Towards a novel meta-modeling approach for dynamic multi-level instantiation

Zoltan Theisz Gergely Mezei
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
zoltan.theisz@gmail.com, gmezei@aut.bme.hu

Abstract. Instantiation is one of the core concepts of all current meta-modeling approaches and thus of all model-driven frameworks, nevertheless its semantics is left to the discretion of the particular modeling methodology. This sloppiness often results in various incompatibility and limitation issues. In this paper, we introduce a new, dynamic, multi-level instantiation technique that precisely formalizes the essence of instantiation. The dynamic nature allows the partial instantiation of certain meta-elements, while keeping the rest of the elements at a higher abstraction level. Supporting this level of flexibility is often required in complex IT systems e.g. in the telecommunication domain. The described solution is not limited by implementation details, it can be used as a common platform for any modeling tool.

Keywords: Meta-modeling; Instantiation; Abstract State Machines

1 Introduction

In model-based development techniques, instantiation is the very essence of any meta-modeling technique: by definition, the instantiation operation defines the semantical linkage between the meta-modeling and the modeling layer. The exact nature of the instantiation process has a dominant influence on the flexibility, the expressiveness and the limitations of the resulting modeling approach. If the operation of instantiation is not properly and precisely defined, the various modeling frameworks may tend to interpret it differently and thus they may become incompatible. It is usually taken for granted – at least in the practice – that instantiation equals to the well-known relation between classes and objects. The similarity may look obvious, but they must not be treated as being equivalent since their disparity becomes easily noticeable e.g. in the case of multi-level instantiation of MOF [1] [2]. We can distinguish two kinds of instantiations: shallow instantiation means that information is defined on the nth modeling level and it is used at the immediate instantiation level, while deep instantiation allows defining information on the nth modeling level and use it on the (n+x)th ($x > 0$) modeling level [3]. Although multi-level modeling solutions are getting more and more popular, deep instantiation methods are rarely used. The core of the issue is due to the fact that if each layer has to be instantiable, then there must also be a means of being able to add new attribute

and operation definitions to the model definition. There are two options to support it: one can either bring the source of this information along through all model layers (and 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. In practice, the concept of potency notion and dual field notion [3] [4] were introduced as such solutions. However, this level of flexibility does not always suffice. For example, the on-going modernization of the state-of-the-art telecom technology will propel the changes far beyond our current imagination.The future definitely lies in the Cloud and the whole telecom superstructure will become totally software driven, while the hardware infrastructure will purely consist of programmable network fabrics of routers and switches and clusters of software defined data centers of virtually infinite processing capacity. In this scenario, scheduled and gradual instantiation of information models are necessary. Under gradual instantiation we mean that we concertize (instantiate) some attributes and operations of the meta definitions, but not necessarily all of them in one single go. This added dynamism in the instantiation process is the main reason why we do refer to our approach as dynamic instantiation, which establishes a well-regulated dynamic switch between meta levels. The paper describes in a solid, mathematically precise way this new instantiation approach, which is a delicate combination of a model representation framework and a corresponding instantiation mechanism. In the next sections, firstly, an overview is given, then the formal definition is presented followed by a few simple examples. Finally, conclusions are drawn and future directions are highlighted.

2 Dynamic instantiation

In general, the following idea is used when talking about instantiation: let us take a meta definition and process it by instantiating all the defined items. If we have three attribute definitions, then we are, by definition, forced to instantiate all of them. We must not say, for example, that we instantiate only two and leave the last one alone. But, is it really useful, in all practical cases, to follow that rigid instantiation rule without gaining anything in return? In meta-model based Cloud software implementations, instantiation often means concretization. More precisely, services defined at higher modeling layers become concrete service types and concrete service instances on lower levels. For example, a service may be defined only as a template – a domain specific recipe – and as such it tends to have free variables. Obviously, one has to create concrete implementations of that service and consequently all necessary free variables must be gradually substituted for later executability. Hence, the inefficiency of forced substitution of all free variables in one single go may need to be superseded by a more pragmatic behavior: the creation of hierarchies of service templates and the gradual substitution of free variables in a well-scheduled manner. In essence, it would mean to instantiate only some part of the meta definition and leave the rest of it as "free variables" for later concretization. In our dynamic instantiation approach, we permit the transfer of meta definitions

to the next level without full instantiation. This behavior is somewhat similar to the potency notion-based methods where it is also explicitly specified that the instantiation information is there, but we would like to use it only later on a different modeling layer. Also, our instantiation differs, by its very nature, from usual inheritance, although inheritance relations can be redefined by it as well. In essence, our instantiation idea allows the holistic manipulation of multi meta-layers and therefore we can describe also such modeling scenarios with "free variables" that are beyond the reach of inheritance due to its constrainment to one single modeling layer. Moreover, our approach makes it possible to add new attributes to existing model elements, or remove unwanted attributes, provided their meta definition has been provisioned for this later conretization. The instantiation process also enforces type constraints by e.g. specifying the instance value(s) of selected attribute(s). To sum up, our dynamic instantiation takes advantage of a permissive instantiation approach, whereby attributes are not enforced to be always instantiated in a single go. However, we did not want to lose completely the concretization feature, therefore, we do insist on instantiating at least one "free" attribute at a time whenever a model element is to be instantiated. In the sequel, the precise mathematical model of these ideas will be formalized in their full exactness.

3 Formal syntax

In the following, we present a formalization of our dynamic instantiation technique. The formalization is referred to as Dynamic Multi-Layer Algebra (DMLA) and it is based on Abstract State Machines (ASM, [5]). Basically, an ASM formalism defines an abstract state machine and a certain set of connected functions that specify the transition logic between the states. DMLA consists of three major parts: The first part defines the modeling structure and shows the ASM functions operating on this structure. The second part is an initial set of constructs, built-in model elements (e.g. built-in types) that is necessary to use the basic 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 is self-contained in structure and can work with different bootstraps. Moreover, the bootstrap selection seeds the concrete meta-modeling capability of our generic DMLA.

3.1 Data representation

In our approach, the model is represented as a Labeled Directed Graph. Each model element such as nodes and edges can have labels. Attributes of the model elements are represented by these labels. Since the attribute structure of the edges follows the same rules applied to nodes, the same labeling method is used for both nodes and edges. Moreover, for the sake of simplicity, we use a dual

field notation in labeling that represents Name/Value pairs. In the following, we refer to a label with the name N of the model item X as X_N.

We define the following labels: (i) X_{Name} (The name of the model element), (ii) X_{ID} (A globally unique ID of the model element), (iii) X_{Meta} (The ID of the metamodel definition), (iv) $X_{Cardinality}$ (The cardinality of the model element, it is used during instantiation as a constraint. It determines how many instances of the model element may exist in the instance model.), (v) X_{Value} (the value of the model element (used in case of attributes only as described later), (vi) $X_{Attributes}$ (A list of attributes)

Due to the complex structure of attributes, we do not represent them as atomic data, but as a hierarchical tree, where the root of the tree is always the model item itself. Nevertheless, we handle attributes as if they were model elements. More precisely, we create virtual nodes from them: 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 defined as discussed above. After the structure of the modeling elements has been introduced, we can now define the Dynamic Multi-Layer Algebra itself.

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

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

Note that for the sake of simplicity we model the cardinality as a pair of lower and upper limits. Obviously, this representation could be easily extended to support ranges (e.g. "1..3") as well. The label Attrib is an indexed list of IDs, which refers to other entities. Now, let us have a simple example:

```
Person_ID = 12, Person_Meta= 123, Person_Cardinality = {0, inf},
Person_Value = undef, Person_Attrib = []
```

The definition formalizes the entity Person with its ID being 12 and the ID of its metamodel being 123. Note that in the algebra, we do not specify that the universe of IDs uses the universe of natural numbers, this is only a possible implementation here for sake of pure illustration purposes. Of course, we could have also used, for example, URIs, or GUIDs for achieving the same goal. The only requirement imposed on the universe is that it must be able to identify its elements uniquely. One can instantiate

any number of the Person entity in the instance models and it has no components and value defined. In the sequel, for the sake of easier reading, we are going to use a more compact representation with equal semantics (Note both tuples and lists share the same square bracket notation):

```
["Person", 12, 123, [0, inf], undef, []].
```

3.2 Functions

Functions are used to define rules to change states in ASM. In DMLA, we rely on *shared* and *derived* functions. The current attribute configuration of a model item is represented using *shared* functions. The values of these functions are modified either by the algebra itself, or by the environment of the algebra (for example by the user). *Derived* functions represent calculations, they cannot change the model, they are only used to obtain and restructure existing information. Thus, derived functions are used to simplify the description of the abstract state machine. The vocabulary $\sum$ of the Dynamic Multi-Layer Algebra formalism is assumed to contain the following characteristic functions:

- **Name**(U_{ID}): U_{String} – The function takes an ID of an entity and returns the name of the element. If no model element is defined by the given ID the function returns *undef*.
- **Meta**(U_{ID}): U_{ID} – The function takes an ID of an entity and returns the ID of the meta definition of the element. If no model element is defined by the given ID the function returns *undef*.
- **Card**(U_{ID}): [U_{Number},U_{Number}] – The function takes an ID of an entity and returns the minimum and maximum cardinality of the element as an array of two elements. If no model element is defined by the given ID the function returns *undef*.
- **Attrib**(U_{ID}, U_{Number}): U_{ID} – The function takes an ID of an entity and an index and returns the attribute of the entity at the given index position. If no model element is defined by the given ID or no attribute is defined at the index position the function returns undef.
- **Value**(U_{ID}): U_{Basic} – The function takes an ID of an entity and returns the data stored in the Value label of the entity. If no model element is defined by the given ID the function returns *undef*.

  Note that the functions are not only able to query the requested information, but they can also update the information based on the usual ASM syntax. For example, we can update the meta definition of an entity by assigning a value to the Meta function: $\text{Meta}(\text{ID}_{ConcreteObject}) := \text{ID}_{NewMetaDefinition}$
- **Contains**(U_{ID},U_{ID}): U_{Bool} – The derived function takes an ID of an entity and the ID of an attribute and checks whether the entity contains the attribute directly or indirectly. If there are no entities defined by the given IDs the function returns false.
- **DeriveFrom**(U_{ID},U_{ID}): U_{Bool} – The derived function checks whether the entity identified by the first parameter is an instantiation of the entity specified by the second parameter. The function checks along all meta levels, not just one above, thus, for example the meta-meta element can also be checked by the function. If no of these entities is found the function returns false.

The definitions of the functions are the following:

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

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

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

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

Here we assume that the Attrib labels return *undef* when the index is greater than or equal to the number of stored entities.

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

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

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

3.3 The bootstrap mechanism

The aforedefined functions make it possible to query and change the model. However based only on these constructs, it is hard to use the algebra (especially in the case of a self-describing multi-level architecture) due to the lack of basic, built-in constructs. For example, entities are required to represent the basic types, otherwise we cannot use the label Meta when it refers to a string, because the label is supposed to take its value from U_{ID} and not from U_{String}. We need to define the base constructs somewhere inside or outside the core algebra.

Obviously, there is more than one "correct" solution to define this initial set of information. For example, we can restrict the usage of basic types to an absolute minimum, or we can extend them by allowing implementation specific types, such as DateTime to simplify the usage. In this paper, we present a basic set of constructs, referred to as the bootstrap of the algebra. The bootstrap provides a practically useful minimal set of constructs however this set can be freely modified later to adapt to specific situations and one can still get a working algebra. The bootstrap has two main parts: basic types and principal entities.

Before describing the bootstrap in detail, it is worth mentioning that the bootstrap and the instantiation mechanism cannot be defined independently of each other. When an entity is being instantiated there are constructs to be handled in a special way. For example, we can check whether the value of an attribute violates the type constraint of the metamodel only if the algorithm can find and use the basic type definitions.

3.3.1 Basic types

The built-in types of the Dynamic Multi-Layer Algebra 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 and annotation attributes

Besides the basic types, we also define three principal entities: Attribute, Node and Edge. They act as the root meta elements of attributes, nodes and edges, respectively. All three principal entities refer to themselves by meta definition (more precisely, they are self-referring among themselves). Thus, for example, the meta of Attribute is the Attribute entity itself.

We should also mention that attributes are not only used as simple data storage units, but also to create annotations, which are to be processed by the instantiation. Similarly to basic types, we can define special attributes with specific meaning. By adding these special attributes to entities, we can fine-tune their handling. We refer to these attributes as annotation attributes from now on. We define three annotation attributes: AttribType, Src and Trg.

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. AttribType is only applied for attributes.

Src and Trg are used both as type constraints and data storage units to store the source and target node of an edge, respectively. The constraint part restricts which nodes can be connected by the edge, while the data storage contains its current value. The constraint is expressed by AttribType, while the actual data is stored in the Value field. This is only applied for edges.

In the following, we illustrate the concepts by a few simple examples. For the sake of legibility, we rely on the notation `ID_Attribute` to refer to the ID of the attribute and similarly to other entities.

3.3.3 Simple attribute

Our first example is rather simple, we define an attribute Age:

```
["Age", ID_AgeAttribute, ID_Attribute,[1,1], undef,
  [["AgeType",ID_AgeType,ID_AttribType,[0,1], ID_Number,[]]]
]
```

Age refers to `ID_Attribute` as its meta definition. Moreover, it has an entity (AgeType) as a subattribute. AgeType is an instance of the annotation attribute AttribType. It has been added to Age as a type constraint on the value of the instances of Age. The type constraint refers to `ID_Number` expressing that we can specify the age as a number as it is shown in the instance of Age:

```
["Age", ID_ConcreteAgeAttrib, ID_AgeAttribute, [1,1], 23,[]]
```

3.3.4 Complex attribute

The second example describes a complex data consisting of several fields:

```
["Name", ID_NameAttribute, ID_Attribute, [1,1], undef, [
  ["NameType",ID_NameType,ID_AttribType, [0,1], ID_ComplexName, []]
]]
```

As it can be seen, we specify that the value of this attribute is a custom type (ComplexName). The definition of the referred type is the following:

```
["ComplexName", ID_ComplexName, ID_Attribute, [1,1], undef, [
  ["FirstName", ID_FirstName, ID_Attribute, [1,1], undef, [
    ["FNType",ID_FNType, ID_AttribType, [0,1], ID_String,[]]
  ]],
  ["LastName", ID_LastName, ID_Attribute, [1,1], undef, [
    ["LNType", ID_LNType, ID_AttribType, [0,1], ID_String, []]
]]]]
```

In ComplexName, we define fields (FirstName, LastName) as nested attributes with their own type. By instantiating Name, we can obtain the following for example:

```
["Name", ID_Name, ID_NameAttribute, [1,1], ID_ConcreteName, []]
["ConcreteName", ID_ConcreteName, ID_ComplexName, [1,1], undef, [
   ["FirstName", ID_FirstName2, ID_FirstName, [1,1], "John", []],
   ["LastName", ID_LastName2, ID_LastName, [1,1], "Smith", []]
]]
```

On this instance level, we have an instance of ComplexName that is referred to from Name. In Concrete Name, we instantiate both the FirstName and the LastName attributes and set their values.

3.3.5 N-layer instantiation

DMLA provides two techniques to support n-layer attributes: transferring the attributes and instantiating them. Transferring an attribute means that the definition is copied from the meta-type to the instance object. No modification is allowed, since copy means in this context that we re-use the entity from the meta type. In contrast, when an attribute is instantiated, it becomes more concrete, for example, its value can be set (as already explained in the previous examples). Let us illustrate the difference on a simple example. We define a type Person with an attribute Aunt and add a type constraint enforcing that the value of Aunt must be of type Person.

```
["Person", ID_Person, ID_Node, [0, inf], undef, [
   ["Aunt", ID_Aunt, ID_Attribute, [0,2], undef, [
       ["DType",ID_DType,ID_AttribType,[0,1], ID_Person,[]]
]]]
```

When instantiating the type Person, we can decide to transfer the attribute Aunt to the next level without instantiating it. This allows us to specify, concretize the information stored in Aunt later. In this case, we can get an instance as follows:

```
["JakeSmith", ID_JakeS, ID_Person, [1, 1], undef, [
   ["Aunt", ID_Aunt, ID_Attribute, [0,2], undef, [
       ["DType",ID_DType,ID_AttribType,[0,1], ID_Person,[]]
]]]]
```

Note that the attribute Aunt is not a clone of the original Aunt, but they are the same (as their ID shows). Obviously, instead of transferring the attribute, we can also decide to instantiate it and create one instance of the attribute:

```
["JohnSmith", ID_JohnS, ID_Person, [1, 1], undef, [
   ["Aunt", ID_JohnAunt, ID_Aunt, [1,1], ID_JaneSmith, []]
]]
```

We can also omit the attribute, if the person does not have an aunt, since its minimum cardinality allows us to do so:

```
["JaneSmith", ID_JaneS, ID_Person, [1, 1], undef, []]
```

However, the cardinality also allows us to create more than one Aunt instances in Person entities. This is, where the flexibility of dynamic instantiation shows its value: we can also say that we would like to instantiate only one Aunt attribute and keep the other possible attribute instance for later use:

```
["JillSmith", ID_JillS, ID_Person, [1, 1], undef, [
   ["Aunt", ID_JillAunt, ID_Aunt, [1,1], ID_JaneSmith, []],
   ["Aunt", ID_AuntCopy, ID_Aunt, [0,1], undef, [
       ["DType",ID_DType,ID_AttribType,[0,1], ID_Person,[]]
]]]]
```

This is similar as if we had transferred the attribute, but its cardinality had to be changed, therefore we cannot reuse the entity from the metamodel verbatim, so we have created a new one instead. This new attribute is basically a clone of its meta definition, but the ID, Meta and Cardinality labels have been modified. By instantiating this hybrid object we can create objects such as:

```
["MySelf", ID_MySelf, ID_JillS, [1, 1], undef, [
  ["Aunt", ID_JillAunt, ID_Aunt, [1,1], ID_JaneSmith, []]
  ["Aunt", ID_JillAunt2, ID_Aunt, [1,1], ID_JaneSmith2, []]
]]
```

We can also specify that from now on, we do not accept Persons in general as the value of Aunt, but only a specific instance of the Person type. For example we can modify our earlier example:

```
["JohnSmith", ID_JohnS, ID_Person, [1, 1], undef, [
   ["Aunt", ID_JohnAunt, ID_Aunt, [1,1], ID_JaneSmith, [
       ["DType2",ID_DType2,ID_DType,[0,1], ID_JaneS,[]]
]]]]
```

This definition reconfigures the attribute Aunt by adding a new type constraint to it. From this point on, we can use only instances of the entity JaneSmith as the value in the instances of the entity JohnSmith:

```
["MySelf", ID_MySelf, ID_JohnS, [1, 1], undef, [
   ["Aunt", ID_MyAunt, ID_JohnAunt, [1,1], ID_MyAunt, []]
]]
["MyAunt", ID_MyAunt, ID_JaneS, [1,1], undef, []]
```

Thus the entity MySelf can refer to a specific instance of JaneSmith, not to the type. The same thread of thinking applies equally well to attributes which have cardinality [0, inf], e.g. in the case of an alternative definition of type Person:

```
 ["Person", ID_Person, ID_Node, [0, inf], undef, [
   ["Relative", ID_Relative, ID_Attribute, [0,inf], undef, [
      ["DType3",ID_DType3,ID_AttribType,[0,1], ID_Person,[]]
]]]]
```

Here, different kinds of relatives can be introduced later, such as children, uncles, cousins etc. This flexibility through "late binding" of attribute type and value is one of the main advantages of dynamic N-layer instantiation.

3.4 Instantiation

Based on the structure definition of the algebra and the bootstrap, we can represent our models as states of the DMLA now. The next step is to define an instantiation operation that takes an entity and produces a valid instantiation of it. Instantiation is rarely unambiguous, one can usually create several 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 manipulating their values also from the outside of the algebra. However, the functions do not validate these manipulations because that would be a considerably complex task. In a practical implementation of our proposed mechanism, such situations can be easily controlled based on change notification and a transaction handling policy. Nevertheless, it may be possible that an invalid model definition occurs. Hence, in our approach, we distinguish between valid and invalid models, where validity checking is based on formulae describing different properties of the model. Therefore, in the following, we assume that whenever external actors change the state of the algebra, the formulae are evaluated. The evaluation result is important indeed since instantiation must only work on valid models. Therefore, the instantiation process is specified via validation rules (set solution) that ensure that if an invalid model may result from an instantiation (a particular point solution), it is rejected and an alternative instantiation is selected and retried.

Validation is based on several, more or less atomic formulae. Most of the formulae take an Instance entity and a MetaType entity and they check whether the Instance entity is a valid instance of the MetaType entity. The only exception, formula φ_{Meta}, takes only one parameter and validates if the given entity has enough valid instances according to the cardinality label. The formulae we rely on are the following:

- $\varphi_{LabelCheck}$: The Meta label of the Instance refers to the ID of MetaType.
- $\varphi_{AttribSrc}$: All attributes of the Instance must be a clone, a copy, or a valid instantiation of an attribute of the MetaType. 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. Attributes must not violate the cardinality constraint defined by the meta definition.

- $\varphi_{EntityIns}$: Instance must always have at least one instantiated (sub)attribute, or its value must be set.
- $\varphi_{ValueCheck}$: If a component is a direct or indirect instantiation of Attribute and it has a value, 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.
- φ_{Edge}: Each entity that is a direct, or indirect instantiation of Edge must have Src and Target attributes, which must be a valid direct or indirect instantiation of the appropriate attributes of Edge.
- $\varphi_{IsValid}$:True, if all the other instance checker formulae (formulae with two parameters) return true
- φ_{Meta}: The number of valid instances of the parameter entity does not violate the cardinality constraint.

The meta definition describes the structure, a set of requirement, or in other words the constraints the instance must conform to. As mentioned before, in most of the cases, these constraints do not have a unique solution, but there are several ones. There exist many options, decision points during a usual instantiation step. For example, if we have an attribute, we can transfer it, or instantiate it, or we can set its value in certain circumstances. Therefore, instantiation is not a procedure which has a simple input parameter and produces the required instance object. It is rather similar to a procedure that uses instructions as input where the instructions consist of a selector and a bound action. For example, if we have a complex attribute, then a selector can select one of its components and apply the action on the selected components. The input of the instantiation algorithm is a set of such instructions and we apply them one by one during the instantiation step. We model these instructions as a tuple $\{\lambda_{selector}, \lambda_{action}\}$ with abstract functions. The function $\lambda_{selector}$ takes an ID of an entity as parameter and returns with a possibly empty list of IDs referring to the selected entities. The function λ_{action} takes an ID of an entity and applies an action on it. We do not specify here which kind of actions λ_{action} can execute. The only limitation we set is that it must invoke only functions previously defined on the ASM. As mentioned, the functions $\lambda_{selector}$and λ_{action} are abstract in nature, which allows us to handle them as black boxes and do not have to specify their inner mechanisms explicitly. Although this choice may seem a bit restrictive, we have intentionally decided to follow this abstract approach because any concrete formalization such as specifying a formal model for the selection and action languages would be too heavy for the scope of this paper. Nevertheless, having acknowledged that limitation, we take it for granted now that operations can be added to the bootstrap similar to attributes and those operations can be shaped to specify selectors and actions. Hence, based on the above two abstract functions, the instantiation algorithm can be defined as follows:

Algorithm 1 The instantiation algorithm

```
1: rule Instantiate(ID_SubjectEntity, Instructions)
2: for all λ_selector, λ_action in Instructions do
3:     for all SelectedEntity in λ_selector(ID_SubjectEntity) do
4:         λ_action(SelectedEntity)
```

4 Conclusion and future works

The motivation of the paper was to create a precise formal instantiation approach providing a solid based for metamodeling environments and at the same time offering a solution to the challenges of modern, multi-layer, dynamic IT systems. We have formally introduced the new multi-level modeling approach addressing these challenges. The approach relies on three theoretical building blocks: a precise structural representation based on ASM semantics, a generic bootstrapping mechanism which defines the initial structural elements of the meta-modeling algebra, and a dynamic instantiation process that abstractly specifies the relevant validation formulae. The approach is theoretically solid and practically easily implementable; hence, our intention is to use it in the future as a potential framework for precise multi-level meta-modeling. The main mechanisms of the approach were illustrated by a few simple examples.

We have several plans on how to continue this research. Our research directions will target the detailed investigation of practical bootstrapping solutions, including refined operations of the ASM semantics itself, and the elaboration of formal models for the selection and action languages needed for any practical implementation of the dynamic instantiation process. Also, we consider to apply DMLA to redefine legacy instantiation concepts. On the practical side, we plan to use DMLA as a foundation for the implementation and management of meta-model based Cloud service solutions in the domains of distributed industrial automation, modern virtualized telecommunications networks and cooperative mobile applications operating on individualized any-media sources.

Acknowledgments

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-20120013) organized by VIKING Zrt. BalatonfĂźred.

References

[1] OMG, "Meta-object facility." http://www.omg.org/mof/.

[2] OMG, "Mda - the architecture of choice for a changing world." http://www.omg.org/mda/.

[3] T. Atkinson, Colin; Kuhne, "The essence of multilevel metamodeling," *The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, 2001.

[4] T. Atkinson, Colin; Kuhne, "Rearchitecting the uml infrastructure," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 2002.

[5] *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.