Multi-level Dynamic Instantiation for Resolving Node-edge Dichotomy

Zoltan Theisz[1] and Gergely Mezei[2]
[1]*Huawei Design Centre, Athlone, Ireland*
[2]*Dept. of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, Hungary*

Keywords: Meta-modeling, Instantiation, Multi-level Modeling, Node-edge Dichotomy, ASM.

Abstract: The core idea of metamodel-based model construction is well established. However, there are different meta-modeling approaches relying on various modeling structures and instantiation procedures. Although, in general, these approaches offer similar features, they are sometimes incompatible with each other. Therefore, a precise abstract definition of instantiation is needed. The paper describes an abstract modeling framework, which is easily customizable in order to adapt it to different multi-level modeling techniques. The framework consists of an abstract modeling structure, basic built-in constructs, and a dynamic instantiation procedure. The paper demonstrate the flexibility of the approach by a specific bootstrap that is explicitly designed for the rebalancing of the node-edge antagonism, which is mostly the origin of many reification patterns applied in current meta-model designs. Although the proposed solution to the node-edge dichotomy is only an example of our multi-level meta-modeling approach, it is per se a valuable achievement showing that it can be done in a more elegant manner than it is usually expressed in other state-of-the art modeling frameworks.

1 INTRODUCTION

State-of-the-art telecom service management operates on Cloud-based software components such as Virtual Network Functions (VNF) (NFV, 2013). The central piece of the service orchestrator is a flexible modeling core that enables gradual instantiation of meta-objects through various stages, corresponding to different levels of detail set by the individual stakeholders representing the collaborating partners of a telecom service ecosystem. Contemporary such solutions usually consist of ad-hoc pattern-based multi-level meta-modeling implementations, which are reliant on either relational database techniques or XML technologies to enable jumps between meta-levels via proprietary domain specific promotion and demotion operations. Therefore, the integration of these model-based systems is ad-hoc as well.

Although multi-level meta-modeling is a well-researched topic, its industry quality tool support has not been validated or established yet. Moreover, most of the current approaches address predominantly design-time aspects of multi-level meta-modeling and they also do so only from the particular point of view of clabjects and the potency notion (Atkinson and Kühne, 2003). However, in Cloud-based software component management, both components and connections must be treated equally: all concepts of multi-level meta-modeling have to be applied uniformly in order to facilitate automatic and transparent reification of edges as nodes and vice versa.

In this paper, we aim to illustrate how a novel Abstract State Machine (ASM (Boerger and Stark, 2003)) based algebraic formalism for multi-level meta-modeling can be applied to resolve the node-edge dichotomy which is prevalent with many meta-modeling techniques. The paper is structured as follows: Section 2 introduces the technical background; it covers both the modeling requirements originating from state-of-the-art Cloud software management and the most prominent solutions of multi-level meta-modeling. Then, in Section 3, we introduce our Dynamic Multi-Layer Algebra (DLMA) approach in some detail. Next, in Section 4, DMLA abstract syntax is applied to a simplified modeling example to showcase the power of this formalism to unify the handling of nodes and edges on the same abstraction levels. Finally, Section 5 concludes the paper and also addresses some of our future research goals.

Theisz, Z. and Mezei, G.
Multi-level Dynamic Instantiation for Resolving Node-edge Dichotomy.
DOI: 10.5220/0005690802740281
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 274-281
ISBN: 978-989-758-168-7

2 BACKGROUND

In this section, first, we shortly introduce the most well-known management standard for Cloud-based software components and also elaborate on its modeling requirements. Then, the major instantiation techniques and the most prevalent multi-level meta-modeling approaches are summarized and discussed from the perspective of the paper. Finally, the concept of partial instantiation is described.

2.1 Cloud Base Software Systems

Cloud based software management experienced a rapid change and a relatively quick consolidation during the last couple of years. Currently its de-facto standard model is called the Topology and Orchestration Specification for Cloud Applications (TOSCA (OASIS, 2013)). The main concepts of TOSCA are depicted in Figure 1.

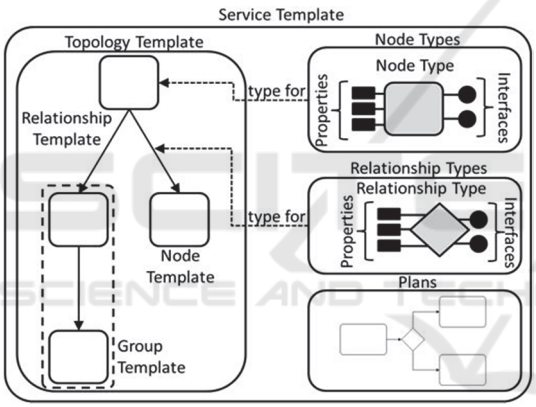

Figure 1: TOSCA structural elements of a service template and their relations.

The TOSCA meta-model defines both the structure of the service components and their management interfaces. A Node Type specifies the properties of a software component with the operations which are available to manage it. Similarly, a Relationship Type defines the semantics and the properties of the relationship between or among Node Types. A Topology Template consists of a set of Node Templates and Relationship Templates that together define the topology model of a cloud based service in the form of a graph. The nodes and edges are instances of Node Types and Relationship Types.

A Node Template grants two features: 1) it specifies the occurrence of a Node Type as a component of a service, 2) it imposes usage constraints, for example, the number of allowed instances of that component which can occur there in run-time. A Relationship Template specifies the occurrence of a particular relationship between nodes in a Topology Template, including the direction of the relationship. Also, it may define further constraints similarly to Node Templates. Hence, Node Template and Relationship Template must be handled equally from the perspective of their modeling role in the Topology Template. Thus, it also means that designers of a specification may have to model similar features repeatedly for both node-related and relationship-related items. Moreover, by analyzing practical TOSCA examples, it may easily turn out that solution architects prefer to reify relations as nodes and connect them together by further "virtual" relations due to the lack of proper multi-level meta-modeling support. May this reification be to be repeated on various abstraction levels, the topology templates can become seriously entangled and it may not be clear later if a particular node stands for itself or its sole purpose of reification has been some other relations supposed to be present by a meta-level below. In other words, the modeling challenge is to come up with a solid foundation that is able to explicitly tackle these ad-hoc meta-level promotion/demotion design decisions and thus to consolidate them into simple design patterns. Also, Node and Relationship Types must be handled equally, that is, the node-edge dichotomy ought to be tackled upfront by any serious meta-modeling solution. On the practical side, that robust meta-modeling framework is to be implemented in such a way that it can operate on models both in design-time and run-time via dynamic and partial instantiation of modelled elements of Service Templates residing at any particular modeling level.

2.2 Meta-modeling and Instantiation

Meta-modeling systems are based on instantiation, that is, we create our models by instantiating meta-models. Therefore, instantiation is the essence of any meta-modeling discipline. Although the precise definition of instantiation ought to be the keystone of any viable modeling technique, it is usually taken for granted that instantiation is a simple relation similar to the well-known one between classes and objects. The similarity looks obvious, but the two relations are only similar on the surface.

Our approach is based on the common conceptual foundation of multi-level instantiation techniques, so firstly we summarize two of such state-of-the-art multi-level approaches in order to position our work correctly within this domain.

2.2.1 Linguistic vs Ontological Instantiation

Linguistic and ontological instantiation were introduced in (Atkinson and Kühne, 2003). Linguistic instantiation is the relationship between model elements on different levels of a multi-level meta-modeling hierarchy. Linguistic instantiation usually describes meta-modeling relations between types and objects. In contrast, ontological instantiation defines the semantics of modeling constructs. Therefore, for each modeling level, an additional ontological instantiation relationship is defined that associates model elements of that particular meta-level to each other, according to a certain concern of the problem domain. In the case of MOF, the model elements on M1 are linguistic instances of their meta-elements on M2, however, a particular M1 (or M2) model element can also be an ontological instance of another element of the same M1 (or M2) level Although the MOF architecture does not facilitate both of these two meta-modeling dimensions equally, it does provide a mechanism for expressing ontological relationships by its reliance on stereotypes and profiles. Namely, stereotypes and profiles defined on M2 level express ontological instantiation relationships that can be flexibly used as extra annotation(s) on M1 model elements. Although this approach may be utilized to allow an ontological unification of nodes and edges, no practical solutions of this kind has been published yet.

2.2.2 Multi-level Instantiation

Provided that only two modeling levels are available, instantiation has to be interpreted between these two levels only. However, in a multi-level meta-modeling architecture, we may be able to use instantiation across multiple levels. These techniques can distinguish between two kinds of instantiation: shallow instantiation means that the information is defined on the n^{th} modeling level and it is used on the immediate instantiation $(n+1)^{th}$ level, while deep instantiation allows defining the information on the n^{th} modeling level and use it on the $(n+x)^{th}$ $(x > 0)$ modeling level (Atkinson and Kühne, 2001). Although multi-level modeling solutions are getting more and more popular, these deep instantiation methods are rarely utilized. For example, MOF defines a four layer architecture, but it supports only shallow instantiation. This results in rigid, inconsistent relations between meta-types, types and objects, which is one of the reasons why node-like and edge-like elements are kept separate due to the lack of enough meta-levels.

Having multi-level meta-modeling in mind, one needs to safeguard that each meta-level should be instantiable by some means of being able to add new attributes and operations to the existing models. There are two options available: one can either bring the source of the information along all model levels (and eventually use it wherever 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 (Atkinson and Kühne, 2002) (Atkinson and Kühne, 2001) 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 through clabjects which is controlled only by the explicit definition of a potency limit allowing a preset number of meta-levels an element can be instantiated at. The only drawback of this solution is that the modelled elements will be forced to live until the attached non-negative numbers have been decremented, by each instantiation, to zero. Hence, the potency solution is both too liberal and too restrictive at the same time: on one hand, it is too liberal because at each meta-level the full potential of meta-model building facilities is available, thus an arbitrary number of new model elements can be injected into the model at will and without any potential restriction. It is definitely good so for design-time modeling, however it may wreak havoc in dynamic run-time instantiation if it is used in an uncontrolled fashion. On the other hand, it is too restrictive in its treatment of the number of meta-levels a model element can be located at because the modeler must know in advance at which levels the information will be needed and set its potency value accordingly. In other words, instead of explicitly encoding the required nature of semantics vis-a-vis the instantiation constraints within the model itself and thus making the model agnostic to the artificial division line set between design-time and run-time modeling, potency notion simplifies it down to an integer number and thus positions itself as a pure design-time multi-level modeling approach. Moreover, potency notion favors nodes against edges due to its bias to clabject. Nevertheless, recent techniques such as Dual Deep Instantiation (DDI) (Neumayr et al., 2014) consistently extended the potency notion to edges via its application to the end-points. Unfortunately, some limitations still remain to be overcome: firstly, since DDI defines edges by the endpoints it is non-trivial to extend it by edge-owned

attributes, which may also have potency values attached or may be connected to other attributes. In other words, edges have not made equivalent to nodes yet, but on the surface they may look equivalent in particular cases of application domains. For example, in order to cover TOSCA's concept of Relationship Types many more axioms will be needed than as published in (Neumayr et al., 2014). Secondly, DDI is still a clabject-based set theory solution. Hence, the presence of an edge at a particular meta-level is fully controlled by its related potency values. Therefore, those potency values must be carefully set and designed so that only those edges appear at meta-levels that are really needed to be used there. This challenge may invalidate DDI's usage in practical TOSCA-like modeling situations.

3 DYNAMIC MULTI-LAYER ALGEBRA

In this section, we shortly introduce our Abstract State Machine (ASM) (Boerger and Stark, 2003) based multi-level instantiation technique. An ASM formalism defines an abstract state machine and a certain set of connected functions that specify the transition logic between the states of that machine. In more detail, a state represents an attribute configuration, namely, a concrete set of parameter set-value pairs for the functions of the algebra.

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. 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, a concrete bootstrap selection seeds a particular set of meta-modeling capability of the generic DMLA. In effect, the proper selection of the bootstrap elements imposes explicit restriction on the universality 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. 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

In the following, we use the word *entity* exclusively if we refer to an element which has the label structure defined above. 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:

- X_{Name}: U_{String}
- X_{ID}: U_{ID}
- X_{Meta}: U_{ID}
- $X_{Cardinality}$: $[U_{Number}, U_{Number}]$
- X_{Value}: U_{Basic}
- X_{Attrib}: $U_{ID}[]$

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

```
RouterID = 12, RouterMeta = 123,
RouterCardinality = [0, inf],
RouterValue = undef, RouterAttrib =[]
```

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 (NB: both tuples and lists share the same square bracket notation):

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

3.2 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:

$$Name(ID): \begin{cases} name, if\ \exists X: X_{ID} = ID\ \wedge X_{Name} = name \\ undef, otherwise \end{cases}$$

$$Meta(ID): \begin{cases} Y_{ID}, if\ \exists X, Y: X_{ID} = ID\ \wedge X_{Meta} = Y_{ID} \\ undef, otherwise \end{cases}$$

$$Card(ID): \begin{cases} [low, high], if\ \exists X: X_{ID} = ID\ \wedge \\ \qquad X_{Cardinality} = [low, high] \\ undef, otherwise \end{cases}$$

$$Attrib(ID, Idx): \begin{cases} attrib, if\ \exists X, i: X_{ID} = ID\ \wedge X_{Attrib}[Idx] = \\ \qquad attrib \\ undef, otherwise \end{cases}$$

In these functions we suppose that the Attrib labels return undef when the index is greater or equal to the number of stored entities.

$$Value(ID): \begin{cases} val, if\ \exists X: X_{ID} = ID\ \wedge X_{Value} = val \\ undef, otherwise \end{cases}$$

$$Contains(ID_1, ID_2): \begin{cases} true, if\ \exists c, idx: c = Attrib(ID_1, idx) \wedge \\ \qquad (c_{ID} = ID_2 \vee Contains(c_{ID}, ID_2)) \\ false, otherwise \end{cases}$$

$$DeriveFrom(ID_1, ID_2): \begin{cases} true, \quad \exists x, y: x_{ID} = ID_1\ \wedge\ \exists y: y_{ID} = ID_2 \\ \wedge\ (x_{Meta} = y \vee DeriveFrom(x_{Meta}, y)) \\ false, \qquad otherwise \end{cases}$$

3.3 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. In order to focus on how to tackle the problem of node-edge dichotomy, we will restrict the usage of basic types to an absolute minimum. The bootstrap has two main parts: *basic types* and *principal entities*.

3.3.1 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. Types *Bool*, *Number*, *String* and *ID* are inherited from *Basic*.

3.3.2 Principal Entities

Besides the basic types, we also define two principal entities: *Attribute* and *Base*. They act as root meta elements of attributes, 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. *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. *AttribType* is only applied for attributes. The principal entities of the node-edge dichotomy resolution bootstrap are the following:

```
{"Attribute", ID_Attribute, ID_Attribute,
 [0,inf], undef,
 [
   {"Attributes", ID_Attributes, ID_Attribute,
    [0, inf], undef,[]}
 ]}

{"Base", ID_Base, ID_Base,
 [0, inf], undef,
 [
  {"Attributes", ID_Attributes, ID_Attribute,
   [0, inf], undef,[]}
  {"Links", ID_Link, ID_Attribute,
   [0, inf], undef,[]}
 ]}

{"AttribType", ID_AttribType,ID_Attribute,
 [0,1],undef,
 [
  {"AType", ID_AType, ID_AttribType,
   [0,1],ID_ID,[]}
 ]}

{"Node", ID_Node, ID_Base,
```

```
 [0, 0], undef,
 [
  {"Attributes", ID_Attributes, ID_Attribute,
   [0, inf], undef,[]}
 ]}
{"Edge", ID_Edge, ID_Base,
 [0, inf], undef,
 [
  {"Attributes", ID_Attributes, ID_Attribute,
   [0, inf], undef,[]}
  {"Links", ID_Link, ID_Attribute,
   [2, 2], undef,[]}
 ]}
```

The definition of *Attribute* describes that an attribute can have zero or more attributes as children. The definition of *Base* is very similar to *Attribute*, but it also possesses links. The definition of *AttribType* is an instantiation of *Attribute* and it also uses *AttribType* to restrict its own type. Finally, both *Edge* and *Node* are defined as instances of *Base*, however *Node* must not have links and in the case of *Edge* we set the number of links to 2. Obviously, that number can have any non-negative value, so setting it to 2 has the purpose to represent edges with two end-points.

3.4 Dynamic Instantiation

Based on the structure definition of the algebra and the bootstrap, one can now represent models as states of DMLA. During the instantiation, one may 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 allows manipulation of their values also from outside of the algebra. However, functions do not validate these manipulations. Instead, we distinguish between valid and invalid models, where validity checking is based on formulae. We also assume that whenever external actors change the state of the algebra, the formulae must be evaluated. This dynamic property of the ASM makes it possible that DMLA can be applied both in design- and run-time modeling.

All validation formulae (except φ_{Meta}) take an *Instance* entity and a *MetaType* entity, and then they check if the *Instance* entity is a valid instance of the *MetaType* entity. The formula φ_{Meta} takes only one parameter and validates if the given entity has enough valid instances according to the cardinality label. In the definitions *I* refers to the *Instance* to be validated, while *M* refers to *Meta*, that is, the meta-type of *I*.

3.4.1 Helper Formulae

$\varphi_{IsInstantiated}(C, I)$: $\exists m, i: m = Attrib(Meta(C),i) \wedge \varphi_{IsValid}(I, m)$

$\varphi_{InstCounter}(C,a_1,a_2)$:

$\varphi_{IsInstantiated}(C, a_1) \wedge \varphi_{IsInstantiated}(C, a_2) \wedge Meta(a_1) = Meta(a_2)\} \vee \{\varphi_{IsInstantiated}(C, a_1) \wedge \neg\varphi_{IsInstantiated}(C, a_2) \wedge Meta(Meta(a_1)) = Meta(a_2)\} \vee \{\neg\varphi_{IsInstantiated}(C, a_1) \wedge \varphi_{IsInstantiated}(C, a_2) \wedge Meta(Meta(a_2)) = Meta(a_1)\} \vee \{\neg\varphi_{IsInstantiated}(C, a_1) \wedge \neg\varphi_{IsInstantiated}(C, a_2) \wedge Meta(a_1) = Meta(a_2)\}$

$\varphi_{CardinalityCheck}(C, I)$: $\neg DeriveFrom(I, ID_{Attribute}) \vee Card(Meta(I))[0] \leq Count(a: \exists i: a=Attrib(C, i) \wedge \varphi_{InstCounter}(I, a)) \leq Card(Meta(I))$ (Atkinson and Kühne, 2001)

This first formula simply checks if *I* is instantiated in container *C*. The second formula checks if two attributes in the same container *C* are originated from the same meta definition. Finally, the third formula checks if an attribute *I* violates the cardinality constraints in the specific container *C*. In the definition, we use the function *Count* for the sake of simplicity. This function counts the elements fulfilling the given selector. In this particular case, we count the number of attributes, which have the given meta *M*. Note that we use the formula $\varphi_{InstCounter}$ to count both copied and instantiated elements.

$\varphi_{Typecheck}(T, v)$: $(T=ID_{Bool} \wedge v \in U_{Bool}) \vee (T=ID_{Number} \wedge v \in U_{Number}) \vee (T=ID_{String} \wedge v \in U_{String}) \vee (T=ID_{ID} \wedge v \notin \{free(U_{ID})\}) \vee \varphi_{IsValid}(v, T)$

The fourth formula checks if a specific value *v* violates the type constraint *T*. We apply function free on the universe of IDs to retrieve the set of unused IDs. The last part of the condition is used only if the type itself refers to a non-basic type. In this case, the value must be an instantiation of the referred type.

$\varphi_{AttribCheck}(I, a)$:

$\varphi_{IsValid}(a, Meta(a)) \vee \exists j: Attrib(Meta(I), j) = a \vee \{\exists m, k: m=Attrib(Meta(I), k) \wedge Value(m)=Value(a) \wedge Name(m) = Name(a) \wedge Meta(m) = Meta(a) \wedge (\nexists i: Attrib(a,i) \neq Attrib(m,i))\}$

The fifth formula checks if an attribute *a* is a valid instantiation, a copy, or a clone of a meta attribute of the meta of the container *I*. If it is a clone, then the same entity is used in the *Instance* as in the *MetaType*. If it is a copy, then only the *ID* and *Cardinality* labels can change.

3.4.2 Validation Formulae

$\varphi_{LabelCheck}(I, M)$: $Meta(I) = M$

$\varphi_{EntityIns}(I, M)$: {∃c, idx: Attrib(I, idx) = c ∧ $\varphi_{IsValid}$(c, Meta(c))} ∨ Value(I) ≠ undef

$\varphi_{AttribSrc}(I, M)$: ∄a, idx: Attrib(I, idx) = a ∧ {¬$\varphi_{CardinalityCheck}$(I, a) ∨ ¬$\varphi_{AttribCheck}$(I, M, a))}

$\varphi_{ValueCheck}(I, M)$:

¬DeriveFrom(I, $ID_{Attribute}$) ∨ Value(I) = undef ∨ {DeriveFrom(I, $ID_{AttribType}$) ∧ $\varphi_{IsValid}$(Value(I), Value(M))} ∨ {¬DeriveFrom(I, $ID_{AttribType}$) ∧ ∃t, idx: t = Attrib(M, idx) ∧ DeriveFrom(t, $ID_{AttribType}$) ∧ $\varphi_{Typecheck}$(t, Value(I))}

The first formula checks the correct usage of *Meta* label, the second checks if at least one of the attributes is instantiated, or has its value set. The third formula checks if there is an attribute violating the cardinality constraint or not having a valid meta definition in the meta of the entity. Finally, the fourth formula validates *AttribType* definitions. It returns true if the value is not set.

$\varphi_{Link}(I, M)$: ¬DeriveFrom(I, ID_{Base}) ∨ Card(Meta(I))[0] = Count{∃j, i : DeriveFrom(j, ID_{Link}) ∧ j = Attrib(I, i) }

The fifth formula checks if instances of *Base* have the correct number of links. That formula ensures that *Node* instances have zero links and *Edge* instances have two links. Obviously, if also *N-Edge* had been defined as follows

```
{"N-Edge", ID_N-Edge, ID_Base,
 [0, inf], undef,
 [
   {"Attributes", ID_Attributes, ID_Attribute,
    [0, inf], undef,[]}
   {"Links", ID_Link, ID_Attribute,
    [N, N], undef,[]}
 ]}
```

The fifth formula would be also able to check if N-Edge instances really possess N links as required. Now, with all the formulae combined we get the following:

$\varphi_{IsValid}(I, M)$: $\varphi_{LabelCheck}$(I, M) ∧ $\varphi_{AttribSrc}$(I, M) ∧ $\varphi_{EntityIns}$(I, M) ∧ $\varphi_{AttribType}$(I, M) ∧ φ_{Link}(I, M)

The validity of the new ASM state is checked to evaluate the below formula for all meta-types:

$\varphi_{Meta}(M)$: ¬DeriveFrom(M, $ID_{Attribute}$) ∧ Card(M)[0] ≤ Count(i: $\varphi_{IsValid}$(i, M)) ≤ Card(M) (Atkinson and Kühne, 2001)

4 EXAMPLE

In order to showcase, by a concrete example, how to tackle the node-edge dichotomy in DMLA a simplified network management example will be modelled. Let us assume that we create a model with a generic concept of routers (*RouterType*), a particular router type (*SimpleRouter*) and a router instance (*MyRouter*).

```
{"RouterType", ID_RouterType, ID_Node,
 [0, inf], undef,
 [
   {"IPAddr",ID_IPAddr,ID_Attribute,
    [0,inf],undef,
    [
     {"IPType",ID_IPType,ID_AttribType,
      [0,inf], ID_String,[]}
    ]
  }]}

{"SimpleRouter", ID_SimpleRouter, ID_RouterType,
 [0, inf], undef,
 [
   {"IPAddr",ID_IPAddr,ID_Attribute,
    [2,2],undef,
    [
      {"IPType",ID_IPType,ID_AttribType,
       [0,inf], ID_String,[]}
    ]
   }]}

{"MyRouter", ID_MyRouter, ID_SimpleRouter,
 [0, inf], undef,
 [
  {"In", ID_In, ID_IPAddress,
   [1,1], "192.168.0.1", []},
  {"Out", ID_Out, ID_IPAddress,
   [1,1], "192.168.0.2", []}
 ]}
```

The RouterType concept allows router types to have any IP addresses, which SimpleRouter restricts to 2. Finally, MyRouter sets the two IP address attributes to their concrete values. Let us now further assume that there is also a type called *Company* in the model that represents entities which manage routers and may also log those management activities. Management is expressed via a relation and logging by an attribute of Company that contains some of these management relation instances. The meta entities are as follows:

```
{"Company", ID_Company, ID_Node,
 [0, inf], undef,
 [
   {"Log",ID_Log, ID_Attribute,
    [0,inf],undef,
    [
      {"LogType",ID_LogType, ID_AttribType,
       [0,inf], ID_Management,[]}
```

```
    ]
   }]}
{"Management", ID_Management, ID_Edge,
 [0,inf], undef,
 [
  {"Src", ID_ManagementSrc, ID_Link,
   [1,1], ID_Company,
   [
    {"ESType", ID_ESType, ID_AttribType,
     [0,1], ID_Company,[]}
   ]
  },
  {" Trg", ID_ManagementTrg, ID_Link,
   [1,1], ID_RouterType,
   [
    {"ETType", ID_ETType, ID_AttribType,
     [0,1], ID_RouterType,[]}
   ]
  }]}
```

The cardinality restrictions of Management ensure that the Edge instances have only one source and one target end-points. The types of the end-points are also restricted to Company and RouterType. Finally, the fully instantiated entities will be created by fully instantiating all entity attributes:

```
{"MyManagement", ID_MyManagement, ID_Management,
 [0,inf], undef,
 [
  {"Src", ID_MyManagementSrc, ID_ManagementSrc,
   [1,1], ID_MyCompany, []},
  {"Trg", ID_MyManagementTrg, ID_MyManagementStr,
   [1,1], ID_MyRouter, []}
 ]}
{"MyCompany", ID_MyCompany, ID_Company,
 [0, inf], undef,
 [
  {"Log1", ID_Log1, ID_Log,
   [1,1], ID_MyManagement, []}
 ]}
```

The example clearly illustrates that there is no difference in the handling of Nodes and Edges. Both can have attributes, and both can be contained by other entities independently of them being either Nodes or Edges. Also, Edges can connect Nodes, or even Edges, defined at arbitrary meta-levels.

5 CONCLUSION

In this paper, we have formally described a novel multi-level modeling approach that consists of three functional building blocks: a precise ASM based structural algebraic representation, a generic bootstrap mechanism of the initial structural entities, and a dynamic instantiation process that is formalized via validation formulae. The specific bootstrap described in the paper was engineered to resolve the node-edge dichotomy within the formalism. We have demonstrated it by a simple modeling example. Knowing that ASM based approaches are easily implementable, our intention is now to establish a practical, industry-ready software framework for precise multi-level meta-modeling. Also, we aim to reformulate the TOSCA standard in DMLA so that by reaching these two goals a model based reference implementation could be had for the management of meta-model based Cloud service solutions.

ACKNOWLEDGEMENTS

This work was partially supported by the TÁMOP-4.2.1.D-15/1/KONV-2015-0008 project.

REFERENCES

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

Atkinson, C. & Kühne, T., 2002. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation (TOMACS),* 12(4), pp. 290-321.

Atkinson, C. & Kühne, T., 2003. Model-Driven Development: A Metamodeling Foundation. *IEEE Software,* 20(5), pp. 36-41.

Boerger, E. & Stark, R., 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis.* s.l.: Springer-Verlag Berlin and Heidelberg GmbH & Co. KG.

Neumayr, B., Jeusfeld, M. A., Schrefl, M. & Schütz, C., 2014. *Dual Deep Instantiation and Its ConceptBase Implementation.* s.l., s.n.

NFV, 2013. *Network Functions Virtualisation (NFV); Architectural Framewok.* [Online] Available at: http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf"

OASIS, 2013. *OASIS: Topology and Orchestration Specification for Cloud Applications Version 1.0.* [Online] Available at: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

OMG, n.d. *MDA - The Architecture of Choice For A Changing World.* [Online] Available at: http://www.omg.org/mda/

OMG, n.d. *MetaObject Facility.* [Online] Available at: http://www.omg.org/mof/