DMLA 3.0: Towards an Industrial Multi-Layer Modeling Framework

Norbert Somogyi Máté Hidvégi Gergely Mezei
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
{somogyi.norbert, hidvegi.mate, gmezei}@aut.bme.hu

Abstract. Multi-level modeling is an extension of the traditional two-level modeling paradigm allowing for an arbitrary number of classification levels. The main goals of multi-level modeling are to improve readability and reduce the complexity of models by modeling each concept on an abstraction level that is more natural to its purpose. The Dynamic Multi-Layer Algebra (DMLA) is an approach created for multi-level modeling. DMLA has originally been created as a proof-of-concept, but has since proved its feasibility. However, despite its success, DMLA also had various shortcomings that prevented it from being applicable in industrial setups. Apart from being semantically sound and based on well-proven research, industrial needs often include non-functional requirements as well. Thus, a major revision was necessary both at a theoretical and practical level. This paper elaborates on the main concepts of the new version and the reasons behind them.

Keywords: Modeling; Multi-level modeling; DMLA; Multi-layer modeling; Level-blind

1 Introduction

The original motivation behind multi-level modeling was to create a modeling paradigm (i) to reduce accidental complexity [1], and (ii) to improve the general comprehension of models. In this context, accidental complexity means that model elements are created solely for the sake of expressing the multi-level nature of the solution, rather than capturing some aspect of the modeled domain. For this purpose, multi-level modeling has introduced the concept of unlimited instantiation levels and various other notions that are based on the unlimited nature of levels. Approaches that acknowledge the existence of explicit modeling levels are often referred to as *level-adjuvant.* Similar yet highly different approaches have also emerged in the form of *level-blind* approaches, which do not acknowledge explicit instantiation levels, although they can still implicitly implement the concept of levels.

Dynamic Multi-Layer Algebra (DMLA) is a level-blind, multi-layer modeling approach based on the Abstract State Machines (ASM) formalism [2]. DMLA offers a highly flexible and customizable modeling structure and it can easily deal with both design-time and run-time aspects of modeling. To achieve this, the concept of instantiation is dynamic in spirit and the formalism is able to account for explicit model states and not just for simple isolated snapshots. Over the years, DMLA has proved its feasibility and various advantages over multi-level modeling approaches. [3, 4] Nevertheless, in some aspects, further improvements could be made. For example, common industrial needs, such as transaction handling or communication with external applications, should be supported. Thus, in this paper, we present DMLA 3.0, the main goal of which is to become a framework also fit for industrial development.

The paper is structured as follows. Section 2 presents related work, highlighting the main ideas of multi-level modeling and comparing our approach to other multi-level solutions. Section 3 introduces the main concepts, fundamental goals, theoretical difficulties and improvements, components, architecture and functionalities of DMLA 3.0. Section 4 concludes the paper.

2 Related work

Multi-level modeling is a modeling approach that aims to improve upon the shortcomings of traditional modeling approaches, such as OMG's Meta Object Facility (MOF) [5]. The key idea behind multi-level modeling is *deep instantiation* [6]. As opposed to classic approaches with a fixed number of levels, in multi-level modeling, instantiation may concern an arbitrary amount of levels. Instantiation chains are formed, further refining the original concept until a concrete model element is created. Consequently, a model element may be the meta-type of another element on a lower level and the instance of an element at a higher level. For this reason, model elements may be considered as both classes and objects at the same time and are often referred to as *clabjects* [7]. *Potency notion* [8] means labeling elements of the model with non-negative integers. Potency controls the depth of instantiation on a given model element. For example, a potency of 3 implies that the instantiation of the element spans exactly 3 further levels. Over the years, the concept of potency notion has undergone significant revision and has evolved into numerous different variants.

Over the years, many research tools had been created for different multi-level approaches. *Deep Java* [9] is an approach that integrates multi-level modeling into an object-oriented environment. It uses the concept of classic potency notion. *MultEcore* [10] is a potency-based tool that represents any two adjacent levels as an Ecore model directly in the Eclipse Modeling Framework (EMF). In contrast to Deep Java and MultEcore, DMLA 3.0 uses its own ecosystem and does not use potency notion. DMLA 3.0 strives to include support for numerous industrial requirements, such as transaction handling or multi-user support. No other tool is known to take such aspects into consideration, for they focus mainly on the modeling aspects of their approach. Naturally, DMLA 3.0 retains any advantage that previous iterations had, such as being highly flexible and supporting fast-prototyping. Apart from the industrial requirements, DMLA 3.0 also focuses on further improving and solidifying its theoretical formalism.

3 DMLA

In this section, we present the milestones of DMLA incarnations. We briefly introduce DMLA 1.0 and 2.0, and also present DMLA 3.0, introducing its main concepts, fundamental goals, components, architecture and functionalities.

3.1 Core and Bootstrap

The Dynamic Multi-Layer Algebra (DMLA) is a multi-layer modeling framework inspired by the core ideas of multi-level modeling. DMLA is a level-blind approach that does not use the potency notion concept although it can emulate potency. This is why we refer to DMLA as multi-layer instead of multi-level. DMLA consists of two essential basic parts: the Core and the Bootstrap. The Core defines the basic modeling structure describing the model elements (nodes and edges), along with low-level functions to access and modify them [11]. In DMLA, the model elements are called entities. The theoretical formalism of entities is defined in the Core. Every entity is represented as tuples containing various information about the element. The Bootstrap [11] is a set of entities defining basic modeling facilities. The Bootstrap makes it possible to use DMLA to actually create domain models in practice. It defines not only the basic entities, but also the basics of validation and thus the semantics of instantiation itself. Consequently, it is possible to create different Bootstraps following different modeling paradigms. This concept is one of the most important features of DMLA.

3.2 DMLA 1.0 and 2.0

DMLA had two previous incarnations. DMLA 1.0 followed the naive approach of each major modeling concern having been allocated to its own representation in a 6-tuple structure defined over the underlying ASM [2] representation. DMLA 2.0 [12] introduced a major improvement by replacing the 6-tuples by 4-tuples emphasizing validated, modeled behavior over hard-wired constructs. The 4-tuples describe: (i) the ID of the entity, (ii) the ID of the entity's meta entity (instantiation relationship), (iii) values assigned to the element (reference to another entity, or a primitive string, number or bool literal), and (iv) attributes assigned to the element. Entities may contain other entities, in this case, the attributes hold the references to the contained entities. DMLA 2.0 also introduced a completely modeled *operation language*, which made it possible to describe the validation semantics by modeled entities instead of abstract formal functions like in DMLA 1.0. Operations may be contained by entities, similarly to how methods may belong to objects in object-oriented programming languages. The modeled operation language of DMLA 2.0 also made it possible to re-design the validation mechanism. The Bootstrap became self-validated and almost completely self-describing. DMLA 2.0 also introduced slots (value holders) and constraints (reusable validation logic) as part of the Bootstrap.

The flexibility of DMLA was proven by several challenges and research results. However, DMLA 2.0 also had its limits: (i) the self-describing model architecture resulted in a complex boxing mechanism in the Bootstrap, producing many entities solely for technical reasons, (ii) the modeled operation language focused on validation only, (iii) the approach was realized as a closed ecosystem: a single-user application, which had no option to access the models from outside, or to reach external applications from the models. As these compromises do not belong to a fully-fledged modeling environment, a major overhaul of DMLA was deemed necessary.

3.3 Goals

When designing DMLA 3.0, collecting a well-defined list of goals was a necessary first step. Some of these stem from the original motivation behind creating DMLA, while others were added to improve usability and the ability to use DMLA in industrial scenarios. The following goals were identified:

- **Stepwise refinement.** The approach should support creating highly abstract prototypes and refine them step-by-step all the way to concrete products. This has always been one of the main objectives of DMLA. This way fast-prototyping is supported: the ability to quickly create prototype models based on partial information.

- **Strict, self-described validation.** The Boostrap should be able to validate itself and all other (domain) models as well. This way, the model may automatically be kept valid at all times.

- **Support for modification.** There should be a way to modify, create and delete entities using the operation language.

- **Modeled operations.** The operation language should be fully modeled and operation code should be able to be modified similarly to other modeling entities.

- **External collaboration.** The approach should be able to reach external components, and external components should be able to reach the models and even modify them. Naturally, providing external services for the model and vice versa is crucial when used in industrial setups, as this enables the modeling framework to be integrated into a network of services.

- **Multi-user support.** The approach should be able to support multiple users working on the models simultaneously. Similarly to version control systems, the approach should support parallel editing of models and various other artefacts in a convenient and error free way.

- **Generated applications.** Code generation based on the models should be supported.

3.4 Main components

Driven by the goals, we have re-designed most of the components of DMLA. The Core remains unchanged (both its definition and its role), it still describes the structure of tuples and the basic functions (e.g. querying a part of the tuple) of the ASM formalism. The role of the Bootstrap also remains unchanged, but its content has been completely rewritten. Regarding the relationship between the Bootstrap and the Core, several shortcomings of the previous DMLA implementation have surfaced over the years.

Starting from DMLA 2.0, a virtual machine (an interpreter) was used to emulate the ASM in practice to manipulate the entities and execute operations. The main issue stems from the fact that the operation language of DMLA needs a set of programming statements (e.g. conditional branch or iterations) to build the operations. These statements can easily be modeled as entities, but defining their semantics is not an obvious task. From a theoretical point of view, one can say that the underlying ASM supports these statements, but in practice, one must define how exactly to handle the statements. In DMLA 2.0, this issue was solved by a hard-wired mapping between the modeled programming statement entities and the underlying virtual machine language, but this solution is fragile and not elegant. Especially considering that several Bootstraps and several virtual machine implementations may exist.

As a solution, in DMLA 3.0, we have introduced the Bootstrap Core Interface (BCI), which is an abstraction over the Core and is practically a set of constructs. Every element listed in the BCI must have a corresponding implementation in the Bootstrap. It is worth mentioning that besides the statements, the BCI contains several other constructs to be able to emulate the ASM completely: (i) built-in ASM functions to manipulate the tuples, and (ii) entry points for entity validation. The main advantages of using the BCI are: (i) it has an explicit list of supported statements and built-in functions and (ii) the virtual machine and the Bootstraps are now independent of each other. The virtual machine defines the semantics of all BCI constructs; thus, it can execute operation statements easily. Moreover, Bootstraps have a reference only to the BCI, not to the virtual machine directly, which simplifies locating errors and is also advantageous for security aspects.

3.5 Bootstrap 3.0

The Bootstrap plays a key role in DMLA, thus, having an efficient Bootstrap is mandatory. The previous Bootstrap had been created as a proof-of-concept solution and it had several compromises. For example, it used several layers of boxing of entities in order to handle self-describing validation. While designing the new Bootstrap, these shortcomings were avoided wherever possible.

3.5.1 Components of entities

In DMLA 2.0, all entities had validation operations used to refine the instantiation. Moreover, entities were composed of slots used both as value holders (such as fields in the object-oriented world) and for operations. It was possible to add constraints (re-usable validation logic) on slots in order to customize their instantiation. As constraints themselves were also entities, they were able to have slots (modeling the parameters of the constraint logic), and the lifecycle of entities was ruled by customizing the validation operations of the constraints. Although this hierarchy was adequate, it had some drawbacks. (i) Fields and operations were not differentiated, although their constraints and lifecycles were highly different (e.g. a field must always have a type and a cardinality, while an operation has a signature). (ii) Entities were not able to have constraints directly. (iii) The lifecycle validations of constraints were designed to be highly general leading to overly complicated definitions.

In DMLA 3.0, the Bootstrap defines that domain entities are composed of four attributes: (i) fields, (ii) operations, (iii) constraints and (iv) annotations. All of the different attribute types have their own roles and are handled differently. Fields and operations are separated, due to their different life-cycles. It is possible to create entities which do not allow adding new fields to their instances, but the ability of adding operations cannot be disabled. This behavior reflects that the structure of the entity may be sealed but it is always possible to extend its dynamic behavior. Constraints can be added to entities directly, they define a restriction on the structure or the value of the entity. They can be added to any entity, but they cannot be omitted later during the instantiation. This reflects the fact that if we add a restriction to an entity, then this restriction should be obeyed by all of its direct and indirect instances as well. Similarly to many modeling and programming environments, DMLA 3.0 introduces annotations to alter how the entity should be processed. Some of the annotations instruct the Bootstrap (e.g. the validation), while others are processed by the underlying virtual machine (e.g. BCI mapping). Annotations can be freely added to or removed from any entities.

3.6 Structural validation and entity modification

Each entity is a direct or indirect instance of the entity Base, which has an operation called *Validate*. All entities inherit this, thus all entities can be validated. Naturally, entities tend to override the original definition of Validate and customize the semantics of instantiation. However, instances may only tighten the validation rules, but are not allowed to relax them. The validation operation is a BCI construct: whenever the virtual machine proceeds to check whether an entity is valid, it looks up the correct operation based on the BCI mapping and executes that operation. Thus, the virtual machine decides when it validates an entity, but it is the Bootstrap that decides what validation means.

Since the semantics of valid instantiation depend on the Bootstrap, not on the virtual machine or the BCI, the validation of the structure of entities is also Bootstrap-dependent. Naturally, the Core representation cannot be altered, entities are represented by 4-tuples. However, the Bootstrap can decide how it handles the attributes of entities. For example, the four parts that entities are composed of and the validation rules mentioned above are completely modeled by the Bootstrap and described by its operation language. The virtual machine and even the BCI is unaware of this structure. It would be possible to create another Bootstrap introducing for example derived properties without changing the virtual machine or the BCI.

At this point it is worth mentioning that the Core (and therefore the BCI) offers only low-level functions to create entities. A new tuple can be created, but it will be completely empty. Usually, this is not sufficient as we expect entities to have fields, operations, constraints and sometimes annotations as defined in their meta-entity. The Core cannot guarantee this, since it does not know about the four attribute types that entities are composed of and the instantiation mechanisms in general. To overcome this, the Bootstrap was extended with advanced creation operations, which instantiate attributes as expected.

3.6.1 Contracts

In DMLA, every entity has exactly one meta-entity refining the rules of instantiation, including the constraints imposed upon the structure and the values of instance entities. Although this mechanism works well, introducing the inheritance relationship becomes impossible as both the base entity (inheritance) and the meta-entity (instantiation) would define rules e.g. on the fields of the entity and these rules could conflict with each other. The situation is even worse considering that fields refer to their meta-field directly. However, using inheritance would be useful in many scenarios especially to group entities.

To overcome this issue, DMLA 3.0 introduces *Contracts*. Contracts are very similar to object-oriented interfaces in the sense that they define a structural pattern that each entity adhering to this contract must include. There are two differences between the meta-entity and the Contract: (i) The meta-entity acts as a full specification, namely, attributes not mentioned in the meta-entity are considered invalid, while Contracts do not add any restrictions on such attributes. (ii) The meta-entity validates its slots based on their meta-slots. In contrast, Contracts apply a constraint-based validation only: if the constraints of the given slot are satisfied, then the slot is considered valid. Based on these differences, a Contract accomplishes a flexible grouping mechanism on entities.

3.7 DMLA Engine

The technology that enabled DMLA 2.0 to create, modify and validate models on-the-fly is called GraalVM [13]. GraalVM is a new generation Java Virtual Machine and development kit which provides ahead-of-time compilation and efficient support for custom programming languages as well. The enhanced language support is achieved by Truffle [14]. Truffle is an API on top of GraalVM allowing the implementation of arbitrary programming languages such as the modeled operation language of DMLA, while also allowing interoperability between the different Truffle languages by its polyglot nature. The implementation of DMLA 3.0 is also based on GraalVM and Truffle.

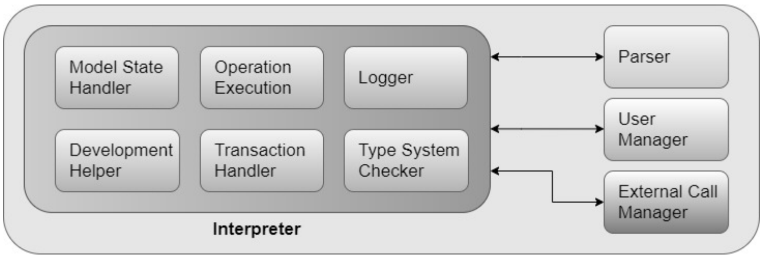

Figure 1: DMLA 3.0 Engine

3.7.1 Architecture

The architecture of the DMLA 3.0 *Engine* is depicted in Figure 1. The Engine contains four subcomponents: (i) the Parser, (ii) the User Manager, (iii) the External Call Manager, and (iv) the Interpreter. The Parser component is responsible for constructing the tuple representation from a user-friendly textual or graphical representation and building the Truffle Abstract Syntax Trees (AST) from the operation definitions. The User Manager component handles user registration, authentication and authorization including permission handling. The External Call Manager creates a bridge between the model space and external applications. Finally, the Interpreter is the main component of the Engine. It has several subcomponents: (i) the Model State Handler acts as a model repository and ensures validation of models, (ii) the Operation Execution evaluates operation definitions, (iii-iv) the Logger and the Development Helper components help developing models and localizing errors, (v) the Transaction Handler grants consistency and validation even in a multi-user environment, and (vi) the Type System Checker helps using and validating the unique, modeled type system of DMLA.

3.7.2 Validation

In earlier iterations, validation always meant checking the relations and structure of entities. Given a snapshot, it was evaluated whether each entity is a valid instance of its meta-entity. However, by introducing the operation language, moving the focus from validation to more general model manipulation and opening the ecosystem of DMLA, three kinds of validations emerged: (i) structural, (ii) dynamic, and (iii) environmental validation. Structural validation is always applied on a snapshot of the models; thus, entities are handled as read-only. In contrast, dynamic validation defines rules for scenarios when the model is currently changing or has been changed. Finally, environmental validation formulates rules for the execution environment (DMLA Engine). These rules are not directly part of the models, but affect their execution. Structural validation rules are completely modeled, dynamic validation is partially modeled, while environmental validation rules are encoded solely in the underlying DMLA Engine.

Structural validation Structural validation is based on the Validate operation of the entities mentioned earlier. There are two different scenarios to handle: (i) single entity validation, and (ii) full model validation. Single entity validation means validating an entity against its meta-entity. In contrast, full model validation validates all entities, which usually happens when starting the DMLA Engine. In this scenario, the Interpreter calls the single entity validation on each entity one-by-one. Structural validation may be triggered by an interaction with the user, or automatically by the virtual machine (e.g. when loading models).

Dynamic validation Dynamic validation consists of rules to be enforced during (i) the change of model entities, and (ii) the execution of operations. When an entity is to be changed, the Interpreter must evaluate the annotations applied on the entity. For example, the read-only annotation means that the tuple representing the entity cannot be changed. Note that structural validation cannot be used here, since both the old and the new version of the entity would possibly be a valid instance of its meta-entity. However, the modification should be forbidden. The validation of such annotations is currently handled by the Interpreter. Another case when dynamic validation is used occurs when executing operations. For example, type checking should be applied on variables and parameters of operations, when the operation is run. The main challenge stems from the fact that the type system of DMLA is rather dynamic. Instead of having a somewhat static type hierarchy as usual, the concept of type is encoded in the semantics of instantiation. This means that the Interpreter cannot decide on itself whether a certain value belongs to a certain type. It must call the validation defined by the Bootstrap or the domain model in order to answer that question. The following points must be validated when executing an operation: (i) the type of the parameters whenever an operation is called, or an operation returns a value, (ii) variable assignment, (iii) typecasting. The Interpreter handles these validations automatically and transparently based on the modeled type conformance checks and reports any type conformance errors.

Environmental validation There are several scenarios, when environmental validation is needed, such as user management or transaction handling. Users should be able to log in, apply queries, modify the model, while others are simultaneously working on the same model. Although users and their permissions could be modeled by the Bootstrap, that would actually not be desirable as users of the framework are definitely not part of the domains. Therefore, they are handled completely by the Engine as part of the environmental validation.

Similarly to users, transactions are handled by the Engine. DMLA 3.0 offers a very simple but efficient method for handling conflicts caused by simultaneous user actions or by action sequences leading to invalid entities. All operations are executed in a transaction. A transaction is started when the engine starts the execution of the operation and it is committed at the end of the execution. The prerequisite of a successful commit is a successful model validation. If the commit fails for any reason, the changes are rolled back, and an exception is thrown by the Interpreter. Transactions are not modeled as they are part of the execution environment, not the (domain) models. Avoiding inconsistent model states caused by transactions are granted by the environmental validation.

3.8 External interfaces

In the previous iterations, DMLA was used only as a modeling system with strict, multi-layer validation. Moreover, the validation had to be modeled completely, e.g. there was no option to use an external constraint solver. DMLA operated in a closed ecosystem with highly limited ways to communicate with external applications. In contrast, DMLA 3.0 aims at communicating and collaborating with external applications.

Applications may want to use different ways of communication with DMLA, e.g. REST API, TCP connection or direct API calls. It is clearly not desirable to add these technical details to the model. Especially, since this information is not part of the domain, but a configuration setting of the execution environment. DMLA 3.0 introduces Connection Points (CP) identified by an ID (string). The External Call Manager (ECM) component of the Engine supports a set of protocols, one of which the applications must implement. A CP contains technical details based on the selected protocol (e. g. IP address) needed by the ECM to communicate with the external application. Therefore, CPs act as a bridge between the inner entities of DMLA and the outside world.

There are four different scenarios to support regarding the direction of communication. (i) DMLA operation calls external application, (ii) external application calls DMLA operation, (iii) external application reads the structure of an entity, (iv) code generation based on DMLA entities. In the first case, the modeler defines an operation with an empty body and specifies the CP to use. If the operation is called, the ECM identifies the external application and forwards the input parameters set by the call. These operation calls are always executed asynchronously, but it is possible to specify a callback function to be executed when the external call returns. In the second case, the identifier of the CP is used as an annotation on the modeled DMLA operation, which is to be called by the external application. Based on the annotation, the ECM can identify the operation, pass the parameters given by the external application and call the operation. Moreover, it is possible to register a second CP, which is used to send the result of the DMLA operation back to the external application. In the third case, no operation needs to be executed, but the structure of an entity is to be queried. The solution is based on the built-in *Serialize* function available in all entities. Serialize creates a JSON representation of the structure of the entity. The Interpreter calls Serialize on the specified entity and returns the results to the external application. The fourth case is an advanced version of the third one. The user obtains the JSON representation of the specified entity and generates code using this data. The code generation is based on DMLA Structure Description Language (DSDL), a template-based model transformation language created specifically for this purpose.

4 Conclusions

Dynamic Multi-Layer Algebra (DMLA) is a multi-layer modeling approach aiming at providing an efficient solution for multi-level modeling for industrial applications. In this paper, we have presented the goals, concepts and features introduced in the new iteration, DMLA 3.0. The motivation for a significant revision stemmed partly from the original goals set in the previous iterations and partly from being able to support typical industrial requirements. While the formal base was kept unchanged, the Bootstrap and the virtual machine was completely rewritten to make editing domain models more efficient and convenient. Although the paper presented the changes in the software design and architecture as well, these changes were caused mainly by theoretical improvements. We have also presented the main issues and their solutions in several key aspects. DMLA 3.0 is superior compared to earlier versions in almost all aspects. Besides being easier to use in practical scenarios, it is more rigorously formalized and at the same time more flexible as well. Currently, DMLA 3.0 is under development. We plan to implement the approach as elaborated in this paper and prove its features by several case studies.

Acknowledgments

The work presented in this paper has been carried out in the frame of project no. 2018-1.1.2-KFI-2018-00062, which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.1.2 KFI. funding scheme. It was also supported by the BME Artificial Intelligence TKP2020 IE grant of NKFIH Hungary (BME IE-MI-SC TKP2020) and by the Ministry of Innovation and the National Research, Development and Innovation Office within the framework of the Artificial Intelligence National Laboratory Programme.

References

[1] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," *Software & Systems Modeling*, vol. 7, pp. 345–359, Jul 2008.

[2] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., 1st ed., 2003.

[3] G. Mezei, Z. Theisz, D. Urbán, and S. Bácsi, "The bicycle challenge in dmla, where validation means correct modeling," in *Proceedings of MODELS 2018 Workshops: 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018* (R. Hebig and T. Berger, eds.), vol. 2245 of *CEUR Workshop Proceedings*, pp. 643–652, CEUR-WS.org, 2018.

[4] F. A. Somogyi, G. Mezei, D. Urbán, Z. Theisz, S. Bácsi, and D. Palatinszky, "Multi-level modeling with DMLA - A contribution to the MULTI process challenge," in *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pp. 119–127, IEEE, 2019.

[5] "OMG: MetaObject Facility." http://www.omg.org/mof/, 2005. Accessed:2021-04-23.

[6] C. Atkinson and T. Kühne, "The essence of multilevel metamodeling," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, (Berlin, Heidelberg), pp. 19–33, Springer-Verlag, 2001.

[7] C. Atkinson and T. Kühne, "Meta-level independent modelling," in *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, pp. 1–4, 2000.

[8] F. A. Somogyi, Z. Theisz, S. Bácsi, G. Mezei, and D. Palatinszky., "Multi-level modeling without classical modeling facilities," in *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pp. 393–400, INSTICC, SciTePress, 2020.

[9] T. Kühne and D. Schreiber, "Can programming be liberated from the two-level style: Multi-level programming with deepjava," *SIGPLAN Not.*, vol. 42, pp. 229–244, Oct. 2007.

[10] F. Macías, A. Rutle, V. Stolz, R. Rodríguez-Echeverría, and U. Wolter, "An approach to flexible multilevel modelling.," *Enterprise Modelling and Information Systems Architectures*, vol. 13, pp. 10:1–10:35, 2018.

[11] G. Mezei, F. A. Somogyi, Z. Theisz, D. Urbán, S. Bácsi, and D. Palatinszky, "A bootstrap for self-describing, self-validating multi-layer metamodeling," in *Proceedings of the Automation and Applied Computer Science Workshop*, pp. 28–38, 2019.

[12] 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).

[13] D. Bonetta, "Graalvm: Metaprogramming inside a polyglot system (invited talk)," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*, META 2018, (New York, NY, USA), pp. 3–4, Association for Computing Machinery, 2018.

[14] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," *SIGPLAN Not.*, vol. 52, pp. 662–676, June 2017.