Multi-Level Modelling in the Modelverse

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

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

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

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

1 Introduction

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

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

2 Background

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

2.1 Deep Instantiation and Deep Characterization

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

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

2.2 Tool Support

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

3 The Modelverse: Overview

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

3.1 Architecure of the Modelverse

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

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

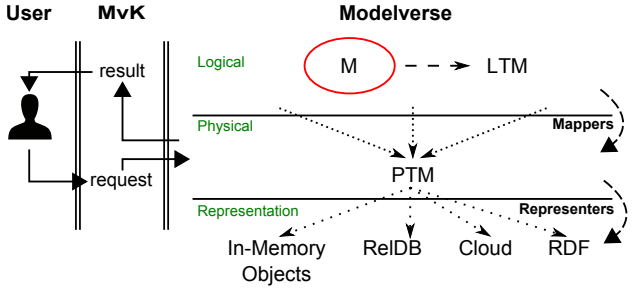

Fig. 1. The framework on which the development of the Modelverse is based.

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

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

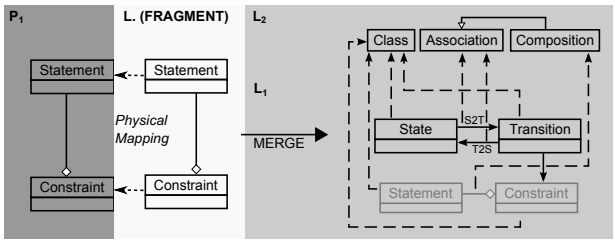

Fig. 2. An example of a modelling language fragment.

3.2 Modular Language Design

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

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

4 Case Study

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

4.1 A Visual Notation

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

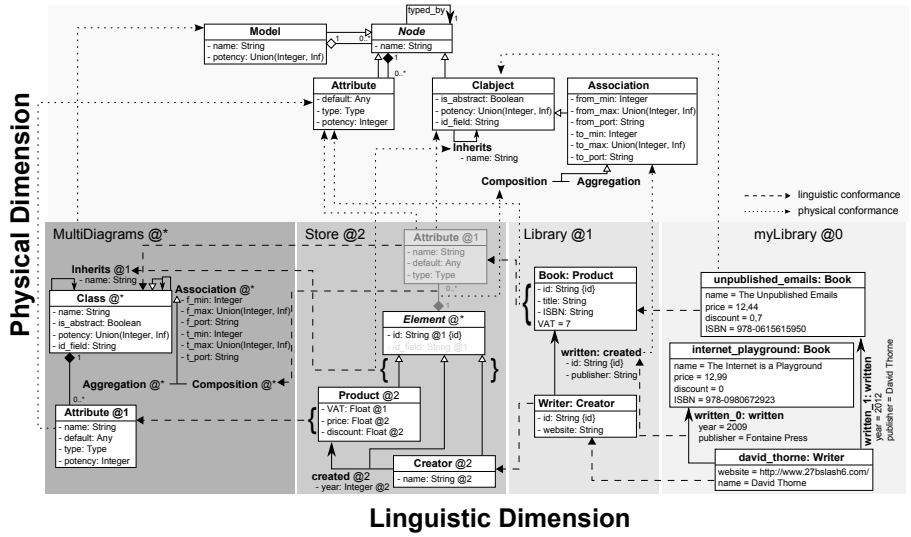

Fig. 3. The example modelling hierarchy.

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

4.2 Physical Representation

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

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

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

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

   Attribute:
    name = 'id'
    type = String

   Attribute:
    name = 'id_field'
    type = String

  Class:
   name = 'Product'

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

   Attribute:
    name = 'price'
    type = Float

   Attribute:
    name = 'discount'
    type = Float

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

  Class:
   name = 'Creator'

   Attribute:
    name = 'name'
    type = String

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

  Association:
   name = 'created'

   Attribute:
    name = 'year'
    type = Integer

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

Listing 1. Textual notation for Model Store

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

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

   Attribute:
    name = 'id'
    type = String

   Attribute:
    name = 'title'
    type = String

   Attribute:
    name = 'ISBN'
    type = String

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

   Attribute:
    name = 'id'
    type = String

   Attribute:
    name = 'website'
    type = String

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

   Attribute:
    name = 'id'
    type = String

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

Listing 2. Textual notation for Store Library

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

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

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

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

Listing 3. Textual notation for myLibrary

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

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

4.3 A Textual Representation: the HUTN

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

5 Conclusions and Future Work

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

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

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

References

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