Validated Multi-Layer Meta-modeling via Intrinsically Modeled Operations

Gergely Mezei, Dániel Urbán
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
{gmezei, urban.daniel}@aut.bme.hu

Zoltán Theisz
Huawei Design Centre,
Dublin, Ireland
zoltan.theisz@huawei.com

***Abstract*— State-of-the-art meta-model based technologies are facing increasing pressure under new challenges originating from practical industrial applications. Dynamic Multi-Layer Algebra (DMLA) is a novel generic modeling approach that intends to meet these challenges by combining multi-level meta-modeling and validated instantiation. This paper describes the core ideas and techniques that have been applied to DMLA in order to create a modular model intrinsic validation of instantiation. The approach is also illustrated by a simple example.**

Keywords*— *validated meta-model; modular constraints; multi-level modeling

I. Introduction

Meta-model based solutions have been gradually gaining acceptance in many complex industrial applications, for example in the domains of telecommunications, IoT and cloud management systems. Many of these applications rely on EMF technologies to provide facilities for type safe storage and manipulation of millions of configuration and control parameters. Nevertheless, those practical model-based solutions are seriously hampered by a legacy, that is, meta-model induced database schemas must be maintained by design-time derived toolchains such as EMF/CDO. Thus, these solutions are hardly capable of automatically coping with a massive amount of frequently changeable instance data that is mostly regulated by a slowly, but steadily evolving set of type information controlled by some product life-cycle schemes. Therefore, an integrated validation mechanism of design- and run-time models should become part of any such industrial solutions. Also, the validation mechanism shall be both modular and capable of maintaining the instance data and the meta-model schemas in sync irrespective of their abstraction levels and design or run-time nature. This practicality requirement asks for a proper multi-level meta-modeling technique, which supports the validation of the instantiation step not as an afterthought, but as a direct consequence of the underlying multi-level modeling formalism.

The paper describes our proposed solution, the Dynamic Multi-Layer Algebra (DMLA), which represents a modular, operation-based, multi-level meta-modeling approach with a self-described, model intrinsic validation mechanism to automatically evaluate every potential change of its encompassing multi-level meta-model. Most of our ideas and solution techniques are the results of our hands-on experiences we have gathered by analyzing real industrial information models for several years. As a consequence, the current DMLA version (v2.1), is not a theory any longer; we have also implemented major parts of it as an executable demo, which is available for download at [1].

II. Related Work

Practical meta-model based applications show an increased interest for both state-of-the-art and innovative modeling technologies. OMG's Meta-Object Facility (MOF) is still the dominant meta-modeling approach used for industrial applications. There are two reasons for this: 1) MOF's four layer modeling architecture is easily comprehensible; 2) the Eclipse Modeling Framework (EMF) has been maturing during the recent years. However, new practical challenges, for example, full life-cycle management of model-based instances may require full-fledged multi-level meta-modeling. These alternative techniques advocate an explicit distinction between linguistic and ontological meta-models [2] [3] and can also differentiate between shallow and deep instantiation [4]. Deep instantiation is more prevalent because it can effectively reduce accidental complexity in the domain models. For example, in the case of potency notion [4], every class and attribute gets a potency assigned, which indicates the remaining levels the model elements can get through before reaching their finally instantiated status. In a sense, potencies are simple non-negative numbers, but, in their effect, they represent the current level of abstraction. They are decremented at each instantiation and when they reach 0 no further instantiation is allowed. Potency notion has been successfully implemented in the EMF based tool Melanee [5]. However, despite the many advantages potency notion provides, it also showcases some disadvantages due to its Orthogonal Classification Architecture (OCA) [6] heritage. Namely, OCA takes it for granted that all meta-model management facilities, i.e. introducing a new attribute to a clabject, are operational on each metalevel, thus, the instantiation step is oversimplified; it is merely controlled by a single integer value. Melanee tried to diversify this naïve counting by distinguishing the concepts of durability and mutability, but the constraining of the instantiation is still only described by integer values.

An important step forward to achieve expressive multi-level meta-modeling is the Lazy Initialization Multilayered Modeling framework (LIMM, [7]). This approach enables the definition of model elements at the meta level, at the application level, or one can simply declare them as data. In effect, LIMM associates flags to the model elements in order to control how they are to be used in the successive layers. An attached flag can take three values: it restricts, allows or enforces the initialization of a model element in subsequent layers. In a sense, the potency value evolves into a simple indicator of life-cycle status, which incorporates more than only checking if the value is positive when meta-level transitions of model elements are being evaluated.

DMLA aims to combine multi-level meta-modeling with dynamic model manipulation, which also necessitates non-trivial description of constraints on the instantiation steps through meta-levels. Also, DMLA incorporates a fully self-modeled operation language, which is currently implemented as a lightweight external DSL, called DMLAScript. Hence, although on the surface, DMLA looks very similar to XMF and XModeler [8], it clearly advocates a different architectural design. Firstly, DMLA's main focus lies on the multi-level meta-modeling of data; thus, it considers operations also as data (of a particular meta-level) that can be mixed in into the rest of the meta-model. Secondly, DMLA separates the DSL representation of DMLAScript from its internal meta-modeling formalism. Thirdly, DMLA's formal foundation is ASM-based, that is, it is executable by design [9]. Nevertheless, both approaches rely on the concept of a self-describing multi-level meta-model and the core idea of having a bootstrap. However, as long as XMF's meta-model utilizes higher order functions to process syntax and to provide a basic executable language which extends OCL syntax and semantics, in DMLA, the operation language is a mere facilitator to generate meta-model elements for the bootstrap. As a result of this setup, the operations are constrained only by the other elements of the bootstrap; thus, either they can be directly interpreted by the underlying ASM formalism or they must be translated and later executed by some ASM compatible run-time platform(s).

III. The Dynamic Multi-Layer Algebra

The Dynamic Multi-Layer Algebra (DMLA) is a multi-level modeling framework that consists of two major parts: (i) the Core, a formal definition of the modeling structure and its management functions; (ii) the Boostrap, an initial set of predefined modeling entities.

A. The Core

The definition of the Core is based on Abstract State Machines (ASM, [9]). In our case, the states of the state machine are snapshots of the models, while transitions represent modification actions between these states (e.g. deleting a node). The model is represented as a Labeled Directed Graph. Each element of the model such as nodes, edges or even attributes can have labels. These labels are used either to hold data (e.g. concrete literal value of an attribute) or to express relations (e.g. containment) between the elements. Because attributes may have complex structure, we represent them as hierarchical trees. Also, for the sake of simplicity, we will use a dual field notation for labelling of Name/Value pairs, that is, a label with the name N of the model element X is referred to as X_N. We defined the following labels: (i) X_{ID}: globally unique ID of model element; (ii) X_{Meta}: ID of the meta-model definition; (iii) X_{Values}: values of the model element; (iv) $X_{Attributes}$: ordered set of contained attributes.

Definition – The superuniverse |A| of a state A of the DMLA consists of the following universes: (i) U_{Bool} containing logical values {*true/false*}; (ii) U_{Number} containing rational numbers and a special symbol ∞ representing infinity; (iii) U_{String} containing character sequences of finite length; (iv) U_{ID} containing all possible entity IDs; (v) $U_{\underline{Basic}}$ containing elements from $\{U_{Bool} \cup U_{Number} \cup U_{String} \cup U_{ID}\}$. Additionally, all universes also 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_{ID}: U_{ID}, (ii) X_{Meta}: U_{ID}, (iii) X_{Values}: $U_{Basic}[]$ (contained primitive values), (vi) X_{Attrib}: $U_{ID}[]$ (reference to entities).

In ASMs, functions are used to rule how one can change the states. In DMLA, we rely on shared and derived functions. The current attribute configuration of a model element is represented via shared functions. The values of these functions can be modified either by the algebra itself, or by the environment of the algebra (e.g. by the user). Derived functions represent calculations which cannot change the model; they are only used to obtain and to restructure existing information. The vocabulary ∑ of DMLA is assumed to contain the following characteristic functions: (i) Meta(U_{ID}): U_{ID}, (ii) Attrib(U_{ID}, U_{Number}): U_{ID}, (iii) Value(U_{ID}, U_{Number}): U_{Basic}. The functions are used to access the values stored in the corresponding labels. These functions are not only able to query the requested information, but they can also update it. For example, one can update the meta definition of an entity by simply assigning a value to the Meta function (although the new relation may be invalid based on the instantiation rules). Moreover, there are two other derived functions: (i) Contains(U_{ID}, U_{ID}): U_{Bool} and (ii) DeriveFrom(U_{ID}, U_{ID}): U_{Bool}, which check containment and instantiation (transitive) relations, respectively.

B. The Bootstrap

In a nutshell, the Core is the formalism, while the Bootstrap is the practical foundation for DMLA. The Bootstrap is an initial set of modeling constructs and built-in model elements (e.g. built-in types) which are necessary in order to adapt the abstract modeling structure to practical applications. The main idea behind separating the Core and the Bootstrap is to improve flexibility, but also to keep the formal definition: the algebraic part is relatively fixed and structurally self-contained; its purpose is to isolate itself from the certain particularities of the various bootstraps. This design

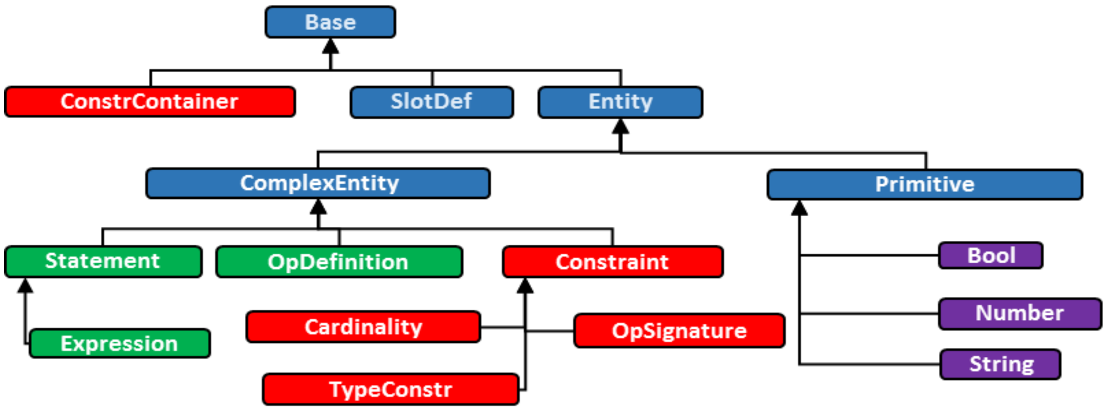

Fig. 1 – Main elements of the Bootstrap

makes it possible to replace the Bootstrap; hence, one can not only customize the basic modeling entities, but one can also re-define even the semantics of what valid instantiation means. A particular bootstrap seeds the meta-modeling facilities of generic DMLA formalism, thus, one may create a Bootstrap for simulating potency notion, another one for LIMM and a third one for power type behavior, etc.

The main elements of the current bootstrap (**Fig. 1**) can be categorized into four groups: (i) basic entities (blue boxes), (ii) built-in types (purple boxes) representing the primitive types available in DMLA, (iii) entities used in facilitating the introduction of operations in DMLA (green boxes), and (iv) validation related entities (red boxes).

1) Basic entities

Basic entities are the enablers of multi-level meta-modeling in DMLA. They create the root of the meta hierarchy all other modelled entities rely on.

The *Base* entity is at the very top of the hierarchy, thus all other entities are instantiated from it (directly or indirectly). *Base* defines that entities can have slots (defined by *SlotDefs*) and *ConstraintContainers*. Slots represent substitutable properties, which are syntactically similar to class members in OO languages. *ConstraintContainers* (and the contained *Constraints*) are used to customize the instantiation validation formulae. Moreover, *Base* has two other slots, reserved for validation of those formulae, which enforce the basic mechanisms of instantiation validation for multi-level modeling as explained later.

The *SlotDef* entity is a direct instantiation of *Base*. It is used to define slots. Slots can contain *ConstraintContainers*, which grants them the capability to attach constraints to the containment relations defined by the slot. Moreover, *SlotDef* overrides the validation slots inherited from *Base*.

The *Entity* entity is another direct instance of *Base*. *Entity* is used as the common meta of all primitive and user-defined types. *Entity* has two instances: *Primitive* (for primitive types) and *ComplexEntity* (for custom types). All domain relevant entities further instantiate *ComplexEntity*.

2) Built-in Types

The core entities needed to represent the universes of ASM in the bootstrap are: *Bool*, *Number* and *String*. All these types refer to sets of values in the corresponding universe. For example, the entity *Bool* has been created so that it could be used to represent *Boolean* type values within the model. Built-in types are relied on when a slot is filled by a concrete value and that value is not a reference to another model entity, but a primitive, atomic value. All built-in types are instances of Primitive.

3) Operations

All these entities representing the grammar of the operation language are defined in the AST subpart of the bootstrap under *ComplexEntity*. Moreover, there are also some extra-grammar entities defined here that deal with ASM execution semantics of those operations by specifying for example the invocation mechanism and the handling of return values and variables. This aspect of DMLA is not discussed in this paper in detail.

4) Validation

In DMLA, the validation logic relies on the selection of two type specific formulae based on the meta-hierarchy of the element to be validated. These two types are referred to as alpha and beta. The *Base* entity contains the default alpha and beta formulae, which can be constrained by the instances via their own specialized definition of valid instantiation, provided that does not contradict the standard validation rules imposed by *Base*. The validation mechanism is detailed later.

IV. Validation and Operations

In DMLA, if a model entity claims another entity as its meta the framework automatically validates if there is indeed a valid instantiation between the two. In DMLA 1.0 [11] instantiation was simply validated by a fixed set of general usage formulae, but in DMLA 2.0 [10], those formulae have been modularized by introducing a bootstrap compatible representation thereof. Since these formulae can directly influence the current semantics of the instantiation, model modification has got modularized and DMLA's instantiation has become effectively self-defined by model interpretation.

While implementing DMLA 2.0, we realized that by introducing operations into our framework we could describe validation formulae and their modular extensions, the so-called constraints, by attaching operations onto certain DMLA entities. Herewith, automatic model validation became a core feature. By further experimenting with this feature, we were able to establish a very compact though flexible validation system.

When we introduced operations into DMLA, the first technical issue was related to their representation. It was clear that operations must be described by DMLA modeling elements only, similar to other modeling entities in the Bootstrap. We have decided to base our representation on Abstract Syntax Tree (AST), where operations consist of entities representing their roles in the grammar such as expressions or statements. For example, the conditional statement ("if") has three child attributes: a condition (expression), a true branch (statement), and an optional false branch (statement). Since all of these subparts are DMLA entities, validation rules are applied to them similarly to other entities, which created the self-describing facility of DMLA.

The second technical issue considered how and when the operations must be executed. DMLA's practicality agenda aims at a virtual machine (DMLA VM) implementation similar to Java VM in order to interpret and execute operations. We have not reached this goal yet, but we have already defined and implemented an engine capable of parsing AST related DMLA entities and of producing executable Java code from the model. Currently, the generated code can be automatically integrated into the program, which is running the Core of DMLA's ASM implementation. Since the validation logic of the bootstrap is in its entirety described in DMLA operations, its semantics can be completely and consistently updated by simply changing the model.

The last technical issue focused on the effective manipulation of the operations within DMLA. Since DMLA 2.0's formal entity syntax is 4-tuple based [10], operations must be specified accordingly. Nevertheless, we realized that it would become cumbersome to produce realistic models by only relying on the 4-tuple representation. Even simple statements and operation calls may require dozens of entities that refer to each other in a complex entanglement. Hence, we implemented a simple XText-based DSL language with concrete syntax for DMLA, the so-called DMLAScript. The language design has borrowed syntax ideas mainly from Java, but the repertoire of language constructs is limited by the needs of DMLA. Although DMLAScript looks like being part of DMLA, it is not: DMLAscript is pure syntactic sugar above DMLA's 4-tuple representation. Nevertheless, by being able to specify validation logic in DMLAScript, our productivity increased enormously. Currently, DMLAScript descriptions are parsed into 4-tuples, which are then input to DMLA ASM's standard Java code generator. In other words, code generator (semantics) and language parsing (syntax) work independently, but hand-in-hand in the current toolchain implementation.

A. Flexible instantiation

In DMLA, the validation logic of instantiation relies on the selection of two type specific formulae based on the hierarchy of the element to be validated. We refer to these two types of formulae as alpha and beta. The alpha type formulae have been designed to validate an entity against its instances, by simply checking if the instantiation relation can be verified between the two entities (meta and instance). During validation, the framework iterates over the entities of the model, and invokes the alpha type validation on every entity and its meta entity. In contrast, the beta type formulae are in context checks: they are used when an entity has to be validated against multiple related entities, typically the attributes of an entity. For example, cardinality-like constraints shall be evaluated by beta formulae due to the underlying one-to-many relation thereof. Note that the exact validation rules provided by the alpha and beta formulas are Bootstrap-dependent, thus, it is easy to re-interpret the instantiation logic by only modifying these formulae.

Modular validation in DMLA works via compatible constraint extension. It means that entities can copy or extend the validation logic of their meta entity, which grants a very high level of flexibility without any loss of expressivity. The integration of operation ASTs into the Bootstrap allowed it to contain executable logic. Therefore, any model entities may provide their own specialized version(s) of valid instantiation, provided there is no contradiction with the standard validation rules imposed by their meta type (meta formulae are automatically validated by the framework).In parallel to validation rules, constraint specification was also modularized in order to avoid repeated definitions by introducing a generic Constraint entity. Constraints describe reusable validation logic that can be attached to any entity. It is important though to mention that the validation of constraints is special because it is not enough to validate the (Constraint) entity itself, but also the entity the constraint is referring to. For example, a range checker constraint added to a slot describing a number attribute must validate the value of the attribute, not the constraint (definition) itself. This is why we added special formulae to Constraints: the constraint-alpha and constraint-beta aimed to validate the entity containing the constraint. However, constraints are also special due to their life cycle. Thus, in order to achieve self-describing multi-level validation we needed constraints which are able to govern their own (customized) life-cycle. E.g. a constraint can decide if its re-instantiation is valid, or not. This feature is encoded in two other operations (lifecycle alpha and beta). This feature is similar to a self-managed, customizable potency notion.

In summary, the validation of the Bootstrap is based on three pairs of formulae: 1) the alpha and the beta type validation formulae, which are applied to every entity of the Bootstrap; 2) the ConstraintAlpha and the ConstraintBeta formulae, which are extensions of the container entity's alpha and beta formulae; 3) the Constraint-LifeCycleAlpha and the ConstraintLifeCycleBeta formulae, which manage and validate the DMLA correct life-cycle of Constraint instances

```
operation Bool ID::PersonAlpha(ID instance) {
   //Access the value of the slot containing the name of the person
   ID fullName = call $GetRelevantAttributeValue(instance, $Person.FullName);
   If (fullName==null)
           return true;        // If no name is specified yet => valid
   Object[] firstNames=call $GetRelevantAttributeValues(fullName, $ComplexName.FirstName);

   //Access the first name values contained by the ComplexName
   if(firstNames ==null || size(firstNames)<2) return true; //not specified/has less than 2 first names

   //Ensure the first names do not match
   return index<Object>( firstNames, 0) != index<Object>( firstNames, 1);
}
```

Code 1 – The PersonAlpha operation

B. Validation example

In order to showcase how DMLA and its validation framework in practice, let us take the following example: we are creating the meta model of a person that has a name which consists of one or two first names and a single last name. There is also a constraint imposed on every person, namely that a person cannot have matching first names: e.g. “Bob Smith” and “Bob Rob Smith” are valid, but “Bob Bob Smith” is not. In order to turn this specification into DMLA entities, only a few steps are required. As usual in modeling, one has to create customized composite entities to represent these concepts. In the current Bootstrap, the entities are instances of ComplexEntity, which enables having an arbitrary amount of attribute slots within.

First, the ComplexName entity is defined to encapsulate the parts of a person name. It has two slots: one for the first name(s) with [1..2] cardinality, and another one for the last name with [1..1] cardinality; both being of type String. Secondly, the Person entity is defined, which contains a single slot with [1..1] cardinality, and is an instance of ComplexName. Now, the structure having been set up, validation follows.

As explained earlier, validation in DMLA is based on two operation types: the alpha and the beta type formulae. The core validation logic is defined in the alpha and beta formulae defined inside entity Base, which is the root meta of the Bootstrap (all other entities are direct or indirect instances of Base). Since in the example the validation logic can be evaluated on a single Person instance without considering any of its context (matching name constraint is contained within Person), one only has to override the alpha formula. In order to do so, an additional operation must be attached to Person describing the customized alpha validation logic. The alpha formula (Code 1) is a simple operation, it accepts two IDs, an instance ID and a meta ID, and it returns true if the meta-instance relation is valid in this regard. The logic of the operation is written in DMLAScript. When it comes to execution, the validation logic is first translated into 4-tuples, that is, into DMLA native entities, and then the tuples are compiled into Java code that represents the ASM compliant behavior in JVM carrying out the validation.

V. Conclusion and future work

DMLA went through various stages during the last few years, from the pure theoretical foundation to the implementation of a highly modular and practical multi-level meta-modeling framework of industrial focus. Although the current implementation is still pre-alpha, model validation has become flexible and modular enough due to DMLA’s powerful self-describing formalism. By now, we have started producing realistic bootstraps to cover industry induced use cases. Currently, our research goals aim at introducing executable semantics via operations, polishing 4-tuple generation, and streamlining the Java based execution engine for DMLA’s ASM virtual machine.

References

[1] "DMLA Website," [Online]. https://www.aut.bme.hu/Pages/Research/VMTS/DMLA. [Accessed 23 04 2017].

[2] J. D. Lara, E. Guerra and J. S. Cuadrado, "When and How to Use Multilevel Modelling," Journal ACM Transactions on Software Engineering and Methodology, vol. 24, no. 3, 2014.

[3] M. Gutheil, K. Bastian and C. Atkinson, A systematic approach to connectors in a Multi-level Modeling Environment, vol. 5301, Lecture Notes in Computer Science, 2008, pp. 843-857.

[4] C. Atkinson and T. Kühne, “The Essence of Multilevel Metamodeling,” The Unified Modeling Language. Modeling Languages, Concepts, and Tools, vol. 2185, pp. 19-33, 2001.

[5] C. Atkinson and R. Gerbig, Melanie: Multi-level modeling and ontology engineering environment, New York, USA: ACM, 2012, pp. 7:1 - 7:2.

[6] C. Atkinson, M. Gutheil and B. Kennel, "A Flexible Infrastructure for Multilevel Language Engineering," IEEE Transactions on Software Engineering , vol. 35, no. 6, pp. 742 - 755, 2009.

[7] F. Raque Golra and F. Dagnat., "The Lazy Initialization Multilayered Modeling Framework," in ICSE 2011 : 33rd International Conference on Software Engineering, Honolulu, 2011.

[8] T. Clark, C. G.-P. and B. Henderson-Sellers, "A Foundation for Multi-Level Modelling," Proceedings of the Workshop on Multi-Level Modelling at ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, vol. 1286, pp. 43-52, 2014.

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

[10] D. Urbán, Z. Theisz and G. Mezei, "Formalism for Static Aspects of Dynamic Metamodeling," Periodica Polytechnica Electrical Engineering and Computer Science, vol. 61, no. 1, pp. 34-47, 2017.

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