The bicycle challenge in DMLA, where validation means correct modeling

Gergely Mezei[1], Zoltán Theisz[2], Dániel Urbán[3], Sándor Bácsi[4]

[1,3,4] Budapest University of Technology and Economics, Budapest, Hungary
[2] *evopro systems engineering Ltd., Hauszmann Alajos str. 2, Budapest, Hungary*
[1]gmezei@aut.bme.hu, [2]zoltan.theisz@evopro-group.com,
[3]urb.daniel7@gmail.com, [4]bacsi.sandor93@gmail.com

Abstract. Driven by growing popularity of multi-level meta-modeling, newer and newer approaches have been recently created. Since there is no de facto standard approach yet how to handle multi-level modeling relations, many contending technical solutions and corresponding methodologies exist nowadays. Therefore, the bicycle challenge was put together to encourage the multi-level research community to apply those techniques on a realistic looking lab example set-up and thus be able to compare the advantages and disadvantages thereof in a controlled manner. Hence, this paper describes our particular way of solving the challenge, that is, we are focusing on our multi-level approach, the Dynamic Multi-Layer Algebra (DMLA). Our solution modeled all mandatory and several of the optional requirements; moreover, all of them have been automatically validated for correctness as well. We explain the solution with the help of patterns we relied on during modeling. Correspondence between the patterns and the requirements is clearly indicated and propped up by examples taken from requirements and illustrated by their modeled manifestation in text.

Keywords: meta-modeling, multi-level, dynamic

1 Introduction

Multi-level meta-modeling approaches have been matured a lot recently. There are many approaches [2][4][5] available, some in tools [6] as well. However, concept standardization within the community is still an ongoing process that can be effectively fostered by standard challenges and workshop discussions. Therefore, the bicycle challenge is and will be a strong catalyst of any advancement of multi-level modeling.

In this paper, we describe the essential parts of our model solution in Dynamic Multi-Layer Algebra (DMLA) [11] in accordance to the suggested format description of the bicycle challenge. Although we wanted to comply as much as possible with the prescribed structure of the challenge paper, we had to combine the original Model Design and Discussion sections in order to provide a logical flow of model explanation. Hence, the paper is structured in the following manner: In Section 2, we describe the main principles of DMLA in a nutshell, then, in Section 3 we are putting them into the context of the bicycle challenge. Here, we first conceptualize and then discuss in detail those

DMLA modeling patterns which we benefitted from in our candidate model solution to the bicycle challenge. We link the patterns to some of their most representative requirements in order to indicate how our model is structured. Finally, in Section 4, we conclude the paper and shortly mention future research directions.

2 The Dynamic Multi-Layer Algebra

Dynamic Multi-Layer Algebra (DMLA) [7][8][9][10] is our multi-level modeling framework that 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. 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 are functions, which manipulate the model graph, for example, they can create new model entities. These definitions [8] form the Core of DMLA, which is defined over an Abstract State Machine (ASM) [3]. The states of the state machine represent the snapshots of dynamically evolving models, while transitions (e.g. deleting a node) stand for modifications 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 Bootstrap itself is swappable, so even the semantics of a valid instantiation can be re-defined. DMLA is also fluid in its way of multi-level modeling: it does not require to instantiate all entities of a model at once. Hence, instantiation steps are independent by design; therefore each entity can refer to any other entity along the meta-hierarchy unless cross-level referencing is found to be contradictory during model validation.

Validation in DMLA is intuitive: 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 entities. All validation formulae can be modularized by being introducing directly into the Bootstrap. Since these formulae directly influence the proper semantics of instantiation, DMLA's instantiation is self-defined by design via the model per se. The technical facility enabling this self-described meta-modeling is based on operation reification. Operation definitions are modeled by their AST representation as tuples, which are later translated into executable code by the framework.

During some practical modeling projects with DMLA, we had realized that large number of entities and their complex relations are more than challenging to be defined merely by directly producing and manipulating 4-tuples. For example, the Bootstrap and our model solution to the bicycle challenge consist of 13091 entities all together. In order to simplify creating models, we introduced a scripting language, DMLAScript, with its an Xtext-based workbench into the DMLA framework. DMLAScript is a domain-independent external DSL language to help automate 4-tuple production. Thus, the modeler must only deal with creating scripts for models in DMLAScript.

3 The bicycle challenge

Bicycle challenge is a comprehensive modeling exercise aiming to let the competing multi-level approaches demonstrate their capabilities on a made up modeling challenge. The competition started at the MULTI workshop in 2017 with one contribution [12], which was based on a multi-level extension of EMF. The 2018 version of the challenge [1] added more requirements and also fine-tuned the existing ones. Applying DMLA to the challenge was a work of approximately two man-days. The validation mechanism of DMLA helped us a lot from the beginning, so finding contradicting or faulty constructs were straightforward. Indeed, we were automatically warned that two requirements of the challenge contradict each other: "A professional racing bike has a professional race frame which…has a minimum weight of 5200 gr." and "Rocket-A1-XL which is a professional race frame. The Rocket-A1-XL has a weight of 920.0 gr." In order to fix it and get a valid model, we had to change minimum to maximum weight.

3.1 Model design and discussion

In DMLA, every entity has a meta-entity that describes its features. The instantiation relationship between a meta-entity and its instances is strictly validated, that is, no feature can be added that does not have a corresponding meta-feature; and no feature can be removed, unless the meta allows this behavior by optionality. Validation is based on three types of formulae: alpha, beta and gamma. 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 all entities of the model and invokes alpha type validation on every entity and its meta-entity. In contrast, beta type formulae are in-context checks: they are used when an entity has to be validated against multiple related entities: typically those are the attributes of an entity. For example, cardinality-like constraints are evaluated by beta formulae due to their underlying one-to-many relation. Gamma formulae are used when validation cannot be applied locally, but it requires checking global conditions, e.g. the uniqueness of an identifier.

Entities have slots similarly to attribute fields or properties in OOP languages. The slots of an entity set up its structure, for example a Bicycle has a Seat slot containing a reference to the entity Seat. A slot may have constraints added for fine-tuning its behavior. There are several kinds of constraints: 1) type constraint restricts the type of the values to be put in the slot (e.g. at filling the slot Wheel in a bicycle entity, one can only use instances of Wheel entity there), 2) cardinality constraint prescribes the allowed number of instances within a given slot (e.g. a tricycle may have three wheels), 3) operation signature constraint specifies the signature of the operation defined in the slot it is assigned to. It must be reemphasized that in DMLA operation definitions are built based on AST representation, but one can apply a specific function ("call") on them to execute them at validation. Knowing that built-in constraints are also self-modeled in the Bootstrap, when the Bootstrap is changed, the validation rules of the Bootstrap will be re-applied to itself. Eventually a fixed point must be found for any valid model in DMLA. Of course, any Bootstrap must be self-validated by design. The current DMLA

2.1 Bootstrap contains an extendable constraint mechanism with some pre-defined practical constraint elements. One of these constraints is the MustFillOnce, which is explained in detail later in the patterns section.

Last but not least, we want to put DMLAScript into the right perspective before we introduce the most relevant modeling patterns in DMLA that has been applied for the challenge as well. DMLAScript is a pure syntactic sugar above DMLA's 4-tuple representation; therefore it is independent of any practical domain models, however its language elements are AST modeled in the Bootstrap. Thus, theoretically, Bootstrap's modeling semantics firmly anchors DMLAScript. Practically, all DMLA models (even the Bootstrap) are built in DMLAScript and then turned into 4-tuples. Thus, we show only DMLScript code fragments in the paper when we explain details of the bicycle challenge. The full DMLA model of the bicycle challenge can be downloaded from [11] both in DMLAScript and in XML serialized 4-tuple representation.

3.2 Entity structure

Pattern 0: Prohibition of features or their relation can be simply left out since validation will fail if disallowed entities occur in the model.

DMLA is a validation based prescriptive modeling approach, that is, those requirements, which state disallowed features are automatically satisfied at all meta-levels by simply not being put into the model. The pattern is used in requirements "A racing fork does not have a suspension. It does not have a mud mount either." and „A mountain bike may have a rear suspension. *That is not the case for city bikes*.".

Pattern 1: Gradual type constraining is supported by restricting constraints on slots.

DMLA's basic tenet is to build multi-level meta-models along controlled reduction of design abstraction. Modeling entities, which have an internal structure, describe their setup by slots. A slot represents a feature of the entity. At the topmost abstraction level, one may not have much information about a slot, so it is used merely as a placeholder. Later, by instantiating an entity, one can override the constraints applied on the slot by restricting the structure and/or the behavior. This pattern is heavily used in all domain modeling scenarios and thus also during solving the challenge. For example, a classic bike has a frame component modeled by slots. When instantiating Bicycle and creating the RaceBike entity, we narrow the type constraints applied on slot Frame so that it could contain only instances of the RaceFrameComponent entity (**Fig. 1**). Obviously, a RacingFrameComponent is an instance of FrameComponent, thus the constraint does not contradict the meta definition. Type constrains automatically satisfy that the concretization is always consistent whenever the validation succeeds. This feature gives practical value to DMLA when industry sized models are being built.

The pattern is used e.g. in requirement "A racing bike has a racing fork and racing frame."; "A professional racing bike has a professional race frame".

```
1060    RaceBike: ClassicBike
1061    {
1067        @T: Bicycle.Frame.T =
1068        Type : Bicycle.Frame.T.Type
1069        {
1075            slot Type : Bicycle.Frame.T.Type.Type = $RacingFrameComponent;
1078        };
1079        @Bicycle.Frame.C
1080        slot Frame: Bicycle.Frame;
```

Fig. 1. Overriding the type constraint in RaceBike in the slot Frame

Pattern 2: Create new slots by dividing general purpose slots, when new features are needed. Keep the original slot if adding new features may be required later or omit it otherwise.

A usual entry point of domain definition is the ComplexEntity in the Bootstrap. ComplexEntity has a slot called Children. Its cardinality allows any number of instances (0..*) of any practically available type (any entities derived from the Base entity that is at the root of meta hierarchy). This setup perfectly suits the highest abstraction levels, where one still does not know what kind of slots are needed for the concrete entities. All along the instantiation chain, one can restrict, divide and even omit the Children slot. Dividing a slot into several instances may need an explanation. While getting more and more concrete along the instantiation chain, the slots gain more and more concrete information. For example, one may have to introduce a new feature in entity Configuration so that it could have components. In this case, one may create specific slots while keeping the general Children entity intact for later features or omit it to deny the introduction of other features in the instances of Configuration. Specific slots deriving from Children may have additional constraints, e.g. the slot Component accepts only instances of Components and BasicParts, but not Base as firstly defined.

The pattern is not specific to the Children slot: it is usual to have a slot with less restrictions and one wants to divide it into several more specific slots when introducing new features. For example, a Bicycle as an instance of Configuration needs to specify that it can have only frame, fork, seat and wheel components. This means that the slot Configuration.Components is turned into a Frame, Fork, Seat and Wheel slot in Bicycle. It is also worth mentioning that constraints of the more general slots are often restricted again while dividing the slot into multiple parts. For example the slot Wheel accepts only the instances of the WheelComponent, not Component/BasicPart.

The pattern is used many times in the challenge, e.g. “A configuration…is composed of components.”; „A bicycle is built of components like a frame, a fork, two wheels, and so forth, each of which being a component.”; „Every category of bicycle except for racing bikes may be equipped with an electric motor. Electric bikes need enforced brakes and a battery.”; „A mountain bike or a city bike may have a suspension.”;”A mountain bike may have a rear suspension.

Pattern 3: Mandatory slots are modeled by cardinality 1..1. They must be kept all along the whole instantiation chain. Optional slots are modeled by cardinality 0..1. They can be omitted on any level. Optional-mandatory slots are modeled by cardinality 0..1 and the MustFillOnce constraints at the same time. They can be omitted at any level as far as their value has already been set earlier along the instantiation hierarchy.

There are features that must be available at all modeling levels, e.g. if a Component has a Weight attribute, then all component instances, even concrete (physical) components must have the Weight attribute. On the other hand, there are optional slots that may be kept, or omitted during instantiation. For example, a Mountain bike may have Suspension, but it is not mandatory. However, there is also a third option: a slot is optional so it can be omitted later, but its value must be set at least once before being eliminated. For example, the regular price of a bicycle is handled this way. Here, one can set the price at the category level, at the bicycle model level, or at any other levels; however, once one has set it, it is no longer needed. Nevertheless, one cannot have a bike without regular price. So, when the regular price of a certain, physical bike is needed it can be fetched from its meta hierarchy and there is no need to multiply this information at all levels above. This third option cannot be set by simply setting the cardinality of the slot, since it should be 0..1. Instead, a special constraint referred to as MustFillOnce (MFO) is applied here. The MFO constraint validates the slot and does not allow it to disappear before having a concrete value.

The pattern is used e.g. in requirements: ”A mountain bike may have a rear suspension.”; „A racing frame is specified by top tube length, down tube length, and seat tube length.”; „A racing bike can be certified by the Union Cycliste Internationale (UCI).”; “Each bicycle model has a regular sales price.”.

Pattern 4: Inheritance between the entities is imitated by instantiation.

Instantiation and inheritance relations look similar on the surface, for example both are having a grouping and substitution goals. But, the main difference between the two is that instantiation is vertical, while inheritance a horizontal in nature. The fluid metamodeling behavior conceals the difference within the semantics of DLMA’s formalism of instantiation: one can instantiate an entity and still use both the entity and its instance(s) interchangeable. Although we use instantiation, we can stay at the same level of abstraction. That is why we have not introduced inheritance per se into DMLA, but used also instantiation in its stead. For example, the requirement “a mountain bike is a bicycle” provides a MountainBike entity instantiated from Bicycle.

The feature is used e.g. in requirement: “There are different categories of bicycles, such as mountain bike, city bike, or racing bike”; „A professional racing bike is a racing bike”; „A customer is a natural person or an organization.”.

Pattern 5: Enum and bitfield like requirements are modeled as an enum type with its slot-less instances.

In DMLA, the modeling of enums seems a bit strange at first: enum types do not have slots, the concrete enum values are the instances of the enum type. This mechanism suits our validation approach, since slots have a type constraint stating that the type is the enum type and thus the slot can have only concrete enum values later on.

The pattern is used e.g. in requirements:” There are different categories of bicycles, such as mountain bike, city bike, or racing bike, for different purposes such as mountain, city, or race. A racing bike is not suited for tough terrain.”; “A racing frame is made of steel, aluminum, or carbon.”

3.3 Operations and customized validation

Pattern 6: Slots can represent signature driven operation definitions at all concretization levels.

Operations are specified by their AST that are constructed from modeling entities encoding operational logic which acts upon the data part of the model. Operation definitions are stored in slots similar to usual modeling features, but they have two specialties: their type constraint is set to OperationDefinition and they have an OperationSignature constraint. The signature constraint delivers semantics that is similar to its usual programming language peers. For example one can define an operation signature that has context information (e.g. "this") and a numeric return value and then state that the average regular price operation will use this signature. From this point on, one can replace the inner part of the operation (i.e. its implementation), but not its signature (interface) is kept. For example, the entity Bicycle may have a method which checks if the given bike is suitable for a certain condition, however the instances of Bicycle may overwrite this by changing the original constraint.

The pattern is used e.g. in requirements: "A configuration is a physical artefact" (in checking the conditions of being physical as explained later)"; "The average actual sales price for a bike model."; „The average actual sales price for a bike category.

Pattern 7: Alpha formula completes entity's type semantics beyond meta-hierarchy.

When validating an instantiation relation, it is usually enough to check if all components have their meta counterparts and the cardinalities of the slots are satisfied. However, there are cases, where custom validation logic is needed, for example, if we require that the size of the front and rear wheel must be equal. This logic can be described by an alpha formula in DMLA. In this particular case, a validation formula is added to the Bicycle entity to trigger condition checks on both wheels (by using their slots). The validation mechanism of the Bootstrap makes it sure that the alpha formulae of all the metas in the hierarchy are "called" within the context of the current entity [9,10]. That mechanism enforces that more concrete instances must obey to all type requirements along their meta hierarchy.

The pattern is used in requirements: "Front wheel and rear wheel must have the same size"; "A racing frame is made of steel, aluminum, or carbon."; "A carbon frame allows for carbon wheels or aluminum wheels only"; "A professional racing bike has a professional race frame which is made of aluminum or carbon and has a minimum weight of 5200 gr." (**Fig. 2**)

```
1431    operation Bool ID::ProRaceFrameAlpha(ID instance){
1432
1433        ID material = call $GetRelevantAttributeValue(instance, $BicycleComponent.Material);
1434        if (material!=null  && material!=$Material_Carbon
1435                            && material!=$Material_Aluminium) return false;
1436
1437        Number weight= call $GetRelevantAttributeValue(instance, $Component.Weight);
1438        if (weight!=null&& weight>5200) return false;
1439
1440        return true;
1441    }
```

Fig. 2. The alpha validation of the ProRaceFrame entity

Pattern 8: Global validation requirements are satisfied by gamma formulae.

Alpha and beta formulae are used in validating the instantiation relationship, but they are only local and pivot around themselves. However, there are cases, when this is not enough though. For example, all Components must have a unique serial number. Uniqueness is not a local feature, so it must be checked within the entire model. Gamma formulae are essential whenever all the instances of an entity must be enumerated in the model in order to check some global characteristics thereof.

The pattern is used in requirement: "A component has properties, for example weight, size, colour, unique serial number.".

Pattern 9: Soft validation, i.e. filtering features are supported by operations attached to the entities.

DMLA has a highly customizable, flexible validation mechanism, however it is strict and thus failing the validation criteria automatically produces invalid models. For business analytics like functionalities or domain specific validation of entity concepts, one may need a "softer" method that is rather a filtering mechanism than a validation. Here, the modeled entities may be filtered against query criteria. For example, each bicycle (concrete instances, models and even categories) can be tested against a certain feature, e.g. whether it is suitable for tough terrain. To model this, we have added an operation to the Bicycle entity. Of course, the instances can freely override the default logic, which is: the bike is suitable for everything. The feature does not follow the logic of validation (e.g. alpha formulae), thus overridden operation implementations may even contradict their meta definition at will, the validation will not stop.

The pattern is used in requirements: "There are different categories of bicycles, such as mountain bike, city bike, or racing bike, for different purposes such as mountain, city, or race. A racing bike is not suited for tough terrain. A racing bike is suited for races."; "Challenger A2-XL is a professional racing bike model for tall cyclists".

Pattern 10: Custom validation can be driven by flags. If a flag is presented, validation is turned off, if a flag is omitted, validation is switched on.

A core part of the bicycle challenge is to check whether a model element is physical. In the DMLA group, we had discussed this requirement in detail in order to find the best solution for it. At first sight, being a physical artefact seemed to be a structural condition: an entity is a physical thing if all of its components are physical. However, later we had realized that this was not enough. An unassembled bicycle is not a bicycle (taking its function into account) even if it contains physical components only. Therefore, we have invented a twofold mechanism for this purpose: every BicycleEntity has an optional slot AbstractEntity, and an operation CheckPhysical. If the slot is left out during instantiation, the user states that the given entity is not abstract any longer. In this case, we validate this statement by calling the CheckPhysical operation that is intended to validate if the entity is physical according to its structural parts (all primitive values are set, all non-primitive components are physical). The mechanism is supported by an alpha validation formula of BicycleEntity that checks if the AbstractEntity slot has been left out and it calls the CheckPhysical operation in that case. This validation is not a

soft validation, so not only the last (most specific) implementation of CheckPhysical is called, but all of its meta versions as well. It is worth mentioning that it is quite comfortable since, for example, Configuration checks if its Components are physical and thus, we do not need to validate if Seat, Fork, etc. are physical in Bicycles (since the slots originate from slot Components).

The pattern is used in requirement: "A configuration is a physical artefact".

Pattern 11: Derived properties are added to entities as operations. In case of summary, the instantiation chain can be used in a layer-transparent fashion.

The challenge has five optional requirements all focusing on price-based calculations by derived properties. We only solved the first four of them due to their similarity. The solutions follow the same pattern. The requirements are covered by operations. Each operation is applied on an instance of Bicycle and returns a number as described by their operation signature (Pattern 6). Users can call the operation at any abstraction level and the operation will calculate the result based on the entities below that level. This means that we do not have a separate operation for bike category and bike model, the calculation is independent from the level, the only difference is that the user calls the same method on a category (e.g. race bike), or a model (Challenger A2-XL). Inside the operations, all model entities are fetched and selling transactions (SellingActs) which are physical artefacts are selected, thus the calculation is run only on physical bikes that are already sold, and not on abstract bikes, on generic (non-physical) selling act templates, or on bikes stored in the shop. From SellingAct, a sold bike is obtained and it is checked if it is an instance of the current entity processed (e.g. whether it is an instance of the given bike model). If that is the case, the number of bikes and also their accumulated price (actual selling price, or regular price obtained from the meta hierarchy of the bike depending on the actual requirement) get incremented. The operations clearly demonstrate how easily one can add calculations or similar algorithms to the existing modeling entities. The pattern is used in optional requirements of the challenge.

4 Conclusions

Current implementation of DMLA 2.1 seems to be able to easily cope with light industrial application similar to the bicycle challenge. As an illustration for this, we have extended the original challenge in our solution, and modeled tricycles, tandem bikes and unicycles as well in order to showcase the capabilities of our approach even further [11]. Obviously, DMLA also does have limitations. The most important thereof is the speed of the validation in the current implementation [10], which does not allow DMLA, in its current implementation, to function as a models@run-time framework yet. Hence, we are working on its parallel and incremental validation now.

Regarding the challenge, our main issue was how to present our solution in an easily understandable way since ours is an "executable model" with many lines of "model code" where every detail does count. We decided to upload the commented version of the whole solution onto the DMLA website [11]. It is worth reading it through line by line to appreciate each of the patterns mentioned. Currently, we are working on a lighter

DMLAScript syntax for purposes of domain modeling without Bootstrap relevance. We also benefited a lot from the challenge because we realized that DMLA is not mature yet when it comes to handle complex inheritance trees similar to state-of-the-art UML models. Although it may be the Achilles heel of any pure multi-level modeling, we will definitely have to address it properly in DMLA if we really want to believe in its modeling capacity for real industrial set-ups in the future.

Acknowledgement

This work was performed in the frame of FIEK 16-1-2016-0007 project, implemented with the support provided by evopro Systems Engineering and the National Research, Development and Innovation Fund of Hungary, financed under the FIEK 16 funding scheme. The research has been also supported by the European Union, co-financed by the European Social Fund. (EFOP-3.6.2-16-2017-00013).

References

1. MULTI 2018 Homepage, https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/
2. Atkinson, C., Kuehne, T.: Model-driven development: A metamodeling foundation. IEEE Software, 20(5):36–41, (2003).
3. Borger, E., Stark, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag New York, Inc. (2003).
4. de Lara, J., Guerra, E.: Deep Meta-modelling with MetaDepth. Objects, Models, Components, Patterns. 6141, pp. 1-20. (2010).
5. Atkinson, C.,, Gerbig, R.: Melanie: Multi-level modeling and ontology engineering environment, In MW '12 Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards, Article No. 7 (2012).
6. Multilevel Wiki, http://homepages.ecs.vuw.ac.nz/Groups/MultiLevelModeling/MultiTools
7. Theisz, Z., Mezei, G. An Algebraic Instantiation Technique Illustrated by Multilevel Design Patterns., MULTI@MoDELS, Ottawa, Canada (2015).
8. Urban, D., Theisz, Z., Mezei, G.: Formalism for Static Aspects of Dynamic Metamodeling, Periodica Polytechnica, vol. 61, no. 1, pp. 34-47, (2017).
9. Mezei., G., Urban, D., Theisz, Z.: Validated Multi-Layer Meta-modeling via Intrinsically Modeled Operations, MULTI@MoDELS, Austin, Texas, USA (2017).
10. Urban, D., Theisz, Z., Mezei, G.: Self-describing Operations for Multi-level Meta-modeling, MODELSWARD 2018, Madeira, Portugal (2018).
11. DMLA Homepage, https://www.aut.bme.hu/Pages/Research/VMTS/DMLA
12. Macias, F., Rutle, A., Stolz, V.: Multilevel Modeling with MultEcore: A Contribution to the MULTI 2017 Challenge, MULTI@MoDELS, Austin, Texas, USA (2017).