# A bootstrap for self-describing, self-validating multi-layer metamodeling

Gergely Mezei Zoltán Theisz Dániel Urbán
Sándor Bácsi Ferenc A. Somogyi Dániel Palatinszky
*Department of Automation and Applied Informatics*
*Budapest University of Technology and Economics*
{gmezei, ztheisz, daniel.urban, sandor.bacsi, somogyi.ferenc, daniel.palatinszky}@aut.bme.hu

**Abstract.** In the age of Industry 4.0, the ability to be highly flexible and at the same time rigorously precise is the principal requirement to become part of the smart revolution. Therefore, although classical metamodeling approaches do offer a relatively higher standard of flexibility compared to state-of-the-art object-oriented paradigms, they still fail to support effective step-wise refinement required by the smart industry. Multi-level metamodeling aims to address this issue by increasing the number of modeling layers. By managing to create a self-describing and self-validating approach, one may end up providing a perfect solution to the challenges and also grant model validity in real. In recent years, our research group has created such an approach called the Dynamic Multi-Layer Algebra (DMLA). DMLA is a multi-layer metamodeling formalism with a sound mathematical background. In order to achieve the needed flexibility, we have designed a bootstrap of model entities that are both self-describing and self-validating by design. This paper introduces the elements of the bootstrap by mainly focusing on those metamodeling ideas which we have considered essential for any realistic multi-level metamodeling approaches of such kind.

**Keywords:** Metamodelling; validation; DMLA; multi-layer; multi-level; bootstrap

## 1 Introduction

Industry challenges of today can sometimes only be answered by beyond state-of-the art software development paradigms. The legacy approaches of model-based software engineering (MBSE) and in particular the ones of applied domain-specific modeling have often turned out not being either flexible of precise enough in practice; hence, new methods are to be sought for. As a contemporary trend in MBSE, two and/or four level metamodeling frameworks are being frequently extended to an arbitrary number of levels, which could automatically lead to higher flexibility of expressiveness, but sometimes only at the cost of losing modeling precision. Nevertheless, for a modeling approach to be rigorous enough is indeed a mandatory feature for being able to support the continuous evolution of requirements industry 4.0 solutions are based on. Although the modeling community has well understood the challenge, current

research in multi-level metamodeling clearly indicates that there exists no universally accepted standard yet that is fully embraced by all researchers and tool vendors. On the contrary, this research field is in full (r)evolution and plenty of more or less compatible academic and practical approaches emerge. One of these novel approaches is the Dynamic Multi-Layer Algebra (DMLA) which has been created in our research group for last four years. DMLA supports multi-level modeling behavior in a peculiar, but naturally obvious way. Unlike in many current modeling tools and methodologies, in DMLA, the modeling rules (both structural and operational) of the approach are explicitly modeled in so called bootstraps. That is, DMLA's bootstraps are self-describing assets and because they also precisely define the validation logic of its own by themselves they are also self-validating. Thus, the modeling paradigm defined by DMLA is automatically kept consistent and valid at all times.

This paper explains the standard bootstrap of DMLA, the so-called Bootstrap, which is the keystone of any validated, self-describing multi-level metamodeling in DMLA. The paper is organized as follows: Section 2 gives a brief summary on bootstrapping in other modeling systems. Next, Section 3 introduces the formal theory behind DMLA and then it also elaborates on the details of the Bootstrap. Finally, Section 4 concludes the paper and sets out future research.

## 2 Related work

Bootstrapping of a language or modeling hierarchy, in general, means that there is a process where all initial elements are to be set up. This approach is a rather usual step in many modeling and programming environments such as in Java's bootstrap class loader. In compiler technology, it is also a common method to develop a language compiler firstly in an existing language, then, to rewrite it in the new language, and finally to re-compile it by itself. However, in the field of MBSE and in particular in multi-level modeling, having a self-describing bootstrap is not such a common practice yet.

Smalltalk [1] is an object-oriented, dynamically typed reflective programming language. Reflective means here that the language is defined in itself in a causally connected way. Pharo [2] is inspired by Smalltalk, but aims to overcome its limitations and improve its concepts. Both approaches are capable of changing their own language definition, but they are programming languages with inherent typing and OOP mechanisms, and cannot offer the unabridged freedom of a real modeling language.

The Meta Object Facility (MOF [3]) defines its bootstrap as part of the MOF standard. According to the standard, the bootstrap is self-describing, but this statement is hard to verify since the standard does not provide any formal proves and the informal textual description, in English, is hard to be considered mathematically sound due to its complex self-referencing.

An open-source industrial-strength Meta-Programming System (MPS) [4] [5] is provided by the company Jetbrains. MPS provides several meta-languages

covering a wide range of language-design elements in order to support comprehensive language design. All the meta-languages within MPS are defined by MPS itself making the platform quasi bootstrapped. Nevertheless, MPS heavily relies on built-in executable language called Base Language, which resembles to Java, which is the source of any further language extension and development within MPS. Hence, although MPS's self-describing bootstrap is an interesting approach, MPS does not enable modifications to the Base Language without re-implementation of the framework and it does not support multi-level hierarchies, either.

# 3 DMLA

The Dynamic Multi-Layer Algebra (DMLA) [6][7][8] is a multi-layer modeling framework based on Abstract State Machines (ASM, [9]). DMLA consists of two parts: (i) the Core containing the formal definition of modeling structures and its management functions; (ii) the Bootstrap having a set of essential reusable entities of any modeled domains in DMLA. We have intentionally separated the two parts because the algebra must be self-contained in structure and must be able to get used with different bootstraps.

## 3.1 The Core

According to the definition of the Core, the model is represented as a Labeled Directed Graph. Each model element such as nodes and edges can have labels. Attributes of the model elements are represented by these labels. Since the attribute structure of the edges follows the same rules applied to nodes, the same labeling method is used for both nodes and edges. We define the following labels on each model entity X: (i) $X_{ID}$ (a globally unique ID of the model element), (ii) $X_{Meta}$ (ID of the meta-entity definition), (iii) $X_{Value}$ (list of concrete values), (iv) $X_{Attributes}$ (list of contained attributes).

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

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

Entity definitions are based on the above mentioned labels for forming 4-tuples. Besides these tuples, the Core also defines basic functions to manipulate the model graph, for example, they can create new model entities or query existing ones. Generally, in ASM-based approaches, functions are used to rule

how one can change states in the ASM. In DMLA, we rely on shared and derived functions. The current attribute configuration of a model item is represented using shared functions. The values of these functions are modified either by the algebra itself, or by the environment of the algebra. Derived functions represent calculations which cannot change the model; they are only used to obtain and to restructure existing information. The vocabulary $\sum$ of DMLA is assumed to contain the following characteristic functions: (i) Meta($U_{ID}$): $U_{ID}$, (ii) Value($U_{ID}$, $U_{Number}$): $U_{Basic}$ and (iii) Attrib($U_{ID}$, $U_{Number}$): $U_{ID}$. The functions are used to access the values stored in the corresponding labels. Note that the 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 new value to the *Meta* function.

Moreover, there are two derived functions: (i) Contains($U_{ID,}U_{ID}$): $U_{Bool}$ and (ii) DeriveFrom($U_{ID}$,$U_{ID}$): $U_{Bool}$. The first function takes an ID of an entity, the ID of an attribute and checks if the entity contains the attribute. The second function checks whether the entity identified by the first parameter is an instantiation, also transitively, of the entity specified by the second parameter.

## 3.2 The Bootstrap

The functions of the ASM make it possible to query and change the model. However based only on these constructs, it may be hard to use the algebra (especially in the case of a self-describing multi-level architecture) due to the apparent lack of basic, built-in constructs. This issue will be resolved by adding the Bootstrap to DMLA. Before discussing the Bootstrap in detail, we first briefly introduce the main concepts.

### 3.2.1 Main concepts

Instantiation in DMLA means gradual constraining and thus has several peculiarities. Hence, whenever a model entity claims another entity as its meta, the framework automatically validates if there is indeed a valid instantiation between the two entities. However, unlike other modeling approaches, the rules of valid instantiation is not encoded in an external programming language (e.g. Java), it is though modeled by the Bootstrap. Therefore, both the main validation logic and also the constraints used by the validation, like checking type and cardinality conformance [10] can be easily modeled within the Bootstrap. Since the modeled validation logic defines the DMLA's inherent instantiation semantics, the instantiation itself is bootstrap dependent. The Operations needed for encoding the concrete validation logic are modeled by their abstract syntax tree (AST) representation as 4-tuples within the Bootstrap as well.

In DMLA, the multi-level behavior is supported by fluid metamodeling, that is, instantiation is rather independent of metamodel design until a model can be assumed to be valid. More precisely, each modeled entity can refer to any other entity along the meta-hierarchy, unless cross-level referencing is found to be contradictory to the validation rules.

Entities may have attributes referred to as *slots*, describing a part of the entity, similarly to classes having properties in object-oriented programming. Each slot originates from a meta-slot defining the constraints it must take into consideration. When instantiating the entity, all slots are to be validated against their meta-slots. This is how DMLA checks the modeled type and cardinality constraints on the slots by the Bootstrap. Moreover, the Bootstrap can be extended with other application dependent constraints as well.

Besides gradually narrowing the constraints imposed on a slot, DMLA enables the division of a slot into several instances similarly to entities, where one can create many instances of a meta-entity within its cardinality constraint. For example a general purpose meta-slot *Components* can be instantiated to *Display*, *CPU* and *HardDrive*. Moreover, it is also possible to omit a slot completely during the instantiation if that does not contradict the cardinality constraint.

Another important feature of DMLA is that fluid metamodeling is supported at the slot level as well, namely, when an entity is being instantiated, one can decide which of the slots are being instantiated and which are merely cloned, that is being copied to the instance without any modifications. It means that one can keep some of the slots intact while the others are being concretized. Note that this behavior clearly reflects DMLA's way of modeling: one can gradually tighten the constraints on certain parts of the model without having to impose any unrelated obligations on other parts of the model which may not be known at that time.

The main entities of the Bootstrap are depicted in Figure 1.

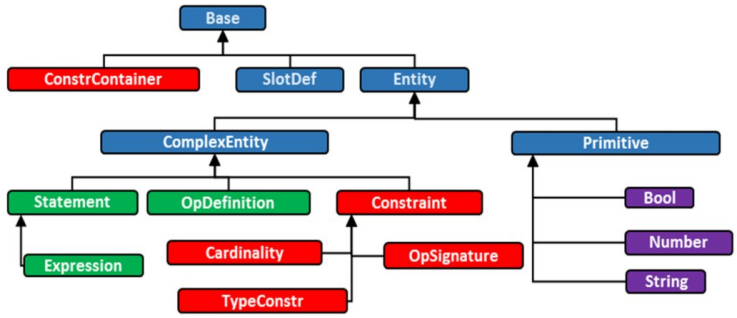


Figure 1: Main entities of the Bootstrap

### 3.2.2 Base

In the Bootstrap, the entity *Base* is the origin of all other entities. *Base* does not have a meta-entity. Since validation is based on checking the rules of the meta, this means that the definition of *Base* itself is exempt from validation due to its obvious existence as the root of the meta-level hierarchy. More precisely, the meta entity of *Base* is the value *undef* known from the ASM universes. The validation rules are always obtained from the meta and *undef* returns an empty

set at this point, which means that *Base* is verified against an empty set of validation rules, thus it is always valid.

*Base* grants three different features for its instances: (i) the ability to have slots, (ii) the ability to have constraints and (iii) it also defines the basic validation formulae. The first two features are granted by containing the *SlotDef* and the *ConstraintContainer* entity directly, meaning that instances of *Base* can have instances of these two entities, i.e. concrete slots and constraints. It is important that *SlotDef* and *ConstraintContainer* are directly contained by *Base*, and not by wrapping them into a slot. This is a necessity since if a slot that describes the ability to have slots had been defined it would have resulted in a circular self-reference that could not be resolved. The situation is similar in the case of constraints, which are mainly used in customizing the instantiation of slots. In other words, in DMLA, a concept must not be defined by itself. Note, however, that this does not mean that *Base* itself cannot have slots or constraint at all. Once *SlotDef* and *ConstraintContainer* are there, *Base* can use these definitions. More precisely, in DMLA, instantiation has no meaning of precedence or succession, that is, the entities reference to each other and validation checks an ASM snapshot of entities against validation rules that have being modeled as slots of validation formulae.

This leads to the third feature of *Base*: the basics of validation. According to the definition of *Base*, each entity may have an alpha, a beta, and a gamma validation formula. The formulae are defined as operations and are stored in slots. Instances may refine these formulae, but each entity is validated against all of its meta-entities. This means that the formulae may be extended making it more rigorous, but they cannot be further relaxed. The validation formulae of *Base* are rather simple, they enforce that all attributes of an entity must have an appropriate meta attribute in its direct meta-entity. This rule applies for directly contained attributes (e.g. *SlotDef* in *Base*) and also for slots (i.e. wrapped, extended attributes). The main difference is that slots may also have additional constraints attached to them. The validation in *Base* calls the validation of these additional constraints as well to ensure the validity of the given slot. However, checking the cardinality or the type of the value of the slot is not part of the validation in *Base*. More precisely, these checks are performed by calling the *Cardinality/Type* constraints, provided they are attached to the slots.

Moreover, in the current version of the Bootstrap, *Base* has a slot called *CanContainValue*. This slot is used to distinguish entities that can contain a value (in Value label of 4-tuple) and those that cannot. Validation is part of the validation formulae of *Base*, and it is triggered by the value of the slot. However, since there are only two entities (*SlotDef* and *ConstraintContainer*) that have this slot set to true, we plan to remove the slot in future versions and encode the ability directly in the validation formulae of *Base*.

### 3.2.3 Slots and their constraints

Once *Base* is defined, we can define the entity *SlotDef* that is a direct instance of *Base*. Although the relation between the two entities are circular (*Base* contains

*SlotDef* that is the instance of *Base*), this does not cause any problem, since their validation is executed not at once. More precisely, when validating *Base*, we can suppose that the definition of *SlotDef* is valid and vice versa.

*SlotDef* is a wrapper that can contain a concrete value (*CanContainValue* is set to true) and constraints applied on this value. To be able to handle constraints, *SlotDef* clones (makes an uninstantiated copy of) the attribute *ConstraintContainer* defined in *Base*. Besides, the validation formulae inherited from *Base* is extended by forbidding the instantiation slots that have no constraints on them. The reason behind is that if a slot has no constraints, then we have no rules at all to validate its value. Recall that in DMLA even type and cardinality checks are described by slots. This means that slots always have at least one constraints to obey. Note however that the validation does not apply to cloning, thus, a slot without attributes can still be cloned and that is what we take advantage of.

In DMLA, constraints are the key to validation and *ConstraintContainer* is the entity used actually to add constraints to entities. From this point of view, *ConstraintContainers* are the bridge between the entity and its constraints. Although we could have attached constraints directly to the entities instead of wrapping them into a *ConstraintContainer*, the current handling is easier by introducing this extra containment level because constraints become separated by this way. Thus, the role of *ConstraintContainer* is rather a technical than a theoretical one. The validation formulae of *ConstraintContainer* prescribes that each of them can contain exactly one *Constraint* as value and also verifies the rules defining the lifecycle of the constraint as will be explained later. The entity *Constraint* is used as the meta-entity of all constraints; it is the instance of *ComplexEntity* explained later. At this point, it is important only to know that *Constraint* inherits the ability to contain slots from *ComplexEntity*. In the case of a constraint, slots are used to customize the behavior. For example, in the case of *CardinalityConstraint*, we have two slots: *Minimum* and *Maximum*.

At this point, it is worth to take a closer look at the constraint-based validation of DMLA. In general, alpha and beta validation formulae are used to describe the instantiation rules of an entity. However, that is not enough in the case of the constraints, where one must be able to validate the instantiation of the entity the constraints are attached to besides instantiation of the constraints themselves. For example, when a type constraint have been instantiated, one may tighten but must not relax its type parameter e.g. it is allowed to change from Number to Integers, but not the other way around. However, the same type constraint, at the same time, must also be able to validate whether the value of the slot it is associated with belongs to the type specified. Therefore, having a single pair of alpha-beta formulae is not an efficient solution here. In order to solve this, constraints have an additional pair of alpha-beta formulae called *ConstraintAlpha* and *ConstraintBeta*. These additional formulae verify the value of the entity the constraint belongs to. Note that the evaluation of *ConstraintAlpha*, and *ConstraintBeta* cannot be initialized by the constraint itself since it needs in-context information (i.e. the entity to validate). In order to handle this properly, in the Bootstrap, the validation of *Base* is defined in such

a way that Whenever an entity detects a *ConstraintContainer* in itself, it also calls the *ConstraintAlpha* and *ConstraintBeta* operations during validation.

In the case of slots, the instantiation can be customized by adding constraints to the slot. However if the behavior of a constraint would have to be customized, this option is not available. Although the validation formulae of the constraint is given, but it is not enough. For a particular constraint (e.g. *TypeConstraint*), the rules of instantiation are set, but it is hard to describe the rules of the lifecycle of the constraint. For example, one is allowed to omit *TypeConstraint* of a slot only if the slot has a concrete value (and it is not instantiated any further). It is very hard to describe such a behavior by the alpha formulae.

Therefore, the Bootstrap defines constraints on constraints. Similarly to the definition of *Base*, one should not define that a *Constraint* may have *Constraint*s in it, since it would result in self-reference. Nevertheless, by our experience in DMLA, such a general, highly flexible solution is not needed at all. Instead it is enough to define lifecycle-based validators, which are similar to constraints, but their flexibility is more limited. In order to support this feature, *Constraint*s have a list of life cycle validators (similarly to constraints in a slot) and a third pair of alpha-beta validation formulae referred to as *LifeCycleAlpha* and *LifeCycleBeta* to invoke these validator operations. The evaluation of the *LifeCycleAlpha* and *LifeCycleBetha* formulae are managed by the *Constraint-Container* which is wrapping the *Constraint*.

### 3.2.4 Entity and further instances

By having the definition of *Base*, *SlotDef* and Constraint, all basic structural elements are there. Next, *Entity* is a direct instance of *Base*. *Entity* is used as a semantic ancestor of primitive and complex types. Primitive types represent the basic types of the underlying ASM universes, while complex types are used in practical scenarios to represent all kinds of domain specific types. The role of *Entity* is important due to the type constraints. Namely, without *Entity*, there were not a common ancestor for the types, thus the modelers would have to decide every time between the above two type options (basic and custom types) whenever a new slot is being defined. Such a kind of a restriction would have been very annoying.

*PrimitiveEntity* is an instance of *Entity*, it has an instance to each dedicated universe of the ASM (e.g. *Number*, *String*, *Bool*). It is important to clarify that the instances of *PrimitiveEntity* are used in *TypeConstraint*s and not as a value holder. For example, if we have a slot Name, then the value of the slot will be "John", while the value of the type parameter of its *TypeConstraint* will be the entity *String*.

*ComplexEntity* is practically the origin of all user entities. It is a direct instance of *Entity* and has a general-purpose slot (Children) that can later be used to model all kinds of features. The cardinality constraint of the slot allows creating any number of instance of the slot, while the type constraint allows any type available (by setting the type parameter to *Base*). Based on *ComplexEntity*, DMLA has a highly flexible, but at the same time highly customizable and

rigorous entry point for modeling. There are three built-in constraints in the Bootstrap: the type, cardinality and operation signature constraints, all of them are instances of the *Constraint*. A common setting for all three constraint types is that they are marked as permanent by using a *LifeCycleValidator*. Being permanent means that the constraint can be omitted only if all other constraints are omitted, meaning that the slot has been set to its final form of instantiation.

### 3.2.5 Type, Cardinality and Operation Signature

The *TypeConstraint* is used to validate a concrete value of a slot against a type given at the slot definition. It has the *IsInclusive* flag meaning that it should or should not accept a value that is the specified type itself. For example, the *TypeConstraint* defines the slot to accept type *Bicycle*. If the *IsInclusive* flag is set, the value *Bicycle* is accepted as well its instances. The alpha validation of *TypeConstraint* checks if the type parameter is not relaxed, while the *ConstraintAlpha* validation verifies the value of the slot against the type parameter. Note that the validation defined by *ConstraintAlpha* is activated (checked) only when the value of the slot is set.

The *CardinalityConstraint* is used to customize the cardinality of the instances of an entity or a slot. The constraint has a *Minimum* and a *Maximum* parameter. Similarly to *TypeConstraint*, the alpha validation of the constraint ensures that if an existing cardinality is overwritten (refined), then it should not relax the original condition. For example, if the original maximum parameter is already set to a concrete value, it cannot be overwritten to infinity. Unlike in the case of *TypeConstraint*, the *ConstraintAlpha*, and *ConstraintBeta* formulae are activated even if the concrete value of the associated slot is not set yet. This is necessary since if an entity has a slot and the slot is divided into several slots during instantiation, then the lower and upper limits of the instance slots have to be accumulated. For example a *Machine* may have 1..200 *Component*s and one of its instances, a *Keyboard*, may have 99..109 *Button*s, an optional *Cord* (0..1) and a *PlasticBody* (1..1), so calculation of the lower and upper limits results in (99+0+1..109+1+1).

The *OperationSignature* constraint is used to validate the signature of operations. The constraint contains three slots: the return type, the context type and a slot to define parameters. Although it may be thought that a simple *TypeConstraint* is enough for all of the parameters, it is not. The difference is that more information is needed about the type. For example, the return type of an operation may be the ID of an entity, or it can also be an array of values. Therefore, instead of setting the type parameter of the given *TypeConstraint* directly to the type expected, wrapping it by a so-called *VariableType* entity is needed. *VariableType* can wrap both usual type information and also the additional information mentioned above. The validation formulae of *OperationSignature* work similarly to the formulae of *TypeConstraint*s, restriction is allowed during instantiation and *ConstraintAlpha*/*Beta* is triggered only if the value of the slot is set.

### 3.2.6 AST entities

By now, all modeling entities that are strictly required for modeling the static structure of metamodels have already been introduced, nevertheless the Bootstrap does contain more. One of the unique features of DMLA is that it has a built-in operation language, which makes the underlying ASM functionality available for the Bootstrap. This means that the expressions and statements composing an operation are also defined as entities in the Bootstrap. In order to achieve this, every language element has a corresponding entity and when operation is defined, its abstract syntax tree is build from the instances of these entities. For example, all conditional statements refer to *If* as their meta. In order to simplify the writing of operations, we have developed a script language called DMLAScript. DMLAScript is pure syntactic sugar around 4-tuples, the scripts written in DMLAScript are translated to entities (structural and AST) composed of 4-tuples and later evaluated in a Java executor for carrying out the validation. Note that although the current executor implementation is based on Java, the concepts of DMLA are not, thus it is possible to port it to another languages such as C#, or JavaScript. We have chosen Java only since it was easy to connect a Java-based executor to the XText-based DMLAScript workbench.

### 3.2.7 Other entities

Besides the aforementioned entities, the Bootstrap contains many technical elements, but these elements do not alter or extend the basic mechanisms of DMLA. Although the Bootstrap has been created only as a proof-of-concept, we did successfully used it in many practical use cases including the Bicycle challenge. Minor fixes and extensions were required, but so far, DMLA's principal concepts has been still withstanding the test of time.

## 4 Conclusion

Self-defining language environments always tend to be more flexible and more consistent than environments defined by an external language. Multi-level modeling is a potential solution for the industrial challenges of model-based software engineering in the age of Industry 4.0. Creating an approach that could support both multi-level metamodeling and has a completely self-describing bootstrap would be very advantageous from many practical perspectives. Our novel approach, the Dynamic Multi-Layer Algebra (DMLA) aims at this goal.

The paper has introduced the core concepts of DMLA and elaborated the basic building blocks of its standard Bootstrap in detail. We do believe that by having been discussing the details behind DMLA's formal foundation and its entity hierarchy within the Bootstrap it has become clear how flexible yet rigorous multi-level modeling can be done.

In the future, we plan to streamline the Bootstrap and optimize it. Self-validation plays a key role here, since DMLA has a fix point by it self-validation design: whenever modeling errors may occur during bootstrap modifications

they will be caught as models get automatically invalid if self-validation cannot be kept self-consistent.

## Acknowledgments

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

## References

[1] G. Casaccio, S. Ducasse, L. Fabresse, J.-B. Arnaud, and B. Van Ryseghem, "Bootstrapping a Smalltalk," in *Smalltalks*, (Buenos Aires, Argentina), Nov. 2011.

[2] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009.

[3] "Omg metaobject facility." http://www.omg.org/mof/, 2005. Accessed: 2019-03-20.

[4] F. Campagne and F. Campagne, *The MPS Language Workbench, Vol. 1.* USA: CreateSpace Independent Publishing Platform, 1st ed., 2014.

[5] A. Prinz. and A. Shatalin., "How to bootstrap a language workbench," in *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pp. 347–354, INSTICC, SciTePress, 2019.

[6] "Dmla webpage." https://www.aut.bme.hu/Pages/Research/VMTS/DMLA, 2019. Accessed: 2019-04-20.

[7] D. Urbán, G. Mezei, and Z. Theisz, "Formalism for static aspects of dynamic metamodeling," *Periodica Polytechnica Electrical Engineering and Computer Science*, vol. 61, no. 1, pp. 34–47, 2017.

[8] D. Urbán, Z. Theisz, and G. Mezei, "Self-describing operations for multi-level meta-modeling," in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pp. 519–527, 2018.

[9] R. Boerger, Egon; Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[10] Z. Theisz, D. Urbán, and G. Mezei, "Constraint modularization within multi-level meta-modeling," in *Information and Software Technologies - 23rd International Conference, ICIST 2017, Druskininkai, Lithuania, October 12-14, 2017, Proceedings*, pp. 292–302, 2017.