Attribute Algebra for N-layer Metamodeling

GERGELY MEZEI, TIHAMÉR LEVENDOVSZKY, HASSAN CHARAF
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
Goldmann Gyorgy tér 3, Budapest, H-1111
HUNGARY

Abstract: Metamodeling is a popular method to apply software modeling. A metamodel acts as set of rule for its instantiation, the model. The instantiation mechanism, originally restricted to two-layers, is more and more often generalized to *n-layer*. One of the most important issue in supporting n-layer instantiation in metamodeling is to create and handle n-layer attributes. The paper presents the N-layer Attribute Algebra, the formal description of our n-layer attribute structure and the transformation used in instantiation.

Key–Words: Metamodeling, N-layer, Attribute, Formalism, Abstract State Machines

1 Introduction

Using metamodels is a well accepted method both in creating visual languages, and reusing existing domains by modifying the metamodel. Models are instantiations of the appropriate metamodel, which means that a metamodel defines the topology and the attribute structure of their models. The issues of two-layer instantiation can be divided into two groups: the issues related to topological rules (e.g. the instantiation of the association in the UML class diagram [1] as links in the UML object diagram), and the issues of attribute-related instantiation rules (e.g. the UML class attributes can have values in the object diagram). In the two-layer case, the solution to these issues have been standardized by OMG [2]. However, extending the two-layer instantiation to n-layer is a natural need. This extension generates a couple of new problems to solve. The topological problems have been discussed and solved [4], but extending the attribute instantiation concepts to an n-layer instantiation mechanism is an open issue.

The paper presents the result of our research in creating a generic attribute structure that is able to describe the attributes of visual languages in n-layer metamodeling. The paper [3] has presented the implementation details of our research, thus, this paper focuses on the theoretical aspects of the research. The paper elaborates a mathematical formalism to describe the attribute structure and the instantiation method. The formal description is referred to as *N-layer Attribute Algebra*.

Section 2 introduces Abstract State Machines, which is the formalism technique we used in creating our algebra. Section 3 presents the algebra itself, while Section elaborates the instantiation transformation 4. In Section 5 we conclude and summarize the paper.

2 Backgrounds

In [5], ASMs are introduced as follows. ASMs are finite sets of *transition rules* of the form

$$\textbf{if } Condition \textbf{ then } Updates$$

which transform abstract states. Where *Condition* (referred to as guard) under which a rule is applied, is an arbitrary predicate logic formula without free variables. The formula of *Condition* evaluates to *true* or *false*.

Updates is a infinite set of assignments in the form of

$$f(t_1, \ldots, t_n) := t,$$

whose execution is understood as changing (or defining if there was not defined before) the value of the occurring function f at the given arguments.

The notion of *ASM states* is the classical notion of mathematical structures where data come as abstract objects, i.e., as elements of sets (domains, universes, one for each category of data) which are equipped with basic operations (partial functions) and predicates (attributes or relations).

The notion of *ASM run* is the classical notion of computation in transition systems. An ASM computation step in a given state consists of executing simultaneously all updates of all transition rules whose guard is *true* in the state, if these updates are *consistent*. A

set of updates is called *consistent* if it contains no pair of updates with the same location, e.g. with the same parameter set. For the evaluation of terms and formulas in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used.

Simultaneous execution of an ASM rule *R* for each *x* satisfying a given condition φ:

$$\textbf{forall } x \; with \; \varphi \;\; R,$$

where φ is a Boolean-valued expression and *R* is a rule. We freely use abbreviations, such as *where*, *let*, *if then else*, *case* and similar standard notations which are easily reducible to the basic definitions above.

A priori no restriction is imposed either on the abstraction level or on the complexity or on the means of the function definitions used to compute the arguments and the new value denoted by t_i, *t* in function updates. The major distinction made in this connection for a given ASM *M* is that between *static* functions which never change during any run of *M* and *dynamic* ones which typically do change as a consequence of updates by *M* or by the environment. The dynamic functions are further divided into four subclasses. *Controlled* functions are dynamic functions which can directly be updated by and only by the rules of *M*. *Monitored* functions are dynamic functions which can directly be updated by and only by the environment. *Interaction* or *shared* functions are dynamic functions which can directly updated by rules of *M* and by the environment. *Derived* functions are dynamic functions which cannot be directly updated either by *M* or by the environment, but are nevertheless dynamic because they are defined in terms of static and dynamic functions.

2.1 The mathematical definition of ASM

In an ASM state, data is available as abstract elements of domains which are equipped with basic operations represented by functions. Relations are treated as *Boolean*-valued functions and view domains as characteristic functions, defined on the superuniverse which represents the union of all domains.

Definition 1 *A vocabulary (also called signature)* Σ *is a finite collection of function names. Each function name* f *has an* arity*, which is a non-negative integer representing the number of arguments the function takes. Function names can be static or dynamic. Nullary function names are often called constants; but the interpretation of dynamic nullary functions corresponds to the variables of programming. Every ASM vocabulary is assumed to contain the static constants* undef*,* True *and* False*.*

Definition 2 *A state* $\mathfrak{A}$ *of the vocabulary* Σ *is a nonempty set* X*, together with interpretation of the function names of* Σ*, where* X *means the superuniverse of* $\mathfrak{A}$ *. If* f *is an n-ary function name of* Σ*, then its interpretation* $f^{\mathfrak{A}}$ *is a function from* X^n *into X; if* c *is a constant of* Σ*, then its interpretation* $c^{\mathfrak{A}}$ *is an element of* X*. The superuniverse* X *of the state* $\mathfrak{A}$ *is denoted by* $|\mathfrak{A}|$*.*

The $elements$ of the state are the elements of the superuniverse of the state and according to the definition, the parameters of the functions are also elements of the superuniverse.

Definition 3 *An abstract state machine M consists of a vocabulary* Σ*, an initial state* $\mathfrak{A}$ *for* Σ*, a rule definition for each rule name, and a distinguished rule name called the main rule name of the machine.*

3 The N-layer Attribute Algebra

N-layer Attribute Algebra represents the attributes of a model item in a tree structure, where the root of the tree is the model item itself. All model items and attributes have a unique identifier (ID) in order to make the identification and reference handling simpler.

We represent the current attribute configuration of a model item using *shared* functions. The value of the functions an changed either by the algebra itself, or by the environment of the algebra (for example by the user). A state of the algebra represent an attribute configuration, namely a concrete set of parameter set-value pairs for the functions. For example, there is a state S_1, in which we have a model item m_1 with no attributes defined. If a new attribute is defined or an old attribute is modified/deleted, then the value of the functions and the current state of the algebra changes. A possible next state of S_1 is S_2, which contains the same model item, but in this case m_1 has an attribute a_1.

The function $Meta(ID)$ returns the meta item/attribute of the model item/attribute obtained as parameter. The value of the function changes usually when a new model item/attribute is created. The function $IsAttribute(ID)$ is used to differentiate between attribute and model item IDs, it returns true, if the parameter is the ID of an attribute. The function $Name(ID)$ results in the name of the model item, or attribute retrieved as parameter. The function $Root(ID)$ obtains the root of the attribute tree, namely the model item. The function $Attribute(ID, N, I)$ is used to obtain the value of an attribute. The first parameter selects the model item, or the attribute (p) which is the parent of the attribute

we are trying to find. Then, the function collects children nodes of p that has the name defined in the second parameter. The third parameter selects the I^{th} element from this list. If p cannot be found, or it does not have the children requested, the function returns $undef$.

Definition 4 *The vocabulary OCLASM of the OCLASM formalism is assumed to contain the following characteristic functions (arities are denoted by dashes): Meta/1, IsAttribute/1, Name/1, Root/1, Attribute/3.*

Definition 5 *The superuniverse $|\mathfrak{A}|$ of a state $\mathfrak{A}$ of the N-layer Attribute Algebra is the union of four universes:*

- U_{Bool} *containing logical values* {*true/false*}

- U_{Number} *containing ration numbers* $\{\mathbb{Q}\}$

- U_{String} *containing character sequences of finite length*

- U_{ID} *containing all the possible IDs.*

3.1 Valid and Invalid Models

The N-layer Attribute Algebra distinguish valid and invalid models, where checking of validity is based on formulas describing different properties of the model (the formulas are shown at the top of the next page). When external systems are changing the attribute tree using the functions of the algebra, these formulas are automatically re-checked.

In order to simplify the definition of the formulas, a helper function ($\varphi_{IsUnique}(ID, Name)$) is introduced. The function $\varphi_{IsUnique}(ID, Name)$ shows whether the $Name$ attribute is unique among the child attributes of the model item/ parent attribute defined by ID. Using this helper function the following properties can be defined: (i) each model item has a unique *InstanceName* attribute, (ii) each attribute with the name 'Attribute' has *Name*, *MinMul*, *MaxMul* and *Type* children; these children are unique, (iii) two *Attribute* definitions with the same *Name* cannot exist (iv) each attribute with the name 'ComplexType' has a *Name* child and one ore more *Attribute* children,(v) two complex type definitions with the same *Name* cannot exist, (vi) $Attribute$s and $ComplexType$s cannot use the reserved words of the algebra as $Name$. Valid models are models fulfilling all six conditions.

4 The Transformation

In order to solve the issues of n-layer attribute handling, it is not enough to create an attribute structure, but an instantiation transformation is required as well. The instantiation transformation is described by a rule of the algebra. When creating a new model item, then this rule is used to initialize the attribute structure of the item according to the attributes defined in the metamodel. To simplify the definition of the instantiation rule, we define helper functions as follows:

$$GetAttributes(P) = \{\forall ID : \exists I : ID <> undef \wedge ID = Attribute(P, `Attribute`, I)\}$$

$$ChildCounter(P, N) = I_1 : Attribute(P, N, I_1) <> undef \wedge (\nexists I_2 : I_2 > I_1 \wedge Attribute(P, N, I_2) <> undef)$$

$$Add(P, N) = ID : \{ID = new\, U_{ID} \wedge IsAttribute(ID) = true \wedge Attribute(P, N, ChildCounter(P, N) + 1) := ID\}$$

$$ComplexAttribute(A) = Attribute(A, `Type', 0)\ in \{`Bool`, `Number`, `String`, `ID`\}$$

$$GetComplexType(A, N) = Attribute(Attribute(Root(A), `ComplexType`, 0), N, 0)$$

The function $GetAttributes$ obtains the children of the selected model item/attribute P, with the name 'Attribute'. The function $ChildCounter$ counts how many children the selected parent P has with the given name N. The function Add creates a new child attribute for P in the attribute tree, where the name of the new attribute is set to N. The function $ComplexAttribute$ is used to check whether an attribute is of the simple type. The function $GetComplexType$ retrieves the first *ComplexType*, which is defined in the model item $Root(A)$ and have the name N.

The first parameter of the rule $TransformAttributes$ is the *local root*, the attribute, or model item whose children we want to process. The second parameters is the meta item, or attribute of this local root.

Firstly, if the local root is a model item, then the name of the model item is set to $InstanceName$ attribute of the metamodel item. Moreover, an empty $InstanceName$ attribute is added to the model item automatically in order to obey the rules of valid models.

In the next step, the list of $Attribute$s are traversed and processed one-by-one. The current attribute definition in the metamodel is selected by A.

$$\varphi_{IsUnique}(ID, Name) = \begin{cases} true, & \text{if } \exists I_1 \wedge \nexists I_2 : \; Attribute(ID, Name, I_1) <> undef \\ & \qquad \wedge Attribute(ID, Name, I_2) <> undef \wedge I_1 <> I_2 \\ false, & \text{otherwise} \end{cases}$$

$$\varphi_{InstName} = \begin{cases} true, & if\ \forall ID : \; if(\neg IsAttribute(ID)) \rightarrow \varphi_{IsUnique}(ID,' InstanceName') \\ false, & otherwise \end{cases}$$

$$\varphi_{Attr} = \begin{cases} true, & if\ \forall ID : \; if(Name(ID) =' Attribute') \rightarrow (\varphi_{IsUnique}(ID,' Name') \wedge \\ & \varphi_{IsUnique}(ID,' MinMul') \wedge \varphi_{IsUnique}(ID,' MaxMul') \\ & \wedge \varphi_{IsUnique}(ID,' Type')) \\ false, & otherwise \end{cases}$$

$$\varphi_{AttrName} = \begin{cases} true, & if\ \forall ID_1 : \; if(Name(ID_1) =' Attribute') \rightarrow (\nexists ID_2 : \\ & Name(ID_2) =' Attribute' \wedge ID_1 <> ID_2 \\ & Attribute(ID_1,' Name', 0) = Attribute(ID_2,' Name', 0)) \\ false, & otherwise \end{cases}$$

$$\varphi_{CplxType} = \begin{cases} true, & if\ \forall ID : \; if(Name(ID) =' ComplexType') \rightarrow \\ & (\varphi_{IsUnique}(ID,' Name') \wedge \\ & (\exists Idx : Attribute(ID,' Attribute', Idx) <> undef)) \\ false, & otherwise \end{cases}$$

$$\varphi_{TypeName} = \begin{cases} true, & if\ \forall ID_1 : \; if(Name(ID_1) =' ComplexType') \rightarrow (\nexists ID_2 : \\ & Name(ID_2) =' ComplexType' \wedge ID_1 <> ID_2 \\ & Attribute(ID_1,' Name', 0) = Attribute(ID_2,' Name', 0)) \\ false, & otherwise \end{cases}$$

$$\varphi_{ReservedNames} = \begin{cases} true, & if\ \forall ID : \; if(Name(ID) in \{' Attribute',' ComplexType'\} \rightarrow \\ & (Attribute(ID,' Name', 0) \notin \\ & \quad \{' InstanceName',' Attribute',' ComplexType'\}) \\ false, & otherwise \end{cases}$$

$$\varphi_{ValidModel} = \varphi_{InstName} \wedge \varphi_{Attr} \wedge \varphi_{AttrName} \wedge \varphi_{CplxType} \wedge \varphi_{TypeName} \wedge \varphi_{ReservedNames}$$

Algorithm 1 The $TransformAttributes$ rule

```
1: rule TransformAttributes(ID, ID_Meta)
2: if (¬ IsAttribute(ID)) then
3:    Name(ID) =
          Attribute(ID_Meta,'InstanceName',0)
          Add(ID,'InstanceName')
4: for all A in GetAttributes(ID_Meta) do
5:    N = Attribute(A,'Name',0)
6:    for i= 0 TOAttribute(A,'MinMul',0) do
7:       Child= Add(ID,N)
8:       Meta(Child)= A
9:       if ComplexAttribute(A) then
10:         TransformAttributes(
                  GetComplexType(A),Child)
```

The rule creates attributes ($Child$) according to the minimum multiplicity ('MinMul') property of A.

Note that the number of descendant levels of an attribute in the attribute tree is not limited by the algebra. The children of an attribute can have children as well. For example, the attribute a_1 is of complex type c_1. C_1 has two sub-attributes a_2 and a_3. N-layer Attribute Algebra does not limit nesting of complex types, the type of a_2 or a_3 can be a complex type as well. In the instantiated model, attributes created from a_1 have two children attributes as defined in (a_2, a_3) and they can have grandchildren or other descendants as well. Therefore, if the type defined in A is a complex type, then the rule processes the definition of the type recursively.

The presented transformation rule can create a basic, initial and valid attribute structure for model items. Note that this attribute structure can be modified later both by external systems (representing for example the user), or by other, internal algorithms as

long as the formula $\varphi_{ValidModel}$ remains true.

UML class models have an instantiation definition standard, which is applicable only to class diagrams. In case of UML, we cannot use the same instantiation in class diagrams and for example in statechart diagrams. In contrast, the rule $TransformAttributes$ can be used in a transparent way between modeling layers. It does not need any special information, or property of the metamodel in order to be applied. The only restriction to the rule is that it can process only valid metamodels, where validity is defined by the $\varphi_{ValidModel}$ formula.

4.1 Re-Instantiated attributes

There are cases, when an attribute is defined on the metamodel level, it is mainly used on the model level, but it is also required on the instantiation level of the model. This is the case for example with UML class diagram - object diagram relationship. The structure of class diagrams is defined in MetaClass model. This model defines for example that classes can have class attributes and operations. An instantiation of the MetaClass model is a class diagram. In class diagrams, we can have classes with class attributes defined. For example, the class C_1 has a class attribute with string type and the attribute name 'MyVar'. The instantiation of class diagrams produces object diagrams. An instantiation of C_1 is the object diagram O_1. Here, we should be able to set the *value* of 'MyVar'. Recall that MyVar was defined in C_1, but the structure of class attributes originally came from the MetaClass model. In other words, we have defined the structure of an attribute on the n^{th} level, we have set some properties of the attribute on the $(n+1)^{th}$ level, but we need the attribute on the $(n+2)^{th}$ level as well. This requirement is referred to as *re-instantiation* of an attribute.

The transformation rule $TranformAttributes$ does not support re-instantiation. It processes only the $Attribute$s of the selected metamodel. In order to support re-instantiation, the rule must be extended. We use a helper function $GetChildren$ to simplify to extend the transformation rule.

$$GetChildren(P) = \{\forall ID : \exists I, N : ID <> undef \wedge ID = Attribute(P, N, I)\}$$

The main difference between $TransformAttributes$ and its improved version $TransformAttributes_2$ is that two new children (***ReInstantiate*** and ***ReInstantiationName***) are added to each attribute having complex type. Using the example cited earlier, MetaClass model defines an $Attribute$ with a $Name$ tag 'ClassAttribute', where the $Type$ tag of the attribute is 'ClassAttributeType'. Moreover, MetaClass has a $ComplexType$ with a $Name$ tag 'ClassAttributeType' (Fig. 1).

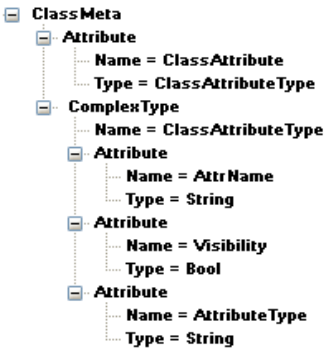

Figure 1: Attribute tree example: MetaClass

In C_1, these attributes are instantiated (Fig. 2). The result is a $ClassAttribute$ attribute, which has several children, such as 'Visibility' or 'AttributeType'. The rule $TransformAttributes_2$ adds two children (***ReInstantiate*** and ***ReInstantiationName***) automatically to $ClassAttribute$. Moreover, note that $\varphi_{ValidModel}$ allows to add $Attribute$ children to an arbitrary complex attribute. These additional $Attribute$ children are useful if the original attribute definition ($ClassAttribute$ in this case) must be extended in order to use on the re-instantiated level. This is the case with object diagram O_1 that requires the ability to set the *value* of class attributes.

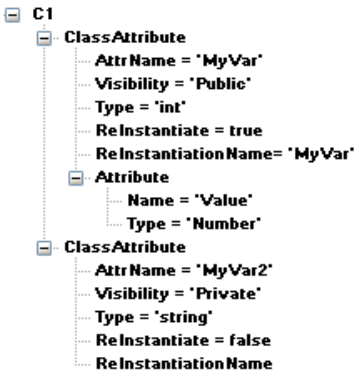

Figure 2: Attribute tree example: Class diagram

The boolean value ***ReInstantiate*** on the n^{th} modeling level indicates whether the attribute should take part in instantiation to the $(n+1)^{th}$ level. In the example, if we set ***ReInstantiate*** in $ClassAttribute$ to *true*, the attribute definition will be available in O_1. Since C_1 can have more than one $ClassAttribute$, it is essential to distinguish them in O_1. Different $ClassAttribute$s are transformed to attributes

with different name, where the name is defined in *ReInstantiationName*. Original components of $ClassAttribute$ are copied to the re-instantiated level O_1 (Fig. 3).

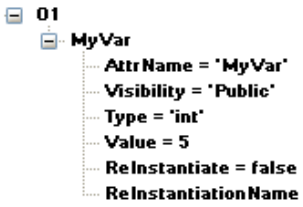

Figure 3: Attribute tree example: Object diagram

Algorithm 2 The $TransformAttributes_2$ rule

1: **rule** $TransformAttributes_2(ID, ID_{Meta})$
2: **if** ($\neg$ ISATTRIBUTE(ID)) **then**
3: NAME(ID) = ATTRIBUTE(ID_{Meta},'InstanceName',0)
4: ADD(ID,'InstanceName')
5: **for all** A in GETCHILDREN(ID_{Meta}) **do**
6: **if** Name(A)='Attribute') **then**
7: N = ATTRIBUTE(A,'Name',0)
8: **for** i= 0 TO ATTRIBUTE(A,'MinMul',0) **do**
9: $Child$= ADD(ID,N)
10: **if** COMPLEXATTRIBUTE(A) **then**
11: TRANSFORMATTRIBUTES(GETCOMPLEXTYPE(A),$Child$)
12: ADD($Child$,'ReInstantiate')
13: ADD($Child$,'ReInstantiationName')
14: **else**
15: **if** (GETCHILDREN(A)<>undef $\wedge$ Attribute(A,'ReInstantiate')=true $\wedge$ Attribute(A,'ReInstantiate')<>'') **then**
16: $Child$= ADD(ID, Attribute(A,'ReInstantiationName'))
17: **for all** A_2 in GETCHILDREN(A) **do**
18: **if** Name(A_2)<>'ReInstantiate') $\wedge$ Name(A_2)<>'ReInstantiationName') **then**
19: $Child_2$= ADD($Child$,Name(A_2))
20: TRANSFORMATTRIBUTES(GETCOMPLEXTYPE(A_2),$Child_2$)
21: ADD($Child_2$,'ReInstantiate')
22: ADD($Child_2$,'ReInstantiationName')

5 Conclusion

Metamodeling is one of the most focused research field in software modeling. Original concepts describing two-layer metamodeling structures are not flexible enough to fit the requirements of today. One of the largest open issues in creating real n-layer metamodeling solutions is to define an n-layer instantiation mechanism. This paper has presented a solution, the N-layer Attribute Algebra to solve this issue. The paper introduced an attribute structure, which is highly generic and modeling layer-transparent. The attribute structure has been formalized using Abstract State Machines; the definition of valid models has been created as well. In order to show that the attribute structure can really be used in an n-layer instantiation chain, the paper has elaborated a transformation rule ($TransformAttributes$) capable of applying the instantiation. The rule has been formalized as well. The paper has defined re-instantiated attributes using a simple example. Re-instantiated attributes are used in more, than two modeling layers. The instantiation rule has been extended ($TransformationAttributes_2$) in order to support re-instantiated attributes as well.

The paper has presented an N-layer Attribute Algebra in a formal way. The algebra has been implemented in Visual Modeling and Transformation System (VMTS, [6]) and it has been successfully used in several domains of interest, e.g. resource editors for mobile phones, UML 2.0 models, feature modeling or visual model transformation languages. Formal definitions and flexible constructs of the algebra ensures that the solution is not only useful in VMTS, but in any other n-layer metamodeling system as well.

References:

[1] UML, http://www.uml.org/

[2] OMG, http://www.omg.org/

[3] G. Mezei, T. Levendovszky, L. Lengyel, H. Charaf, "A Flexible Attribute Instantiation Technique for Visual Languages", IASTED on SE, February 15-17, 2005, Innsbruck, Austria, pp. 355-359.

[4] T. Levendovszky, L. Lengyel, H. Charaf, "A UML Class Diagram-Based Pattern Language for Model Transformation Systems", WSEAS Transactions on Computers, Issue 2, Volume 4, February, 2005, ISSSN 1109- 2750, pp. 190-195

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

[6] VMTS Web Site, http://www.vmts.aut.bme.hu/