Periodica Polytechnica
Electrical Engineering
and Computer Science

61(1), pp. 34-47, 2017
DOI: 10.3311/PPee.9547
Creative Commons Attribution

RESEARCH ARTICLE

Formalism for Static Aspects of Dynamic Metamodeling

Dániel Urbán[1], Gergely Mezei[1*], Zoltán Theisz[2]

Received 02 June 2016; accepted 16 October 2016

Abstract
The viability of any multi-level meta-modeling technology clings on the correct and exact definition of the underlying instantiation concept. Although multi-level instantiation has been well researched and conceptualized, current methodologies mostly limit their scopes to design-time modeling. Hence, state-of-the-art meta-modeling seriously neglects important practical needs of run-time modeling.

In this paper, we present two incarnations of our Dynamic Multi-Level Algebra (DMLA), a modular, semantically correct multi-level meta-modeling formalism consisting of (i) an algebraic ASM foundation, (ii) a flexibly replaceable bootstrap mechanism for defining modelled entities, (iii) a corresponding set of logical validation formulae for establishing instantiation semantics. The expected features of DMLA clearly derive from practical modeling needs so that the resulting mechanism be applicable for both design-time and run-time modeling. The core components of the theory are explained both individually and by comparing their two versions of DMLA. Hence, the aim of this style of presentation is to showcase the balance struck between parsimoniousness and wanted modularity within our semantically correct, practical multi-level meta-modeling approach. A simplified networking model is also included to demonstrate the approach.

Keywords
dynamic instantiation, multi-level meta-modeling, algebraic formalism, modular meta-modeling, meta-modeling lifecycle

[1] Department of Automation and Applied Informatics,
Faculty of Electrical Engineering and Informatics,
Budapest University of Techology and Economics
H-1117 Budapest, Magyar tudósok krt. 2., Hungary

[2] evopro Innovation Ltd.,
H-1116 Budapest, Hauszmann Alajos str. 2, Hungary

[*] Corresponding author, e-mail: gmezei@aut.bme.hu

1 Introduction

Metamodeling has become a well-established software engineering methodology that has standardized the way software architects build practical models for various complex software intensive applications on industrial scale. The models may serve many purposes, but the most important thereof are the different domain specific analyses, congruent model transformations and almost full-automatic code generation. Although modeling tools have matured a lot and the Eclipse Modeling Framework (EMF) [1] started to dominate the technology, the core paradigm still relies on the four level semantics of OMG's MOF [2]. Nevertheless, the number of meta-levels turned out to become rather limited. In theory, three meta-levels are available for modeling, taken for granted that level M3 is fixed by the OMG. However, in practice, only level M2 and M1 are freely available in design-time and only M0 is used in run-time explicitly. Moreover, the separation between design-time and run-time modeling is kept quite rigid; there is no automatic mechanism available that validates M0 models against their M1 meta-model. Obviously, specific deployments, for example highly configurable adaptive systems may allow case-by-case application of such solutions; however, these are merely exceptions rather than the deployment of state-of-the-art technologies. In summary, one may claim that EMF provides, by default, only two modeling levels implemented in a single threaded design environment.

Taken into account that concurrent model manipulation is also needed at run-time, Models@runtime solutions resurfaced to solve the challenge of run-time concurrent model management. Nevertheless, these technologies do not facilitate standard modeling tools such as model transformation or flexible code generation. Hence, a unified multi-level modelling framework for both design- and run-time applicability is still missing. This paper focuses on the fundamentals of a solution capable of fixing the aforementioned shortcomings of current modeling approaches.

Since instantiation is the real essence of any metamodeling discipline, the best way to start formalizing a multi-level metamodeling approach is through a semi-formal definition of the requirements imposed on the instantiation. All current instantiation approaches share the following principle: let us take a meta-definition and process it by instantiating all defined items of the definition. In other words, it means that, for example, if we have three attribute definitions, then we are forced to instantiate all those three variables at the same time. We cannot decide, for example, to instantiate only two of them right now and keep the last item in the state of being uninstantiated.

Dynamic Multi-Level Algebra (DMLA) is the formal definition of our multi-level modeling approach which acknowledges all requirements presented above. DMLA offers a rather flexible and customizable modelling structure and it can easily deal with both design-time and run-time aspects of modelling. To achieve this, the concept of instantiation is dynamic in spirit and the formalism is able to account for explicit model states and not just for simple isolated snapshots. The basic mechanism of DMLA is based on Abstract State Machines (ASM).

DMLA has two incarnations: the first one (DMLA 1.0) followed the naive approach of each major concern having been allocated to its own slot of representation in a 6-tuple structure defined over the ASM representation. We aimed to use DMLA 1.0 for experimentation in order to test and compare the expressiveness of DMLA to other multi-level modeling techniques [3-5]. Although the approach proved its merit we could also recognize some of its technical shortcomings which were later fixed by DMLA 2.0, mainly in the fields of modularity, flexibility and conceptual harmonization.

Although a short introduction of DMLA 1.0 is given in the paper, the main focus is on DMLA 2.0. The formal definition of the modelling structure, the mechanisms of instantiation (including validation formulae to ensure validity of models) is elaborated in detail. A short, illustrative case study is also presented in order to illustrate the concepts in practice.

2 Related Work

Multi-level metamodeling has enjoyed its renaissance during the last couple of years thanks to the reemerging interest in flexible modeling approaches and the slight disillusion of incumbent four level MOF and two level EMF modeling techniques. Although there are many flavors of multi-level modeling, all possess some facilities that enable instantiation across multiple levels. In general, there are two variants of instantiation:

1) shallow instantiation where the information is defined at modeling level n and this information is directly used at the immediate instantiation level n+1,
2) deep instantiation that allows to define some information on the nth modeling level which can later be used on the (n+x)th (x > 0) modeling level [6].

One of the earliest and probably the most well-known deep instantiation approach is the so called potency notion [6]. The core idea is both simple and genuine: non-negative numbers are attached to all model elements which are then decremented by each instantiation until they reach 0, where no further instantiation is permitted. The approach works well and has been successfully implemented even in EMF by Melanee [7]. Nevertheless, potency notion bears also some disadvantages due to its Orthogonal Classification Architecture (OCA) [8] because OCA assumes that all meta-model management facilities are universally accessible in such a way as if almost the usual full MOF potential were available at all meta-levels. In general, instantiation is only controlled by one explicitly defined scalar value, however, Melanee subdivides it into potency, durability and mutability. Nevertheless, potency derived values can be decremented only synchronously, on each model elements at the exact moment of an instantiation, which results in a preset number of meta-levels each element can be instantiated at. This inflexibility requires fine-grained harmonization of the instantiation process throughout the entire metamodel, which makes modular design of complex industrial applications rather difficult to achieve. The apparent fragility of the method stems from the fact that potency notion is both too permissive and too restrictive at the same time. It is too permissive because at each meta-level the full potential of all meta-model building facilities is available, but it is also too restrictive because the model designer must know in advance at which meta-level the final instances will be needed and the potency values must be set accordingly. Also, although OCA does not prescribe it, potency notion is slightly asymmetrical since it prefers nodes to edges within the meta-model building process. Nevertheless, potency allocation at the end-points can be consistently extended towards the edges via DDI [9], the rigidity of the original approach has not been effectively relaxed thereby, that is, nodes still remain to be preferred to edges.

By OCA becoming the mainstream of multi-level metamodeling there seems to be a tendency to implement deep instantiation frameworks by predominantly relying on the clear differentiation between linguistic and ontological instantiation. Note that the terminology "ontological" in this context only relates to the things that exist in the universe of discourse of the domain to be modeled and it has nothing to do with contemporary ontological research [10]. In fact, OCA has introduced an explicit ontological representation into the original linguistically defined MOF meta-model interpretation. Nevertheless, this technique has not novel at all since UML also allowed the usage of both class and corresponding object diagrams in the same model: both being represented as instances of meta-concepts. However, their association had to be taken for granted and implemented implicitly by any UML compatible tooling. In order to extend UML's special interpretation of combined linguistic-ontological metamodeling, OCA generalizes ontological representation in such a way that the ontological instantiation [11] must enable any particular M1 (or M2) model element to become an ontological instance of another element of the same M1 (or M2) level) multi-level. Hence, although MOF meta-levels are still linguistically defined in OCA, the domain semantics needed for deep multi-level instantiation can be theoretically introduced within the frame of any existing MOF compatible tools. Nevertheless, the approach has been only sporadically used and only Melanee succeeded in this endeavor. Nevertheless, OCA has been implemented also by metaDepth [12], one of the most successful frameworks among tools enabling building of systems with arbitrary number of meta-levels through predominantly ontological deep metamodeling. A clear application of showcasing the benefits of multi-level instantiation over classical MOF techniques has been demonstrated by comparing two-level and multi-level modeling methods through the example of CloudML [13] and some well-known design patterns used in practical metamodeling [14].

Finally, one must not forget that UML's standard profiling mechanism also showcases some mechanisms to mimic shallow multi-level modeling via mixed multi-level concepts such as meta-classes representing descriptors (e.g. Classifier) or items (e.g. InstanceSpecification). One of the best known application of these concepts can be found in MARTE [15], which clearly demonstrates the relevance of and the need for multi-level modeling for industrial solutions. Hence, multi-level metamodeling, if it is done in the proper way, may revitalize also classical modeling disciplines.

3 Multi-level metamodeling in practice

Multi-level metamodeling technologies have matured a lot during the years, however their practical applicability is still rather limited. Some of the reasons behind this retard may be that usually, contemporary industry solutions still consist of ad-hoc pattern-based multi-level meta-modeling implementations, which are also combined with either relational database techniques or XML technologies to enable meta-level shifts between meta-levels via proprietary domain specific promotion and demotion operations. Furthermore, current multi-level modeling frameworks are mainly visualization driven (e.g. Melanee, XModeler [16]) or though textual (e.g. metaDepth), but still not capable enough to meet the scalability needs of real industrial applications, or simply prefer design-time focus to run-time modeling aspects due to their EMF heritage. However, in practical industrial modeling, there is a definitive need for ecosystem thinking when it comes to build large-scale model-based applications. In principle, traditional model-based software components have been designed without effectively considering life-cycle and inter-component integration. Therefore, all modeling activities had to be finished before the application was about to be deployed. However, in the era of Cloud-deployed component applications, the instances of a meta-model-based application - a potentially infinite number of them - may remain active and thus must be kept alive for an extended period of time. Moreover, entities, besides being instances of a metamodel, may also play the role of templates, i.e. a meta-model, for other instances running within the same application. Hence, recursive meta-model nesting is not an exception now, but it is the real nature of modern Cloud-aware component software solutions. In effect, the design-time and run-time aspects of modeling are to be mixed and even blended; therefore, meta-models cannot and must not be sealed after design-time instantiation. As a consequence, the concept of instantiation must be multi-level and such that it covers both design-time and run-time perspectives.

The abstract requirements for a genuine representation of practical multi-level instantiation can be summarized as follows:

- Instantiation copies the structure of the meta-definition and decides whether to instantiate child elements one by one
- Instantiation of a type results in a compatible value within the same meta-definition slot
- Instantiation of a cardinality limit results in a more restricted value imposed on the same meta slot
- Instantiation of type constraints results in further restriction(s) on the set of valid instances

In practice, the consequence of the above instantiation rules will result in a system that enables the automatic creation of a hierarchy of meta-definition templates where template variables are gradually substituted in a well-orchestrated sequence of execution. Also, optional template variables may even be simply left out during the same instantiation process. The proposed instantiation process also covers both design- and run-time aspects of meta-model based software development by allowing the integrated and semantically correct co-existence of all model elements, from the very abstract top-level concepts down to their most concrete executable instances.

Dynamic Multi-Level Algebra (DMLA) is the formal definition of our multi-level modeling approach, which acknowledges all the requirements presented above. It supports dynamic instantiation. The instantiation is explicitly given by a meta-reference that directly connects the instances to their meta-definition. In order to provide a uniform representation for both meta- and instance data, the formalism applies multi-slot tuple encoding. The tuple encoding must support composability by allowing the explicit representation of substructures as children. This feature is vital since it formalizes the template induced origin of complex meta-definitions. Moreover, the multiplicity of substructures is determined by the explicitly given cardinality information that can also be instantiated by further restrictions on their allowed domains. Similarly, non-cardinality related constraints can be attached to the substructures and they can be instantiated by imposing further limitations on their domains of satisfiability. Finally, substructures can also be represented by simple, built-in datatypes whose instantiation is accomplished as a traditional value selection which is validated by domain checks on the permitted values.

4 Dynamic Multi-Layer Algebra 1.0

In this section, we introduce the base concepts of our original Dynamic Multi-Layer Algebra (DMLA). DMLA is a multi-level instantiation technique based on Abstract State Machines (ASM, [16]). It consists of three major parts: The first part defines the modeling structure and defines the core ASM functions operating on this structure. In essence, it 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 practical applications. 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, any concrete bootstrap selection seeds the concrete metamodeling 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.

4.1 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. For simplicity, we use a dual field notation in labelling of 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:

- X_{Name}: the name of the model element
- X_{ID}: a globally unique ID of the model element
- X_{Meta}: the ID of the meta-model definition
- $X_{Cardinality}$: the cardinality of the model element, it is used during instantiation as a constraint. It determines how many instances of the model element may exist in the instance model.
- X_{Value}: the value of the model element (used in case of attributes only as described later)
- $X_{Attributes}$: 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: 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, e.g. attributes have a name. Moreover, they may also have a value. This is the reason why we have defined the Value label. In order to avoid any misunderstanding, in the following, we are going to use the word *entity* exclusively if we refer to an element which has the label structure. Let us now define the algebra itself.

Definition The superuniverse $|\mathfrak{A}|$ of a state $\mathfrak{A}$ of the Multi-Layer Algebra consists of the following universes:

- U_{Bool} containing logical values {true/false}
- U_{Number} containing rational numbers $\{\mathbb{Q}\}$ and a special symbol ∞ representing infinity
- U_{String} containing character sequences of finite length
- U_{ID} containing all the possible entity IDs
- 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) $X_{Cardinality}$: $[U_{Number\,,}\ U_{Number}]$, (v) X_{Value}: $U_{Basic,}$ (vi) X_{Attrib}: $U_{ID}[]$. The label Attrib is an indexed list of IDs, which refers to other entities.

The label Attrib is an indexed list of IDs, which refers to other entities. Now, let us have a simple example:

$$\mathbf{Router}_{ID} = 12,\ \mathbf{Router}_{Meta} = 123,$$
$$\mathbf{Router}_{Cardinality} = [0,\ \infty],\ \mathbf{Router}_{Value} = \mathbf{undef},$$
$$\mathbf{Router}_{Attrib} = []$$

This definition formalizes the entity Router with its ID being 12 and the ID of its meta-model being 123. In the sequel, we will rely on a more compact representation with equal semantics.

```
{"Router", 12, 123, [0, ∞], undef, []}
```

4.2 Functions

Functions are used to rule how one can change states in the 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. *Derived* functions represent calculations which cannot change the model; they are only used to obtain and to restructure existing information. The vocabulary $\sum$ of DMLA is assumed to contain the following characteristic 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 labels. We suppose that the Attrib labels return undef when the index is greater or equal to the number of the stored entities. Note that the functions are not only able to query the requested information, but they can also update it. For example, one can update the meta definition of an entity by simply assigning a value to the Meta function.

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.

4.3 Bootstrap Mechanism

The ASM functions define the basic structure of our algebra. The functions allow to query and change the model. However, based only on these constructs, it is hard to use the algebra due to the lack of basic, built-in constructs. For example, entities are required to represent the basic types; otherwise one cannot use label Meta when it refers to a string since the label is supposed to take its value from U_{ID} and not from U_{String}. We need to define those base constructs somewhere inside or outside of the core algebra. Obviously, there may be more than one "correct" solution to define this initial set of information. Here, we restrict the usage of basic types to an absolute minimum. The bootstrap has two main parts: *basic types* and *principal entities*.

4.3.1 Basic Types

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.

4.3.2 Principal Entities

Besides the basic types, we also define two principal entities: *Attribute* and *Base*. They act as root meta elements of attributes, and combined node and edge meta-types, respectively. Both principal entities refer to themselves by meta definition. Thus, for example, the meta of *Attribute* is the *Attribute* entity itself. For example, the definition of *Attribute* describes that an attribute can have zero or more attributes as children.

```
{"Attribute", ID_Attrib,ID_Attrib,[0,inf],undef,
 [
  {"Attributes",ID_Attribs,ID_Attrib,[0, ∞],undef,[]}
 ]}
```

The third principal entity, *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 is mandatory if the given attribute is to be instantiated, otherwise *AttribType* can be omitted. The definition of *AttribType* is an instantiation of *Attribute* and it also uses *AttribType* to restrict its own type.

```
{"AttribType", ID_AType,ID_Attrib, [0,1], undef,
 [
  {"AType", ID_ATypeType, ID_AType, [0,1],ID_ID,[]}
]}
```

4.4 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 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 formulae (detailed and explained in [4, 5]) defines valid instantiation as follows:

1. All attributes of the Instance must be a *clone*, a *copy*, or a valid *instantiation* of an attribute of the MetaType.
2. If it is a *clone*, then the same entity is used in the Instance as in the MetaType. The definition is transferred to the next level without any modification.
3. If it is a *copy*, then the ID label must be, while the Cardinality label may be changed.
4. If it is an *instantiation*, then it must always have at least one instantiated (sub)attribute, or its value must be set.
5. The accumulated (copied, instantiated) instances must not violate the cardinality constraint defined by the meta definition.
6. If a component is a direct or indirect instantiation of Attribute and it has a Value set, then its meta definition must have an AttribType component and the type of value must match the type defined by AttribType. The only exception to this formula are attributes deriving from AttribType itself, for which we validate the Value field against the Value of meta definition directly.

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 $\lambda_{selector}$ 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.

5 DMLA 2.0

Although the original version of DMLA possesses a universal modeling capability, we found a few features which can be improved for greater flexibility and easier usage. Three main factors drove us in this direction. The first aspect regards the universality of our system: we recognized that by lifting certain features from our core data representation into the bootstrap makes the foundation of DMLA even more generic. The source of the second improvement was that we recognized that the mandatory concretization rule (rule #4 in the list of formulae for DMLA 1.0) can be avoided. The third improvement fixes a few cases of ambiguity we have found in the original version of DMLA. Hence, we established a more compact type system, which resulted in much cleaner validation formulae and also eliminated the ambiguities.

These changes affect multiple areas of the original algebra, however the role of the basic components (ASM-based core, bootstrap and instantiation mechanism) and the principal ideas behind the theory remain intact. In this section, we introduce the changes we have applied to DMLA, and we will also examine the effects these changes may have on the semantics and flexibility of the original DMLA mechanisms.

5.1 Data representation and functions

The structurally most important change we carried out was the reduction of the number of labels used in the representation of entities. Choosing the number of labels in the representation is a practical trade-off between flexibility, usability and the level of abstraction. The more labels one has, the more built-in features one can rely on in the bootstrap level. However, if the labels are fewer, the bootstrap becomes more customizable. More precisely, although the labels limit the expressiveness of the core, they also help to impose essential constraints one must follow in the models. In the extreme case of having only one label (a universal set of general attributes), the resulting ASM would necessitate a full behaviour customization within its bootstrap. However, this representation would be too general for the purposes of generic multi-level metamodeling: it would neither restrict the usage of attributes, nor define precisely their semantics. Also, the identification of entities and meta relationships would become part of the bootstrap, which would go against the general principles of DMLA.

We decided to reduce the number of labels from six to four, by leaving out labels X_{Name} and $X_{Cardinality}$. This move unlocks a few new possibilities in customizing the bootstrap, while it still encompasses all the main concepts as built-in, mandatory features.

The sole purpose of having label X_{Name} was to improve the legibility of the model, it had no other semantic meaning. We realized that by using a more readable, string-based ID convention, we can replace it perfectly.

On the contrary, the label $X_{Cardinality}$ does have a real and important semantics in meta-modeling. Nevertheless, in DMLA, cardinality constraints are validated by the validation formulae, which anyhow depend on the bootstrap due to some special treatments of selected entities (e.g. AttributeType). Therefore, it turned out to become too rigorous to restrict the format and thus the expressiveness of cardinality constraints by the core of DMLA. For example, if someone wanted to define valid cardinality ranges instead of the ASM default of a lower and an upper limit, (s)he would have to unnecessarily modify the core. In order to overcome this limitation, we moved the cardinality from the core ASM representation as reified feature into the bootstrap by adding it as an attribute to all containment slots in the model entities.

The new ASM representation has four labels: X_{ID} identifies and references to model elements. Since DMLA's main focus lies on multi-level instantiation, X_{Meta} encodes type – instance relationships. As mentioned before, these two labels could be a part of a generic attribute set, but since these are relevant in most use-cases, extracting them makes our concept easier to use. Keeping the $X_{Attributes}$ label is a necessity in order to provide composability of entities. Finally, X_{Value} serves to differentiate contained attributes from fully-instantiated values resulting in simple and clear validation formulae. The only minor change between the original and the new ASM formalism is to enable X_{Value} to become a list, making it possible to store multiple values in the slot.

The above changes do not affect the data representation of the superuniverse. The modified X_{Value} label is now defined as $U_{Basic}[]$, therefore we also changed the Value(U_{ID}) function to Value($U_{ID,}$ U_{Number}), allowing direct indexing of list of values. Also, the derived functions Name(U_{ID}) and Card(U_{ID}) have been removed.

The Router example expressed by the new representation:

```
{"Router", "RouterMeta", undef, []}
```

5.2 Mandatory Concretization

In DMLA 1.0, instantiation of an entity was valid only if at least one of its meta slots was concretized. The restriction was included (encoded) in the validation formulae. The original purpose of mandatory concretization was the aim to avoid unlimited instantiation chains, for example by instantiating an element again and again, without changing anything in it.

When DMLA 2.0 was created, we reconsidered the constraint. We have realized that by allowing infinite cardinality in entities, unlimited chains may occur even if the condition is satisfied. Thus, we have removed the constraint on mandatory concretization from the formulae.

However, we have also recognized that by extending the current bootstrap of DMLA 2.0 at some key points, one can easily impose the original concretization constraint again. These key points are also mentioned in the paper.

5.3 Bootstrap Mechanism

DMLA 2.0 regularized the exceptional cases of the original DMLA considering the validation formulae, the instantiation procedure and also created a bootstrap that handles these cases more uniformly. The key changes applied to the bootstrap are as follows:

- We have created a new entity *Base* that is root meta entity of every model element. It has two instances, *Entity* and *SlotDef*. *Entity* is used as the root entity for all usual model entities, while *SlotDef* represents a slot definition to be filled later with concrete values. *SlotDef* acts as a wrapper for all constraints on the contained attributes and references.
- *Since Entity* is the meta of every entity except slot definitions, one can use *Entity* in type constraints as a reference instead of their IDs. Therefore, we have removed the *ID* and the *Basic* types.
- Since the cardinality label has been removed from the ASM, it had to be re-introduced into the bootstrap, thus, we have created a *Cardinality* entity.
- *SlotDef* wraps all constraints of a given slot as mentioned earlier. This means that it supersedes the original principal entity *AttribType*. However, *SlotDef* does not only take over the role of simple type validation, but also enables for example a wrapping for *Cardinality* constraint and a generic extension point for defining further constraints (e.g. range restrictions) on attributes.
- We have also changed how validation formulae work. Instead of using the same formula for each model entity, we now allow entity dependent formulae, i.e. a formula may act differently for different entities. For example when validating *Base*, we can be less restrictive, than when validating a constraint. This way, validation is more modular and easier to customize.
- The mechanism of entity validation is split into two separate kinds of formulae now. The first (α) kind is used to validate the instantiation of an entity against one of its instances. The second (β) kind has to evaluate the validity of instantiation in context, checking an entity against a list of entities, consisting of clones and instances. The second kind of formulae can validate for example whether the cardinality constraint is violated by the list of candidates. Note that if we wanted to add the mandatory concretization check from DMLA1.0 to the bootstrap, we could add a third (γ) kind of formulae, which would evaluate the concretization in context and call this third kind of validation from the appropriate α formulae, respectively.

5.3.1 The Base entity

In DMLA 2.0, the *Base* entity has become the root meta of every entity. In order to eliminate apparent self-meta recursion, the meta of the *Base* entity is set to *undef* since principal entities were anyhow treated exceptionally regarding their meta relation in the original DMLA as well.

Since *Base* acts as the root meta, it must be based on the most flexible structure DMLA may enable, that is, it consists of an arbitrary amount of slots of any type. This is expressed by adding the *SlotDef* entity as an attribute to *Base*.

We have also added an additional slot called *IsPrimitive* in order to draw a clear distinction between instances of *Base*. *IsPrimitive* attribute marks basic types (e.g. string), while every other entity eliminates this slot, when instantiating *Base*. Realizing this structure, the illustration of Base is the following:

```
{"Base", undef, undef,
[
"SlotDef",
{"IsPrimitive", "SlotDef", undef,
 [ ["BOOL"], [0, 1] ]
 }
]}
```

5.3.2 The Entity entity

In DMLA 2.0, *Entity* entity is the root meta of every entity except the slots. Flexibility is a key feature here similarly to *Base*, but we must restrict the contained elements type to *Base*. This restriction is applied via an instance of *SlotDef*, as explained later. The *Entity* entity also clones the *IsPrimitive* flag of the *Base*, since every basic type is an instance of *Entity*.

```
{"Entity", "Base", undef,
[
{"Children", "SlotDef", undef,
 [ ["Base"], [0, ∞] ]
  },
 "IsPrimitive"
]}
```

5.3.3 The SlotDef entity

The *SlotDef* entity is meant to represent a slot of the containing entity. In this bootstrap, it has two parts: a type constraint and a cardinality constraint. In the future, we are going to create other bootstraps containing more constraints (e.g. range check for integers, or regular expression constraints on strings).

The *SlotDef* has two ways of instantiating: either a value is provided, which conforms to the constraints set in the meta *SlotDef*, or no value is provided, but some of the constraints of the *SlotDef definition is concretized*. The wrapper of *SlotDef* feature clearly formalizes the generic instantiation principle of DMLA. In effect, *SlotDef* controls and helps validating the contained values, therefore *SlotDef* is the only element capable of containing a value.

For example, two possible instances of the aforementioned *Children* slot of the *Base* would be:

```
//concretized SlotDef
{“SampleEntity”, “Entity”, undef,
[
 {“SpecChild”, “Children”, undef, [
 [“Base”],//clone from the meta
 [1, 2]   //concretized cardinality instance
 ]}
]}

//filled-in SlotDef
{“SampleEntity2”, “Entity”, undef,
[
 {“SpecChild2”, ”Children”,
  [“ChildID”], undef}
]}
```

SlotDef makes DMLA 2.0 more powerful, consistent and parsimonious in labels since one single ASM label spawns a separate entity in the bootstrap in order to represent every further constraint in the model (type and cardinality). The bootstrap itself can handle the concretization of the constraints as instantiations rather than delegate it as exceptions within validation formulae [1]. This also provides a hook for further extensions: for example, one may create an alternative representation of cardinality instead of the default min-max pair, simply as an instance of the *Cardinality* entity.

The following definition is the definition of *SlotDef*. Note that the parts of the *SlotDef* entity refers to SlotDef as their meta, making it self-descriptive, but not self-meta recursive.

```
{“SlotDef”, “Base”, undef,
[
{“TypeConstraint”, “SlotDef”, undef,
 [[“Base”],[1,1]]},
{“CardConstraint”, “SlotDef”, undef,
  [[“Cardinality”],[1, 1]]}
]}
```

Although the description notation used for the entities are expressive and compact, we have also created a more intuitive graphical notation to visualize the entities. The definition of *Entity, SlotDef and the examples can be* visualized as:

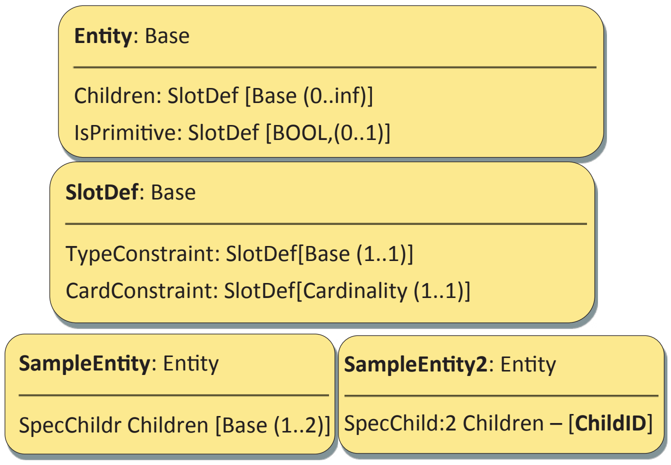
Entity: Base
Children: SlotDef [Base (0..inf)]
IsPrimitive: SlotDef [BOOL,(0..1)]
SlotDef: Base
TypeConstraint: SlotDef[Base (1..1)]
CardConstraint: SlotDef[Cardinality (1..1)]
SampleEntity: Entity
SpecChildr Children [Base (1..2)]
SampleEntity2: Entity
SpecChild:2 Children – [**ChildID**]

Fig. 1 Visual notation: Entity, SlotDef, examples

5.3.4 The Cardinality entity

Since cardinality has been removed from the ASM as a label, it is now used as a constraint. To achieve this, it must be defined as an entity within the bootstrap. *Cardinality* is an instance of *Entity* and it eliminates the *IsPrimitive* slot expressing that it is not a primitive basic type. By default, we have kept the original min-max semantics, however, more generic concept may also be possible to be further concretized by instantiation. Thus, the current structure of the *Cardinality* entity looks like as:

```
{“Cardinality”, “Entity”, undef,
[
{“CardMin”, “Children”, undef,
 [[“NUMBER”], [1, 1]]
]},
{“CardMax”, “Children”, undef,
 [[“NUMBER”], [1, 1]]
]}
]}
```

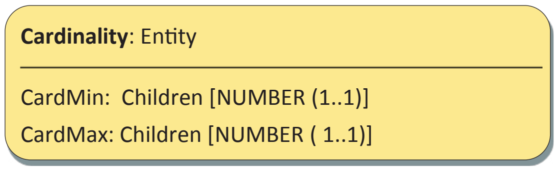
Cardinality: Entity
CardMin: Children [NUMBER (1..1)]
CardMax: Children [NUMBER (1..1)]

Fig. 2 The Cardinality entity

5.3.5 Type conformity

An important difference between DMLA 1.0 and 2.0 is the way we handle the type constraint. In the bootstrap, the value provided in the *TypeConstraint* attribute of the *SlotDef* is a restriction on the value of *SlotDef* instances. If the value is an ID, the referred element has to be a direct, or indirect instance of the provided type. If the value is a built-in type, the respective basic type has to pass the check. Since the *Base* entity is the topmost meta of every element in the model, setting the *Base* as the type of the attribute equals to the use of the *Basic* type in the original version. Note that since the value of the *TypeConstraint* element is restricted to *Base* instances, and the *SlotDef* is a *Base* instance, the *SlotDef* itself can also be used as a type restriction.

5.3.6 Basic types

The bootstrap of DMLA 2.0 has also built-in basic types as DMLA 1.0. The types and their definitions are the following:

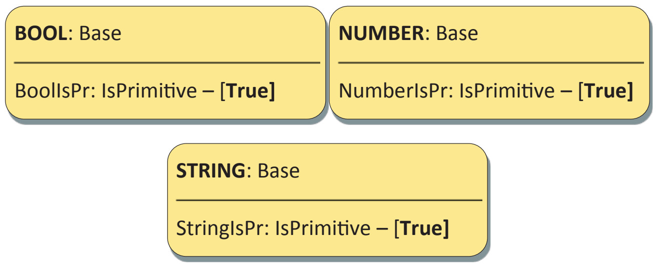
BOOL: Base
BoolIsPr: IsPrimitive – [**True**]
NUMBER: Base
NumberIsPr: IsPrimitive – [**True**]
STRING: Base
StringIsPr: IsPrimitive – [**True**]

Fig. 3 Basic types

5.4 Example of Node-Edge rebalancing

As a simple, though practically quite useful example of a customized bootstrap, let us now illustrate how easy it is to solve the so called node-edge dichotomy [1] by the new DMLA 2.0 formalism. For the sake of compactness, we use the graphical formalism for the definition of both Node and Edge.

Node is a direct instance of *Entity*. The only difference between *Node* and *Entity* is that *Node* eliminates the *IsPrimitive* flag.

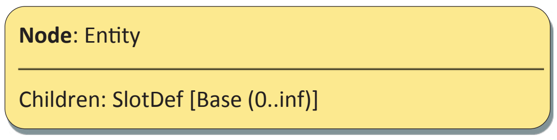

Fig. 4 The Node entity

Edge is also a direct instance of *Entity*. It keeps the *Children* collection to be able to define attributes for edges, however – at the same time – it also instantiates the *Children* attribute to create two slots representing the end points of the edge. Note that *Edge* is a good example to dynamic instantiation (part of the slot definitions are kept untouched for later usage, while others are instantiated).

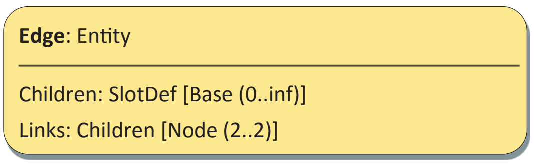
Edge: Entity
Children: SlotDef [Base (0..inf)]
Links: Children [Node (2..2)]

Fig. 5 The Edge entity

5.5 Validation formulae

The concept of validation remained basically the same: we use the validation formulae to differentiate between valid and invalid states of the ASM. However, correspondingly to the modified bootstrap, we carried out some changes. We introduced a simplification step and discarded the concept of *copies* from our model. This does not cause any problem since the concretization of a cardinality constraint is defined as an instantiation, instead of an exceptional branch of the formulae (as it is in DMLA 1.0).

Further modification in formulae can be summarized as follows:

The first change is that the formulae are now dependent on the meta of the validated entity. This means that now we have specialized formulae for certain sub-trees of our meta-hierarchy. This is mostly needed because of the different characteristics of the *Base* and the *SlotDef* entity. The *Base* entity has the general concept of instantiation, meaning that the instantiation may concretize the meta entity. On the other hand, the *SlotDef* entity has a different behaviour, because it has two ways of instantiation: providing a value or concretizing the constraints. These formulae were also provided for the type constraint and cardinality constraint of *SlotDef*, and the minimum and maximum slots for *Cardinality*. These specialized cases of the formulae are capable of referring to the more generic cases, which makes these sub-formulae extensions and not disjoint expressions.

The second change results in the fact that the validation of every element is split into two main formulae as mentioned earlier. The first formula (alpha type formula) has to validate a meta entity against one instance entity, checking if the instance violates any constraints. The second formula (beta type formula) has to validate a meta entity *in its context*, which means that it has to validate a list of clone and instance entities. This formula is mostly needed to check the cardinality constraint of the meta element. The *in context* checks (beta formulae) are evaluated while validating the first type (alpha formula) of the validation formula, checking the validity of every child of the meta entity against the relevant children of the instance entity.

5.5.1 Helper formulae

The formula *DeriveOrEq* checks if the entity I equals M, or if I is an nth level instance of M.

$$\mathbf{DeriveOrEq}(\mathrm{I},\mathrm{M}): \mathit{DeriveFrom}(I,M) \vee I = M \tag{1}$$

The formula *InstanceOf* checks if entity I is a direct instance of entity M.

$$\mathbf{InstanceOf}(\mathrm{I},\mathrm{M}): \mathit{Meta}(I) = M \tag{2}$$

The formula *CloneOf* checks whether the two elements are equal (clones). The formula is used only to increase the legibility of the formulae.

$$\mathbf{CloneOf}(\mathrm{ID}_1,\mathrm{ID}_2): ID_1 = ID_2 \tag{3}$$

The formula *ChildrenByMeta* obtains a set containing all attributes of C that are equal to M or an nth level instance of M.

$$\mathbf{ChildrenByMeta}(\mathrm{C},\mathrm{M}): \{a \mid \exists i : a = Attrib(C,i) \wedge DeriveOrEq(a,M)\} \tag{4}$$

The formula *ChildrenByMeta* selects a single attribute of C that is equal to M or an nth level instance of M.

$$\mathbf{ChildrenByMeta}(\mathrm{C},\mathrm{M}): a \mid \exists i : a = Attrib(C,i) \wedge DeriveOrEq(a,M)) \tag{5}$$

The formula *ValueCount* counts the number of values of I.

$$\mathbf{ValueCount}(\mathrm{I}): \left|\{v \mid \exists i : v = Value(I,i) \wedge v \neq undef\}\right| \tag{6}$$

The formula *HasValue* returns true if I has at least one filled in value.

$$\mathbf{HasValue}(\mathrm{I}): \exists a,i \mid a = Value(I,i) \wedge a \neq undef \tag{7}$$

The formula *IsSlotPart* returns true, if I equals to or derives from *TypeConstraint* or *CardConstraint*.

$$\mathbf{IsSlotPart}(\mathrm{I}): DeriveOrEq(I, ID_{CardConstraint}) \vee DeriveOrEq(I, ID_{TypeConstraint}) \tag{8}$$

The formula *IsSlotInstance* returns true if I equals to or derives from *TypeConstraint*, *CardConstraint*, *CardMin* or *CardMax*.

$$\mathbf{IsSlotInstance}(\mathrm{I}): IsSlotPart(I) \vee DeriveOrEq(I, ID_{CardMin}) \vee DeriveOrEq(I, ID_{CardMax}) \tag{9}$$

5.5.2 Validation formulae

The formula $\varphi_{\mathrm{IsValid}}$ checks if I is a valid instance of M. I has to have M as its Meta element, and also needs to validate against the proper alpha type formula.

$$\varphi_{\mathrm{IsValid}}(\mathrm{I},\mathrm{M}): InstanceOf(I,M) \wedge \varphi_{ValidInstance}(I,M) \tag{10}$$

The formula *ValidInstance* selects the proper alpha type formula and validates instance I and the meta element M. The selection is based on the type of I, which is M. As the formula shows, *SlotDef* parts are to be processed separately ($\alpha_{SubstitutableConstraint}$), the validation must be customized in these cases and slot definitions themselves require special handling as well ($\alpha_{SlotDef}$). Otherwise, we can use the default validation applicable for all Base instances.

$$\begin{aligned}
&\varphi_{\mathrm{ValidInstance}}(\mathrm{I},\mathrm{M}): \\
&\big(DeriveOrEq(M, ID_{CardMin}) \wedge \alpha_{CardMin}(I,M)\big) \vee \\
&\big(\neg DeriveOrEq(M, ID_{CardMin}) \wedge \\
&\quad \big(\big(DeriveOrEq(M, ID_{CardMax}) \wedge \alpha_{CardMax}(I,M)\big) \vee \\
&\quad \big(\neg DeriveOrEq(M, ID_{CardMax}) \wedge \\
&\qquad \big(\big(IsSlotPart(M) \wedge \alpha_{SubstitutableConstraint}(I,M)\big) \vee \\
&\qquad \big(\neg IsSlotPart(M) \wedge \\
&\qquad\quad \big(\big(DeriveOrEq(M, ID_{SlotDef}) \wedge \alpha_{SlotDef}(I,M)\big) \vee \\
&\qquad\quad \big(\neg DeriveOrEq(M, ID_{SlotDef}) \wedge \\
&\qquad\qquad \big(DeriveOrEq(M, ID_{Base}) \wedge \alpha_{Base}(I,M)\big) \\
&\qquad\quad)) \\
&\qquad)) \\
&\quad)) \\
&)
\end{aligned} \tag{11}$$

The formula *ValidContext* selects the proper beta type formula and validates the attribute *a* and the *Children* elements. The selection is based on the meta of the entity *a*. Similarly to the alpha formulae, parts of *SlotDef* and *SlotDef* itself require custom validation.

$$\begin{aligned}
&\varphi_{\mathrm{ValidContext}}(\mathrm{a},\mathrm{Children}): \\
&\big(IsSlotInstance(Meta(a)) \wedge \\
&\beta_{SubstitutableConstraint}(a, Children)\big) \vee \\
&\big(\neg IsSlotInstance(Meta(a)) \wedge \\
&\quad \big(\big(DeriveOrEq(Meta(a), ID_{SlotDef}) \wedge \\
&\quad \beta_{SlotDef}(a, Children)\big) \vee \\
&\quad \big(\neg DeriveOrEq(Meta(a), ID_{SlotDef}) \wedge \\
&\qquad \big(DeriveOrEq(Meta(I), ID_{Base}) \wedge \beta_{Base}(I, Children)\big) \\
&\quad)) \\
&)
\end{aligned} \tag{12}$$

The formula *TypeConformity* checks if value V conforms to the type T. If value V is an ID, it has to be equal to T or an nth level instance of T. If value V is a primitive type, the corresponding built-in ID has to fulfill the condition.

$$\begin{aligned}
&\varphi_{\mathrm{TypeConformity}}(\mathrm{T},\mathrm{V}): \\
&T \neq undef \wedge V \neq undef \wedge \\
&\big(\big(V \in U_{Bool} \wedge DeriveOrEq(ID_{BOOL}, T)\big) \vee \\
&\big(V \in U_{Number} \wedge DeriveOrEq(ID_{NUMBER}, T)\big) \vee \\
&\big(V \in U_{String} \wedge DeriveOrEq(ID_{STRING}, T)\big) \vee \\
&\big(V \in U_{ID} \wedge DeriveOrEq(V, T)\big)\big)
\end{aligned} \tag{13}$$

The helper formula $ValidContextMeta$ calls the beta formula of the ith attribute of M, and checks all relevant attributes of I (based on their meta) for the call.

$$\begin{aligned}\varphi_{\text{ValidContextMeta}}\left(\text{I},\text{M},\text{i}\right):\ & \exists a \mid a = Attrib\left(M,i\right)\wedge \\ & \varphi_{ValidContext}\left(a, ChildrenByMeta\left(I,a\right)\right)\end{aligned} \tag{14}$$

The formula $CardMin$ returns the minimum cardinality contained by the *SlotDef I*.

$$\begin{aligned}&\varphi_{\text{CardMin}}\left(\text{I}\right): val \mid \exists c,m: \\ &\quad c = ChildByMeta\left(I, ID_{CardConstraint}\right)\wedge \\ &\quad m = ChildByMeta\left(Value\left(c,0\right), ID_{CardMin}\right)\wedge \\ &\quad val = Value\left(m,0\right)\end{aligned} \tag{15}$$

The formula $CardMax$ returns the maximum cardinality contained by the *SlotDef I*.

$$\begin{aligned}&\varphi_{\text{CardMax}}\left(\text{I}\right): val \mid \exists c,m: \\ &\quad c = ChildByMeta\left(I, ID_{CardConstraint}\right)\wedge \\ &\quad m = ChildByMeta\left(Value\left(c,0\right), ID_{CardMax}\right)\wedge \\ &\quad val = Value\left(m,0\right)\end{aligned} \tag{16}$$

The alpha formula of the *Base* element consists of several parts. The first part checks the validity of all of the attributes of *I* with the beta formulae. The second part checks if all of the attributes of *I* are related to the attributes of *M*. Note that the equality in α_{Base2} expresses that the two sets ($I_{Attributes}$ and the calculated union) consists of the same elements. The third part checks if the *I* element has a value filled in, which is prohibited for the *Base* element.

$$\alpha_{\text{Base}}\left(\text{I},\text{M}\right): \alpha_{Base1}\wedge\alpha_{Base2}\wedge\alpha_{Base3} \tag{17}$$

$$\alpha_{\text{Base1}}\left(\text{I},\text{M}\right): \nexists i \mid \neg\varphi_{ValidContextMeta}\left(I,M,i\right) \tag{18}$$

$$\begin{pmatrix}\alpha_{\text{Base2}}\left(\text{I},\text{M}\right): j \mid \forall a : \exists i : a = Attrib\left(M,i\right) \\ I_{Attributes} = \bigcup \wedge j = ChildrenByMeta\left(I,a\right)\end{pmatrix} \tag{19}$$

$$\alpha_{\text{Base3}}\left(\text{I},\text{M}\right): \neg HasValue\left(I\right) \tag{20}$$

The beta formula for the *Base* element. The in-context validity is always true in the case of the *Base* element, since the *Base* element does not have any constraints on the in-context instantiation and cloning of an element.

$$\beta_{\text{Base}}\left(\text{a},\text{Children}\right): true \tag{21}$$

The alpha formula of the *SlotDef* element. The formula prohibits the instantiation of a *SlotDef* that has any values filled in. Otherwise, if instance *I* has no values, the formula delegates to the alpha formula of the *Base* element. If instance *I* has a value, it is validated against the type and the maximum cardinality constraint of M. Note that minimum cardinality cannot be checked here, since it would require context information.

$$\begin{aligned}&\alpha_{\text{SlotDef}}\left(\text{I},\text{M}\right): \neg HasValue\left(M\right)\wedge \\ &\quad\left(\left(\neg HasValue\left(I\right)\wedge\alpha_{Base}\left(I,M\right)\right)\vee\right. \\ &\quad\left.\left(HasValue\left(I\right)\wedge\alpha_{SlotDef1}\left(I,M\right)\wedge\alpha_{SlotDef2}\left(I,M\right)\right)\right)\end{aligned} \tag{22}$$

$$\begin{aligned}&\alpha_{SlotDef1}\left(\text{I},\text{M}\right): \nexists v \mid \exists i : v = Value\left(I,i\right)\wedge\neg\varphi_{TypeConformity} \\ &\quad\quad\left(Value\left(ChildByMeta\left(M, ID_{TypeConstraint}\right),0\right), v\right)\end{aligned} \tag{23}$$

$$\alpha_{SlotDef2}\left(\text{I},\text{M}\right): ValueCount\left(I\right)\leq\varphi_{CardMax}\left(M\right) \tag{24}$$

The beta formula of the *SlotDef* element. If the attribute *a* is the *SlotDef* entity itself, the formula delegates to the *Base* beta formula, since the *SlotDef* does not contain a concrete cardinality constraint yet. Otherwise, if attribute *a* has a value, the context can only contain the clone of *a*, since a filled in *SlotDef* cannot be instantiated or discarded. If *a* has no value, the validity depends on the cardinality constraint in the element. The elements of *Children* are counted based on the number of filled in values, or the provided cardinality constraints. This results in two numbers, the possible minimum and maximum number of values. These values are to be checked against the cardinality in the *a* element.

$$\begin{aligned}&\beta_{SlotDef}\left(a, Children\right): \\ &\quad CloneOf\left(a, ID_{SlotDef}\right)\wedge\beta_{Base}\left(a, Children\right)\vee \\ &\quad\neg CloneOf\left(a, ID_{SlotDef}\right)\wedge\left(\beta_{SlotDef1}\vee\beta_{SlotDef2}\right)\end{aligned} \tag{25}$$

$$\begin{aligned}&\beta_{SlotDef1}\left(a, Children\right): \\ &\quad HasValue\left(a\right)\wedge Count\left(Children\right)=1\wedge \\ &\quad Children\left[0\right]=a\end{aligned} \tag{26}$$

$$\begin{aligned}&\beta_{SlotDef2}\left(a, Children\right): \\ &\quad\exists low, up, min, \max \mid \neg HasValue\left(a\right)\wedge \\ &\quad low = \varphi_{CardMin}\left(a\right)\wedge up = \varphi_{CardMax}\left(a\right)\wedge \\ &\quad min = \sum m \mid \forall c : \exists i : c = Children\left[i\right]\wedge \\ &\quad\quad\quad\left(\left(\neg HasValue\left(c\right)\wedge m = \varphi_{CardMin}\left(c\right)\right)\vee\right. \\ &\quad\quad\quad\left.\left(HasValue\left(c\right)\wedge m = ValueCount\left(c\right)\right)\right)\wedge \\ &\quad max = \sum m \mid \forall c : \exists i : c = Children\left[i\right]\wedge \\ &\quad\wedge low\leq min\wedge up\geq max)\end{aligned} \tag{27}$$

The alpha formula of the *TypeConstraint* and *CardConstraint* elements. The only changes compared to the *SlotDef* alpha formula is that these elements allow the instantiation of an element with a filled in value thus concretizing the value even more. For example, the cardinality [1..2] is a valid instance of the cardinality [1..4]. The value of *I* has to be an nth level instance of the value of *M*. The formula also restricts the value for the ID universe.

$$\begin{aligned}
&\alpha_{SubstitutableConstraint}(I,M):\\
&(\neg(ValueCount(M)=1 \wedge ValueCount(I)=1) \wedge\\
&\quad \alpha_{SlotDef}(I,M)) \vee\\
&(ValueCount(M)=1 \wedge ValueCount(I)=1 \wedge\\
&\quad ((\exists mv, iv : mv = Value(M,0) \wedge mv \in U\_ID \wedge\\
&\quad iv = Value(I,0) \wedge iv \in U_{ID} \wedge DeriveFrom(iv,mv)))
\end{aligned} \tag{28}$$

The beta formula of the *TypeConstraint*, *CardConstraint*, *MinCard* and *MaxCard* elements. This formula accepts the instantiation of a filled in element. If the element is instantiated, the context can only contain the instance (and no clones).

$$\begin{aligned}
&\beta_{SubstitutableConstraint}(a, Children):\\
&(ValueCount(a)=1 \wedge (\nexists c, i : c = Children[i] \wedge c = a) \wedge\\
&\quad Count(Children)=1) \vee\\
&(\neg(ValueCount(a)=1 \wedge (\nexists c, i : c = Children[i] \wedge c = 0))\\
&\quad \wedge \beta_{SlotDef}(a, Children))
\end{aligned} \tag{29}$$

The alpha formula of the *CardMin* element. The formula allows the instantiation of an already filled in element with a greater minimum value.

$$\begin{aligned}
&\alpha_{CardMin}(I,M):\\
&(\neg(ValueCount(M)=1 \wedge ValueCount(I)=1) \wedge\\
&\quad \alpha_{SlotDef}(I,M)) \vee\\
&(ValueCount(M)=1 \wedge ValueCount(I)=1 \wedge\\
&\quad (\exists mv, iv : mv = Value(M,0) \wedge iv = Value(I,0)\\
&\qquad \wedge iv < mv))
\end{aligned} \tag{30}$$

The alpha formula of the *CardMax* element. It allows the instantiation of an element with a lower maximum value.

$$\begin{aligned}
&\alpha_{CardMax}(I,M):\\
&(\neg(ValueCount(M)=1 \wedge ValueCount(I)=1) \wedge\\
&\quad \alpha_{SlotDef}(I,M)) \vee\\
&(ValueCount(M)=1 \wedge ValueCount(I)=1 \wedge\\
&\quad (\exists mv, iv : mv = Value(M,0) \wedge iv = Value(I,0)\\
&\quad \wedge low \le min \wedge up \ge max)
\end{aligned} \tag{31}$$

6 Evaluation

In this section, we give an evaluation of our approach by summarizing and comparing its features to other solutions in the field. As first, we present a short example to show, how DMLA 2.0 works in practice, then we give an overview of DMLA 1.0 and 2.0 differences and finally we place DMLA 2.0 in the field multilevel modeling.

6.1 Simple Router Example

As a simplified concrete example, let us describe a network. In this network, a generic Router concept is concretized by various types of routers that have many instances deployed in the network. That situation is a frequently reoccurring one in network management, which may challenge state-of-the-art meta-model based tools resulting in an ad-hoc solution. Nevertheless, the situation is easy to be represented in [1]. In particular, DMLA 2.0 even facilitates formally correct definitions in its graphical notations as follows.

In the example, four entities are defined: the first one introduces IPType. It specifies the address as a String and it also has an IsIPv4 flag showing whether we use IPv4, or IPv6.

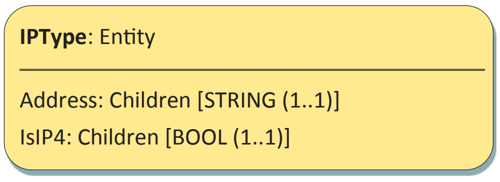
IPType: Entity
Address: Children [STRING (1..1)]
IsIP4: Children [BOOL (1..1)]

Fig. 6 The IPType entity

Other entities define the instantiation hierarchy starting with *RouterType*, which instantiates Node by restricting the types of Attributes. Then, a particular router type, *SimpleRouter*, further restricts the cardinality of *IPAddresses* to two.

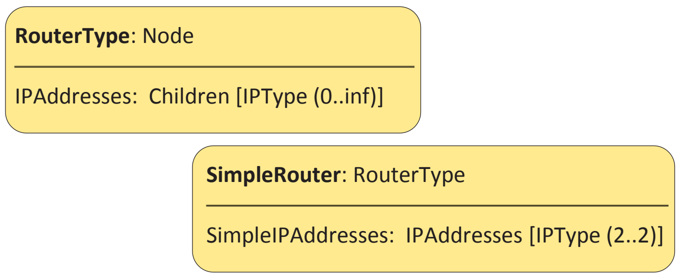
RouterType: Node
IPAddresses: Children [IPType (0..inf)]
SimpleRouter: RouterType
SimpleIPAddresses: IPAddresses [IPType (2..2)]

Fig. 7 RouterType and SimpleRouter

Finally, the *MyRouter* instance sets the concrete IP addresses according to the definition of *IPType*. Note that In and Out attributes are set by instances of the IPType entity rather than by a primitive expression (e.g. a string literal).

In a similar vein, other entity hierarchies can be created: one that represents the companies, which manage the router(s). Companies may also have (any number of) logs. Each of the logs consists of exactly one entry modeled as a string.

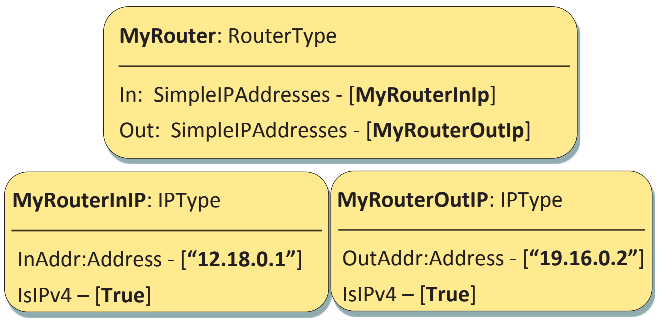
MyRouter: RouterType
In: SimpleIPAddresses - [MyRouterInIp]
Out: SimpleIPAddresses - [MyRouterOutIp]
MyRouterInIP: IPType
InAddr:Address - ["12.18.0.1"]
IsIPv4 – [True]
MyRouterOutIP: IPType
OutAddr:Address - ["19.16.0.2"]
IsIPv4 – [True]

Fig. 8 Concrete router and its attributes

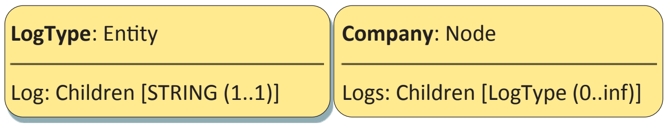
LogType: Entity
Log: Children [STRING (1..1)]
Company: Node
Logs: Children [LogType (0..inf)]

Fig. 9 The Company and LogType entities

It is also possible to create concrete companies. Here, we used the ability to instantiate attributes partially. We add a concrete log entry to the company, but we do not remove the attribute slot. Therefore, the instances of MyCompany can also add further log entries to themselves.

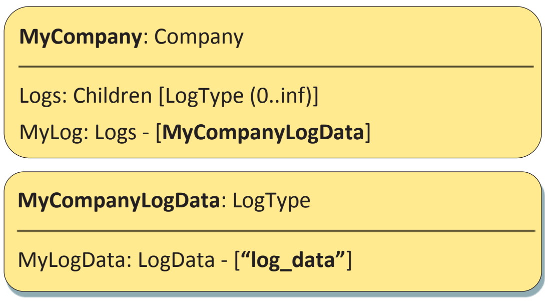
MyCompany: Company
Logs: Children [LogType (0..inf)]
MyLog: Logs - [MyCompanyLogData]
MyCompanyLogData: LogType
MyLogData: LogData - ["log_data"]

Fig. 10 A concrete company

Finally, we can create a third hierarchy expressing the relations between router management and router by instantiating Edge. As it is shown, the type of the source and target links are set at the metalevel, while their concrete value is set at the instance level.

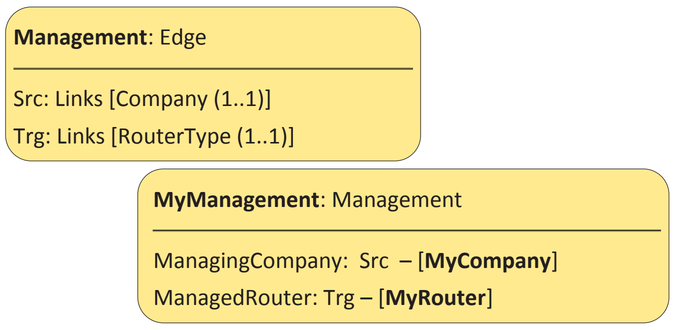
Management: Edge
Src: Links [Company (1..1)]
Trg: Links [RouterType (1..1)]
MyManagement: Management
ManagingCompany: Src – [MyCompany]
ManagedRouter: Trg – [MyRouter]

Fig. 11 Management - the link between

As the example clearly demonstrates, the DLMA notation is compact, formal and easily customizable by the bootstrap.

6.2 DMLA 1.0. vs DMLA 2.0

Although we have already mentioned the differences between the two major versions of DMLA when presenting DMLA 2.0, for the sake of clarity, we give a short summarization here.

The most important difference is that DMLA 1.0 uses a 6-tuple to represent the data and the modeling relations, while DMLA 2.0 is based on a 4-tuple. The DMLA 2.0 solution is more compact (entity names are omitted) and more flexible (cardinality handling is not wired in the core of the algebra) at the same time.

Modeling relations are restricted by their metamodel but a simple type-instance relation is not always enough, additional constraints are needed. Cardinality is a good example to this, but format expressions (e.g. the format of email addresses), range limits (e.g. pick any number under 100) and other kind of constraints may also be useful in many cases. DMLA 1.0 emphasized the role of cardinality and encoded the attribute type restrictions in AttribType [2], which is a strict, but rigid solution. In DMLA 2.0, constraint handling is raised to a whole new level, by introduction SlotDef and allowing to extend the type of constraints later. It is out of the scope of this paper, but we should mention that we successfully managed to extend constraints by range and format constraints with only a slight modification in the bootstrap presented in this paper. Such extensions would be much harder to add to DMLA 1.0.

Another good example to the flexibility of constraint handling in DMLA 2.0 is that the mandatory concretization condition (hard wired in DMLA 1.0) is omitted in the presented bootstrap; however, it could be also easily added by modifying the formulae .

The principal entities and built-in types are also reformed in DMLA 2.0. This is mainly a pure consequence of the aforementioned changes. However, we should also mention that by introducing Base as the basis of all modeling entities, the meta relations can be expressed more elegantly by using the instance-of relation instead of referencing to model elements by their ID.

Finally, the structure of validation formulae (the separation of α and β formulae) of DMLA 2.0 allows us to control instantiation in a more sophisticated (precise, compact and flexible) way.

To sum up, DMLA 1.0 can be considered as an experimental prototype of our approach for dynamic, multilevel modeling with great expression power. It contained a very promising set of ideas, but it still featured some unpolished edges. In contrast, DMLA 2.0 is a more advanced, flexible approach which is also much easier to be used and implemented. Hence, in DMLA 2.0, the concepts have been revisited and the original format has been changed for the better representation wherever it served to eliminate accidental complexity.

7 Conclusion

Despite the growing need for model-based software development and the essential role of instantiation is meta-modelling approaches, current state-of-the-art techniques face difficulties at expressing domain specific design rules within models, probably due to their lack of adequate dynamic multi-level instantiation. Dynamic Multi-Layer Algebra (DMLA) is a novel ASM-based algebraic formalism that enables formally correct and expressive multi-level metamodeling for combined design- and run-time applications. Although the original DMLA formalism worked well and also demonstrated the benefits resulting from a balanced structural separation of ASM, bootstrap and formulae parts, the representation itself turned out unnecessarily complex. In this paper, we have highlighted the shortcomings and came up with a clarified, more compact and more streamlined formalism, which exhibits a 4-tuple representation instead of the original 6-tuple one. The improvement is semantically relevant because the type and cardinality constraints have become more uniform and formally specified by the respective formula extensions, which opens the door to further relaxation of the current structure of e.g. cardinality constraints. Furthermore, we introduced the concept of SlotDef, which is an easy-to-use extension point to add additional instantiation validators to the system. SlotDef makes it also possible to introduce OCL like multi-level constraints into the meta-model. However, complex constraint introduction and the relaxation of the type conformity formula are still ongoing research. We are currently also working on a practical implementation of the formalism in C# and Xtext so that real-life industrial automation and telecom management models could be tested for their flexible representation and semi-automatic run-time processing. The results of the practical experimentation may further shape and improve the formalism.

References

[1] "Eclipse Modeling Framework (EMF)." [Online]. Available from: https://eclipse.org/modeling/emf/.

[2] OMG, "MetaObject Facility." OMG, [Online]. Available from: http://www.omg.org/mof/.

[3] Theisz, Z., Mezei, G. "An Algebraic Instantiation Technique Illustrated by Multilevel Design Patterns." In MULTI@MoDELS, Ottawa, Canada, 2015.

[4] Theisz, Z., Mezei, G. "Multi-level Dynamic Instantiation for Resolving Node-edge Dichotomy." In: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 2016.

[5] Theisz, Z., Mezei, G. "Towards a novel meta-modeling approach for dynamic multi-level instantiation." In: Automation and Applied Computer Science Workshop, Budapest, Hungary, 2015.

[6] Atkinson, C., Kühne, T. "The Essence of Multilevel Metamodeling." *The Unified Modeling Language. Modeling Languages, Concepts, and Tools.* 2185, pp. 19-33, 2001.

[7] Atkinson, C., Gerbig, R. "*Melanie: Multi-level modeling and ontology engineering environment*." ACM, New York, USA. Article No. 7. 2012 https://doi.org/10.1145/2448076.2448083

[8] Atkinson, C., Gutheil, M., Kennel, B. "A Flexible Infrastructure for Multilevel Language Engineering." *IEEE Transactions on Software Engineering*. 35(6), pp. 742–755. 2009. https://doi.org/10.1109/TSE.2009.31

[9] Neumayr, B., Jeusfeld, M. A., Schrefl, M., Schütz, C. "Dual Deep Instantiation and Its ConceptBase Implementation." In: Proceedings of the 26th International Conference on Advanced Information Systems Engineering, Thessaloniki, Greece, 2014. pp. 503-517. https://doi.org/10.1007/978-3-319-07881-6_34

[10] Atkinson, C., Gerbig, R., Kühne, T. "Comparing multi-level modeling approaches." *Proceedings of the 1st Workshop on Multi-Level Modelling.* 1286, pp. 53-61. 2014.

[11] Atkinson, C., Kühne, T. "Model-Driven Development: A Metamodeling Foundation." *IEEE Software*. 20(5), pp. 36-41. 2003. https://doi.org/10.1109/MS.2003.1231149

[12] de Lara, J., Guerra, E. "Deep Meta-modelling with MetaDepth." *Objects, Models, Components, Patterns*. 6141, pp. 1-20. 2010. https://doi.org/10.1007/978-3-642-13953-6_1

[13] Rossini, A., de Lara, J., Guerra, E., Nikolov, N. "A Comparison of Two-Level and Multi-level Modelling for Cloud-Based Applications." *Modelling Foundations and Applications*. 9153, pp. 18-32. 2015. https://doi.org/10.1007/978-3-319-21151-0_2

[14] de Lara, J., Guerra, E., Cuadrado, J. S. "When and How to Use Multilevel Modelling." *Journal ACM Transactions on Software Engineering and Methodology*. 24(3), Artice No. 12. 2014. https://doi.org/10.1145/2685615

[15] OMG, "UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems." [Online]. Available from: http://www.omg.org/spec/MARTE/1.1/. [Accessed: 1st October 2016]

[16] Clark, T., Gonzalez-Perez, C., Henderson-Sellers, B. "A Foundation for Multi-Level Modelling." In: Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems. 1286, pp. 43-52. 2014.

[17] Börger, E., Stark, R. "*Abstract State Machines: A Method for High-Level System Design and Analysis*." Springer-Verlag Berlin Heidelberg. 2003. https://doi.org/10.1007/978-3-642-18216-7