Self-describing Operations for Multi-level Meta-modeling

Dániel Urbán[1], Zoltán Theisz[2] and Gergely Mezei[1]
[1]*Budapest University of Technology and Economics, Budapest, Hungary*
[2]*evopro systems engineering Ltd., Hauszmann Alajos str. 2, Budapest, Hungary*

Keywords: Meta-modeling, Multi-level Modeling, Operation Language, Self-validation.

Abstract: Any meta-modeling discipline, similar to programming languages, will, sooner or later, feel the need for some operational language in order to express constraints for model validation and/or action semantics for executable modeling. Multi-level meta-modeling is no exception in this regard. However, it does provide the facility to formalize the operation language within the meta-modeling framework, thus the language syntax and semantics fits perfectly well the intended need of the modeling environment. Moreover, if the modeling framework is flexible enough in the principles, the model validation can be specified and also applied to the operation language as well. In this paper, we shortly introduce such a modeling formalism, DMLA, and then describe in relative detail the design and the current realization of its operation language, DMLAScript, which enables the multi-level meta-modeling framework to effectively tackle realistic domain models.

1 INTRODUCTION

Multi-level meta-modeling is enjoying a renaissance after more than ten years of simmering. The paradigm has been rediscovered in recent years due to various factors: i) multi-level meta-modeling techniques for describing data models have been evolving a lot since the first introduction of potency notion, ii) research quality tool support is widely available from universities and research institutes, iii) mainstream meta-modeling is facing increasingly challenging problems as industry has started adopting research solutions and tried to use them in real problem settings. Indeed, contemporary meta-modeling is a mature technology, that is, there exist no known theoretical limits of its applicability to whatever shape the particular application domains might come up with and whatever complexity they present. The only real practical headache nowadays is connected to the forecasting of the adaptation and later maintenance costs the candidate solutions will require. In effect, it is only due to the additional cost of accidental complexity, which derives from the selection of a particular modeling technology, provided the details and the scope of the problem have been thoroughly investigated. Obviously, multi-level meta-modeling is not a silver bullet either; nevertheless, this paradigm aims to minimize that accidental complexity by taking advantage of an unlimited number of meta-levels in order to properly allocate the right abstraction detail to each of them. The rest is modeling as usual: instantiation plays exactly the same role as in the case of state-of-the-art modeling methods such as UML or EMF Ecore. However, there is though a significant difference: the leveling is not prescribed by a methodology, but it is only encouraged and directly influenced by the aimed solution(s) of the domain.

Although multi-level meta-modeling is a very promising technique, it does have its own problems and limitations. Currently, the most serious of those issues are: 1) the general lack of customizable syntax and precise semantics of operations acting on multi-level models, 2) a self-contained and self-describing multi-level meta-modeling framework that can bootstrap without explicitly referring to any other legacy modeling techniques, and 3) a semantically correct validation framework for multi-level models that is formally anchored in precise definition of the underlying instantiation process. Our approach, Dynamic Multi-Layer Algebra (DMLA), aims to address these problem areas by a formal algebraic foundation based on a novel precise conceptualization of the instantiation process and a related flexible tuple representation of multi-level model entities, all within a totally self-contained bootstrapping mechanism. A particularly important bootstrap of the methodology is the self-describing validation

Urbán, D., Theisz, Z. and Mezei, G.
Self-describing Operations for Multi-level Meta-modeling.
DOI: 10.5220/0006656105190527
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 519-527
ISBN: 978-989-758-283-7

framework that also incorporates a full-fledged operation language, which is entirely specified by AST entities in DMLA. The language grammar is used for formalizing the validation rules of the bootstrap, including also those of such rules that are to be applied to the AST entities per se. Without self-circularity, this revolutionary validation approach works flawlessly and facilitates a programming like creation of multi-level models. Having the details of the validation framework already published in (Urbán, et al., 2017b), this paper concentrates only on the details of the operation part of the bootstrap, by describing the main design ideas of syntax and semantics for the operation language and its direct application within DMLA's flexible auto-validation mechanism.

2 RELATED WORK

Meta-modeling usually focuses on the systematic modeling of data of a particular application domain. Although the modeling approaches are fully aware of the need of operations within the data structure, MOF and EMF Ecore standard modeling solutions only allow signature modeling in EClass. Various research techniques intended to rectify this situation, the most notable of them being Kermeta (Muller, et al., 2005) (KerMeta, 2017) and recently the GEMOC (Combemale, et al., 2013a) (Combemale, et al., 2013b) framework. Although their solution looks promising, there are still some limitations remaining: for example in the case of Kermeta, there is a need for a complex model promotion due to restricted numbers of ECore's meta-modeling levels. In the case of the GEMOC approach the beauty is fading by the Xtend/Java semantics woven into the Ecore meta-models in order to turn them executable.

Multi-level meta-modeling promises to simplify many of the issues that originate from accidental complexity. For example, its usage becomes very instructive when solving the discordance between the 4-level nature of eMOF and Kermeta's quest for a language meta-model in Ecore. The effective handling of accidental complexity relies on the explicit differentiation between linguistic and ontological meta-models (Lara, et al., 2014) (Gutheil, et al., 2008) and the facility of deep or strict instantiation (Atkinson & Kühne, 2001). For example, potency notion (Atkinson & Kühne, 2001) assigns a potency value to every class and attribute, which clearly indicates the remaining levels they can get through before getting fully instantiated. Melanee (Atkinson & Gerbig, 2012) has further refined potency notion by distinguishing the concepts of durability and mutability. However, in essence, the basic ideas of Orthogonal Classification Architecture (OCA) (Atkinson, et al., 2009) remains in place, thus, it is taken for granted that all meta-model management facilities are fully and non-restrictively operational on each meta-level. Hence, the instantiation step is heavily simplified; it is controlled by simple integer values and no sophisticated constraint handling can be carried out.

More versatile multi-level meta-modeling approaches are metaDepth (Lara & Guerra, 2010) and XModeler (Clark, et al., 2015). Both include an operational language to extend multi-level modeling with operations. metaDepth uses EOL, a language of the Epsilon family, for constraint and action specification. Although it nicely complements metaDepth, it also showcases the same problem already mentioned in the case of GEMOC regarding its reliance to an external language. XModeler has a much more advanced solution for operation integration: XMF's meta-model facilitates higher-order functions in order to process syntax and to provide a basic executable language (XOCL), which relies on OCL syntax and extends it semantics. However, XOCL is fixed in its syntax and semantics; thus, it is not easy to be extended by new features. Also, being part of the XMF (Clark, et al., 2015) modeling framework, every domain model must express its semantics in XOCL. On the contrary, in the approach presented in this paper, the operation language mainly serves as a facilitator to efficiently generate meta-model elements. As a result of this design, the operations are defined and constrained only by the entities found in the bootstrap of the particular application domains.

3 THE DMLA APPROACH

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 pre-defined modeling entities. In DMLA, the model is represented as a Labeled Directed Graph, where all model elements have four labels: a unique ID of the element, a reference to its meta, a list of concrete values, and a list of contained attributes. Besides the 4-tuples representing the model entities, there exist also functions to manipulate the model graph, for example to create new model entities. These definitions (Urbán, et al., 2017a) form the Core of DMLA, which

is specified over an Abstract State Machine (ASM) (Boerger & Stark, 2003). Thus, in DMLA, the states of the state machine are snapshots of the dynamically evolving models, while transitions (e.g. deleting a node) represent modification actions between those states.

The Bootstrap is an initial set of modeling constructs and built-in model elements (e.g. built-in primitive types) which are needed to adapt DMLA's 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 approach formal. This way, the Bootstrap is becomes swappable, thus even the semantics of valid instantiation can be re-defined. Namely, each particular bootstrap seeds the meta-modeling facilities of the generic DMLA formalism.

Validation in DMLA is simple in theory: whenever a model entity claims another entity to be its meta, the framework automatically validates if there is indeed a valid instantiation between the two. The validation formulae can be modularized by introducing them directly into the Bootstrap. Since these formulae directly influence the actual semantics of instantiation, every model validation gets modularized and DMLA's instantiation becomes effectively self-defined by the model per se. However, in practice, the key success factor to achieve this self-validated, self-describing behavior relied on the consistent introduction of operations. In DMLA, operations are modeled internally within the bootstrap by a self-contained operation language.

3.1 The Bootstrap

The ASM functions define the basic structure of the algebra and they also allow to query and change the model. However, relying only on these pure mathematical constructs, it would be rather hard to use the algebra in any practical modeling scenarios. Hence, the concept of the Bootstrap was introduced, which is a flexible and swappable layer for defining any needed modeling entities. For example, the modeling entities of the current bootstrap (Figure 1) can be categorized into four groups: (i) basic types (blue boxes) providing a basic structure for multi-level meta-modeling, (ii) built-in types (purple boxes) representing the primitive types available in DMLA, (iii) entities facilitating the introduction of operations into DMLA (green boxes), and (iv) validation related entities (red boxes).

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 exact definitions are available at (DMLA Website, 2017). 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:

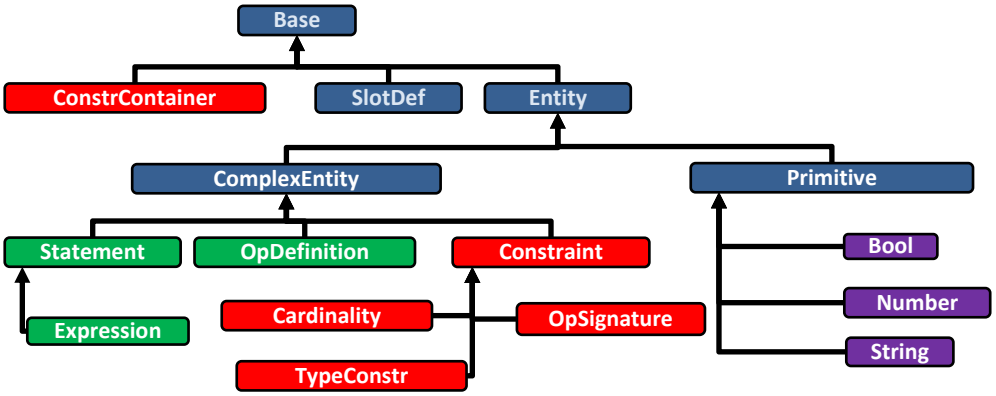

Figure 1: The elements of the Bootstrap.

Primitive (for primitive types) and *ComplexEntity* (for custom types). All domain relevant entities further instantiate *ComplexEntity*.

The built-in types represent the universes of ASM in the Bootstrap: *Bool*, *Number* and *String*. All these types refer to sets of values in the corresponding universes. 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 with a concrete value and that value is not a reference to another model entity, but it is a primitive, atomic value. All built-in types are instances of *Primitive*.

Operations are the primary focus of this paper. All these entities, representing the grammar of the operation language, are defined in the AST subpart of the bootstrap below *ComplexEntity*. Moreover, there are also some extra-grammar entities here, which deal with ASM execution semantics of the operations by specifying for example the invocation mechanism and the handling of return values and variables. More details are described in Section 4.

In DMLA, the validation logic relies on the selection of two type specific formulae (referred to as alpha and beta) based on the meta-hierarchy of the element to be validated. The alpha type formulae is constructed to validate an entity against one of its instances, by checking whether the instantiation relation between the two elements can be verified. In contrast, the beta type formulae are in-context checks: they are mainly needed in case an entity has to be validated against multiple related entities, e.g. in case of cardinality. The *Base* entity contains the default alpha and beta formulae, which can be customized by the instances, provided that they do not contradict the standard validation rules imposed by the *Base*. The validation aspect of the Bootstrap has been discussed in detail in (Urbán, et al., 2017b).

4 OPERATIONS

For any practical modeling technique, having a consistent and powerful operation language is more than a desired feature. Such a feature enables models to be truly self-contained by incorporating the semantics and the dynamic nature of the models as their integral part, instead of relying on an externally provided substitute. However, in most current modeling approaches, this is achieved by importing an external language into the modeling universe, thus they deal with black box semantics. While it is an improvement compared to inflexible static solutions, it still may not be enough: (i) since the language is an external asset, so the self-describing nature of the technique is violated; (ii) the main concepts of the technique - such as instantiation or validation – are not available in the imported language in a genuine way, or they may even have a different interpretation unbeknownst to the modeler.

While the desire to include an operation language within a modeling technique is well understood, another important aspect is if and how such languages can be integrated. In this section, we present our process of how intrinsically modeling a full-fledged operation language and augmenting the original DMLA framework with it, all within the frame of the original DMLA concept domain.

As a result, our approach has clearly separated the various concerns and issues of such a language, and tackled them head-on one by one: (1) Evaluated the different possibilities of modeling the AST (NB: DMLA's bootstrap enables such design by itself), (2) Chose the abstraction level of the language, (3) Created the specification and the model entities of the AST, (4) Evaluated the need of a DSL, (5) Integrated the language into the framework, (6) Analysed the execution methods of the modeled code.

4.1 Modeling an AST

Since most modeling frameworks claim to have a universal foundation to describe models – and this is even more prevalent in multi-level modeling – technically the issue is not be a problem at all. Essentially, code is only data at another meta-level, that is, instances of an AST meta-model. Since the static aspects of DMLA and the selected bootstrap are well formed (Urbán, et al., 2017a), it is relatively simple to create model elements for a programming language as nodes of an AST.

4.2 Abstraction Level

It is an important challenge to choose the abstraction level of the language adequately. In DMLA, accessing and manipulating entities of the model can be achieved at two levels: (i) at the level of the tuples, or (ii) at the level of the bootstrap.

Since everything in DMLA is a tuple, a language can be easily created that operates on the tuples of the model. It also means that the type system and other semantic concepts introduced in the bootstrap cannot become part of the language. Hence, this solution results in a low-level, though universal solution, which is independent of the bootstrap and not automatically reflected in the semantics of the language.

Another solution is to operate on the bootstrap level. This means that the type system and all concepts of the bootstrap are tightly integrated into the language itself, for example, one could use the constraints of the bootstrap for declaring variables.. This solution results in a high-level, yet rather complex language.

We have selected the first option. It may look less elegant, but it results in an efficient low-level language, operating directly over the tuples of the underlying DMLA mechanism. Also, for reasons of practicality, we chose an imperative approach.

4.3 Modeling the AST

We collected some requirements to be imposed: (i) the language shall be bootstrap-agnostic, thus the type system shall mirror the ASM prescribed one, i.e. primitives, IDs and Any. Multi-dimensional arrays of types shall be allowed as well. (ii) The built-in functions of the ASM are to be made available to enable operations with tuples. (iii) Usual programming language constructs, i.e. conditionals, loops and functions must to be supported.

In order to satisfy the above requirements, we implemented the following language constructs: (i) types: Any, ID, string, number and bool (and their multi-dimensional arrays); (ii) variables; (iii) sequences (block); (iv) conditionals (if); (v) loops (while, for, foreach); (vi) type check and cast (is, as); (vii) arithmetical and logical operators (+ - * / || &&); (viii) index operator; (ix) functions and function calls – preferably with the concept of "this", a dedicated parameter; (x) return.

With this list of constructs established, modeling of the AST could be properly carried out. The constructs are defined as bootstrap entities. For example, the If construct is defined as follows: (i) the construct is the instance of *Statement, (ii)* it has a *Condition* slot with *Expression* type and [1..1] cardinality, (iii) it has a *Then* slot with *Statement* type and [1..1] cardinality, (iv) it has an *Else* slot with a *Statement* type and [0..1] cardinality.

All constructs could be modeled in similar ways in the bootstrap. In the end, we implemented an operation language that now exists as a static specification within the model. Using this language, programs can be created to indirectly manipulate tuple representations, and any code written in this language will be stored in the bootstrap.

4.4 DSL for Operations

4.4.1 Evaluating the Need for a DSL

At this point, model representation of "code" is provided in the Bootstrap. Integration of the language into the framework (e.g. migrating the validation logic of the Bootstrap) has no obstacles in its way. But the task looks difficult, to say the least.

It is important to emphasize that the AST representation of code is effusive. Comparing the textual representation of code snippets of any programming language to its equivalent AST representation shows clear gap in terseness.

Let us take the below code example that results in 9 tightly connected tuples when written in the operation language of DMLA as follows:

```
if(true) return 1;
```

This factor of flatulence indicates that writing real "code" would be nearly impossible for any modeller. Constructing nodes of an AST is a very cumbersome low-level method of "coding": in a sense, when 4-tuples are being produced in DMLA, it would look like programming in an assembly language or even directly producing byte-code. This is why the question of creating a DSL is relevant in this context: it helps turn any theoretical solutions to practical implementation. Since our goal was not only to define the language, but we also wanted to tightly integrate it into the already existing modelling framework. To achieve this goal, we had to produce real code, thus we decided to create a DSL for the abstract language syntax: DMLAScript was born.

DMLAScript had to be a practically applicable operation language over DMLA. Therefore, it must be able to effective produce 4-tuple entities. Thus, the most important aspect of DMLAScript's language design is maximal efficiency of tuple production. However, there are other design constraints imposed on it so that DMLAScript could become genuinely part of the DMLA framework: 1) the structure of entities shall be expressed as "data definitions", 2) operational logic shall be programmed as "code", and finally 3) validation logic of the bootstrap must be given by the DSL.

In order to better appreciate the task of an optimal operation language design, it is important to re-emphasize that DMLAScript is not a necessity out of DMLA per se, nonetheless without it we cannot imagine that any practical modelling scenarios can be tackled adeptly. Hence, DMLAScript is effectively a

facilitator of efficient entity modelling, which creates the illusion of a programming language over DMLA.

With DMLAScript, we simply provided a textual DSL over the constructs of the operation language already defined in the Bootstrap. We borrowed most of the syntax ideas from Java. We implemented the DSL and its mapping onto Bootstrap entities (tuples) within Xtext. With DMLAScript, the previous "code" example is as simple as it was written there.

This style of coding looks much more natural and it is easier to use for the modeler than to create the corresponding tuples manually and hook them together along their IDs. Since the syntax of DMLAScript follows modern imperative languages, it is both easy to use for programming and to get parsed for execution. Our current DSL tool relies on transforming the above code snippet into instances of AST nodes, in the end producing the 9 tuples.

4.4.2 Syntax of DMLAScript

Before proceeding to any further in the process of language design, we will show a few simple code examples to introduce the syntax of DMLAScript.

```
Entity1 : Entity2 {
  slot E1Slot : Entity2.E2Slot = 1;
  AnotherSlot;
}
```

In the first example, we declare an entity with the ID *Entity1*. The meta of *Entity1* is *Entity2*. Between the braces, we have the attributes of *Entity1*, namely, there is a slot with the ID *E1Slot*. *E1Slot* is defined inline, nested in *Entity1*. The meta of *E1Slot* is *Entity2.E2Slot*. The constant value 1 is assigned to the value of *E1Slot*. *Entity1* also has a second slot with the ID *AnotherSlot*. *AnotherSlot* is not defined inline: it must already be defined somewhere else in the code, and it is only referenced here to be included as attribute of *Entity1*.

It is important to keep in mind that in DMLAScript, the indexing feature of Xtext is heavily used. Entities in the code have fully qualified names using their parent packages and entities. It means that the produced ID of the tuple generated from the definition of *E1Slot* will look like "*Entity1.E1Slot*". This is important to keep in mind because there are a lot of apparently colliding IDs in the code of the Bootstrap (DMLA Website, 2017), while in reality IDs are affected by the index and the imports, and will be fully unfolded in the tuple generation step.

```
Entity1 : Entity2 {
  @ConstrContainer1
  @ConstrContainer2: MContainer1 =
    $SomeConstraint1;
  @ConstrContainer3: MContainer2 =
    SomeConstraint2: MConstraint {
        slot ConstraintSlot:
       MConstraint.Slot = true;
    };
  slot SomeSlot: Entity2.E2Slot =
    $SomeEntity;
}
```

In the second example, not only an entity is declared with slots, but there are also constraint containers defined on the slot. *Entity1* is the instance of *Entity2*, and has a single slot, *SomeSlot*. *SomeSlot* is the instance of *Entity2.E2Slot*, and its value is a reference to the entity *SomeEntity*. *SomeSlot* has three attributes, all of them indicated above the slot: *ConstrContainer1*, *ConstrContainer2* and *ConstrContainer3*. *ConstrContainer1* is an already defined entity. *ConstrContainer2* is an entity, which is defined inline, its meta is *MContainer1*, and its value is set to the entity *SomeConstraint1*. *ConstrContainer3* is also defined inline, its meta is *MContainer2*, and its value is set to a constraint entity defined inline. This constraint entity is called *SomeConstraint2*, its meta is *MConstraint*. It has one attribute, namely the slot *ConstraintSlot*, which is the instance of *MConstraint.Slot*, and its value is set to true.

This example shows how the basic entities of the bootstrap are used in the language. Most entities defined in the DSL are instances of *ComplexEntity*; they contain *SlotDef* instances as attributes; *SlotDef* instances have values; and *SlotDef* instances contain *ConstraintContainer* instances as attributes; finally, *ConstraintContainer* instances contain *Constraint* instances as values.

```
operation void Method1();
operation Bool Method2(Number p1);
operation String ID::Method3(Bool[] p1);
```

In the third example, we have three operation signatures. The operation *Method1* has a void return type, and no parameters. *Method2* has a *Bool* return type (this refers to the primitive entity *Bool*) and one parameter called *p1* with the type *Number* (primitive). *Method3* has a *String* return type (primitive), has a *context* with the type ID – which will be the type of "*this*" inside the operation - and also has a single parameter called *p1* with the type of *one dimensional Bool array*.

```
operation String Example(Number p1){
  if(p1 < 0) return "negative";
  while(p1 > 0) --p1;
  call $SomeMethod();
  return "something";
}
```

In the fourth example, we define an operation with its body. The operation has the ID *Example*, has a *String* return type, and has a single *Number* parameter called *p1*. If *p1* is less than 0, the operation returns the string "negative", otherwise, it decrements *p1* to 0 in a while loop. After that, it invokes another method with the ID *SomeMethod*, and then returns the string "something". Note that modifying operation parameters (such as decrementing *p1*) has no effect on the caller site, since parameters are all passed by value in DMLA.

```
operation void OpToContain() { }
EntityWithOperation : MetaEntity {
  slot MethodSlot : MetaEntity.Slot =
    $OpToContain;
}
```

In the final, fifth example, we define an operation with the ID *OpToContain*. Then, we define an entity called *EntityWithOperation*. This entity has a single slot called *MethodSlot*, which has its value set to *OpToContain*. As it can be seen, it is very straightforward to reference operations.

4.4.3 Design of DMLAScript

Let us describe now the generic design ideas behind DMLAScript in order to show how the language has coped with the original requirements.

Firstly, DMLAScript rebalances the very pointer like nature of the 4-tuple representation of related DMLA entities. Namely, instead of manually dealing with all the IDs, which weave the model entities together along the meta, the attribute, and even the value references, the model designer can simply rely on a local context around the entities when he carries out his modeling task. Also, when describing the structure of the entities, DMLAScript applies annotation-like constructs to attribute slots in order to simplify the specification of the constraints defined on the slots. These two features create the feeling of a high level entity definition language that enables efficient production of the corresponding 4-tuples.

Secondly, DMLAScript follows the imperative semantics of Java-like notations. We introduced a special syntax for direct referencing of entities, which enables us to connect seamlessly the entity definition part to the operational part of DMLAScript. Taken into account that operations are also represented as entities in DMLA, entity referencing works both ways: from entity definition to operation logic and back. Hence, the modeller should not be aware of those two aspects, DMLAScript looks like an integral language for multi-level meta-modeling, which also happens to be used to define itself in the bootstrap.

Thirdly, DMLAScript also provides a language to express DMLA's validation logic for multi-level meta-modelling. It is established by a pre-set navigation scheme of validation logic invocations that is seamlessly embedded into the basic entities of the bootstrap, via their alpha and beta slots. In effect, the bootstrap has been set up by DMLAScript in such a way that it can carry out its own validity check. The default validation behaviours can be constrained via instantiation, for example, cardinality logic and/or type validation logic can be added to entities via alpha and beta operations written in DMLAScript.

4.5 Integration of the Language

With the syntax and general design and its embedding being detailed, it is obvious that any operation logic can be produced and efficiently stored as tuples in DMLA models. All that thanks to DMLAScript and its parsing module. The next goal was to express the already existing (validation) semantics and dynamic logic with DMLAScript.

Our aim was to migrate the validation logic, which had been previously residing at the level of the Core and the ASM formalism. Moreover, we recognized that the new operation language would enable not only the migration of the validation formulae, but also the full modularization of the underlying evaluation logic. Since operations are handled as data in DMLA, data structures can also contain references to operations as the design required it. This feature allowed us to create truly self-contained entities in the bootstrap, which are not only containing their structure and data, but they can also prescribe their own custom validation logic at ease (Urbán, et al., 2017b).

4.6 Execution of DMLAScript

With DMLAScript genuinely integrated into DMLA's modeling fabric, the only open issue to be solved was how to tackle the dynamics of the language, that is, how to execute, for example, the validation logic represented in 4-tuples. Essentially, there are three ways to answer this challenge: (i) rely on an interpreter, (ii) generate executable code in another language, or (iii) generate a directly executable binary. The third option results in a rigid, platform dependent solution that is more complex than the first two, thus we dropped this idea. While running the language within an interpreter is a rather flexible solution, it though requires a well-established infrastructure, such as some kind of a virtual machine. On the contrary, generated executable code is again

quite a rigid solution, but, at least, it does not require complex a runtime framework.

Since the current technical solution is our first implementation drop, mostly we aimed to validate our assumptions in practice; hence we decided to provide a quick prototypical execution framework, by generating Java code from the AST instances stored in the DMLA models.

The framework itself has been programmed in Java and it consists of a model repository, which contains the tuples of the model and a symbol table for built-in and custom operations. The runtime can generate code from the tuples, compile it, and load the compiled code dynamically. It is important to note that the generated code currently takes into account only the Core and the Bootstrap, so it is independent of DMLAScript syntax and its Xtext module. Hence, the syntax of DMLAScript is currently handled by an external tool, and thus should be thought of only as syntactic sugar over DMLA's operation language.

5 CONCLUSIONS

Model-driven development has become a feasible option to create and maintain complex systems. However, static modeling solutions are not always sufficient any longer in the modern era of industrial applications. Thus, the demand for dynamic modeling techniques became a natural tendency in many fields. Although extending static models with external operation languages and execution frameworks can sometimes meet the requirements, it would be more elegant, and also due to its design more verifiable and customizable, to build the mechanism of operations directly into the modeling framework. From the theoretical perspective, representing operations as modeled entities has been already researched and well understood in detail, but a seamless, self-describing and non-circular integration of these ideas into a fully functional modeling framework has not been implemented up till now.

Our approach, the Dynamic Multi-Layer Algebra (DMLA) provides such a practical solution for the challenge. DMLA features a highly customizable, multi-layer modeling and validation structure that allowed us to build a fully modeled operation language into it. In general, this language enables programming with operations over modeling entities, but its real strength only gets to the surface when it comes to specifying the validation formulae of multi-level instantiation in particular. That ability results in a fully self-describing, self-validation modeling framework, which can validate even its own language definition. Moreover, since the operation language can be part of any modeled domain, it may be further extended or customized.

Currently, the DMLA environment provides as default a high level, Java-like operation language, DMLAScript, which is suitable to keep the specification of the operation logic within manageable size. In the future, we are investigating ways to speed up the current validation process by parallel execution. We are evaluating the possibilities for optimizing the core operations of the validation by parallelizing them with GPU support, which could strike a balance between the flexibility of the bootstrap and the performance of its execution.

REFERENCES

Atkinson, C. & Gerbig, R., 2012. Melanie: Multi-level modeling and ontology engineering environment. *Proceedings of the 2Nd International Master Class on Model-Driven Engineering: Modeling Wizards*, pp. 7:1 - 7:2.

Atkinson, C., Gutheil, M. & Kennel, B., 2009. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering*, 35(6), pp. 742 - 755.

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

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.

Clark, T., Sammut, P. & Willans, J., 2015. *Super-Languages: Developing Languages and Applications with XMF*.

Combemale, B. et al., 2013b. Reifying Concurrency for Executable Metamodeling. *Software Language Engineering. SLE 2013. Lecture Notes in Computer Science*, Volume 8225, pp. 184-203.

Combemale, B. et al., 2013a. Bridging the Chasm between Executable Metamodeling and Models of Computation. *Software Language Engineering. SLE 2012, Lecture Notes in Computer Science*, Volume 7745, pp. 184-203.

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

Gutheil, M., Bastian, K. & Atkinson, C., 2008. A systematic approach to connectors in a Multi-level Modeling Environment. *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, Volume 5301, pp. 843-857.

Kermeta Website [Online] http://diverse-project.github.io/k3/ [Accessed 23 04 2017].

Lara, J. d. & Guerra, E., 2010. Deep Meta-modelling with MetaDepth. *Objects, Models, Components, Patterns*, Volume 6141, pp. 1-20.

Lara, J. D., Guerra, E. & Cuadrado, J. S., 2014. When and How to Use Multilevel Modelling. *Journal ACM Transactions on Software Engineering and Methodology*, 24(3), pp. 12:1-12:46.

Muller, P.-A., Fleurey, F. & Jézéquel, J.-M., 2005. Weaving Executability into Object-Oriented Meta-Languages. *Lecture Notes in Computer Science*, Volume 3713, pp. 264 - 278.

Urbán, D., Theisz, Z. & Mezei, G., 2017a. Formalism for Static Aspects of Dynamic Metamodeling. *Periodica Polytechnica Electrical Engineering and Computer Science*, 61(1), pp. 34-47.

Urbán, D., Theisz, Z. & Mezei, G., 2017b. Validated Multi-Layer Meta-modeling via Intrinsically Modeled Operations. *4th International Workshop on Multi-Level Modelling, 2017, https://www.wi-inf.uni-duisburg-essen.de/MULTI2017/#program (Accepted, presented, publication in progress).*