Towards Evolutionary Multi-layer Modeling with DMLA

Sándor Bácsi[a], Dániel Palatinszky[b] and Máté Hidvégi[c]
Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Muegyetem Rkp. 3., Budapest, Hungary

Keywords: Multi-layer Modeling, Multi-level Modeling, Graal, Truffle, Virtual Machine, Interpreter, Compiler, DMLA.

Abstract: State-of-the-art meta-model based methodologies are facing increasing pressure under new challenges originating from practical applications. In such cases, there is a strong need for approaches that support continuous, fine-graded, incremental refining of concepts. To address these challenges, our research group started working on a new modeling framework, the Dynamic Multi-Layer Algebra (DMLA) a few years ago. DMLA follows a completely new modeling paradigm, referred to as multi-layer modeling. Multi-layer modeling is originated from multi-level modeling and offers a highly flexible abstraction management approach in a level-blind fashion through its advanced deep instantiation and evolutionary snapshot management. One of the key features of DMLA is its self-validation mechanism based on a built-in, completely modeled operation language. Our initial solution had its limitations, since interactive editing was not supported, modelers could interact only with a single snapshot of the model. To overcome the limitations, we have created a virtual machine and an interpreter. In this paper, we present the novel architecture of our solution and demonstrate the feasibility of our approach by a walk-through of the concrete model management steps of an illustrative example to show the benefits of evolutionary model editing in DMLA.

1 INTRODUCTION

In the software industry, the development of applications routinely begins with setting up the most relevant design aspects based upon the main requirements of the given project. Stakeholders and customers try to specify their needs and expectations at the beginning of the project, but in most cases these initial requirements change during later phases of the development. As the project evolves, the requirements may change, thus new constraints may reduce the original flexibility of the software. In many problem domains, it would be beneficial to support this evolutionary nature of software development by applying model-based solutions since it allows the application of changes on a higher abstraction level without changing the lower, dependant structure. One way to solve the problem of continuously evolving requirements is the so-called model-driven engineering (MDE). However, the legacy approaches of MDE and in particular the ones of applied domain-specific modeling have often turned out not being either flexible or precise enough in practice, therefore, there is a need for new methods and techniques.

As a contemporary trend in MDE, two and/or four level modeling frameworks are being frequently extended to an arbitrary number of levels, which could lead to higher flexibility of expressiveness (Atkinson and Kühne, 2001). For a multi-level modeling approach to be rigorous enough is indeed a mandatory feature for being able to support the continuous evolution of requirements that Industry 4.0 solutions are based on. Although the emerging multi-level approaches may solve the challenge of supporting gradual refinement by providing an unbounded number of domain classification levels, this is only one aspect of the industrial demands.

Besides being able to describe the static aspect in a fine-graded way, the other key aspect is introducing dynamic features to models such as querying, modifying, simulating or executing the model definitions. In classic modeling approaches, the workflow usually consists of creating and saving the snapshot of the meta-model, and then creating the instance models based on the meta-model. One can only create the model in a very strict way, without any possibilities for dynamic modification like adding, deleting, or modifying model elements continuously. The

[a] https://orcid.org/0000-0002-4814-6979
[b] https://orcid.org/0000-0002-6168-3963
[c] https://orcid.org/0000-0003-2129-7897

Bácsi, S., Palatinszky, D. and Hidvégi, M.
Towards Evolutionary Multi-layer Modeling with DMLA.
DOI: 10.5220/0010347403440349
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 344-349
ISBN: 978-989-758-487-9

need can arise for an iterative, dynamic process where we can modify the model real-time in an evolutionary way by stepping between the different abstraction levels of the domain.

The idea of the Dynamic Multi-Layer Algebra (DMLA) (of Automation and Informatics,) was born out of the combination of these two modeling aspects, namely, multi-level modeling, and dynamic modeling. DMLA has two versions regarding dynamism: an old compiler-based version and a new, interpreter-based version. In this paper, we introduce the basics of DMLA, discuss why we needed two versions of execution engine and then demonstrate the feasibility of the new version of DMLA by a walk-through of the concrete model management steps of an illustrative example.

The paper is organized as follows: Section 2 presents the background and the related work showing other approaches with or without dynamism. Section 3 describes the basics of DMLA while Section 4 presents the short history and current state of the DMLA workbench. Section 5 presents the illustrative example and concluding remarks are outlined in Section 6.

2 BACKGROUND AND RELATED WORK

Until recently, the usual way of meta-modeling was often based on OMG's Meta Object Facility (MOF) (OMG, 2005). MOF allows the use four levels instead of two in order to increase flexibility. Although the extension is useful, having a fixed number of levels ties the hands of domain experts since they are forced to refine the model in a fixed number of steps. The lack of an arbitrary number of levels between the initial specification and the final realization may lead to the phenomenon called accidental complexity (Atkinson and Kühne, 2008). Multi-level modeling aims to minimize accidental complexity by taking advantage of an unlimited number of meta-levels in order to properly allocate the correct amount of abstraction details to each of them. In the past decades, there have been a growing number of proposals for frameworks aimed at supporting multi-level modeling (Kühne and Schreiber, 2007; de Lara and Guerra, 2010; Atkinson and Gerbig, 2016; Clark and Willans, 2012; of Automation and Informatics,).

Multi-level modeling approaches usually support dynamic behavior which requires the definition of model-modifying operations like adding a new entity to the model. A common pattern to fulfill this requirement is the introduction of programmable operations into the model either via an external or internal language. For example, Melanee (Atkinson and Gerbig, 2016) and XModeler (Clark and Willans, 2012) use an external language (variations of OCL), while DMLA (of Automation and Informatics,) uses an internal, modeled approach. Regardless of which approach we choose, the solution to execute the operations would be either compilation or an underlying virtual machine (VM) that is responsible for managing this behavior.

Following the example, XModeler(Clark and Willans, 2012) supports a virtual machine (XMF VM) for this purpose which not only supports modifying operations on the model, it also supports undoing operations based on transactions. DMLA used to be a compilation based solution but the newest implementation - like XModeler - turned towards interpretation based on a new VM technology by Oracle, called GraalVM(Oracle,) which we are discussing later in Section 4.

3 DYNAMIC MULTI-LAYER ALGEBRA

The Dynamic Multi-Layer Algebra (DMLA) (of Automation and Informatics,) is our multi-layer modeling tool under research. The framework allows the users to define and gradually refine the aspects of their domain language in a self-descriptive, validated environment. DMLA consists of two parts: (i) the Core, containing the formal definition of modeling structures and its management functions; (ii) the Bootstrap, consisting of a set of essential entities that can be reused in all domains. The main idea behind separating the Core and the Bootstrap is to improve flexibility, but also to keep the approach formal. This way, the Bootstrap becomes swappable, thus even the semantics of valid instantiation can be re-defined. Namely, each particular bootstrap seeds the metamodeling facilities of the generic DMLA formalism.

The Core has a formal description based on Abstract State Machines (ASM) (Börger and Stärk, 2003). This description defines the structure of model elements as 4-tuples i.e. tuples with four elements. Tuples contain all data of the given entity: the identifier, along with the meta-identifier of the element as well as its attributes and values. Besides these tuples, the Core also defines basic functions to manipulate the model, for example, they can create new model entities or query existing ones. These functions are defined as ASM functions. Models are represented by the underlying ASM: the states of the machine are snapshots of the models: if an entity changes, a new

snapshot and therefore a new state of the ASM is created.

Over the Core, one can define a Bootstrap (Mezei et al., 2019). The Bootstrap describes the basic building blocks of every domain model and therefore acts as a practical base for modeling. A Bootstrap consists of several entities, (i) it improves the raw entity format given by the Core to a more practical level (e.g. by attaching meta-information to references), (ii) introduces constraints to customize validation between model elements, (iii) defines the basic rules of validation, and (iv) has a complete operation language.

The operation language is an essential part of the Bootstrap, since here, unlike many other modeling approaches, the rules of valid instantiation are not encoded in an external programming language (e.g. Java), but it is modeled by the Bootstrap. All the validation rules and therefore even the definition of the instantiation relationship itself are using the operation language. The operation language models the validation algorithms by their abstract syntax tree (AST) representation built from AST-related model entities, e.g. *if* or *statement*.

In order to accelerate the definition of the tuples representing the model entities, we have created a scripting language referred to as DMLAScript. In practice, the Bootstrap, the domain metamodels and models are also defined by DMLAScript. Although it is used to ease the usage of the framework by helping users describe their domain language in a more familiar way, it does not introduce any new features. DMLAScript is merely a syntactic sugar, its scripts are always translated back to tuples, the primary form of model entities. The benefits of this scripting language are most remarkable in defining operations, where even a single line of DMLAScript code may translate to a complex set of connected entities.

4 TOWARDS EVOLUTIONARY MODEL EDITING

When creating the first workbench for DMLA, our primary goal was to create an approach that supports validated step-wise refinement. Validation was a key feature here as we wanted to achieve a self-validated, self-describing meta-modeling approach. Although the workbench fulfilled all of our initial requirements (Urbán. et al., 2018), we needed a more advanced solution. We have realized that using the operation language only to describe the validation logic is a waste of opportunities. There are more possibilities in having a fully modelled operation language. Therefore, we summarized our new requirements and decided to create a new version of the workbench.

Initially, our typical use case was to define a preliminary version of the target domain via gradual refinement, and then validate it. Then, we repeat the process several times, refine the domain step-by-step and validate it every time. In this case, it was not a problem that the model definition and the validation phases are separated. However, in a usual industrial use case, having an interactive, responsive environment is a must. Users tend to apply minor modifications, and expect the model to be up-to-date at all time. So, instead of having *milestones* as in the first workbench, we need small *steps*.

Possibly the most important new demand was to support other kinds of operations: i) queries, and ii) operations altering the model. By opening the door for operations changing the model, we needed a more dynamic view on our current model, i.e. modifications were to be reflected immediately. The first workbench could support a dynamic view but it would not be able to process the modifications as fast as it would be needed in an industrial scenario since the compilation required too much time.

By realizing the new requirements, we started to work on a new workbench for DMLA. The first, and most important decision was to create an interpreter-based workbench. There were several reasons behind this decision. Interpreters offer a natural way to obtain the inner data structure during execution and therefore it is much easier to create an interactive environment, where changes in the model are reflected automatically and immediately by the environment. Moreover, interpretation helped eliminating the long and slow compilation process and allowed us to debug domains with ease. The virtual machine behind the interpreter is also much closer to a native implementation of the underlying Abstract State Machine formalism than the compiled code produced in the old workbench.

Creating an interpreter and virtual machine for an approach like DMLA is a challenging task without a decent framework. After a long search for the perfect technology, we have decided to have GraalVM (Oracle, ; Würthinger et al., 2013) as the base of our new solution. GraalVM is a new generation of Java virtual machine. It aims to build upon the old Java VM by replacing the compiler. The main goal of GraalVM is to be a universal runtime environment with as close to native performance as possible. GraalVM also introduces Truffle (Wimmer and Würthinger, 2012), a language agnostic API that can be used to describe custom languages for Graal while also allowing for interoperability between the implemented languages. Interoperability is a powerful feature that can be used

to make function calls from a Graal-based language to another (e.g., Python to JavaScript) and exchange data between them. This feature can be considered an additional gain, which may be useful in the future, when we introduce new languages besides DMLAScript to describe the tuples.

For interpreters, performance is usually a weak point, however, Truffle supports several optimization techniques like cached function call targets, optimized execution for different parameter types, hand-optimized bytecode. It is also possible to introduce instrumentation and debugging for the target language easily.

With the help of GraalVM and interpretation, several improvements and new features became possible. One of the most important features is on-the-fly command processing by using an interactive command-line interpreter, which we are demonstrating later in Section 5. On top of that, operations are no more limited to validation and queries, modification of the model is also possible.

5 ILLUSTRATIVE EXAMPLE

In this section, we demonstrate interactive model editing in action by presenting the management steps of a simplified model fragment borrowed from the Bicycle Challenge (MULTI, 2018), for which the full solution (described in DMLAScript) is also available (Mezei et al., 2018). Originally, the typical workflow of solving the Bicycle Challenge was to define a preliminary version of the domain via gradual refinement, and then validate it. We did not have an interactive, responsive environment, thus we could not map the requirements step-by-step, in a natural way. In this example, we use a simplified syntax of our command-line interpreter for the sake of clarity. We refine the presented model fragment step-by-step based on the continuously evolving requirements:

REQ1: *A configuration (Config) is composed of components (Comps).*

```
>addEntity(Config, CEntity)
Config>addSlot(Comps,CEntity.Children)
Config.Comps>addTypeCstr(Comp)
Config.Comps>addCardinalityCstr(0,*)
```

In DMLA, multi-level behavior is supported by fluid meta-modeling. Hence, instantiation steps are independent by design. Each entity in the model can refer to any other entity along the meta-hierarchy. Entities may have attributes referred to as *slots* (abstract placeholder for the value), describing a part of the entity, similarly to classes having properties in object-oriented programming. In this example, we add a new entity *Config* to the model, setting its meta-entity to *CEntity (ComplexEntity)*, which is the usual entry point of domain definition in DMLA. *CEntity* has a slot called *Children*. *Children* enables the instantiation of custom slots, like *Comps* in this example. We add a new slot *Comps* to *Config*, setting its meta-slot to *Children*. In DMLA, one may use constraints on slots, like type and cardinality constraints. Type constraint restricts the type of the values to be put in the slot: *addTypeCstr(Comp)* adds a type constraint to slot *Comps*, thus one can only use instances of *Comp* entity there. Cardinality constraint prescribes the allowed number of instances within a given slot. The cardinality constraint has a minimum and a maximum parameter. *addCardinalityCstr(0,*)* adds a cardinality constraint to slot *Comps*, which prescribes that *Configuration* may have zero-to-many number of *Components*.

Note that when a command is executed, the model changes dynamically, therefore the latest state is always available in the background. One could even call a user-defined operation on the model which would give immediate and up-to-date results.

REQ2: *Ncycle is a configuration. Ncycle is built of components like Wheels(1..3), and Seats(1..2). Ncycle serves as an abstract entity, which can be concretized later to more specific bike models (e.g. Bicycle, Tricycle, Unicycle or Tandem bike).*

```
>addEntity(NCycle, Config)
NCycle>addSlot(Wheels, Config.Comps)
NCycle.Wheels>addTypeCstr(Wheel)
NCycle.Wheels>addCardinalityCstr(1,3)
NCycle>addSlot(Seats, Config.Comps)
NCycle.Seats>addTypeCstr(Seat)
NCycle.Seats>addCardinalityCstr(1,2)
```

In this step, we refine entity *Config*, by adding a new entity *NCycle* with the more concretized slots and the narrowed constraints. In DMLA, one may also *divide* a slot into several instances similarly to entities, where we can create several instances of a meta-entity. The general purpose *Comps* slot is divided into a *Wheels* and a *Seats* slot. Note that we also narrow the type and cardinality constraints on both *Wheels* and *Seats* slots considering the new requirement.

REQ3: *A Unicycle is a bike model equipped with exactly one wheel and one seat.*

```
>addEntity(Unicycle, NCycle)
Unicycle.Wheels>addCardCstr(1,1)
Unicycle.Seats>addCardCstr(1,1)
>Validate()
```

We further concretize entity *NCycle* by adding *Unicycle* to the model. We narrow the cardinality constraint on both slots to restrict the allowed number

of instances (one wheel and one seat), while the type constraints remain intact. We also call the validation of the model to check that what we have done so far is valid. 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. The validation checks the latest state of the model, since the model always changes dynamically, when a command is executed. In this case, no model element found to be contradictory during model validation.

REQ4: *There is a demand for special tandem bikes with two wheels and three seats.*

```
>addEntity(Tandem, NCycle)
Tandem.Wheels>addCardCstr(2,2)
Tandem.Seats>addCardCstr(3,3)
>Validate()
```

We concretize entity *NCycle* by adding *Tandem* to the model. In order to meet the new requirement we apply cardinality constraint on both slots to restrict the allowed number of instances (two wheels and three seats). The *Tandem* entity found to be conradictory during model validation, since the validation mechanism of DMLA ensures that if an existing cardinality is overwritten (refined), then it should not relax the original condition. In this example, the original maximum parameter of slot *Seats* is already set to two in entity *NCycle*, it cannot be overwritten to three. This scenario showcases one of the most important features of the interpreter-based version of DMLA: by running the validation, we can get an immediate feedback on the validation errors of latest state of the model.

Altough we could continue the refinement and the concretization of this simple model fragment (e.g. filling out the slots with concrete values), the aim of this section is only to present the most basic features of interactive model editing in the newest workbench of DMLA.

6 CONCLUSIONS

In this paper, we presented the newest workbench of DMLA, our multi-layer modeling approach, highlighting its features regarding dynamism and evolutionary model editing. Although the paper focused on DMLA, we believe that our experiences and conclusions are not DMLA-specific and are worthy of general discussion. In the future, we aim to work on the interactivity of the tool and modernize DMLAScript to improve the ease of usage. Our other goals concerning DMLA include the optimization of the validation, the visualization of the models, and the introduction of transactions to provide reliable units of work that allow correct recovery from validation errors and keep the model consistent.

ACKNOWLEDGEMENTS

This work was performed in the frame of FIEK_16-1-2016-0007 project, implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the FIEK_16 funding scheme.

REFERENCES

Atkinson, C. and Gerbig, R. (2016). Flexible deep modeling with melanee. In *Modellierung 2016 - Workshopband : Tagung vom 02. März - 04. März 2016 Karlsruhe, MOD 2016*, volume 255, pages 117–121, Bonn. Köllen.

Atkinson, C. and Kühne, T. (2001). The essence of multilevel metamodeling. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, «UML» '01, pages 19–33, Berlin, Heidelberg. Springer-Verlag.

Atkinson, C. and Kühne, T. (2008). Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359.

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

Clark, T. and Willans, J. (2012). Software language engineering with xmf and xmodeler. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, volume 2, pages 311–340.

de Lara, J. and Guerra, E. (2010). Deep meta-modelling with metadepth. In Vitek, J., editor, *Objects, Models, Components, Patterns*, pages 1–20, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kühne, T. and Schreiber, D. (2007). Can programming be liberated from the two-level style: Multi-level programming with deepjava. *SIGPLAN Not.*, 42(10):229–244.

Mezei, G., Theisz, Z., Urbán, D., and Bácsi, S. (2018). The bicycle challenge in dmla, where validation means correct modeling. In *Proceedings of MODELS 2018 Workshops*, pages 643–652.

Mezei, G., Theisz, Z., Urbán, D., Bácsi, S., Somogyi, F. A., and Palatinszky, D. (2019). A bootstrap for self-describing, self-validating multi-layer metamodeling. In Dunaev, D. and Vajk, I., editors, *Proceedings of the Automation and Applied Computer Science Workshop 2019 : AACS'19*, pages 28–38.

MULTI (2018). https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/.

of Automation, D. and Informatics, A. DMLA Homepage. https://www.aut.bme.hu/Pages/Research/VMTS/DMLA.

OMG (2005). OMG MetaObject Facility. http://www.omg.org/mof/. Accessed: 2019-03-20.

Oracle. GraalVM. https://www.graalvm.org/.

Urbán., D., Theisz., Z., and Mezei., G. (2018). Self-describing operations for multi-level meta-modeling. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 519–527. INSTICC, SciTePress.

Wimmer, C. and Würthinger, T. (2012). Truffle: a self-optimizing runtime system. In *SPLASH '12*.

Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., and Wolczko, M. (2013). One vm to rule them all. In *Onward!*