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) $\mathrm{X}_{Cardinality}$ ($[\mathrm{U}_{Number,}\mathrm{U}_{Number}]$), (v) X_{Value} (U_{Basic}), (vi) X_{Attrib} ($\mathrm{U}_{ID}[]$).

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: $\text{Meta}(\text{ID}_{ConcreteObject}) := \text{ID}_{NewMetaDefinition}$. 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

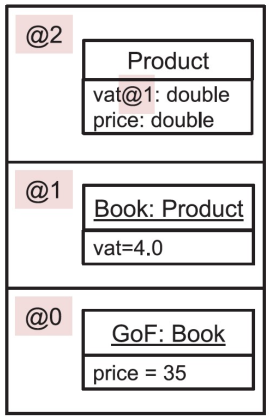
The pattern serves to dynamically add both types and instances to the model. The pattern is broadly applied in network management systems where new device types may be added to the network on-demand, in an ad-hoc fashion, and those types serve to facilitate the management of their instances.

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.

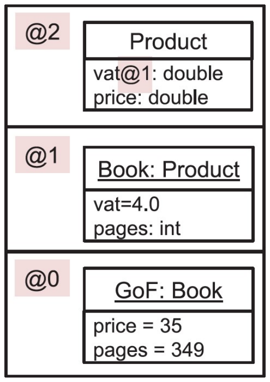


```
Level 2:
{"Product",IDProduct,IDNode,[0,inf],undef,
[
 {"vat", IDvat,IDAttribute,[1,1],undef,
  [{"vatType",IDvatT,IDAttribType,
     [0,1], IDNumber,[]}]
 },
 {"price",IDprice,IDAttribute,[1,1],undef,
  [{"priceType",IDpriceT,IDAttribType,
   [0,1],IDNumber,[]}]
 }
]}

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.