name$=rs
let keyAttr = $rs.attr->any(a|a.isKey=true).name$ in
context relSchemaClass inv keyAttrUnique:
  relSchemaClass.allInstances->forAll(x,y |
    x<>y implies x.keyAttr<>y.keyAttr)

parameter[rs:RelSchema]
let relSchemaClass = $rs.name$ in
let keyAttr = $rs.attr->any(a|a.isKey=true).name$ in
context x,y:relSchemaClass inv 'keyAttrUniqueIn' + #rs#:
  x<>y implies x.keyAttr<>y.keyAttr
```

In these examples it is assumed that there is exactly one key attribute for each class. When these parameterized features become actualized, then in this case the following invariants would be required. It is an open question whether the actualization is implicit or explicit: The actualization mechanism may be considered as being implicit for all actual features that are available in the model, or the actualization mechanism may be considered as being something the modeler has to explicitly ask for.

```
context Town inv keyAttrUnique:
  Town.allInstances->forAll(x,y | x<>y implies x.name<>y.name)
context Country inv keyAttrUnique:
  Country.allInstances->forAll(x,y | x<>y implies x.name<>y.name)

context x,y:Town inv keyAttrUniqueInTown:
  x<>y implies x.name<>y.name
context x,y:Country inv keyAttrUniqueInCountry:
  x<>y implies x.name<>y.name
```

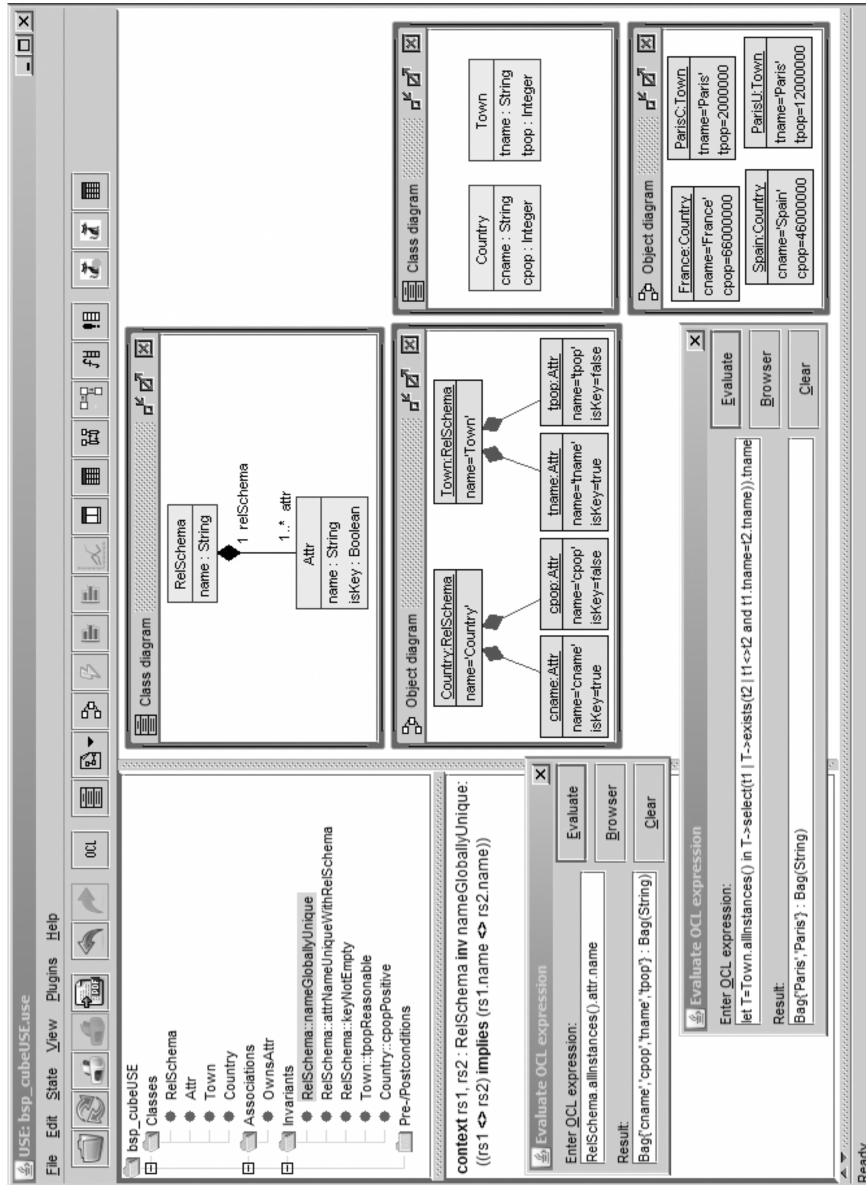


Fig. 2. Example for Multiple Representations of Middle Level Model.

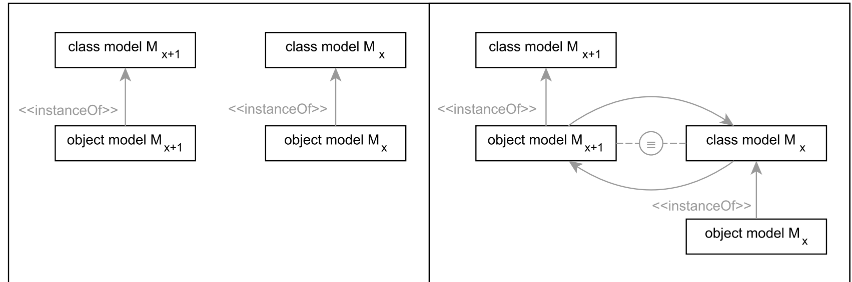


Fig. 3. General Scheme for Multiple Representations of Middle Level Model.

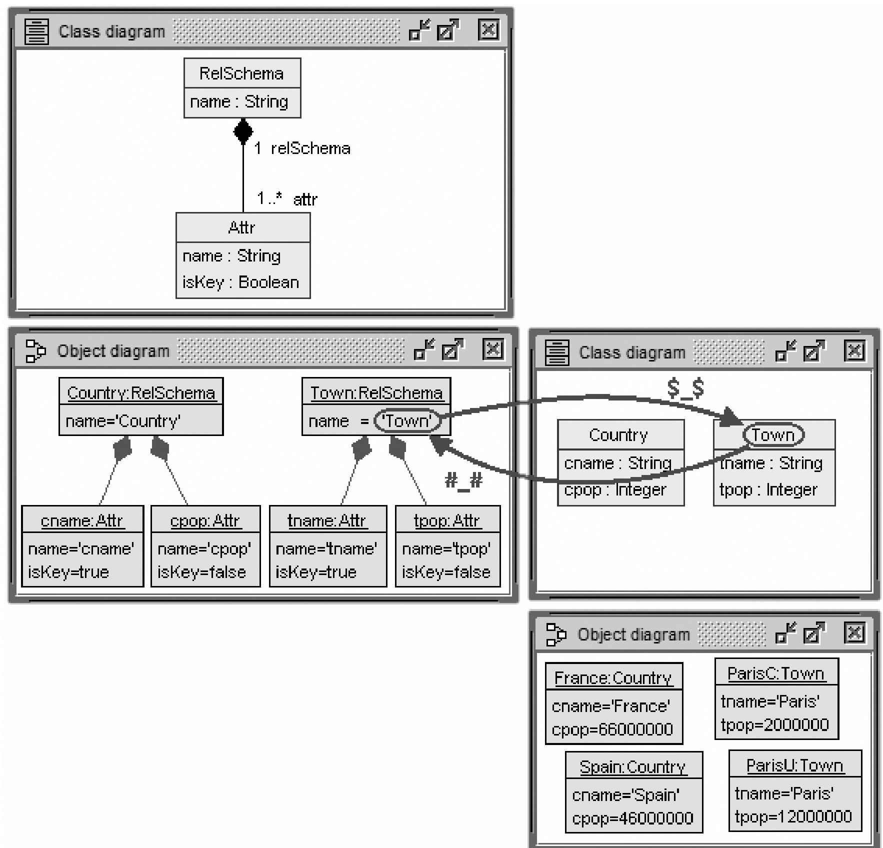


Fig. 4. Operations between Multi-Levels.

4 Conclusion

This paper is based on the idea to describe different metamodels in one model and to connect the metamodels either (a) with generalizations and associations employing appropriate UML and OCL constraints or (b) with special operations allowing to navigate between different multi-levels in a model. The paper does not claim to present final results, but some experiments and ideas in the context of multi-level models.

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, as has been sketched in the second approach in this paper. 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., Gerbig, R., Kühne, T.: A Unifying Approach to Connections for Multi-Level Modeling Foundations. In: Proc. Int. Conf. Models (MODELS'2015). (2015)
4. Atkinson, C., Gerbig, R., Kühne, T.: Opportunities and Challenges for Deep Constraint. In: Proc. Int. Workshop OCL and Textual Modeling (OCL'2015). (2015)
5. Atkinson, C., Grossmann, G., Kühne, T., de Lara, J., eds.: Proc. MODELS'2014 Workshop on Multi-Level Modelling (MULTI'2014). Volume 1286 of CEUR Workshop Proceedings., CEUR-WS.org (2014)
6. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software **20**(5) (2003) 36–41
7. Bézivin, J.: On the Unification Power of Models. Software and System Modeling **4**(2) (2005) 171–188
8. Clark, T., Gonzalez-Perez, C., Henderson-Sellers, B.: A Foundation for Multi-Level Modelling. [5] 43–52
9. de Lara, J., Guerra, E.: Deep Meta-modelling with MetaDepth. In Vitek, J., ed.: TOOLS (48). Volume 6141 of LNCS., Springer (2010) 1–20
10. Gogolla, M.: Employing the Object Constraint Language in Model-Based Engineering. In Gorp, P.V., Ritter, T., Rose, L., eds.: Proc. 9th European Conf. Modelling Foundations and Applications (ECMFA 2013), Springer, LNCS 7949 (2013) 1–2
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming **69** (2007) 27–34

12. Gogolla, M., Favre, J.M., Büttner, F.: On Squeezing M0, M1, M2, and M3 into a Single Object Diagram. In et al., T.B., ed.: Proc. MoDELS'2005 Workshop Tool Support for OCL and Related Formalisms, EPFL (Switzerland), LGL-REPORT-2005-001 (2005)
13. Gogolla, M., Sedlmeier, M., Hamann, L., Hilken, F.: On Metamodel Superstructures Employing UML Generalization Features. In Atkinson, C., Grossmann, G., Kühne, T., de Lara, J., eds.: Proc. Int. Workshop on Multi-Level Modelling (MULTI'2014), http://ceur-ws.org/Vol-1286/, CEUR Proceedings, Vol. 1286 (2014) 13–22
14. Guerra, E., de Lara, J.: Towards Automating the Analysis of Integrity Constraints in Multi-Level Models. [5] 63–72
15. Hamann, L., Gogolla, M.: Endogenous Metamodeling Semantics for Structural UML 2 Concepts. In Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P.J., eds.: MoDELS. Volume 8107 of LNCS., Springer (2013) 488–504
16. 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
17. Igamberdiev, M., Grossmann, G., Stumptner, M.: An Implementation of Multi-Level Modelling in F-Logic. [5] 33–42

Aspect-oriented Concrete Syntax Definition for Deep Modeling Languages

Colin Atkinson[1] and Ralph Gerbig[1]

University of Mannheim
{atkinson, gerbig}@informatik.uni-mannheim.de

Abstract. Multi-level modeling tools provide inherent support for modeling domain scenarios with multiple classification levels. However, as the success of domain-specific modeling tools illustrates users increasingly expect to be able to visualize models using domain-specific languages. It is relatively straightforward to support this using traditional "two-level" modeling technologies, but many of the benefits of multi-level modeling would be lost. For example, in a multi-level context it is not only desirable to define concrete syntax that is applicable over more than just one instantiation level, it should also be possible to customize the visualization of model elements as they become more specialized over instantiation and inheritance levels. In this paper we present an approach for multi-level concrete syntax definition which addresses this need by using aspect-oriented principles to parametrize the visualization associated with model elements. We also explain how this is implemented in the Melanee deep modeling tool.

Keywords: aspect-orientation; deep modeling; concrete syntax

1 Introduction

Since concrete syntax definition is one of the core foundations of domain-specific, model-driven development a large number of tools supporting this capability are available today. Important examples include MetaEdit+ [17] and the Graphical Modeling Framework [11] for graphical languages, XText [9] and Spoofax [12] for textual languages and EMF Forms [8] for form-based languages. Other concrete syntax formats can also be supported such as table-based and wiki-based languages. However, these tools are all based on traditional "two-level" modeling technology which only supports two classification levels. As a result, they have no inherent capability to support the definition of concrete syntax that can "span" (i.e. automatically be applied over) multiple classification levels. In other words, they can only inherently support the application of a concrete syntax definition to instances at the level immediately below. To apply a concrete syntax to two or more levels below its definition, complex transformation and editor generation/deployment steps are needed. This not only complicates the initial defintion and use of concrete syntaxes, it significantly increase the effort involved in maintaining and evolving them.

Deep modeling has the potential to address this problem because it inherently supports multiple classification levels. However, some challenges need to be addressed to support effective level-spanning concrete syntax definition and application within a deep modeling environment. The most significant is to find a suitable tradeoff between the need to define common syntax elements that are suitable across multiple levels whilst providing the flexibility to customize the common syntax elements as and where needed. In other words, the most significant challenge is to allow the concrete syntax used to visualize model elements to reflect the specialization that inherently takes place in the instantiation and inheritance hierarchies in a deep model.

The Melanee [3] deep modeling environment addresses this problem through an aspect-oriented approach which essentially allows concrete syntax definitions to be parametrized. Using this capability it is possible for the concrete syntax applied to a model element to be customized to reflect the specialization that naturally takes place in a deep model, significantly reducing the complexity involved in defining, maintaining, and evolving different concrete syntaxes. The implemented mechanism paves the way towards general concrete-syntax repositories which can be configured for particular domains and application scenarios in a simple and straightforward way.

The remainder of this paper is structured as follows: In the next section (Section 2) the underlying deep modeling approach is introduced. Section 3 then introduces the concept of aspect-oriented concrete syntax definition while Section 4 presents pragmatic observations made during the use of the approach in three different domains. The paper closes with conclusions (Section 5).

2 OCA-based Deep Modeling

The most widely implemented deep modeling architecture is the orthogonal classification architecture shown schematically in Figure 1. Its most obvious difference to the traditional linear modeling infrastructure of the UML is that there are two "orthogonal" classification dimensions. One dimension, the linguistic classification dimension, is represented by the vertical stack of levels labeled L_2 to L_0, and the other dimension, the ontological classification dimension, is represented by the horizontal stack of levels labeled O_0 to O_2.

The linguistic levels essentially capture the organization of the model information from the perspective of how deep modeling is realized in a meta-modeling framework such as EMF. The top most linguistic level (L_2), contains all language constructs available in the deep modeling language, while the middle level (L_1) contains the user-modeled domain languages and is thus often referred to as the domain level. The ontological dimension, in turn, consists of "ontological" levels which are stacked horizontally within L_1. Each model element at this level is classified by exactly one model element at level L_2, indicated by vertically dashed classification lines. The lowest level, L_0, contains the real-world concepts modeled by L_1. The light bulb represents the concept of *EmployeeType* which captures the common properties of all employees. The group of stick-men with a wrench

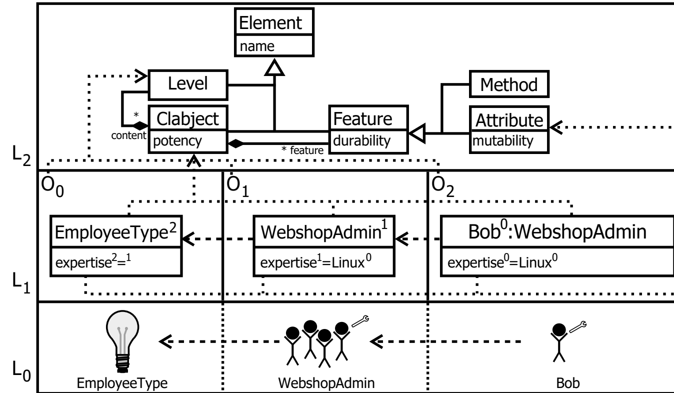


Fig. 1. The orthogonal classification architecture.

in the upper right instantiates this concept as *WebshopAdmin* (a particular type of employee) which is then further instantiated as *Bob* (a particular instance of a *WebshopAdmin*). Classes located in the middle levels (e.g., *WebshopAdmin*) are instances of the classes at higher levels and types for the classes at the lower levels. Hence, they are named "clabjects" which is a concatenation of the terms "class" and "object".

The domain level, L_1, consists of three ontological levels O_0 to O_2 which are intended to represent the domain concepts/objects. Even though three levels are shown here, the number of levels available is not limited. The concrete syntax used in Figure 1 looks similar to the UML but with some modifications to make it level-agnostic. Ontological classification is shown using horizontal dashed lines or the UML colon notation as used in object specifications (e.g. *Bob*). The most important difference to the UML is the association of numerical values with clabjects, attributes and attribute values in the form of superscripts after their names. Depending on whether they are associated with clabjects, attributes or attribute values they are respectively referred to as "potency", "durability" or "mutability". The potency specifies over how many subsequent ontological classification levels a clabject can be instantiated and thus influence the contents of a deep model. In the example *EmployeeType* has a potency of two stating that it can be instantiated at the following two levels, here *WebshopAdmin* with potency one and *Bob* with potency zero. The durability displayed next to the name of an attribute indicates over how many subsequent instantiation steps an attribute endures. In the example, *expertise* has a durability of two so all offspring of *EmployeeType* over the next two levels have to possess such an attribute, here *WebshopAdmin* and *Bob*. Finally, the mutability displayed next to the value of an attribute defines over how many levels the value of an attribute can be changed. The mutability for *expertise* of *EmployeeType* is one, hence it can be changed at the next instance level and is fixed from then on. On level O_1 of the example the *expertise* for all *WebshopAdmins* is set to *Linux* which means that instance *Bob* has to have *Linux* as his *expertise*.

3 Apsect-oriented Concrete Syntax Definition

Aspect-oriented concrete syntax definition employs concepts from aspect-oriented programing languages [14], e.g. AspectJ [13]. More specifically it allows concrete syntax definitions to be parametrized using uniquely-named join-points. Aspects can then be used to contribute to a join-point to customize the notation used to visualize a specialized version of the associated model element. An aspect contains the name of the join-point to which it is applied, a condition determining when the aspect is applied and an "advice" which is introduced to the join-point. The advice can be configured to be added before, after or to replace the join-point. Moreover, multiple aspects can be provided for one join-point.

The meta-model supporting the aspect-oriented definition of graphical concrete syntax in Melanee is shown in Figure 2. The model is unique for each type of concrete syntax format (e.g. graphical, textual, tabular etc.) to best fit the needs of that particular format. Here, for space reasons, the approach is explained only in the context of graphical concrete syntax definition, but Melanee supports the approach for other formats as well. Melanee's algorithm for visualizing model elements is designed in such a way that it can be configured to work with any concrete syntax definition model that supports the join-point and aspect concepts explained below.

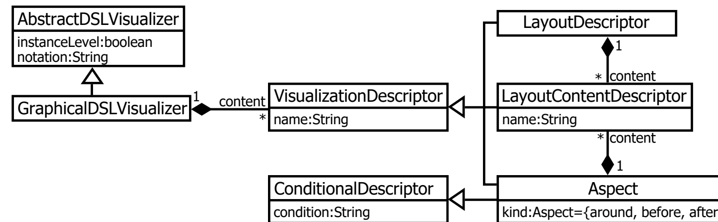


Fig. 2. The graphical concrete syntax definition meta-model.

In Melanee any model element can have multiple *AbstractDSLVisualizers* attached, defining the concrete syntax in a specific concrete syntax format. The *notation* attribute groups visualizers from the same format into "notations" so that families of symbols can be grouped into a given logical "notation". The *instanceLevel* attribute specifies whether a visualizer is applied only to the instance level or to the level at which the visualizer is defined as well. The *GraphicalDSLVisualizer* shown here defines concrete syntax in a graphical format using *VisualizationDescriptors*. This is the base class for all types of elements in a concrete syntax definition. It assigns a name to each element of a concrete syntax definition making each of those potential join-points. To explicitly declare an element as a join-point a unique name has to be assigned to it. A concrete syntax definition consist of three types of *VisualizationDescriptors*: *LayoutDescriptor*, *LayoutContentDescriptor* and *Aspect*. A *LayoutDescriptor* describes the layout which is applied to layout content (e.g. table layout, flow layout etc.). A *LayoutContentDescriptor*, which is contained by a *LayoutDescriptor*, describes the actual displayed

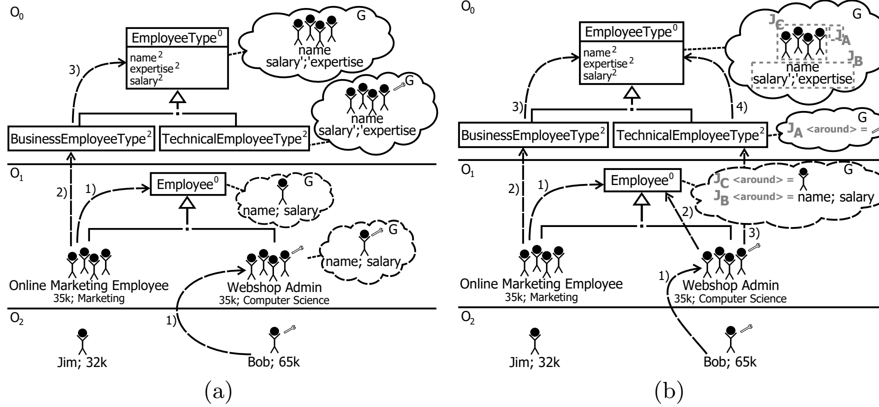


Fig. 3. Visualizer search traces: (a) not aspect-oriented, (b) aspect-oriented.

shape such as a rectangle, circle, label mappings etc. By nesting these two concepts any concrete syntax can be defined as desired. The language is designed to be similar to well known languages for describing UIs such as the Standard Widget Toolkit (SWT), and languages describing concrete syntax for diagrams such as the Graphical Modeling Framework (GMF).

Aspects for join-points are provided by the *Aspect* model element which inherits the *condition* attribute from *ConditionalDescriptors* which are executed when the *condition* holds true. *Aspects* are not only *ConditionalDescriptors* they are also subclasses of *VisualizationDescriptor* so that they can be added to *GraphicalDSLVisualizers*. The *kind* attribute defines the application strategy of the advice (*content*) of the *Aspect*. Three kinds are available – *before* (adding the content before), *after* (adding the content after) and *around* (replacing the content of the join-point). The current meta-model limits the *content* of an aspect to be *LayoutContentDescriptors*. However, this does not limit the expressive power of the model.

Based on the visualizer and aspect-oriented concrete syntax definition approach a special search algorithm is employed to generate the desired visualization for each model element in a deep model. This is based on an algorithm for two-level models [10]. It was first extended to deep modeling in [2] and since then steadily refined to the point where it supports deep aspect-oriented visualization.

Figure 3 shows an example of the "Employee" language with concrete syntax attached to model elements indicated by clouds. The concrete syntax definition uses the meta-model of Figure 2. The order in which clabjects are visited when searching for a model element visualization is indicated by dashed arrows annotated with a number. The search algorithm starts searching at the level of the clabject to visualize. First the clabject itself is visited to determine whether it has an associated visualizer, if not the clabjects in the inheritance hierarchy of the clabject are visited. If no visualizer is found at this level the types of the clabject at the level above are visited. The search continues until a suitable visualizer is found for the clabject. If none is found the pre-defined concrete syntax is used to render the clabject.

The example in Figure 3(a) shows the "aspect-unaware" search trace of the algorithm for *Bob* and *Online Marketing Employee*. *Bob* has no visualizer attached, so the algorithm visits *WebshopAdmin*, the type of *Bob*, which has a visualizer attached. The algorithm stops, returning the stick man icon with a small wrench in the upper right. To find a suitable visualizer for *Online Marketing Employee* its supertype (*Employee*) is visited which contains a visualizer. However, because its *instanceLevel* attribute is set to true (indicated by the dashed cloud) the algorithm continues searching the type level. The direct type, *BusinessEmployeeType*, has no visualizer but its supertype (*EmployeeType*) has. Hence, the search terminates and returns the group-of-stickmen icon to visualize *Online Marketing Employee*.

Figure 3(b) shows the same example but using aspect-oriented concrete syntax definition. Only one visualizer is defined at *EmployeeType* defining the join-points (dashed rectangles) J_A, J_B and J_C. Variations are modeled through aspects of kind *around* along the classification hierarchy, here at *TechnicalEmployeeType* and *Employee*. This focus on modeling variations only reduces the number of fully specified Visualizer from four in Figure 3(a) (*EmployeeType*, *BusinessEmployeeType*, *Employee*, *WebshopAdmin*) to one in Figure 3(b) (*EmployeeType*). The main difference between the two search algorithm versions is that discovered aspects are collected on the way to a model element with an aspect-less visualizer and then merged into the visualizer. The trace for *OnlineMarketingEmployee* is identical to the aspect-unaware version as no aspects are defined along its classification hierarchy. For *Bob* the search ends at *EmployeeType* and merges the collected aspects J_A, J_B and J_C into this visualizer.

Comparing the "aspect-unaware" and "aspect-aware" approaches shows that the search algorithm traverses more model elements when applying the aspect-aware variant of the algorithm. Additionally, the aspect aware version involves additional computational overhead for merging aspects into join-points, leading to a potential performance issue. The impact of this overhead has to be empirically determined by analyzing numerous practical examples. In the following section pragmatic observations about the approach are made.

4 Examples

Because of the novelty of the aspect-oriented concrete syntax definition approach only three languages using this features have been published to date. The first describes executable behavior of robots [5], the second a language for modeling executed environments in a game [1] and the third a distributed extension scenario in the domain of business process modeling and analysis [4].

The model shown in Figure 4 is part of a deep robot behavior modeling and execution framework in [5]. The excerpt focuses on the behavior modeling part of the model. At the highest level, O_0, the goal is to provide a generic language for modeling robot behavior. This is then used to create a robot behavior modeling language for a specific type of robot at O_1, here humanoid robots. One of the goals is to make the concrete syntax as configurable as possible to reflect the type of modeled behavior. For example, the *Move* action should show the coordinates alongside a small walking pictogram, while the *Gesture* action should show a

textual description of the gesture to do alongside a corresponding pictogram. To achieve this the concrete syntax definitions at the O_0 level offer join-points (J_A, J_B, J_C) which are used by the robot behavior language at O_1 for concrete syntax customization. The language is then used at level O_2 to model robot behavior. The behavior modeled in this example first instructs the robot to walk, then detect a ball and finally make an agree or disagree gesture.

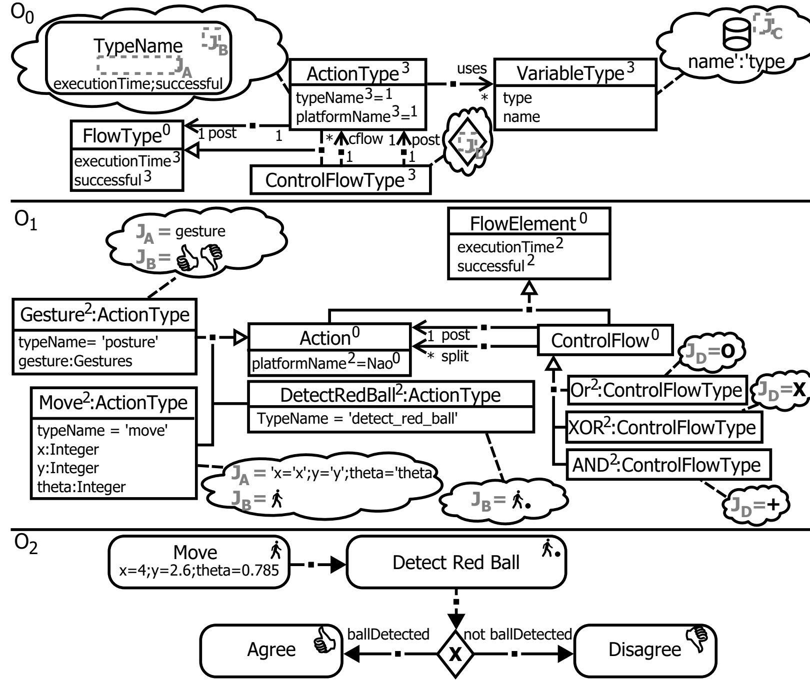


Fig. 4. Robot behavior modeling language after [5].

To realize the language without aspect-oriented concrete syntax definition features it would be necessary to define a complete visualizer for each model element at level O_1 (e.g. *Move*, *XOR*) even though the concrete syntax at level O_1 is only a slight variation of the one defined at O_0. This unnecessary replication of visualizers would significantly increase the accidental complexity involved in developing, maintaining and evolving deep models [6, 7]. A change made to the general language at level O_0 would need to be synchronized with all visualizer definitions at level O_1. Aspect-oriented concrete syntax definitions avoid this problem by making it possible to specify a generic concrete syntax definition on level O_0 that can be customized at level O_1. At level O_1 the variations are modeled only and changes to the generic syntax at level O_0 are automatically promoted to instances without any manual effort. This reduces the number of manual changes to the concrete syntax definitions that have to be made after a change and thus reduces the overall complexity involved.

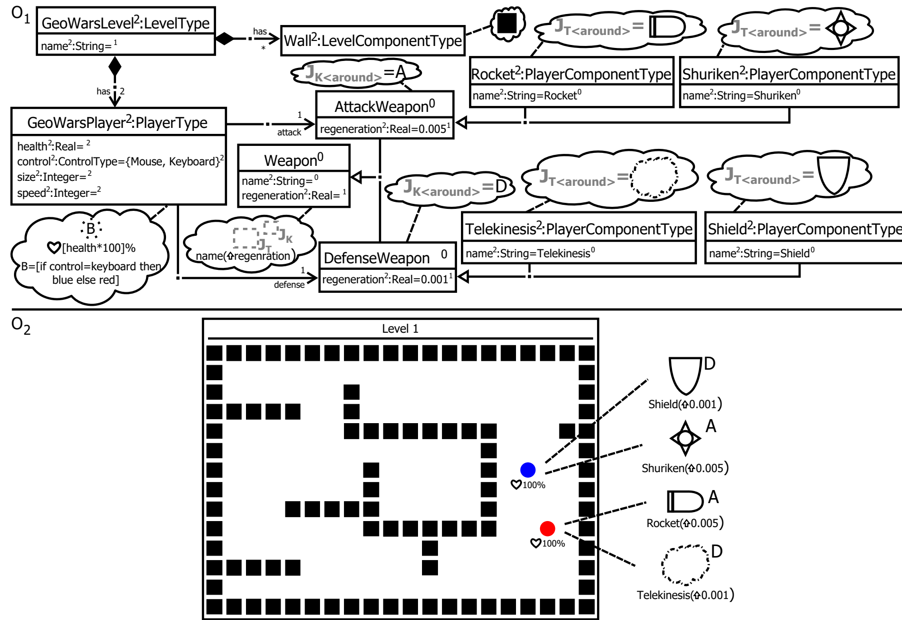


Fig. 5. Game modeling language after [1].

In contrast to the previous example where the customization takes place along the classification hierarchy, in Figure 5 aspect-orientation is used to model concrete syntax variations in an inheritance hierarchy. The example shows a model for a game environment with a focus on how the inheritance hierarchy is used to determine the concrete syntax of *Weapons* in an aspect-oriented style. At the center of each *Weapon* symbol is an icon indicating the type of weapon (e.g. *Shuriken*). The character at the top right of the symbol indicates the weapon kind (A: *AttackWeapon*; D: *DefenseWeapon*) and the text at the bottom gives the name and details of the weapon such as *regeneration* information. The only common part of all weapon visualizations is the information at the bottom. The weapon kind and type change per subclass. To optimize the concrete syntax definition, only the weapon details at the bottom are defined in the visualizer associated with *Weapon* along with two join-points — one for weapon type (J_T) and one for weapon kind (J_K). The weapon kind is associated with the *AttackWeapon* and *DefenseWeapon* subclasses since it is fixed for all of their subclasses. On the other hand, the weapon types which change for each weapon are defined locally at *Telekinesis*, *Rocket*, etc. The lowest level O_2 shows an instance of a modeled game environment which can be executed. Again the aspect-oriented concrete syntax definition reduces the number of elements involved in concrete syntax definition as only variations are modeled instead of the whole concrete syntax.

In the examples presented so far, aspect-oriented concrete syntax definition is used to reduce accidental complexity. Figure 6, however, shows a scenario that cannot be supported without this mechanism. It contains three packages created by independent parties and distributed over the internet — 1. *BPMN*:

providing a business process modeling language, 2. *MDPE*: supporting model-driven performance analysis and 3. *Security*: supporting business process security modeling.

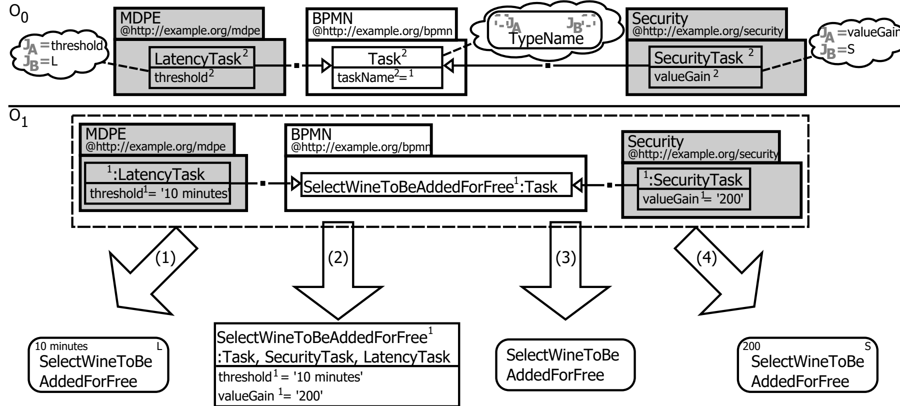


Fig. 6. Distributed business process modeling language after [4].

The *BPMN* language was defined with modeling language and concrete syntax extension in mind. To enable the latter, join-points J_A and J_B are defined in the concrete syntax definition of *Task*. These two are used by the *LatencyTask* and *SecurityTask* subclasses from the *MDPE* and *Security* packages linked to the *BPMN* package. They introduce new language constructs into the *BPMN* language and extend the concrete syntax. The rectangle at O_1 contains a detailed view of one *Task* which is an instance of *Task*, *LatencyTask* and *SecurityTask* for demonstration purposes. In a real model, however, this would be shown as one model element with multiple classifications. The four rendering options for this model instance, which it is possible to switch between on-the-fly, are shown at the bottom of Figure 6. This example shows how distributed deep models can enhance each others concrete syntax definitions in a decoupled style by using aspect-oriented concrete syntax definition. Even though the example only shows a linking depth of one, chains of linked models which contribute towards one concrete syntax can be envisaged.

5 Conclusion

Deep modeling technologies not only facilitate the definition of models that span more than one class/instance level they also allow domain-specific (i.e. user-defined) modeling languages to be used to represent them. These user-defined modeling languages often use concrete syntax definitions which span multiple classification levels, multiple inheritance levels or even multiple models. In this paper we have shown the potential of aspect-oriented concrete syntax definitions to reduce accidental complexity, positively impact model development, evolution and maintenance, and support new modeling scenarios. The three published languages and one running example described in this paper demonstrate the po-

tential advantages of the approach which is fully implemented in the Melanee tool. The authors are not aware of any other deep modeling tool supporting aspect-aware, concrete syntax definition. Besides Melanee, two other deep modeling tools explicitly support the definition of user-defined concrete syntax — MetaDepth [16], a textual tool, and DPF [15], a graphical tool. Both provide concepts for concrete syntax definition but do not support aspect-orientation.

References

1. Atkinson, C., Gerbig, R.: On the execution of deep-models. Exe 2015 – 1st International Workshop on Executable Modeling Proceedings (2015)
2. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. Software Engineering, IEEE Transactions on 35(6) (2009)
3. Atkinson, C., Gerbig, R.: Melanie: Multi-level modeling and ontology engineering environment. In: 2nd International Master Class on Model-Driven Engineering: Modeling Wizards. pp. 7:1–7:2. ACM, New York, NY, USA (2012)
4. Atkinson, C., Gerbig, R., Fritzsche, M.: A multi-level approach to modeling language extension in the enterprise systems domain. Information Systems (2015)
5. Atkinson, C., Gerbig, R., Markert, K., Zrianina, M., Egurnov, A., Kajzar, F.: Towards a deep, domain specific modeling framework for robot applications. pp. 4–15. MORSE '14, CEUR-WS.org (2014)
6. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software & Systems Modeling 7(3), 345–359 (2008), http://dx.doi.org/10.1007/s10270-007-0061-0
7. Brooks, F.P., J.: No silver bullet essence and accidents of software engineering. Computer 20(4), 10–19 (1987)
8. Eclipse Foundation: Emf forms. http://eclipse.org/ecp/emfforms/index.html (2015)
9. Efftinge, S., Völter, M.: oaw xtext: A framework for textual dsls. In: Workshop on Modeling Symposium at Eclipse Summit. vol. 32 (2006)
10. Espinazo-Pagán, J., Menárguez, M., García-Molina, J.: Metamodel syntactic sheets: An approach for defining textual concrete syntaxes. ECMDA-FA '08, Springer-Verlag, Berlin, Heidelberg (2008)
11. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional, 1 edn. (2009)
12. Kats, L.C., Visser, E.: The spoofax language workbench: Rules for declarative specification of languages and ides. OOPSLA '10, ACM, New York, NY, USA (2010)
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of aspectj. In: Knudsen, J. (ed.) ECOOP 2001 (2001)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented Programming. In: Akit, M., Matsuoka, S. (eds.) ECOOP'97 (1997)
15. Lamo, Y., Wang, X., Mantz, F., Bech, O., Sandven, A., Rutle, A.: Dpf workbench: A multi-level language workbench for mde. In: Proceedings of the Estonian Academy of Sciences, 2013. vol. 62, pp. 3–15 (2013)
16. de Lara, J., Guerra, E.: Deep meta-modelling with metadepth. TOOLS'10, Springer-Verlag, Berlin, Heidelberg (2010)
17. Tolvanen, J.P., Kelly, S.: Metaedit+: Defining and using integrated domain-specific modeling languages. OOPSLA '09, ACM, New York, NY, USA (2009)

Exploring Multi-Level Modeling Relations Using Variability Mechanisms

Iris Reinhartz-Berger[1], Arnon Sturm[2], and Tony Clark[3]

[1]Department of Information Systems, University of Haifa, Israel
[2]Department of Information System Engineering, Ben-Gurion University of the Negev, Israel
[3] Department of Computing, Sheffield-Hallam University, UK

iris@mis.hevra.haifa.ac.il, sturm@bgumail.bgu.ac.il, t.clark@shu.ac.uk

Abstract. Over the last decade multi-level modeling (MLM) approaches have been addressing the need for relaxing the strict constraints on intra- and inter-layer type-instance relationships that are imposed by traditional approaches to meta-modeling. In this paper we explore MLM approaches in the context of Software Product Line Engineering (SPLE), propose a meta-language, and show how it can represent three commonly used variability mechanisms - configuration, parameterization, and template instantiation - within the context of MLM. By this we contribute to simplifying the representation of complex relationships in current MLM approaches and to the formal definition of SPLE variability mechanisms utilizing MLM concepts.

1 Introduction

Promoting models as the primary artifacts in software development, early approaches to Model-Driven Engineering (MDE) refer to four layers of abstraction: data (M0), model (M1), meta-model (M2), and meta-meta model (M3) where elements of Mn are instances of types defined at level Mn_{+1}. Of those layers, three are practically used by MDE practitioners: M2 to build metamodels for general purpose modeling languages (*e.g.*, UML) or Domain Specific Modeling Languages (DSML), M1 to instantiate those metamodels in the form of models, and M0 to process instances of the models. Together these three levels are called a 'golden-braid' [11] and traditional approaches to modeling have used two occurrences (M0-M2 and M1-M3) whilst maintaining strict separation between elements of different levels. Furthermore, most traditional approaches follow a similar definition of the type-instance relationship between levels whereby, for example, an object at M0 structurally conforms to a class at M1 when all the object's properties have names and values corresponding to the names and types defined by the class attributes.

Recently, multi-level modelling (MLM) approaches aim to extend this architecture by enabling modelling at an arbitrary number of levels [12], and propose variations and liberalizations of the traditional type-instance relationship between levels. Current MLM approaches relax these limitations so that multiple golden-braid occurrences

can co-exist within the same model to support an arbitrary number of levels. Type-instance relationships in these cases can be expressed via concepts, such as clabjects [2], power types [1521], and deep meta-modeling [24].

Software Product Line Engineering (SPLE) [10, 22] can be considered as dealing with multi-levels. SPLE uses families of products and subsequent adaptation of each family to produce a single model describing a particular product in the line. Families represent multiple variations in a single definition and adaptation involves the selection of choices amongst the variations and potentially modification and extension of these choices. The different levels handled in SPLE can be classified into domain and application engineering [19]. The *domain engineering* layer includes activities and tasks to create and handle core assets, *i.e.*, artifacts that are intended to be (re)used by more than one product in the family. These assets include both common and variable parts. The *application engineering* layer consists of activities and tasks to adapt and tailor core assets in order to satisfy particular requirements of the products at hand. The reuse between the layers is done systematically applying mechanisms, commonly termed variability or reuse mechanisms [5, 17]. Examples of such mechanisms are configuration, parameterization, and template instantiation.

In this paper, we aim to explore the potential relations between the two worlds – SPLE and MLM. Our hypothesis is that the elements in the SPLE families correspond to *types* and the elements in the particular products correspond to *instances* of types. The relationships derived from the variability mechanisms may refine the type-instance relationship that is variegated and liberalized by current MLM approaches. Thus, the contribution of the paper has two aspects. To MLM, SPLE variability mechanisms can be used to simplify the representation of complex relationships in current MLM approaches, and to SPLE – MLM concepts can be used to more precisely define SPLE variability mechanisms.

In the sequel, Section 2 reviews variations of and extensions to the type-instance relationship in existing MLM approaches. Section 3 discusses commonly used SPLE variability mechanisms – configuration, parameterization, and template instantiation – demonstrating their use. In Section 4 we discuss the ability to identify the relations derived from the variability mechanisms as being special cases of the type-instance relationship. Finally, in Section 5 we conclude and set the ground for future research.

2 Related Work

Recent modeling language research has addressed the limitations of traditional strict-modelling approaches that impose boundaries between elements from different levels of the golden-braid. Several researchers have begun to explore variations on the type-instance relation that is intrinsic to most modeling languages and to relax traditional strictness with the aim of providing a richer notion of 'type' within a model. This section reviews the current advances, broadly categorized as 'multi-level modeling' (MLM), towards this aim.

MLM approaches have been suggested for a variety of uses, including designing models for the use of non-modelers [14] and developing modeling tools [16]. In many cases there is a requirement to allow types and their instances to co-exist, for example

enactment [16] of business processes requires the type of a process and its active instance to be co-represented.

Traditional approaches to the type-instance relationship often focus on classes and their instances. UML, for example, defines that an instance is related to a type by 'instance-of' when the class structurally and behaviorally defines all features of the instance. The relationship may hold between a class and an object or a meta-class and a class. The semantics of a language is usually defined in terms of the 'instance-of' relationship both by intrinsic rules that hold and by (at least in UML) constraints that are expressed by the modeler and attached to the type. Therefore, researchers addressing issues related to model-based language engineering provide mechanisms that affect the intra- and inter-level type-instance relationships.

To address the dual role that a certain element plays in MLM approaches (as an instance of the higher level and a type of the lower level), Atkinson [2] coined the term *clabjects* to simultaneously refer to classes and objects. He further suggested a potency-based multi-level approach to support deep meta-modeling [3]. Following that approach, each element is assigned with a potency number, which indicates the number of levels in which the element can be instantiated. A special case of potency is the intrinsic features, suggested by Frank [13], in which the potency number is one.

Observing that an element may be an instance of two elements residing in different levels, Atkinson and Kuhne [34] suggest the notion of orthogonal classification architecture (OCA). In OCA, elements can be instantiated along the linguistic dimension and orthogonally along the ontological dimension. The linguistic dimension refers to instantiation across levels whereas the ontological dimension refers to instantiation within levels.

Another approach to MLM that does not involve mixing elements from different type levels is to use a pattern to encode the relationship between a meta-class and the class that acts as its instance. Gonzalez-Perez and Henderson-Sellers [15] utilize the notion of power types [21] by introducing the concept of a power type pattern which exists at a single level but represents elements from different levels. The power type pattern is defined as "a pair of classes in which one of them (the power type) partitions the other (the partitioned type) by having the instances of the former be subtypes of the latter." This allows the modeler to capture the semantics of the type-instance within a strict-modelling framework.

To support information integration in heterogeneous information systems, Jordan *et al.* [18] suggested modelling primitives that extend standard specialization and instantiation mechanisms. Particularly, they distinguish between specialization by extension (that supports adding attributes, associations, or behavior) and specialization by refinement (that supports adding granularity to the description), as well as between standard instantiation (in which all attributes must be assigned a value from their domain) and instantiation with extension (which enables adding attributes, behavior, and so on). They further introduce the subset by specification relation for representing "the existence of a class of specification construct that identifies particular subtypes of another type", the membership relation, and the specification by enumeration relation that describes how the extensions of sets of entities are related.

In summary, the need for flexible modeling technologies has led a collection of researchers to seek ways to relax traditional strict-modelling and to open up the 'type-instance' semantics to the modelers. Most recent advances have sought to mix types and instances and have allowed structural features to be annotated in order to influence their instantiation semantics. We see that there is a similarity with the aims of SPLE variability mechanisms, as they require mixed type-levels and offer control of instantiation through variability.

3 Exploring Alternatives for Relationships in MLM

As noted, frameworks of SPLE commonly distinguish between domain and application engineering (*e.g.*, [19], [23]). In both layers, modeling plays a central role to analyze and design artifacts. As an example, consider the model depicted in Fig. 1. The left part of the model describes a library management system (LMS) or a family of such systems. Respectively, this level can belong to the application or domain engineering layers. The model refers to books (titles) written by authors, as well as to the actual book copies that can be checked out and have (physical) location. A book may have up to n copies.

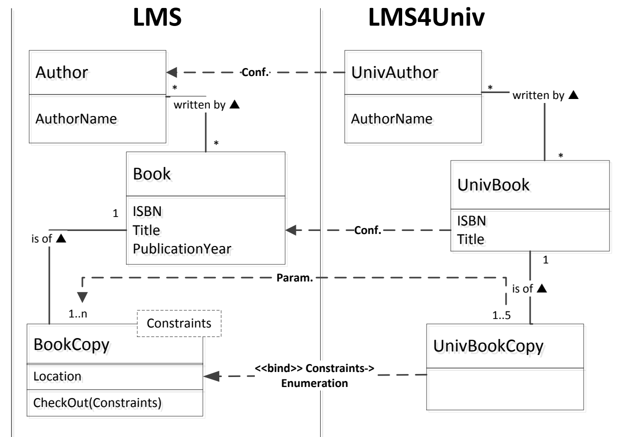


Fig. 1. The LMS example

The second level in our example, partially described in the right part of Fig. 1, refers to library management system(s) in universities (LMS4Univ for short). Here again, the level can be interpreted as specifying a (narrower) domain or an application, and as such it can be considered as an adaptation of the LMS model. Checking-out book copies is constrained according to the (enumerated) user type, *e.g.*, student *vs*. staff, and a book may have up to 5 copies (for economic reasons). In LMS4Univ books have no publication year information.

Note that the two levels depicted in Fig. 1 could be part of a larger chain of levels that includes, for example, a wider domain of Check-In Check-Out applications, or sub-classes of LMS4Univ applications.

For traversing from a higher level to a lower level in SPLE, different variability mechanisms are commonly utilized. These are actually techniques applied to adapt core assets developed in the domain engineering layer to the context of particular products (*i.e.,* artifacts in the application engineering layer). However, they can be used for adapting domains or applications as well. Over the years, different variability mechanisms have been suggested for different development stages, *e.g.*, [1] for implementation, [5], [17] for architecture design, and [6], [7] for reference modeling. We mention here only three common variability mechanisms (see Table 1 for definitions and demonstration of their use through the LMS example):

- In *configuration*, elements of the higher level are selected to be included in the lower level. Partial selections are possible, as in the case of UnivBook which selects only 2 out of the 3 attributes of Book: ISBN and Title.
- *Parameterization* supports assigning values to parameters defined in a higher level. The assignment is done in a lower level. In our example, the maximal number of copies of a certain book is assigned to 5 in LMS4Univ.
- *Template instantiation*, which, in contrast to parameterization that deals with value assignment, deals with type adaptation, is exemplified by constraining book copy check out with enumeration type (which represents user type, *e.g.*, student vs. staff).

Table 1. Common variability mechanisms, their definitions, and use

Variability Mechanism	**Definition**	**Example**
Configuration	Choice between alternative functions and implementations [17]; Modifying selected elements of a core asset based on predefined rules that refer to specific requirements or situations [6, 7].	UnivBook in LMS4Univ (with respect to Book in LMS).
Parameterization	Variation points for features [17]; Data items serving as arguments for distinguished software behavior [5].	Up to 5 UnivBookCopies in LMS4Univ (with respect to up to n BookCopy's in LMS).
Template Instantiation	Type adaptation or selecting alternative pieces of code [17]; Enables filling in product-specific parts in a generic body [5].	UnivBookCopy in LMS4Univ (with respect to BookCopy in LMS).

Next, we explore to what extent the aforementioned mechanisms can be represented by the type-instance relationship and its variations and by this – contribute to simplifying the representation of complex relationships.

4 Variability Mechanisms and Type-Instance Relationships

Our claim is that configuration, parameterization, and template instantiation can be viewed as special cases of the type-instance relationships in the context of MLM. To expand this claim, we first provide a core meta-language that supports our MLM approach.

Consider the simple example shown in Fig. **Error! Reference source not found.**. It uses a MLM approach to model both parametric (as manifested by parameterization or template instantiation) and configurable classes. This is done via the relationship 'of' that works consistently in all cases that are shown. A class defines constraints that must be satisfied by its instances. The relationship 'of' is a declarative statement that the object at its source satisfies the constraints of the class at its target. In the case of BookCopy, ParametricClass is used to model the type parameter used for the method CheckOut. In the case of the UnivBook, the class Book is a *family* specifying that publication year is an optional attribute of book. Therefore, our approach is based on a use of the type-instance relationship which is supported through the consistent use of constraints, the implication of which is a uniform representation for all model-elements. To achieve this *everything is an object* [16].

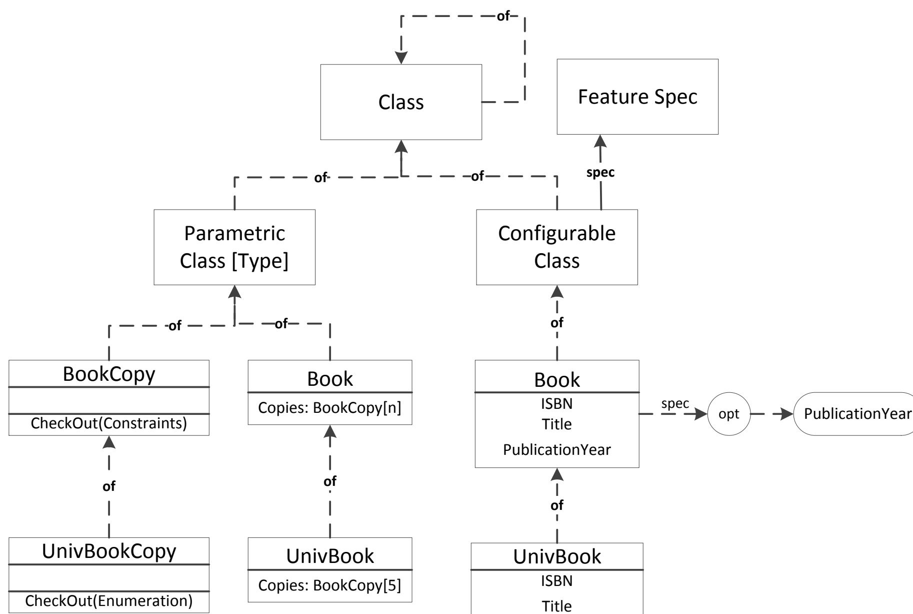


Fig. 2. Configuration, parameterization, and template instantiation expressed via type-instance relationship

Fig. 2 shows a model that contains a kernel meta-language and its extension to support configuration, parametrization, and template instantiation (the last two referred as parametric). Since the basis of our approach is a self-describing meta-language in which everything is an object, the root class is Object. Furthermore, alt-

hough only Class is shown as explicitly inheriting from Object, all classes in the language inherit from the root. To simplify the representation we consider only classes with attributes, hence the use of directed relationships between classes.

Since everything is an object it is possible to access the internal data of any model element through reflection via the attribute named ‘slots’ defined by Object. Inheritance is supported though the ‘parents’ attribute defined by Class and the implication is that all attributes are inherited. An important feature of the approach is defined using constraints on classes. A constraint is a predicate whose ‘check’ operation is supplied with a candidate object.

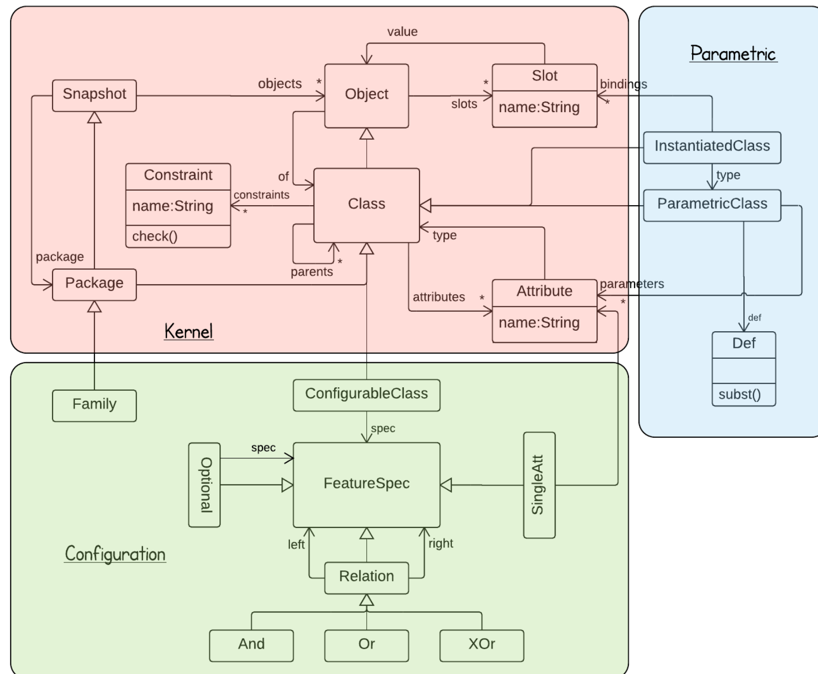


Fig. 2. A Kernel Language

A language definition relies heavily on constraints to specify the ‘of’ relationship that holds between a class and its instances. A key meta-circular constraint that is defined by Class can be paraphrased: ‘An object c is a valid class if it enforces all of its constraints c.constraints when checking o:c for any object o’. Clearly, this constraint allows Class to classify itself and, because of the universal object representation, the object ‘o’ could be a ground instance, a class, or a meta-class.

A snapshot is a container of objects; it can be used as the basis of a wide range of containers where specializations of Snapshot introduce constraints that must hold for the contents. A package is a container of classes (with the associated implied constraint on Package) and a snapshot links to a package that contains the classes that type the contained snapshot-instances. Again, this meta-circularity helps us to ensure that models are objects and that MLM principles work over all levels.

The ‘of’ relationship between a class and an object is defined by the constraints on the class. Since a class may be an extension of the base class Class, it is possible that the constraints used to define a particular occurrence of the relationship ‘of’ might use slot-value information other than those slots defined by Class. Therefore, the ‘of’ relationship can be *overloaded* by the language designer as described below.

Configuration can be expressed via a configurable class that specifies options such that an instance of the class has chosen consistently amongst the available options. Therefore an ‘of’ relationship can hold between a configurable class and its instances if the constraints on the class require the features of each instance to be consistent with the options of the class. Recall that ‘of’ may hold between a ground instance and a class, or a class and a meta-class. Therefore, we may define configuration at any level.

Fig. 2 also shows that configuration is supported via Family – an extension of Package with the implied constraint that a Family is a container of configurable classes. A configurable class defines feature-specifications that are Boolean combinations of attributes. The constraints on a configurable class require that any candidate instance be a class and that the features of the class be consistent with the specification. Therefore, configuration is modeled as a form of type-instance, where the constraints match attributes in the class against feature-specifications in the meta-class.

Parametric model elements define formal parameters or templates that range over element-definitions. When supplied with model elements as actual parameter values or types the formals are consistently replaced within the body of the definitions in order to produce new model elements. Note the term ‘consistently’: the new model elements view the parametric version as a type whose constraints must be satisfied.

Table 2. A MLM Representation for Variability Mechanisms

Variability Mechanism	MLM Instantiation Constraints and Examples
Configuration	*Checks that the structure of the instance is consistent with the variability specified by the type.* Fig. 1 shows LMS as a family with Book as an instance of ConfigurableClass with PublicationYear as an optional attribute. The class UnivBook is a type-consistent instance, and therefore a configuration of Book.
Parameterization	*Assigns a value to a type* ***to create an instance of that type***. Fig. 1 shows that LMS4Univ assigns the value 5 to the parameter n, appearing in LMS and specifying the maximal number of BookCopy associated to a single Book.
Template Instantiation	*Assigns a value to a type* ***to create another type***. The class BookCopy is parametric with respect to the parameter Constraints and the binding of Constraints to Enumeration is shown to produce the instantiated class UnivBookCopy (details of the definition are omitted).

Fig. 2 shows an extension of Kernel with features for parameterization and template instantiation. A parametric class has a collection of typed parameters and a definition. The definition ranges over all model elements and supports an operation ‘subst’ that is supplied with some bindings for the parameters and produces a collection of model elements via consistent substitution. An instantiated class is a normal class that is associated with some parameter bindings. A new constraint on an instan-

tiated class requires that the attributes of the class are consistent with the definition of its class after the bindings have been substituted. Therefore, parameterization and template instantiation are modeled as a form of type-instance, where the constraints match bindings against formal parameters and substitution into a body.

In Table 2 we demonstrate the use of the Kernel Language for applying the variability mechanisms.

5 Conclusions and Future Work

MLM approaches have been proposed in order to relax the traditional strictness requirements on inter- and intra-level type-instance relationships. However, while these proposals are formal, they address the representation of complex relationships to a limited extent. SPLE, on the other hand, distinguishes between different relationships, introducing a variety of variability mechanisms that are more intuitive to the modelers but are less formal in their definitions. The current paper addresses this tradeoff by expressing three key mechanisms to SPLE reuse – configuration, parameterization, and template instantiation – within a type-instance framework. We defined a simple MLM-based kernel-language to show that those mechanisms can be implemented within that framework and can co-exist with other meta-modeling techniques including potency, deep modeling, and power-types (see [8] for an example). This provides a feature-rich, integrated and consistent approach to model-based language engineering. The language used is a much-simplified version of the kernel for the XModeler toolkit [9]. The language is based on a uniform representation for model elements and can support a wide variety of languages that are both general-purpose and domain-specific. We plan to further develop the kernel language and test it in the context of SPLE and to use it as the basis of mixing different MLM approaches with SPLE variability mechanisms.

References

1. Anastasopoules, M. and Gacek, C. (2001). Implementing Product Line Variabilities. Proceedings of the 2001 Symposium on Software Reusability (SSR'01), pp. 109-117
2. Atkinson, C. (1997). Meta-Modeling for Distributed Object Environments. EDOC, pp. 90-101
3. Atkinson, C. and Kühne, T. (2001). The Essence of Multilevel Metamodeling. UML 2001, pp. 19-33
4. Atkinson, C. and Kuhne T. (2003). Model-driven development: A metamodeling foundation. IEEE Software, 20 (5), pp. 36–41
5. Bass, L., Clements, P., and Kazman, R. Software Architecture in Practice. SEI Series in Software Engineering, 3rd Edition, 2012.
6. Becker, J., Delfmann, P., and Knackstedt, R. (2006). Adaptive Reference Modeling: Integrating Configurative and Generic Adaptation Techniques for Information Models. Reference Modeling – Efficient Information Systems Design through Reuse of Information Models, edited by Jörg Becker and Patrick Delfmann, Physica-Verlag HD, pp. 27-58.

7. Brocke, J. (2007). Design Principles for Reference Modelling - Reusing Information Models by Means of Aggregation, Specialisation, Instantiation, and Analogy. Reference Modeling for Business Systems Analysis, edited by P. Fettke and P. Loos, Idea Group Publishing, Hershey, pp. 47-75.
8. Clark, T., Gonzalez-Perez, C., & Henderson-Sellers, B. (2014). A foundation for multi-level modelling. In MULTI 2014–Multi-Level Modelling Workshop Proceedings (p. 43).
9. Clark, T. and Willans, J. (2012). Software language engineering with XMF and Xmodeler. Formal and Practical Aspects of Domain Specific Languages: Recent Developments. IGI Global, USA.
10. Clements, P. and Northrop, L. (2001). Software Product Lines: Practices and Patterns. Addison-Wesley.
11. Cointe, P. (1987). Metaclasses are first class: the objvlisp model. ACM SIGPLAN Notices 22 (12), pp. 156-162.
12. De Lara, J., Guerra, E., and Cuadrado, J. S. (2014). When and How to Use Multi-Level Modelling. ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2), Article no. 12.
13. Frank, U. (2002). Multi-perspective enterprise modeling (MEMO): conceptual framework and modeling languages, Proceedings of the 35th Annual Hawaii International Conference on System Sciences, HICSS), IEEE Computer Society Washington, pp. 72–82
14. Frank, U. (2014). Multilevel Modeling - Toward a New Paradigm of Conceptual Modeling and Information Systems Design. Business & Information Systems Engineering (BISE) 6(6):319-337.
15. Gonzalez-Perez, C. and Henderson-Sellers, B. (2006). A powertype-based metamodelling framework. Software & Systems Modeling 5 (1), pp. 72-90.
16. Henderson-Sellers, B., Clark, T., and Gonzalez-Perez, C. (2013). On the Search for a Level-Agnostic Modelling Language. CAiSE 2013, pp. 240-255.
17. Jacobson, I., Griss, M.L., Jonsson, P. (1997). Software reuse - architecture, process and organization for business. Addison-Wesley-Longman.
18. Jordan, A., Mayer, W., and Stumptner, M. (2014). Multilevel modelling for interoperability. MULTI@MoDELS 2014, pp. 93-102
19. Käkölä, T. (2010). Standards Initiatives for Software Product Line Engineering and Management within the International Organization for Standardization. 43rd Hawaii International Conference on Systems Science (HICSS-43), pp. 1-10.
20. Neumayr, B., Schrefl, M., and Thalheim, B. (2011). Modeling techniques for multi-level abstraction. R. Kaschek, L. Delcambre (Eds.), The Evolution of Conceptual Modeling, pp. 68–92
21. Odell, J. J. (1994). Power types. Journal of Object-Oriented Programming, 2 (2), pp. 8-12
22. Pohl, K., Böckle, G., van der Linden, F. (2005) Software Product-line Engineering: Foundations, Principles, and Techniques, Springer.
23. Reinhartz-Berger, I. and Sturm, A. (2009). Utilizing domain models for application design and validation. Information & Software Technology 51(8), pp. 1275-1289.
24. Rossini, A., de Lara, J., and Guerra, E., Rutle, A., and Wolter, U. (2014). A formalization of deep metamodeling. Formal Aspects of Computing 26 (6), pp. 1115-1152.

Multi-Language Modelling with Second Order Intensions

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. In the last decade, there have been several fundamental advances in the field of ontological and linguistic metamodelling. They proposed the use of megamodels to link abstract, digital and physical systems with a particular set of useful relations; the distinction between ontological and linguistic layers, identification and separation of them; even formalised the act of modelling and the sense and denotation of a language. In this paper, we propose second order intensions and extensions to more closely model linguistical and ontological conformance and mapping.

1 Formal modelling of languages

In the classic theory of formal languages, a *language* L is defined as a set of sequences of alphabet symbols: $L \subseteq \Sigma^*$ [9]. This definition is easily applicable to textual languages (traditionally associated with programming) and visual languages (traditionally associated with modelling). It is also almost trivially generalisable to graph languages by substituting the reflexive transitive closure in the definition by another operation that (usually recursively) constructs all possible valid language instances from symbols of the alphabet. Even then, all manipulations with the language are done as if it were a set of language instances. For example, a *parser* is generally considered a mapping from the textual language to the tree language in that it assigns a valid parse tree to each valid textual input [18]. Hence, the only relation that is needed to formally describe such processes is an *element of* ($\in$ or ε) relation with rare exceptions like generalised parsers [16] that associate one textual input with a set of several possibly valid parse trees. Since in practice such mapping usually gets implemented to output a *representation* of a set instead of the actual set itself, such cases are more sidesteps than real exceptions: the range of a general parser is a set of parse forests, and each output in an element of this set.

Traditional metamodelling abandons the concept of a language in favour of a *modelling layer* [14,15]. The formal arsenal is expanded to a strict hierarchical structure: the lowest layer is too close to real life to formally decompose and study (e.g., raw data, real life objects, concrete systems), the highest layer is so abstract that it is self-descriptive, and the middle layers M_i contain entities that model entities from the layer below (M_{i-1}) and are expressed in languages defined by entities from the layer above (M_{i+1}). Thus, user data is expressed

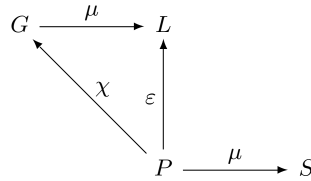


Fig. 1. A grammar G is a model of a language L, it is also a metamodel for the program P to conform to. The said program P is an element of the language L and it models a real system S. (The example demonstrates a megapattern of metamodel conformance [6, Fig.8]).

in user concepts that use UML concepts; UML concepts model user concepts in MOF; and MOF models UML concepts and defines both a language for them and a metalanguage for expressing itself. In this view, a new relation emerges: an instance of (we will denote it as ι), which is more abstract than the ε relation since it works formally even in situations when a set of all valid models cannot be expressed or when it does not make sense conceptually to express it. For example, an object is an instance of its class, and it is much less interesting to consider the set of all possible objects of a class than to investigate the nature of this instantiation and the consequences thereof. Since instantiations is sometimes hard to express universally, we also speak of a conforms to (χ) relation [2]: a model conforms to a metamodel, an object conforms to a class, a program conforms to a programming language, a database conforms to a schema. Since this theory is rooted in the modelling community, relations ι and χ are commonly used together with a *representation of* relation μ used in the conceptual sense: an object Cat models a real cat even though there might be no "language of cats" and the construction of a set of all possible cats is often unnecessary. Relations can be diagrammatically combined to form so called *megamodels* [3,6,5], an example shown on Figure 1.

Formal metamodelling distinguishes between two kinds of instance of relations: the *linguistic* instance of and the *ontological* instance of [1]. This complicates the metamodelling process somewhat but removes most ambiguity associated with the ι relation: the object Fluffy models a very particular cat and it is both an ontological instance of a class Cat (because Fluffy the real cat is a conceptual instance of cats in general) and a linguistic instance of an Object (because it needs to belong to a certain class, to be instantiated in a certain way and obey other constraints typical for all objects but not for all cats). An example is given on Figure 2 and is easily extensible to new ontological levels: cats are ontological instances of species, which are ontological instances of a biological rank [1, Fig.5] — and the formalisation still allows us to distinguish these modelling statements from ones that stay within one ontological level (i.e., that cats are pets, carnivores, mammals, animals, etc — in the object-oriented technological space this is called inheritance). Adding more languages and trans-

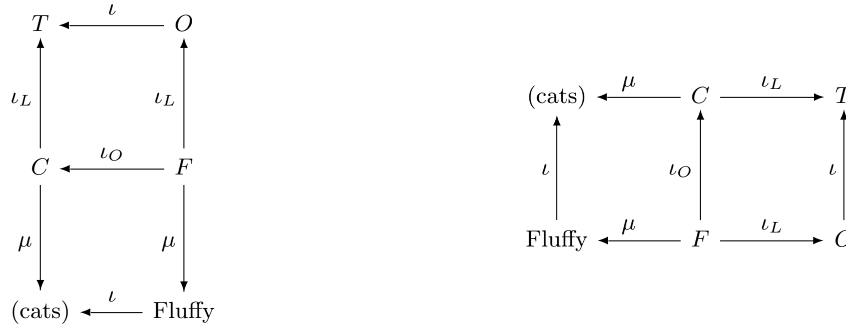


Fig. 2. The linguistic (on the left) and ontological (on the right) metamodelling views on the same megamodel. Fluffy is a cat (instance of a concept of a cat or an element in the set of all cats). Fluffy is modelled (μ) by an object F which is a linguistic instance (ι_L) of an object (O) and an ontological instance (ι_O) of a class Cat (C). The object O is in an instance-type relation to T. Columns on the left and rows on the right roughly correspond to ontological levels; rows in the left and columns on the right roughly correspond to language levels or modelling layers. (The example is a simplified/adapted version from [1, Figs.2,3]).

formations into the megamodel is somewhat more problematic due to the grid nature of the diagrams and to the lack of definitions of ontological instantiation for some languages. Coming back from biology to computer science, this allows us to properly specify that a particular program is a (linguistic) instance of say Java, but an ontological instance of a database application and as such, also obeys a set of structural and behavioural rules.

The next step in refining the theory of metamodelling of megamodels was separating the intensional and the extensional parts of the language [7]. The extensional part is argued to be the set of models allowed in the language. The intensional part models constrains and properties that are characteristic to the instances of the language. If such a distinction is introduced, the meaning of being the "instance of" something becomes ultimately apparent: the model in question must conform (χ) to the intensional part and it is an element (ε) of the extensional part. Since the extension of an abstract system always resides on the lower ontological level, the diagram is also nicely composed of tiles of the meta-level entry and its intentional part on top and its extensional part (the set of valid instances) and the model-level entry that conforms to the intensional part, is an element of the extensional part and is at the same time an instance of the abstract system. An example migrated from the technological space of cats, dogs, breeds and animals, to programming languages, can be seen on Figure 3 (read $\delta_\wedge$ as "has intension" and $\delta_\vee$ as "has extension", as δ was often used for "decomposed in"). In general, tiles like these can always substitute the megapattern from Figure 1, it is in fact its ontology-aware refinement.

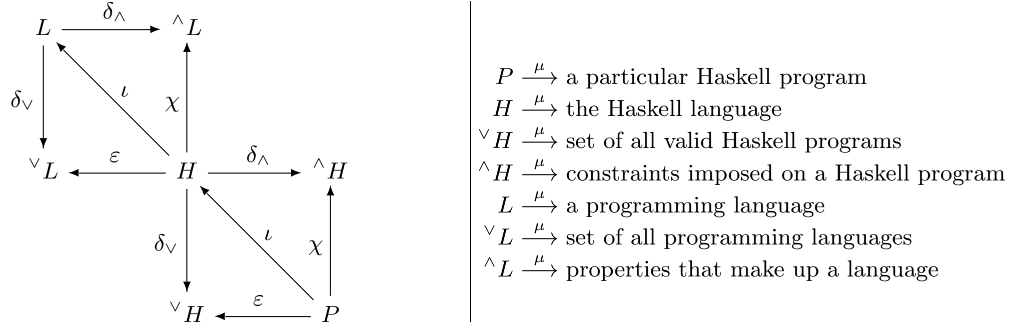


Fig. 3. Two tiles on the left diagram show how an abstract system (a language) H or L is decomposed into an intensional part ${}^{\wedge}X$ and an extensional part ${}^{\vee}X$. Haskell (H) is a programming language (L), so it is an instance (ι) of a language. This means it conforms (χ) to the intension of being a language (${}^{\wedge}L$) and is an element (ε) in the set of all programming languages (${}^{\vee}L$). Similarly, P is a program in Haskell, so it is an instance of H, it conforms to ${}^{\wedge}H$ and is an element of ${}^{\vee}H$. (The example is ported to a more fitting technological space from [7, Fig.7]).

2 The role of a grammar

In the domain of programming languages, one often speaks of a conformance of a program to the grammar of the language in which the program was written in. Is such a grammar the same as the intensional part of the language? ($G = {}^{\wedge}L$?)

The answer given by the formal language theory is *yes* — however, this theory has a slightly different megamodel of the situation: since a language is equated with its extensional part, the "instance of" relation is equated with the "element of" relation. Furthermore, instead of the language being decomposed into two parts, its intension is treated as a *model* of its extension (which is typically infinite, so it helps to have a finite model of it). The result is depicted on Figure 4 in the same style we have used so far. As an example we can consider the technological space of XML: then P is an XML document, G is a DTD or an XML Schema definition, the validator uses G to check $P \xrightarrow{\chi} G$ and programs in XSLT, XQuery, JavaScript and other languages can be written to work on elements of L.

In a more general case, the role of a grammar is the $G \xrightarrow{\mu} L$ chain is called generative or derivational and is used in most proofs in the theory of formal languages and automata. Its role in the $P \xrightarrow{\chi} G$ chain is called analytical and is often utilised by using it prescriptively [8] and generating a parser out of it. What does such a parser do? In the simplest case, it analyses the text of the program and constructs a term that aligns tokens (lexemes) of the input with its understanding of how the structure of any program should look like. This already does not fit our picture at all, and we lack means to express that not only the grammar serves in at least two different roles, but also the language apparently at the same time is a language of strings (that are acceptable inputs

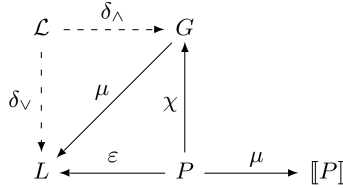


Fig. 4. The formal language theory view on relating languages and instances: the program P is an encoding of a solution $[\![P]\!]$, which is an element of the formal language L and conforms to the grammar G. The grammar G is also a finite model of a (typically infinite) language instance set L. The conceptual language $\mathcal{L}$ is implicit and utterly intangible, and its instantiation is never discussed explicitly.

for a parser) and a language of terms or trees (that are outputs of a parser). An attempt to fit this transformational megapattern into our view is shown on Figure 5: a grammar G serves a model of both two languages (textual and tree language) and the mapping between two representations of the same program (the text and the parse tree). However, since $\mathcal{L}_C$ and $\mathcal{L}_T$ are implicit, we cannot make any statement about the relation between L_T and L_C, which makes the megamodel a bit less useful.

This multipurposefulness of grammars in a broad sense is unfortunate from the modelling point of view, but it explains their omnipresence in software engineering [10]. In terms of modelling modelling[1], $G \overset{\mu}{\leadsto} X$ for all megamodel elements X related to G: they share some intention and can be partially represented by one another [13]. We will explore this intersection in the next section and make it explicit.

In practical software language processing, grammars try to balance in between all these roles, with a varying degree of success. In certain cases, people separate some constraints (traditionally called static semantic rules) that are too hard to express in the chosen grammar formalism and are purely related to $G \xrightarrow{\mu} L$ and $P \xrightarrow{\chi} G$; in other cases (in particular related to mapping between already structured concrete syntax and an improved abstract syntax) there could be several grammars defining separate languages, with $G \xrightarrow{\mu} \tau$ included in one of them or shipped as a third separate artefact.

Interestingly, the role of a language in modelware engineering is slightly different yet also not perfect. Consider the following statement [13]:

$$F \xrightarrow{\mu} L \xrightarrow{\mu} \left\{ M \mid M \xrightarrow{\mu} S \right\}$$

What is stated here is that the formal system F truthfully models a language L which in turn models a set of models M such that they all model the system S.

[1] NB: the original MoDELS 2009 paper used "$\overset{\mu}{\mid\!\leadsto}$" for shared intention instead, we use a much more fitting notation from the extended journal version.

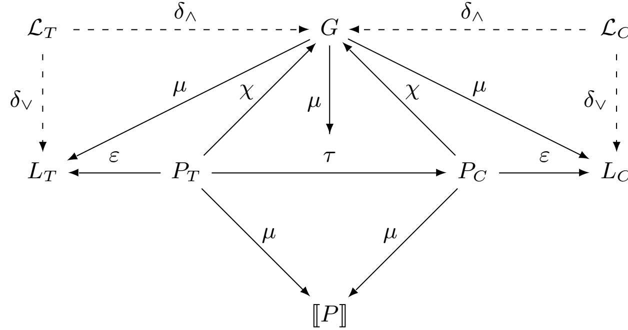


Fig. 5. The classic grammar theory view on relating languages and instances: the text of the program P_T and a concrete parse tree P_C both model the conceptual program (algorithm) $[\![P]\!]$ and both conform (χ) to the grammar G which acts as a model (μ) of both the string language L_T and the tree/term language L_C viewed as the sets of their corresponding instances. The same grammar G simultaneously models the transformation between text instances and tree instances. Conceptual languages $\mathcal{L}_T$ and $\mathcal{L}_C$ remain intangible, denotation $[\![P]\!]$ might exist in a form of flow diagrams.

If represented diagrammatically on Figure 6, we see the main differences between our approach and the method of Muller et al: instead of being decomposed into an intension and an extension, the language is considered to be a model of its extension, and its intension (grammar in a broad sense, some kind of formal model by requirements) is considered to be its model. Furthermore, there is no explicit consideration for the conformance between the models and this formal definition.

3 Second order to the rescue

As we have seen, the grammar of a software language is its intensional part (or approximates it very closely). Let us deconstruct it further. For $^\wedge Collie$, Gašević et al claimed it was a model of the real world intension of the concept of a collie, combining properties such as “has long hair”, “has bushy tail” and “can herd sheep” [7]. For a programming language, the intension is a model of two kinds of properties: essential (“supports parametric polymorphism”, “uses lazy evaluation”, “contains a conditional statement which must contain a condition and a statement”, “variable names should start with a letter”) circumstantial (“functions have comma-separated arguments”, “statement blocks are defined by indentation”, “one statement per line of code”). The essential properties refer to the way the language concepts are constructed and manipulated — therefore, their model is the intension of the intension of the language ($^{\wedge\wedge}L$). The circumstantial properties refer to the way the concepts of the language are represented in the instances — in other words, how language instances (elements of $^\vee L$) are

6

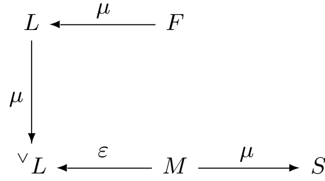


Fig. 6. Modelling modelling modelling: all models M that model a system S, are collected in a set ${}^{\vee}L$. The language L is considered to be a model of its extension, and is in turn modelled by a formal system F which is conceptually its intension which remains disconnected from the model M. (This example is a diagrammatic representation of [13, Fig.17]).

constructed; hence, it is the intension of the extension of the language (${}^{\wedge\vee}L$). The extension of the intension is even more interesting because it is supposed to collect examples of how the intensions can be expressed: it is a collection of all possible syntaxes for a language. To the best of our knowledge such an entity has not yet been formally investigated, but if it also represented as a set, the intension of the extension is an element of that set; otherwise it is still bound to be an instance of ${}^{\vee\wedge}L$. The resulting diagram is depicted on Figure 7 with the decomposition, conformance and instantiation relations.

Since now we have ${}^{\wedge\wedge}L$ to denote the constraints essential to the language L, it should be the same for all related languages. Indeed, if the modelling intent [17] is limited to this essence, and a grammar that respects it, it also models all variants of languages closely without any regard to the choice of ${}^{\wedge\vee}L$. In other words, for $\mu' = \mu/_{{}^{\wedge\wedge}L}$,

$${}^{\vee}L_T \xleftarrow{\mu'} G \xrightarrow{\mu'} {}^{\vee}L_C$$

This result agrees with the "shift in linguistical conformance" by Muller et al [13] (when talking about mapping among models with the same intent) and with the "constant functions all the way down" by Dowty et al [4] (when talking about intensions of intensions). The final diagram is presented on Figure 8: a "real" program can be encoded by a programmer as either text (conforming to ${}^{\wedge\vee}L_T$) or a tree (conforming to ${}^{\wedge\vee}L_C$), and as long as the intension of the intension is preserved ($P_T \xrightarrow{\chi} {}^{\wedge\wedge}L$), the languages stay *same but different* and valid instances can be freely mapped in any direction. The same holds for mappings between abstract syntax and concrete syntax, up to a homomorphism (such mappings often permute arguments and perform other component rearrangements).

4 Concluding remarks and related work

In short, we have proposed to consider four components of software languages: intension-intensions (conceptual constraints to always conform to); extension-

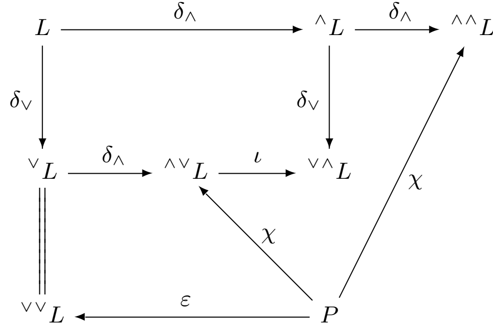


Fig. 7. The language L is decomposed (δ) into its intensional ($^\wedge L$) and extensional ($^\vee L$) parts. They are, in turn, decomposed in the same fashion. The extensional part $^\vee L$ is a set, so the extension of the extension $^{\vee\vee}L$ is the same set. However, the intension of the extension $^{\wedge\vee}L$ represents structural constraints of the elements of the set, while the intension of the intension $^{\wedge\wedge}L$ represents linguistic constraints that refer to the intensional language and not to its accidental representation. The extension of the intension $^{\vee\wedge}L$ represents a model of possible syntactic representations of programs in the language: it is modelled by a simple set, the particular syntax $^{\wedge\vee}L$ is an element of that set, otherwise it is still bound there in an instance of (ι) relation.

extensions (sets of valid language instances in a given notation); intentions of extensions (circumstantial constraints specific to the chosen syntax but not the the language as such); extensions of intensions (models of possible syntaxes compatible with the language core), in a hope that it brings us closer to understanding the nature of modelling languages and various artefacts related to them. In previously existing work, languages are mostly considered either with a fixed syntax or with two or three of them which are also fixed and claimed to be related to different aspects of language processing. This work can serve as a foundation for formal manipulation of languages with multiple syntaxes, or systems where "the same" language is claimed to be used across technical spaces (ORM, parsing, convergence, etc).

For the sake of clarity and conciseness of notation, we have opted for the use of commutative diagrams to represent megamodels instead of using any of the existing megamodelling languages. We have adopted Favre's symbols for relations: μ as "models" or "representation of"; ε as "element of", χ as "conforms to". Atkinson and Kühne did not have a shorthand notation so we used ι_O for "ontological instance of", ι_L for "linguistic instance of" and just ι for "instance of" where the meaning is unknown or universal. We have adopted Montague notation: $^\wedge L$ for intensions and $^\vee L$ for extensions. We also took the liberty of using $[\![P]\!]$ for denotations of single instances — in formal semantic theory denotation is equated to extension but in the current state of multi-level metamodelling we do not yet need to differentiate between $^\wedge P$ and $^\vee P$. However, in the future research on multi-language modelling, we recommend to consider substituting

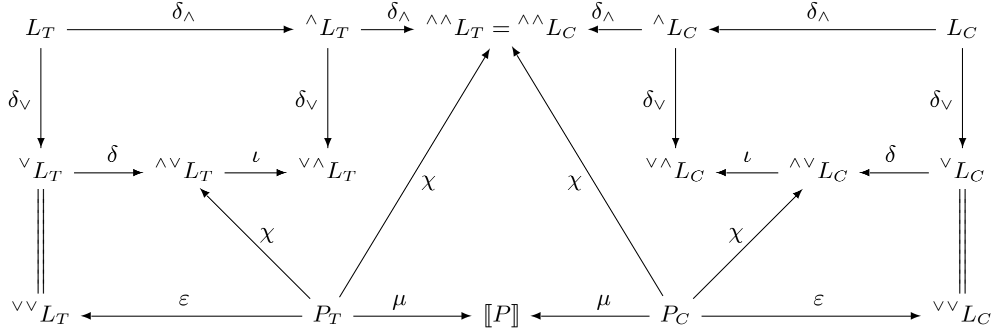


Fig. 8. Each language shown — the textual language L_T and the concrete tree language L_C — are decomposed (δ) into their intensions ($^\wedge L_x$) and extensions ($^\vee L_x$), which are in turn also decomposed into their intensions and extensions. The denotation of a program ($[\![P]\!]$) is modelled by two entities: the program text (P_T) and its concrete parse tree (P_C). They follow the same megapatterns: each P_x conforms (χ) to both the intension of the extension of the corresponding language ($^{\wedge\vee}L_x$) and the intension of the intension of the language ($^{\wedge\wedge}L_x$). Each P_x is also an element (ε) of the extension of the language ($^\vee L_x$). The intension of an extension ($^{\wedge\vee}L_x$) is an instance (ι) of the extension of the intension ($^{\vee\wedge}L_x$), and in the simplest case also just its element.

$[\![P]\!]$ with $^\wedge P$ and using $^\vee P$ for the hypothetical set of all possible representations of a program P.

The notion of commitment to grammatical structure was discussed by Klint, Lämmel and Verhoef [10], and recently was elaborated by a megamodel of various software language engineering artefact kinds such as parse trees, visual models, lexical templates, etc, in the context of (un)parsing in a broad sense [18]. The contribution of this paper to that trend was showing that concrete syntaxes are ontological instances of the abstract syntax (more precisely, $^{\wedge\vee}L \xrightarrow{\iota} {}^{\vee\wedge}L$). The idea that a software language should be allowed to have different syntaxes while staying essentially the same language, is not new but has always been rejected by formalisations.

Atkinson [1], Bézivin [3], Favre [6], Gašević [7], Kühne [11], Muller [13] and many others have made significant contributions to comprehension and refinement of the processes of modelling, metamodelling and megamodelling. This paper is an endeavour to contribute to that trend by making a yet another step in improving the formalisations, as well as by providing concrete examples from software and grammarware engineering, even when they seemed less suitable to discuss ontological matters than the classic Lassie/Fido – Collie – Breed example. We hope such results will both help to identify problems we can solve in the future and bring less meta-minded modelling practitioners and language engineers closer.

Apart from the Fregean perspective on intensional logic, Montague considered a Russelian variant where the extension of the intension (called the "sense denotation") is not a fundamental concept [12]. It has proven to be less useful to him, but in software language engineering this idea has never even been tried yet.

References

1. C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.
2. J. Bézivin. In search of a basic principle for model-driven engineering. *Novatica Journal*, 2004.
3. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.
4. D. R. Dowty, R. E. Wall, and S. Peters. *Introduction to Montague Semantics*. Kluwer Academic Publishers, 1981.
5. J.-M. Favre. Towards a Basic Theory to Model Driven Engineering. In *Proceedings of the Third Workshop on Software Model Engineering*, 2004.
6. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *Electronic Notes in Theoretical Computer Science, Proceedings of the SETra Workshop*, 127(3), 2004.
7. D. Gašević, N. Kaviani, and M. Hatala. On Metamodeling in Megamodels. In *MODELS'07*, pages 91–105, 2007.
8. W. Hesse. More Matters on (Meta-)Modeling: Remarks on Kühne's "matters". *SoSyM*, 5(4):387–394, 2006.
9. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2007.
10. P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM ToSEM*, 14(3):331–380, 2005.
11. T. Kühne and D. Schreiber. Can Programming be Liberated from the Two-Level Style: Multi-Level Programming with DeepJava. In *OOPSLA*, pages 229–244. ACM, 2007.
12. R. Montague. Universal Grammar. *Theoria*, 36(3):373–398, 1970.
13. P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling Modeling Modeling. *SoSyM*, 11(3):347–359, 2012.
14. Object Management Group. *Unified Modeling Language*, 2.1.1 edition, 2007. Available at http://schema.omg.org/spec/UML/2.1.1.
15. Object Management Group. *Meta-Object Facility (MOF^{TM}) Core Specification*, 2.0 edition, January 2006. Available at http://www.omg.org/spec/MOF/2.0.
16. M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
17. E. S. K. Yu and J. Mylopoulos. Understanding "Why" in Software Process Modelling, Analysis, and Design. In *ICSE*, pages 159–168. IEEE Computer Society / ACM Press, 1994.
18. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*, volume 8767 of *LNCS*, pages 50–67, Switzerland, Oct. 2014. Springer International Publishing.

Practical Multi-level Modeling on MOF-compliant Modeling Frameworks

Kosaku Kimura, Yoshihide Nomura, Yuka Tanaka,
Hidetoshi Kurihara, and Rieko Yamamoto

Fujitsu Laboratories, Kawasaki, Japan
{kimura.kosaku,y.nomura,tanaka.yuka,kurihara.hide,r.yamamoto}
@jp.fujitsu.com

Abstract. This paper describes practices for multi-level modeling by only using existing modeling frameworks that comply Meta-Object Facility (MOF). We design modeling patterns for achieving the multi-level modeling methodologies on Eclipse Modeling Framework, and implement the dataflow model by applying the patterns. Moreover, we attempt to compare the patterns regarding the facilitation of developing both our tool and plugins. We found Orthogonal Classification Architecture (OCA) pattern is easier to develop our tool than powertypes pattern, but regarding plugins for our tool, powertypes pattern can define model-to-text transformation templates more simply than OCA pattern.

1 Introduction

Model-driven engineering (MDE) gains productivity of software developments providing several powerful tools for designing, developing or verifying software. Especially, model transformation technologies (i.e., model-to-model and model-to-text) are important for facilitating agile software developments. For the model-to-text transformation enables to generate executable source codes from a model, developers can develop complex applications by using graphical editors.

There are various kinds of graphical editing tools for developing and executing applications, e.g., Extract-Transform-Load [2, 5], Business Analytics [3] and Workflow Management [4]. We also have been developing a graphical editing tool on a cloud platform for facilitating developments of big data processing applications [18]. Figure 1 shows the web interface of our tool.

Many of the tools are based on modeling frameworks and provide automatic generation features for executable source codes. However, extending models of the tools tends to be difficult for third-party developers, and therefore, there have been a few plugins published from developer communities. Nowadays, the meta-models of graphical editing tools have to be easily extensible so that developers can develop more plugins [16].

Meta-Object Facility (MOF)[1] is a standard for MDE provided by Object Management Group (OMG), and Eclipse Modeling Framework (EMF)[2] is one

[1] http://www.omg.org/mof/
[2] https://eclipse.org/modeling/emf/

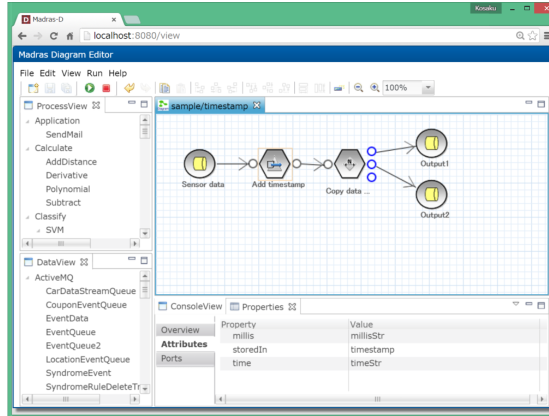


Fig. 1. EMF-based graphical editing tool for developing and executing big data processing.

of mature MOF-compliant modeling frameworks, and there are various toolkits in the EMF community, such as Acceleo[3], Query/View/Transformation (QVT) Operational[4] and ATL Transformation Language[5]. Those toolkits also conform to or follow the OMG's standards. In this paper, we attempt to achieve multi-level modeling on EMF. EMF provides the Ecore metamodel, which is compatible with Essential MOF, and tools for creating models that conform to the Ecore metamodel.

One of the major drawbacks of EMF is that it is hard to define and use a new metamodel located at the same level as the Ecore metamodel, because EMF is basically adequate to create models and objects just based on the Ecore metamodel. If we use our own metamodel, although it is obviously possible to develop a proprietary tool based on it by using the code generation feature of EMF [1, 19], the tool tends to force an unusual manner to developers, and eventually, most of them may feel that "I do not want to use it." This issue is crucial for developing the ecosystem and the community of our tool.

In order to overcome the drawbacks of existing modeling frameworks, various methodologies of multi-level modeling have been proposed such as **Orthogonal Classification Architecture (OCA)** [6, 7, 9, 11], **powertype-based meta-modeling** [13, 14] and **deep instantiation** [12, 17]. The methodologies can provide simple solutions to design metamodels along with models and objects. However, there is little consensus in the literature on fundamental multi-level modeling concepts [10], and therefore, it is still difficult to determine to apply them to industries. For now, multi-level models must be defined by only using existing MOF-compliant modeling frameworks, so we have to clarify a workaround for that.

[3] `http://www.eclipse.org/acceleo/`
[4] `http://www.eclipse.org/mmt/?project=qvto`
[5] `https://eclipse.org/atl/`

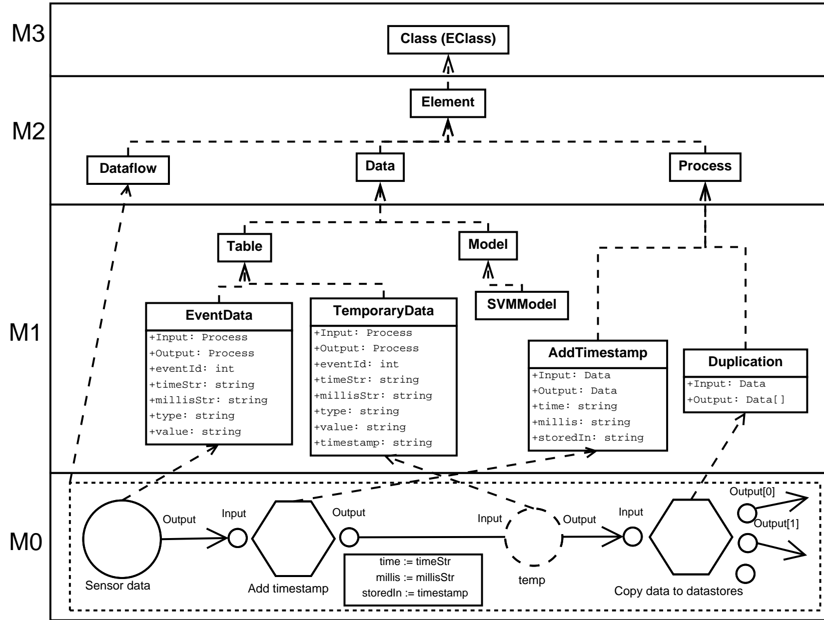


Fig. 2. Hierarchy of dataflow model.

This paper describes practices for achieving multi-level models on EMF. We use a hierarchy of a dataflow model as an example model that is used on graphical editing tools. We design multi-level modeling patterns on EMF, and implement the dataflow model by applying the patterns. Moreover, we attempt to compare the patterns regarding the facilitation of developing both our tool and plugins.

The remainder of this paper is organized as follows. Section 2 describes a model of a graphical editing tool as our motivating example. Section 3 describes patterns for multi-level modeling on EMF. In Sect. 4 we discuss the comparison of the patterns, and our conclusions are presented in Sect. 5.

2 Motivating example: a dataflow model for graphical editing tools

A typical graphical editing tool consists of a palette and a canvas as well as Fig. 1. The palette shows icons representing types of nodes, and the canvas is used to define a diagram by putting a node of the type selected from the palette and drawing an edge between nodes. By using such tool, we can easily develop a data processing application as a flow diagram that consists of nodes and edges representing icons and lines, respectively.

Figure 2 shows the hierarchy of the dataflow model that we want to design. Layer **M3** represents the original Ecore metamodel, and layer **M2** represents the metamodel of the dataflow model. Objects in layer **M2** (i.e., `Dataflow`, `Data` and `Process`) are instances of `Class`. An instance of `Dataflow` composes instances

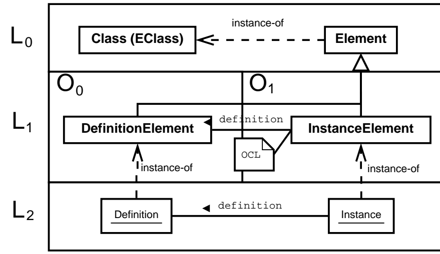


Fig. 3. OCA pattern.

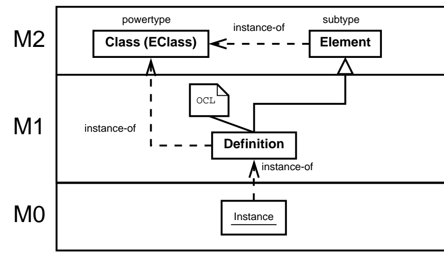


Fig. 4. Powertypes pattern.

of `Data` and `Process`, and represents how data is processed and the order of execution in the processing methodologies as well as the definition in [15].

Layer **M1** represents definitions of types and subtypes of `Data` and `Process` that are displayed on the palette. Classes in layer **M1** are instances of the classes in layer **M2** and have definitions of type names, input ports, output ports and owned properties. In Fig. 2, `Table` and `Model` are instances of `Data`, and `AddTimestamp` and `Duplication` are instances of `Process`. Moreover, `EventData` and `TemporaryData` are subclasses of `Table`, and `SVMModel` is subclasses of `Model`. Those subclasses define their own properties and data schemata for storing databases. A plugin created by a third-party developer defines a new instance of `Process` in layer **M1**, i.e., a new type of nodes in the palette.

Layer **M0** represents an instance of `Dataflow` edited on the canvas in Fig. 1. Objects in layer **M0** are instances of the objects in layer **M1**. Data node `Sensor data` in layer **M0** represents data that is produced and sent by sensors and has the schema defined by `EventData`.

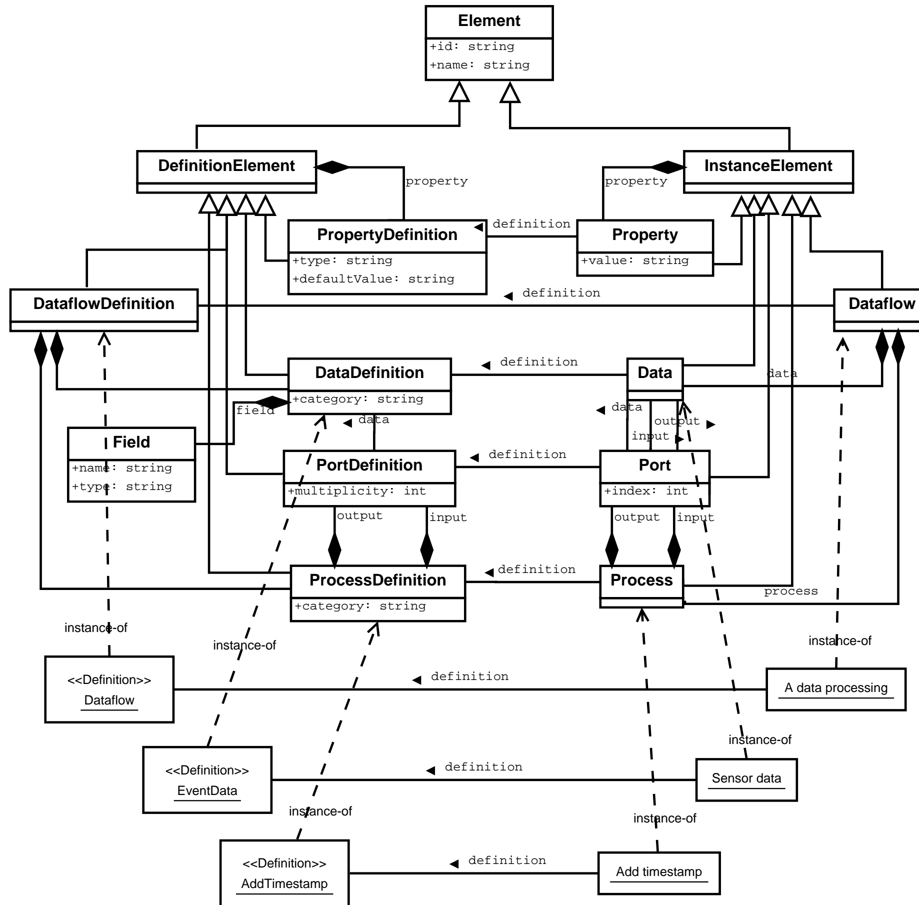
3 Multi-level modeling on EMF

Several multi-level modeling methodologies introduce a new concept of objects. A **clabject** is an object that is both a class and an instance of another class [8]. **Clabjects** sometimes have a **potency** feature that represents the depth to which an attribute can be instantiated [12] and is utilized in **deep instantiation**. In order to achieve multi-level model by only using EMF, we consider that it is difficult to introduce them on EMF, because applying those concepts obviously needs to develop a new modeling editor.

We attempt to implement the dataflow model described in Sect. 2 by applying the following two methodologies: **OCA** and **powertype-based metamodeling**. Figure 3 and 4 show modeling patterns as workarounds for each methodology.

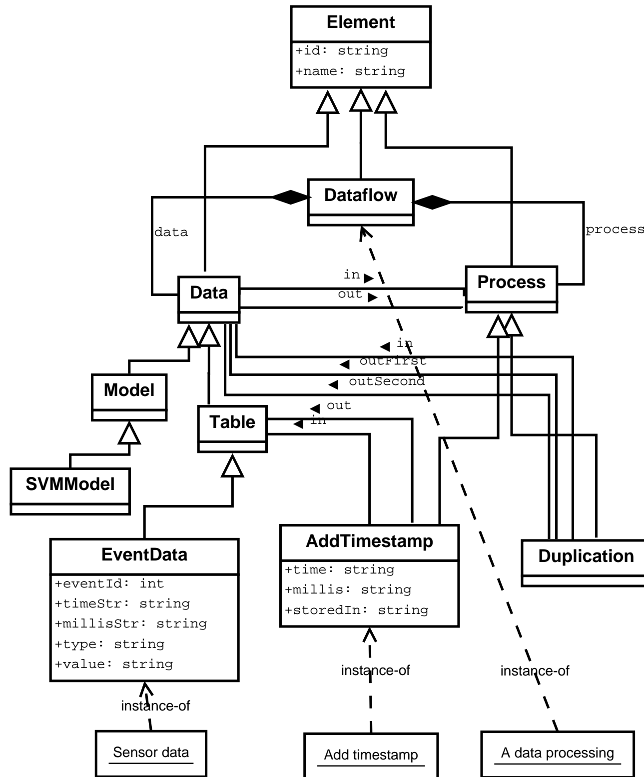
3.1 Model applying OCA pattern

The **OCA** has two dimensions of model layers: *linguistic layers* and *ontological layers*. In Fig. 3, **L** and **O** denotes *linguistic layers* and *ontological layers*, respectively. Layer $\mathbf{L_0}$ contains the Ecore metamodel and class `Element` that is an




```
-- for instance objects of Data
context Data
  inv DataHasDefinition: definition <> null
  inv DataHasValidProperties:
    definition.property->forAll(i | property->exists(definition = i))
  inv DataHasValidFields:
    definition.field->forAll(name <> null and type <> null)

-- for instance objects of Process
context Process
  inv ProcessHasDefinition: definition <> null
  inv ProcessHasValidProperties:
    definition.property->forAll(i | property->exists(definition = i))

  inv ProcessHasValidInputPorts:
    definition.input->forAll(i | input->exists(definition = i))
  inv ProcessHasValidOutputPorts:
    definition.output->forAll(i | output->exists(definition = i))
...
```

Fig. 5. Dataflow model and excerpt of OCL constraints in OCA pattern.




```
context EClass
  def: isA(typeName : String) : Boolean =
    name = typeName or oclIsKindOf(EClass)
    and oclAsType(EClass).eAllSuperTypes->exists(name = typeName)

-- for subclasses of Data
  inv DataHasNoExtraProcessRefs:
    isA('Data') implies eReferences->forAll(
      eReferenceType.isA('Process') implies name.matches('in|out')
    )

-- for subclasses of Process
  inv ProcessHasValidInputPorts:
    isA('Process') implies eReferences->forAll(
      name.matches('^in.*') implies eReferenceType.isA('Data')
    )
  inv ProcessHasValidOutputPorts:
    isA('Process') implies eReferences->forAll(
      name.matches('^out.*') implies eReferenceType.isA('Data')
    )
...
```

Fig. 6. Dataflow model and excerpt of OCL constraints in powertypes pattern.

instance of class `Class`, for defining elements of ontological layers ($\mathbf{O_0}$ and $\mathbf{O_1}$)in layer $\mathbf{L_1}$. Class `DefinitionElement` in layer $\mathbf{O_0}$ and class `InstanceElement` in layer $\mathbf{O_1}$ defines the type and the instance of elements of the dataflow model, respectively. Layer $\mathbf{L_2}$ contains definition objects and instance objects that are instances of class `DefinitionElement` and `InstanceElement`, respectively.

We represent an *ontological instantiation* relationship by a reference to a definition object. The instance object has a reference to the definition object, and the correctness of the relationship between them is verified by constraints written in Object Constraint Language (OCL).

Figure 5 shows the dataflow model that conforms to the **OCA** pattern. Class `Dataflow`, `Data`, `Process`, `Port` and `Property` are instance classes, which are subclasses of class `InstanceElement`, and all of them respectively have their own definition classes, which are subclasses of class `DefinitionElement`. Object `Dataflow`, `EventData` and `AddTimestamp` are definition objects, i.e., instances of the definition classes. Object `A data processing`, `Sensor data` and `Add timestamp` are instance objects, i.e., instances of the instance classes.

Examples of OCL constraints for instance objects of class `Data` and `Process` are shown in the lower part of Fig. 5.

3.2 Model applying powertypes pattern

Powertype-based metamodeling introduces a powertype that is defined as a type whose instances are types inheriting a subtype [14]. While in the original idea, every object in layer **M1** must be a **clabject** that is both an instance of a powertype and a subclass of a subtype, we define an object in layer **M1** of Fig. 4 just as an instance of a powertype, i.e., class `Class`, and use OCL constraints for defining the relationship between the object and a subtype. We define that the object is regarded as a genuine subclass of the subtype if it satisfies the OCL constraints.

Figure 6 shows the dataflow model that conforms to the **powertypes** pattern. As class `Dataflow`, `Data` and `Process` are subclasses of class `Element`, the hierarchy of all classes are represented as inheritance relationships. Class `EventData`, which is a subclass of class `Table`, has attributes that represent data schema. Class `AddTimestamp`, which is a subclass of class `Process`, has attributes that represent parameters of the process. Class `AddTimestamp` also has an input port and an output port as references to class `Table`, which means that it consumes and produces `Table`-typed data.

Object `A data processing`, `Sensor data` and `Add timestamp` are instances of class `Dataflow`, `EventData` and `AddTimestamp` respectively.

Examples of OCL constraints for subclasses of class `Data` and `Process` are shown in the lower part of Fig. 6.

4 Evaluation

We attempt to compare our modeling patterns, **OCA** and **powertypes**, regarding the facilitation of the following developments: developing our tool by

Table 1. Definition of AddTimestamp.

Name	Description
Input	a single port that consumes a subclass of Table
Output	a single port that produces a subclass of Table
time	a formatted date string, e.g., ''yyyy-MM-dd hh:mm:ss''
millis	an integer string of a millisecond value
storedIn	a field name to which a timestamp value is assigned

ourselves and developing plugins for our tool by third-party developers. We consider there are a lot of viewpoints regarding the facilitation, but we have not yet completed the comprehensive evaluation from the viewpoints. In this paper, we concentrate the following two viewpoints: model manipulation for our tool and template description for plugins.

4.1 Model manipulation for our tool

Regarding the development of our tool, we focus on how to manipulate the model on the methodology. The **OCA** pattern can utilize the code generation features of EMF, because we do not need to extend metamodels in layer $\mathbf{L_0}$ of Fig. 3. All objects that are added by plugins for new types of data or processes are located in layer $\mathbf{O_0}$, and they can be manipulated by using automatically generated codes. On the other hand, when we apply the **powertypes** pattern, we have to extend the Ecore metamodel dynamically, so it is difficult to utilize the code generation. We have to manipulate objects in layer **M0** by only using the default Ecore APIs that are not intuitive and troublesome to manipulate.

4.2 Template description for plugins

Regarding the development of plugins, we focus on the description of the model-to-text transformation template for process AddTimestamp in Fig. 1, 2, 5 and 6. Table 1 shows the definition of process AddTimestamp. The process produces a record that is appended a new field named as the string value of storedIn. The new field is assigned a string value of a timestamp that is calculated by using time, and millis of an original record.

Now, we consider a template for producing the following SQL-like processing query.

```
insert into <Output> select <Field of Input>[,<Field of Input> ...],
UDF.timestamp(<time>, <millis>) as <storedIn> from <Input>
```

We use Acceleo, which is an implementation of MOFM2T[6], for generating the query. By applying the **OCA** pattern, the template can be described as follows.

[6] http://www.omg.org/spec/MOFM2T/

```
[template public generate(aProcess : Process) overrides generate
? (definition.name='AddTimestamp')]
insert into [output->any(definition.name='out').data.name/]
select [for (input->any(definition.name='in')
.data.definition.field) separator(',')][name/][/for]
, UDF.timestamp(
[property->any(definition.name='time').value/],
[property->any(definition.name='millis').value/]
) as
[property->any(definition.name='storedIn').value/]
from [input->any(definition.name='in').data.name/]
[/template]
```

By applying the **powertypes** pattern, the template can be described as follows.

```
[template public generate(aProcess : AddTimestamp) overrides generate]
insert into [out.name/]
select [for (_in.eClass().eAttributes) separator(',')][name/][/for]
, UDF.timestamp([time/],[millis/]) as [storedIn/] from [_in.name/];
[/template]
```

By contrasting those descriptions, the **powertypes** pattern can describe the template more simply than the **OCA** pattern. This is because objects in layer **M1** of Fig. 4 are just models of the Ecore metamodel, so we can directly access the attribute values of their instances. This advantage is valid not only Acceleo but also other EMF-based toolkits, and the fact indicates that the **powertypes** pattern is more usable by following the existing common manners of MOF-compliant modeling frameworks.

5 Conclusion

In this paper, we described the practices for achieving multi-level modeling by only using EMF. We defined modeling patterns of the following two methodologies: **OCA** and **powertype-based metamodeling**. We implemented the dataflow model for graphical editing tools by applying the patterns. By comparing the implementations on the two methodologies, We found the **OCA** pattern is easier to develop our tool than the **powertypes** pattern, but regarding plugins for our tool, the **powertypes** pattern can define model-to-text transformation templates more simply than the **OCA** pattern.

We consider we have to achieve the ease of developing plugins for our tool rather than the ease of developing our tool itself, because increasing the number of plugins can benefit our tool and our communities. Although regarding the simplicity of template descriptions, the **powertypes** pattern is a better choice than the **OCA** pattern, further evaluations from other viewpoints are needed for determining the best pattern.

We hope that the multi-level modeling methodology is standardized adequately following the existing common manners of MOF-compliant modeling frameworks.

References

1. Metamodeling with EMF: Generating concrete, reusable Java snippets, http://www.ibm.com/developerworks/library/os-eclipse-emfmetamodel/
2. Pentaho Data Integration, http://community.pentaho.com/projects/data-integration/
3. RapidMiner, https://rapidminer.com/
4. RunMyProcess, https://www.runmyprocess.com/en/
5. Talend Data Integration, http://www.talend.com/products/data-integration
6. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. Software Engineering, IEEE Transactions on 35(6), 742–755 (Nov 2009)
7. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. Software, IEEE 20(5), 36–41 (Sept 2003)
8. Atkinson, C.: Meta-modelling for distributed object environments. In: Enterprise Distributed Object Computing Workshop [1997]. EDOC'97. Proceedings. First International. pp. 90–101. IEEE (1997)
9. Atkinson, C., Gerbig, R.: Melanie: multi-level modeling and ontology engineering environment. In: Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards. ACM (2012)
10. Atkinson, C., Gerbig, R., Kühne, T.: Comparing multi-level modeling approaches. In: MULTI 2014–Multi-Level Modelling Workshop Proceedings. pp. 53–61 (2014)
11. Atkinson, C., Kennel, B., Goß, B.: The level-agnostic modeling language. In: Software Language Engineering, pp. 266–275. Springer (2011)
12. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: UML 2001The Unified Modeling Language. Modeling Languages, Concepts, and Tools, pp. 19–33. Springer (2001)
13. Henderson-Sellers, B., Gonzalez-Perez, C.: The rationale of powertype-based metamodelling to underpin software development methodologies. In: Proceedings of the 2nd Asia-Pacific Conference on Conceptual Modelling - Volume 43. pp. 7–16. APCCM '05, Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2005)
14. Henderson-Sellers, B., Gonzalez-Perez, C.: On the ease of extending a powertype-based methodology metamodel. Meta-Modelling and . WoMM 2006 pp. 11–25 (2006)
15. Kimura, K., Nomura, Y., Kurihara, H., Yamamoto, K., Yamamoto, R.: Multi-query unification for generating efficient big data processing components from a dfd. In: Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on. pp. 260–268. IEEE (2013)
16. Kimura, K., Nomura, Y., Tanaka, Y., Kurihara, H., Yamamoto, R.: Runtime Composition for Extensible Big Data Processing Platforms. In: 2015 IEEE 8th International Conference on Cloud Computing. pp. 1053–1057 (2015)
17. Neumayr, B., Schrefl, M.: Abstract vs concrete clabjects in dual deep instantiation. In: MULTI 2014–Multi-Level Modelling Workshop Proceedings. pp. 3–12 (2014)
18. Nomura, Y., Kimura, K., Kurihara, H., Yamamoto, R., Yamamoto, K., Tokumoto, S.: Massive event data analysis and processing service development environment using dfd. In: Services (SERVICES), 2012 IEEE Eighth World Congress on. pp. 80–87 (2012)
19. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)

An algebraic instantiation technique illustrated by multilevel design patterns

Zoltan Theisz[1] and Gergely Mezei[2]

[1] Huawei Design Centre, Ireland,
zoltan.theisz@huawei.com
[2] Budapest University of Technology and Economics, Budapest, Hungary,
gmezei@aut.bme.hu

Abstract. Multi-level meta-modeling hinges on the precise conceptualization of the instantiation relation between elements of the meta-model and the model. In this paper, we propose a new algebraic instantiation approach, the Dynamic Multi-Layer Algebra. The approach aims to provide a solid algebraic foundation for multi-level meta-modeling, which is easily customizable through different bootstrap elements and a dynamic instantiation procedure. The paper describes the major parts of the approach and also demonstrates their modeling capabilities by covering some of the most-often used design patterns for multi-level modeling.

Keywords: multi-level meta-modeling, dynamic instantiation, design patterns

1 Introduction

Multi-level meta-modeling is enjoying a renaissance thanks to the dynamic modeling needs of contemporary complex software systems. For example, next generation telecom management systems set new challenges towards centralized, model-based extendable network element repositories that must be able to be used both in design- and run-time. The model repository must be open-ended with respect to complex types: both through gradual extension and the introduction of new elements. Therefore, many well-known meta-modeling patterns such as *type-object*, *dynamic features* or the *dynamic auxiliary domain concepts* [5] frequently reappear there. Although potency-based meta-modeling can handle these situations, alternative formalisms may simplify their modeling by allowing more dynamism in the instantiation.

In this paper, we aim to illustrate how such an alternative multi-level meta-modelling approach, referred to as Dynamic Multi-Layer Algebra (DLMA) [7], can be applied equally well to those multi-level meta-modelling patterns. The paper is structured as follows: Section 2 introduces the technical background of multi-level modeling, then, in Section 3, we introduce our DLMA approach in some detail. Next, in Section 4, the approach is illustrated by its targeted application to those three well-know multi-level meta-modeling design patterns that are often observed in real-life applications as analyzed in [5]. Finally, Section 5 concludes the paper with our future research directions.

2 Background

Instantiation lies at the heart of any metamodel-based model development technique. Instantiation is the key operation that defines the semantic linkage between the meta-model and the model level. This linkage can be ontological or linguistic, or even both at the same time, depending on the actual methodology and the available tools used for meta-modelling. Standard approaches prefer the linguistic interpretation which results in a two level architecture, where the metamodel is built first and then it is instantiated into models. This architecture has been implemented, for example, in Eclipse Modelling Framework (EMF) [1], which enforces the definition of the meta-model within one single meta-level by relying on natively available meta-modeling facilities such as type definition, inheritance, data types, attributes and operations. However, it is hard to modify the meta-model once instance models have been built, thus explicit meta-modeling of an ontological interpretation of instantiation is also necessary.

With only linguistic interpretation enforced, the resulting multi-level models may become ad-hoc and usually involve accidental complexity. In order to avoid mixed ontological and linguistic instantiation and to overcome the limits set by the two-meta-level architecture, pure multi-level meta-modeling approaches have been put in action. These techniques can distinguish between two kinds of instantiation: shallow instantiation means that information is used at the immediate instantiation level, while deep instantiation allows to define information on the deeper modeling levels as well. If we need each meta-level to be instantiable, there must be some means to add new attributes and operations to the existing models. There are two options: one can either bring the source of the information along through all model levels (and use it where it may be needed), or one can add the source of that information directly to the model element where it is actually used. The concept of potency notion and dual field notion [3] [2] were introduced as solutions of the problem. Here, elements within a model may not only be instances of some element in the meta-model above, but, at the same time, they may also serve as types to some other elements in the meta-level below. In other words, one assumes the existence of an unrestricted meta-model building facility that is controlled only by the explicit definition of potency limits allowing a preset number of meta-levels an element can be instantiated at. In effect, there are non-negative numbers attached to all model elements that are decremented by each instantiation until they reach 0. In some sense, this solution is both too liberal and too restrictive at the same time: too liberal because at each meta-level the full potential of meta-model building facilities is available, but it is also too restrictive because the modeler must know in advance on which level the information will be needed and set a potency value accordingly.

Although potency allocation at end-points can be consistently extended to connections as well [6], next generation telecom management systems often require more flexibility vis-a-vis setting in advance the allowed number of multi-levels and less universality with respect to the permitted modeling facilities available at each modeling level. In other words, scheduled and gradual instantiation of information modeling is necessary. Under gradual instantiation we mean the

instantiation of some attributes and operations of the meta definitions, and not necessarily all of them in one single go. This added dynamicity in the instantiation is the main driver of our approach. Although our ASM formalism differs from the set theoretical foundation of potency based approaches we believe that it is at least so expressive for practical multi-level meta-modelling and simplifies the implementation of the solution. In order to prop up this conjecture three of the most frequent design patterns for multi-level meta-modelling [5] will be rewritten in DMLA.

3 Dynamic Multi-Layer Algebra

In this section, we shortly introduce our Abstract State Machines (ASM, [4]) based multi-level instantiation technique. Dynamic Multi-Layer Algebra (DMLA) consists of three major parts: The first part defines the modeling structure and defines the ASM functions operating on this structure. In essence, the ASM formalism defines an abstract state machine and a set of connected functions that specify the transition logic between the states. The second part is the initial set of modeling constructs, built-in model elements (e.g. built-in types) that are necessary to make use of the modeling structure in practice. This second part is also referred to as the bootstrap of the algebra. Finally, the third part defines the instantiation mechanism. We have decided to separate the first two parts because the algebra itself is structurally self-contained and it can also work with different bootstraps. Moreover, the a concrete bootstrap selection seeds the concrete meta-modeling capability of the generic DMLA, which we consider as an additional benefit compared to the unlimited and universal modeling capability potency supports at all meta-levels. In effect, the proper selection of the bootstrap elements determines the later expressibility of DMLA's modeling capability on the lower meta-levels.

3.1 Data representation

In DMLA, the model is represented as a Labeled Directed Graph. Each model element such as nodes and edges can have labels. Attributes of the model elements are represented by these labels. Since the attribute structure of the edges follows the same rules applied to nodes, the same labeling method is used for both nodes and edges. Moreover, for the sake of simplicity, we use a dual field notation in labeling that represents Name/Value pairs. In the following, we refer to a label with the name N of the model item X as X_N.

We define the following labels: (i) X_{Name} (the name of the model element), (ii) X_{ID} (a globally unique ID of the model element), (iii) X_{Meta} (the ID of the metamodel definition), (iv) $X_{Cardinality}$ (the cardinality of the model element, which is applied during the instantiation as an explicit constraint imposed on the number of instances the model element may exist in within the instance model), (v) X_{Value} (the value of the model element is only used in the case of attributes!), (vi) $X_{Attributes}$ (a list of attributes)

Due to the complex structure of attributes, we do not represent them as atomic data, but as a hierarchical tree, where the root of the tree is always the model item itself. Nevertheless, we handle attributes as if they were model elements. More precisely, we create virtual nodes from them. Virtual here means that these nodes do not appear as real (modeling) nodes in diagrams but – from the algebra's formal point of view – they behave just like usual model elements. This solution allows us to handle attributes and model elements uniformly and avoid multiplication of labeling and ASM functions. Since we use virtual nodes, all the aforementioned labels are also used for them: attributes have a name, an ID, a reference to their meta definition, a cardinality and they may have attributes as well. Moreover, they may also have a value. By the way, this is the reason why we have defined the Value label.

After the structure of the modeling elements has been briefly introduced, we now define the Dynamic Multi-Layer Algebra itself.

Definition 1. *The superuniverse* $|\mathfrak{A}|$ *of a state* $\mathfrak{A}$ *of the Dynamic Multi-Layer Algebra consists of the following universes: (i)* U_{Bool} *(containing logical values* $\{true/false\}$*), (ii)* U_{Number} *(containing rational numbers* $\{\mathbb{Q}\}$ *and a special symbol representing infinity), (iii)* U_{String} *(containing character sequences of finite length), (iv)* U_{ID} *(containing all the possible entity IDs), (v)* U_{Basic} *(containing elements from* $\{U_{Bool} \cup U_{Number} \cup U_{String} \cup U_{ID}\}$*).*

Additionally, all universes contain a special element, *undef*, which refers to an undefined value. The labels of the entities take their values from the following universes: (i) X_{Name} (U_{String}), (ii) X_{ID} (U_{ID}), (iii) X_{Meta} (U_{ID}), (iv) $\mathrm{X}_{Cardinality}$ ([U_{Number},U_{Number}]), (v) X_{Value} (U_{Basic}), (vi) X_{Attrib} (U_{ID}[]).

Note that we modeled cardinality as a pair of lower and upper limits. Obviously, this representation could be extended to support ranges (e.g. "1..3") as well. The label Attrib is an indexed list of IDs, which refers to other entities.

Now, let us have an example: $\mathrm{Book}_{ID} = 42$, $\mathrm{Book}_{Meta} = 123$, $\mathrm{Book}_{Cardinality} = \{0, \mathrm{inf}\}$, $\mathrm{Book}_{Value} = \mathrm{undef}$, $\mathrm{Book}_{Attrib} = []$ The definition formalizes the entity Book with its ID of 42 and the ID of its metamodel being 123. Note that in the algebra, we do not require that the universe of IDs uses the universe of natural numbers, this is only one possible implementation we use for illustration. In effect, the only requirement imposed on the universe is that it must be able to identify its elements uniquely. Now, one can instantiate any number of the Book entities in the instance model, which will have no components and values defined. For the sake of legibility, we will use a more compact notation from now on without loosing the original semantics:

`{"Book", 42, 123, [0, inf], undef, []}`.

3.2 ASM functions

Functions are used to define rules to change states in ASM. In DMLA, we rely on *shared* and *derived* functions. The current attribute configuration of a model item is represented using *shared* functions. The values of these functions are modified either by the algebra itself, or by the environment of the algebra (for example

by the user). *Derived* functions represent calculations, they cannot change the model, they are only used to obtain and restructure existing information. Thus, derived functions are used to simplify the description of the abstract state machine. The vocabulary $\sum$ of the Dynamic Multi-Layer Algebra formalism is assumed to contain the following characteristic shared functions: (i) **Name**(U_{ID}): U_{String}, (ii) **Meta**(U_{ID}): U_{ID}, (iii) **Card**(U_{ID}): [U_{Number},U_{Number}], (iv) **Attrib**(U_{ID}, U_{Number}): U_{ID}, (v) **Value**(U_{ID}): U_{Basic}.

The functions are used to access the values stored in the corresponding label. Note that the functions are not only able to query the requested information, but they can also update the information. For example, one can update the meta definition of an entity by simply assigning a value to the Meta function: Meta($ID_{ConcreteObject}$) $:=$ $ID_{NewMetaDefinition}$. Moreover, there are two derived functions: (i) **Contains**(U_{ID},U_{ID}): U_{Bool} and (ii) **DeriveFrom**(U_{ID},U_{ID}): U_{Bool}. The first function takes an ID of an entity and the ID of an attribute and checks if the entity contains the attribute. The second function checks whether the entity identified by the first parameter is an instantiation, also transitively, of the entity specified by the second parameter. The detailed, precise definition of the above functions are reported in [7].

3.3 Bootstrap mechanism

The aforementioned functions make it possible to query and change the model. However by only these constructs, it is very hard to use the algebra since it lacks the basic, built-in modeling constructs. For example, entities are required to represent the basic types, otherwise one cannot use the label Meta when it refers to a string because the label is supposed to take its value from U_{ID} and not from U_{String}. To draw a parallel, functions are like empty hardware components. They are useless unless an operation system to invigorates the system.

In DMLA, there is no universal setup for this initial set of modeling constructs. For example, one can restrict the usage of basic types to an absolute minimum, or one can extend them by allowing technology domain or meta-modeling specific types. Also, meta-modeling constructs such as attribute injection or inheritance may be defined explicitly here. Using our previous analogy: we can install different operating systems on our hardware for different purposes. It is worth mentioning that the bootstrap and the instantiation mechanism cannot be defined independently of each other. When an entity is being instantiated there are constructs to be handled in a special way. For example, we can check whether the value of an attribute violates the type constraints of the meta-model only if the algorithm can find and use the basic type definitions. The bootstrap presented in this paper provides a practically useful minimal set of constructs, however that can be freely modified if needed without changing the foundational algebra. The bootstrap has two main parts: basic types and principal modeling entities.

The built-in types of the DMLA are the following: *Basic*, *Bool*, *Number*, *String*, *ID*. All types refer to a value in the corresponding universe. In the bootstrap, we define an entity for each of these types, for example we create an

entity called *Bool*, which will be used to represent Boolean type expressions. Types *Bool*, *Number*, *String* and *ID* are inherited from *Basic*. Besides the basic types, we also define three principal entities: Attribute, Node and Edge. They act as the root meta elements of attributes, nodes and edges, respectively. All three principal entities refer to themselves by meta definition (more precisely, they are self-referring among themselves). Thus, for example, the meta of Attribute is the Attribute entity itself.

```
{"Attribute",IDAttr,IDAttr,[0,inf],undef,
 [{"Attributes",IDAttrs,IDAttr,[0,inf],undef,[]}]
}
```

We should also mention here that attributes are not only used as simple data storage units, but also for creating annotations that are to be processed by the instantiation. Similarly to basic types, we can define special attributes with specific meaning. By adding these annotational attributes to entities, we can fine-tune their handling. We define three annotation attributes: AttribType, Source and Target. AttribType is used as a type constraint to validate the value of the attribute in the instances. The Value label of AttribType specifies the type to be used in the instance of the referred attribute. Using AttribType and setting its Value field are mandatory if the given attribute is to be instantiated. AttribType is only applied for attributes.

```
{"AttribType",IDAttrT,IDAttr,[0,1],undef,[
 {"AType",IDAType,IDAttrT,[0,1],IDID,[]}
]}
```

Source and Target are used both as type constraints and data storage units to store the source and target node of an edge. The constraint part restricts which nodes can be connected by the edge, while the data storage contains its current value. The constraint is expressed by AttribType, while the actual data is stored in the Value field. The complete definition of the boostrap is presented in [7].

3.4 Dynamic instantiation

Based on the structure of the algebra and the bootstrap, we can represent our models as states of DMLA. Now, we will discuss the instantiation procedure that takes an entity and produces a valid instance of it. During the instantiation, one can usually create many different instances of the same type without violating the constraints set by the meta definitions. Most functions of the algebra are defined as shared, which means that they allow manipulation of their values also from outside of the algebra. However, the functions do not validate these manipulations because that would result in a considerably complex exercise. Instead, we distinguish between valid and invalid models, where validity checking is based on formulae describing different properties of the model. We also assume that whenever external actors change the state of the algebra, the formulae are evaluated. The complete definition of validation formulae is presented in [7].

The instantiation process is specified via validation rules that ensure that if an invalid model may result from an instantiation, it is rejected and an alternative instantiation is selected and validated. The only constraint imposed on this

procedure is that at least one instantiation step (e.g. instantiating an attribute, or model element) must succeed in each step. The procedure consists of instructions that involves a selector and an action. We model these instructions as a tuple $\{\lambda_{selector}, \lambda_{action}\}$ with abstract functions. The function $\lambda_{selector}$ takes an ID of an entity as its parameter and returns a possibly empty list of IDs referring to the selected entities. The function λ_{action} takes an ID of an entity and executes an action on it. The actions λ_{action} must invoke only functions previously defined for the ASM. Hence, the functions $\lambda_{selector}$ and λ_{action} can be defined as abstract, which allows us to treat them as black boxes. Also, the operations can be defined a priori in the bootstrap similar to attributes.

4 Multi-level Modeling with DMLA

In our opinion, the most effective way to demonstrate the applicability of DMLA to multi-level meta-modeling problems is through the reproduction of some of the reoccurring practical meta-modeling patterns reported in [5]. DMLA is a multi-level modeling approch, thus we focus only on the potency based formalism of those design patterns without any contemplation on their structure or benefits. Hence, the potency based definition of the those modeling patterns are copied verbatim from [5] and their equivalent DLMA constructs are produced in parallel. Also, the correspondence and/or potential differences between the two multi-level modeling formalisms are briefly explained by the various examples.

4.1 Type-Object pattern

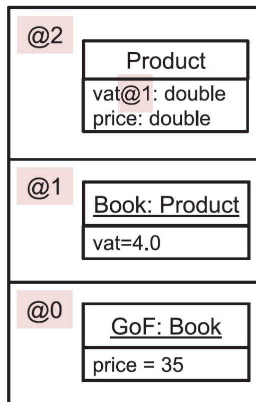
The pattern serves to dynamically add both types and instances to the model. The pattern is broadly applied in network management systems where new device types may be added to the network on-demand, in an ad-hoc fashion, and those types serve to facilitate the management of their instances.

The example below shows the gradual binding of attributes in both type and object level. While potency @1 indicates that *vat* must take its value in the next meta-level, @2 allows *price* to get its value after another meta-level jump.

The DMLA formalism defines *Product* as a Node instance with two Attributes whose value types are defined both as Numbers. Then, during the first instantiation Attribute *vat* is instantiated to 4.0, which is followed by the instantiation of *price* to 35. Since no further instantiation is possible the *GoF* object is ready.

4.2 Dynamic features

The pattern serves to dynamically add new attributes to a type which also become part of each instance of the type. The pattern is broadly applied in network management systems where existing device types may be extended by new features on-demand, in an ad-hoc fashion, and those features are automatically made manageable on all the corresponding instances.

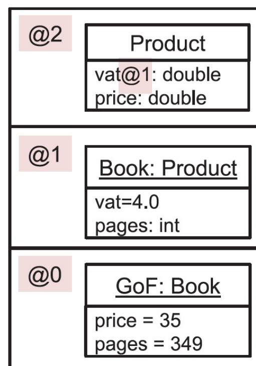



```
Level 2:
{"Product",IDProduct,IDNode,[0,inf],undef,
[
 {"vat", IDvat,IDAttribute,[1,1],undef,
  [{"vatType",IDvatT,IDAttribType,
     [0,1], IDNumber,[]}]
 },
 {"price",IDprice,IDAttribute,[1,1],undef,
  [{"priceType",IDpriceT,IDAttribType,
   [0,1],IDNumber,[]}]
 }
]}

Level 1:
{"Book",IDBook,IDProduct,[0,inf],undef,
[
 {"vat",IDvatC,IDvat,[1,1],4,[]},
 {"price",IDpriceC,IDAttribute,[1,1],undef,
 [
    {"priceType",IDpriceT,IDAttribType,
     [0,1],IDNumber,[]}
 ]}
]}

Level 0:
{"GoF",IDBookC,IDBook,[0,inf],undef,
[
 {"vat",IDvatC,IDvat,[1,1],4,[]},
 {"price",IDpriceC,IDprice,[1,1],35,[]},
]}
```

Fig. 1. The Type-Object pattern

The example below shows the addition of attribute *pages* to *Book* and its later instantiation within *GoF*. The DMLA formalism defines *Product* as a Node instance and further enables the potential addition of an arbitrary number of attributes in it. *Book* introduces the attribute pages and binds its type to Number. It also shuts down the possibility to append more attributes by setting the cardinality of Attribute to zero. Finally, within *GoF*, the pages takes its value as 349.

Similar to the type-object pattern, DMLA can correctly replicate the original potency example. Moreover, it provides the possibility to remove attributes by setting their cardinality to 0. This feature derives from the formal ASM definition of DMLA thanks to the explicit representation of cardinalities there. Hence, even though an attribute may be allowed at some position by its structural definition, it cannot be instantiated if the related upper cardinality is later set to 0.




```
Level 2:
{"Product",IDProduct,IDNode,[0,inf],undef, [
 {"vat", IDvat,IDAttribute,[1,1],undef,
  [{"vType",IDvatT,IDAttribType,[0,1],
    IDNumber,[]}]
 },
 {"price",IDprice,IDAttribute,[1,1],undef,
  [{"pType",IDpriceT,IDAttribType,[0,1],
    IDNumber,[]}]
 },
 {"Attribs",IDAF,IDAttribute,[0,inf],undef,[]}
]}

Level 1:
{"Book",IDBook,IDProduct,[0,inf],undef,[
 {"vat",IDvatC,IDvat,[1,1],4,[]},
 {"price",IDprice,IDAttribute,[1,1], undef,
  [{"priceType",IDpriceT,IDAttribType,
   [0,1],IDNumber,[]}]
 },
 {"pages",IDpage,IDAttribute,[1,1],undef,
  [{"pType",IDpageT,IDAttribType,[0,1],
    IDNumber,[]}]
 }
]}

Level 0:
{"GoF",IDBookC,IDBook,[0,inf],undef, [
 {"vat",IDvatC,IDvat,[1,1],4,[]},
 {"price",IDpriceC,IDprice,[1,1],35,[]},
 {"pages",IDpageC,IDpage,[1,1],349,[]},
]}
```

Fig. 2. Dynamic features

4.3 Dynamic auxiliary domain concepts

The pattern serves to dynamically add new entities to a type whose instances will be correctly related to the instance of the type. Also, the new entities may have attributes and further related entities. The pattern is broadly applied in network management systems where new network concepts are added to device types based on network technology evolution, and those concepts and their instances automatically become part of the management system. Due to the page limits only an excerpt of the DLMA representation of the full design pattern is shown here. In essence, this design pattern is a mixture of the previous two with the extension that the meta-model must provide a possibility to inject new Nodes and Edges at will. Therefore, a "root container" element, let us call it *Domain*, is to be added to the original bootstrap.

```
{"Domain", IDDomain, IDNode, [0, inf], undef,[
 {"Nodes", IDnodes, IDNode, [0,inf], undef, [ ]},
 {"Edges", IDedges, IDEdge, [0,inf], undef, [ ]}
]}
```

Then, arbitrary domain concepts can be introduced dynamically into *Domain* until the model is ready as

```
... {"authors", IDauthors, IDEdge, [0,inf], undef, [
 {"Source", IDautSrc, IDSrc, [1,1], undef, [
  {"SType",IDSType,IDAttribType,[0,1],IDBook,[ ]}]},
 {"Target", IDautTrg, IDTrg, [1,1], undef, [
  {"TType",IDTType,IDAttribType,[0,1],IDAuthor,[ ]}]}, ...
```

5 Conclusion and Future Work

We have applied our novel multi-level meta-modeling approach, DLMA, to three well-known design patterns for deep meta-modeling. During this exercise, our immediate purpose was to illustrate the expressivity of DLMA by rewriting these well-known design patterns that were already published [5] in a mainstream multi-level modeling formalism. Our solution seems to allow higher level of dynamism in instantiation than those existing solutions do, thus it offers a more implementation ready formalisation of instantiation. Moreover, DMLA enables to use different bootstrap alternatives, which may ultimately recreate the full flexibility of state-of-the-art meta-model building facilities modeling professionals of particular technical domains would need. Hence, our concrete goal is to implement the presented approach and to investigate different bootstraps (e.g. adding operations, association classes) to validate the full capability of the approach.

References

1. Eclipse modeling framework (EMF), 2015. https://eclipse.org/modeling/emf/.
2. Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 19–33. Springer-Verlag, 2001.
3. Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20(5):36–41, September 2003.
4. E. Borger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., 2003.
5. Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2):12:1–12:46, 2014.
6. Bernd Neumayr, Manfred A. Jeusfeld, Michael Schrefl, and Christoph Schütz. Dual Deep Instantiation and Its ConceptBase Implementation. In *CAiSE 2014*, volume Vol. 8484 of *LNCS*, pages 503–517, 6 2014.
7. Zoltan Theisz and Gergely Mezei. Towards a novel meta-modeling approach for dynamic multi-level instantiation. In *Automation and Applied Computer Science Workshop*, 2015. http://vmts.aut.bme.hu - Download - Papers.