

The Challenges of Defining and Parsing Multi-Layer DMLA Models

Norbert Somogyi^a and Gergely Mezei^b

Budapest University of Technology and Economics, Műegyetem rkp. 3, Budapest, Hungary

Keywords: DMLA, Multi-Layer Parsing, Domain-specific Languages.

Abstract: Parsing is a principal step of processing languages. The nature of the language tends to determine the challenges that parsers must overcome. For example, parsing procedural or object-oriented languages often require approaches that may be specific to the targeted paradigm. For this reason, the properties of the given language may also affect the capabilities of the parser. The Dynamic Multi-Layer Algebra (DMLA) is a multi-layer metamodeling approach that aims to improve upon the flexibility of traditional meta-modeling methods. In addition, DMLA ensures rigorous validation of domain rules. In DMLA, domain models and validation logic are described in D#, a domain specific language designed specifically for this purpose. Scripts written in D# are parsed and an inner, 4-tuple representation is built. However, due to the multi-layer background and the completely modeled language, parsing DMLA models raises non-conventional difficulties that must be overcome. Reaching a satisfying balance between the expressive power and the concision of the language is also desirable. In this paper, we present the parsing process of DMLA, its peculiarities and the solutions we have employed to deal with them.

1 INTRODUCTION


The original motivation behind multi-level modeling was to create a modeling paradigm (i) to reduce accidental complexity (Atkinson and Kühne, 2008), 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 target domain tasks, rather than capturing some aspect of the 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 (Börger and Stärk, 2003). DMLA offers a highly flexible and customizable modeling structure capable of dealing

with design-time and run-time aspects of modeling. (Mezei et al., 2018; Somogyi et al., 2019)

To describe domain models, DMLA uses a domain specific language (DSL) (Fowler, 2010) called D#. A crucial step of constructing DMLA models is to parse scripts written in D# and to create a representation that conforms to the theoretical ASM formalisms of DMLA. However, due to the nature of the ASM formalism and DMLA itself, parsing and constructing models has some peculiarities and non-conventional difficulties that must be dealt with. In particular, without making the parser aware of specific cases in D# scripts, the language becomes highly complex and uncomfortable for use in practice. This paper reports on these difficulties and the solutions we have employed to deal with them.

The paper is structured as follows. Section 2 highlights the basic concepts of multi-level modeling and the way two well-known multi-level modeling tools define models in practice. Section 3 introduces the main concepts of DMLA and how models are described. Section 4 presents how operations are used to achieve dynamic behaviour in DMLA and the relations between static data and dynamic behaviour. Section 5 presents the difficulties and different aspects of constructing DMLA models. Section 6 concludes the paper and highlights some future work.

^a  <https://orcid.org/0000-0001-6908-7907>

^b  <https://orcid.org/0000-0001-9464-7128>

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) (MOF, 2005). The key idea behind multi-level modeling is *deep instantiation* (Atkinson and Kühne, 2001). 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, 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* (Atkinson and Kühne, 2000).

MELANEE (Lange and Atkinson, 2018) is a domain specific language workbench that supports visualizations of multi-level model editing. It achieves multi-level modeling by extending the traditional 2-level modeling approach of Eclipse Ecore. The visualizations can also be context aware, e.g. the background color of an entity can be dynamically determined. MELANEE uses DeepOCL to query model elements as well as impose and execute constraints upon them. Moreover, it also introduces DeepATL, a multi-level extension of the original ATL (Atlas Transformation Language). This can be used to declare and execute model transformations on existing multi-level models.

XModeler (Clark and Frank, 2020) is a multi-level modeling framework. Similarly to MELANEE, it provides a graphical editor to create models in practice. However, XModeler also supplies *FMML^X*, which is a multi-level modelling and execution language that extends Eclipse XCore. Thus, this language can be used to define multi-level models textually.

When compared to these tools, DMLA supports text-based creation of models only. In MELANEE, models are defined by using various graphical formats. In XModeler, a graphical editor and a textual language can also be used to define models. Thus, the language *FMML^X* also has to be parsed and models must be created based upon it. MELANEE and XModeler use DeepOCL and XOCL, respectively, that are textual, but are used only to declare and execute constraints upon the model or query the model, not to define the model itself. Similarly, although MELANEE uses DeepATL, it is used only to dynamically modify an existing model, not to define it. In contrast, the scripting language of DMLA is com-

pletely modeled and dynamic. This way, more rigorous and flexible validation of models may be achieved when compared to other multi-level tools. The downside of it is that this leads to several challenges that must be dealt with, which do not arise in other tools.

3 BACKGROUND

The Dynamic Multi-Layer Algebra (DMLA) is a multi-layer modeling framework inspired by the core ideas of multi-level modeling. 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 (Mezei et al., 2019). In DMLA, the model elements are called entities. The theoretical formalism of entities is defined in the Core. Every entity is represented as 4-tuples containing various information about the element: (i) the ID of the entity, (ii) a reference to the meta of the given entity, (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.

The Bootstrap (Mezei et al., 2019) 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 refinement itself. Consequently, it is possible to create different Bootstraps following different modeling paradigms. It is also the Bootstrap that defines basic modeling entities that are used to create domain models.

In DMLA, entities are similar to object-oriented classes. An entity has one explicit, direct corresponding meta and may have several indirect metas (the meta of its meta, etc). Marking an entity as its meta means a refinement relationship between the entities that has several peculiarities. For example, suppose one would like to model bicycles. In DMLA, we start at the highest abstraction level and gradually refine the concepts until a concrete product is achieved. Thus, at first, we would define an entity *NCycle*, that contains a frame and any number of wheels representing all kinds of bicycle variants (e.g. unicycles, tandem bikes, etc.). Next, a *Bicycle* would refine *NCycle* by restricting the number of wheels to exactly two. Finally, a *MyFavoriteBicycle* would refine *Bicycle* and assign a concrete frame (e.g. a mountain bike frame) and two concrete wheels as its parts.

Whenever a model entity claims another entity as its meta, the framework automatically validates if there is indeed a valid refinement between the two. Both the main validation logic and the constraints used by the validation are modeled in the Bootstrap, not hard-coded in the framework. Since the modeled validation logic defines DMLA's inherent refinement semantics, the refinement itself is Bootstrap-dependent.

Entities may contain attributes referred to as *slots*, similarly to classes having properties in object-oriented programming. Slots are basic value holders, with customizable life-cycles, e.g. they can be refined, or omitted. Refining a slot means restricting its validation rules (e.g. refining its type), assigning a concrete value to the slot, or dividing it into multiple slots.

Both slots and entities may have additional *constraints* on them. A constraint is a restriction imposed upon the element and thus it helps in restricting the semantics of the given slot, or entity. Constraints are checked automatically during model validation. The most commonly used constraints are *cardinality* and *type* constraints.

As mentioned before, in DMLA, the Bootstrap defines the basic modeling elements that are used to create domain models. The entity *ComplexEntity* is the implicit base of all future entities and its slot *ComplexEntity.Fields* is the implicit base of all future slots. We refer to domain models as the actual models that one wishes to create using DMLA. For example, the former example of bicycles is a domain model, where the entity *NCycle* refines *ComplexEntity*.

4 DYNAMIC BEHAVIOUR

In DMLA, operations are used for describing dynamic behavior and for validation logic. Even the semantics of validation and thus the constraints are defined as operations. Operation logic is composed from atomic programming constructs (e.g. conditional branch, variable declaration) just like in traditional programming languages. However, in DMLA, all of these constructs are modeled entities (e.g. we have an entity "IF"), and thus the abstract syntax tree (AST) built from operation definitions can be represented as a tree of entities. When the operation is executed, this tree built from entities is eventually interpreted and executed by a virtual machine. From this point of view, operations describing the dynamic behavior and semantics can also be handled as modeling structure. For example, operation definitions are translated into tuple-representations similarly to other

entities. Thus, the border between static data and dynamic behavior is particularly thin.

Although, entities may have operations similarly to the methods of object-oriented classes, the coupling between operations and entities are not that strong: operations are handled as global functions referenced from entities. When an operation is added to an entity, a wrapper slot is created and the operation definition is contained by this slot. Wrapping operations in slots separates its life-cycle management from the actual operation definition. This is important due to the shared usage of operations in entities. The wrapper slot makes it possible to add constraints on the operations and thus makes it possible to refine operations during refinement without affecting other entities using the operation definition. In some ways, operation refinement is similar to overriding a method in object-oriented languages. The fundamental difference is that in DMLA, the rules of the type system are parts of the rules of valid refinement and thus they are also defined in the Bootstrap. Essentially, whenever an operation refinement is to be validated, the virtual machine executes the appropriate validation operation defined in the Bootstrap. Therefore, the semantics of valid operation refinement is defined in the Bootstrap, not hard-coded in the framework. Refinement may follow the same semantics as overriding does in object-oriented languages, or it may be entirely different. For example, beyond refining the types of the parameters, it is absolutely valid to create a Bootstrap that allows the addition of parameters to an operation and thus changing its signature from an object-oriented point of view.

It should also be noted that, since operations are wrapped in hidden slots by the parser, refining an operation in fact means refining the wrapper slot as well. This is the second reason why the operation refinement of DMLA may be entirely different from that of object-oriented languages. As the connection between the original and the refining operation is not based on a naming convention, an operation may override another operation despite having a different name. Moreover, an operation may even be split into multiple operations. Taking all this into account implies that parsing D# has to deal with non-conventional difficulties that do not arise in traditional programming languages.

5 PARSING DMLA MODELS

Using the ASM formalism indirectly to create and manipulate DMLA entities would be very troublesome in practice. The reason for this is that it is a

low-level mathematical formalism that is not suitable for practical use indirectly. Instead, DMLA uses a high-level domain specific language called D# to describe models. It is essentially a high-level abstraction over the ASM formalism, but the virtual machine of DMLA relies on the tuple structure, thus the scripts need to be transformed to the tuple format. The parsing process of DMLA is illustrated in Figure 1. In order to create DMLA models, scripts written in the D# language are parsed and various artifacts are created. Firstly, the 4-tuple representations are constructed. Secondly, various auxiliary data-structures are created. Thirdly, special AST-s are built that are later used as the base for interpreting the model on a virtual machine. In terms of the concrete technology stack, the parser and the virtual machine are implemented in the Kotlin language and ecosystem and D# itself is defined using ANTLR (Parr, 2013).

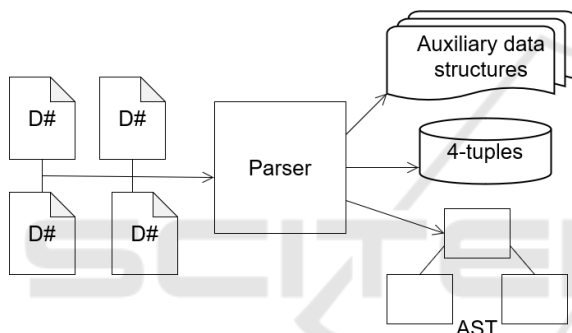


Figure 1: The parsing process of DMLA.

5.1 Defining Domain Models

Listing 1 illustrates domain models using the previously mentioned bicycle example. *NCycle* refines *ComplexEntity*, the base for all domain specific entities. *NCycle* has a slot for its wheels and a slot for its frame. Both have a type constraint and a cardinality constraint defined on them stating that every *NCycle* must have at least one wheel of type *Wheel* and exactly one frame of type *Frame*. *UniCycle* further refines *NCycle* by tightening the cardinality constraint of the wheels slot to having exactly 1 wheel. Similarly *Bicycle* refines *NCycle* to have exactly 2 wheels. It also defines an operation *CheckFrame* which takes a *Frame* as parameter and does something with it. Finally, *RoadBicycle* refines *Bicycle* by dividing the parameter of the *CheckFrame* operation into two instances of *Frame*.

As seen in this example, adding every necessary information to the script leads to an overly complicated code that is highly uncomfortable both to write and read. However, this could be alleviated by adding

```

1  entity ComplexEntity: Base
2  {
3    @Type: Base
4    @Cardinality: 0..*
5    slot Fields;
6  }
7  entity NCycle: ComplexEntity {
8    @Type: Wheel
9    @Cardinality: 1..*
10   slot Wheels:
11     ComplexEntity.Fields;
12
13   @Type: Frame
14   @Cardinality: 1..1
15   slot Frame:
16     ComplexEntity.Fields;
17 }
18 entity UniCycle: NCycle {
19   @Type: Wheel
20   @Cardinality: 1..1
21   slot Wheels: NCycle.Wheels;
22
23   @Type: Frame
24   @Cardinality: 1..1
25   slot Frame:
26     ComplexEntity.Fields;
27 }
28 entity Bicycle: NCycle {
29   @Type: Wheel
30   @Cardinality: 2..2
31   slot Wheels: NCycle.Wheels;
32
33   @Type: Frame
34   @Cardinality: 1..1
35   slot Frame:
36     ComplexEntity.Fields;
37
38   @Type: OperationDefinition
39   @Cardinality: 1..1
40   slot CheckFrameWrapperSlot
41   : Base.Operations =
42     operation void CheckFrame
43       (Frame frame) {...};
44 }
45 entity RoadBicycle: Bicycle {
46   @Type: Wheel
47   @Cardinality: 2..2
48   slot Wheels: NCycle.Wheels;
49
50   @Type: Frame
51   @Cardinality: 1..1
52   slot Frame:
53     ComplexEntity.Fields;
54
55   @Type: OperationDefinition
56   @Cardinality: 1..1
57   slot CheckFrameWrapperSlot
58   : Bicycle.CheckFrameWrapperSlot =
59     operation void CheckFrame
60       (CarbonFrame lowerHalf,
61        GoldFrame upperHalf) {...}
62 }

```

Listing 1: An example domain model defined in D#.

extra functionality into the parser. By handling specific parts of the code specially, the concision of the language could be improved at the cost of its expressive power. The loss of expressive power has

to be compensated by making the parser collect various information about the model automatically, which makes parsing slightly slower as well. Thus, a trade-off has to be made between an easy-to-use language and a simple, efficient parser.

5.2 Slots

When an entity refines another, for each slot in the meta, it must declare whether the slot is cloned (re-used without any modification), or refined. Every other slot not explicitly declared is considered omitted. Notice that much of the complexity seen in the previous example stems from requiring to declare cloned slots in all entities. Let us reverse this thought process: every slot is considered cloned unless they are marked explicitly as omitted and a slot should only be declared if they are actually refined in some way. This way, duplicating the declaration of the slots is entirely avoidable. In the example, lines 23-26, 33-36 and 50-53 could be removed. Therefore, in D#, collecting all slots in the meta entities and cloning them in the instance is the responsibility of the parser. Moreover, when refining a slot, declaring its meta is also redundant. This information can be found automatically by the parser based on a naming convention. For example, in lines 21 and 31 declaring *NCycle.Wheels* as the meta slot is no longer necessary. Naturally, meta-slots are to be declared if the name of the slot is changed during the refinement.

5.2.1 Operations

As mentioned before, entities may define operations, which are wrapped in slots. Operations must always be of type *OperationDefinition*, which is a special entity in the Bootstrap reserved specifically for operations. The cardinality of an operation is always 1..1 and the meta of an operation wrapper slot should be *Base.Operations*, unless it overrides an existing operation. *Base.Operations* is the base of all operation wrapper slots. Finally, the operation definition is set as the value of its wrapper slot. For every operation defining all this information manually is not efficient: all this information can be automatically generated when an operation definition is encountered. The parser can automatically handle this, thus wrapper slots are not needed to be handled manually. In the example, lines 38-41 and 55-58 can be avoided.

5.2.2 Overriding Operations

When an operation is refined, the meta of the refining operation's wrapper slot becomes the wrapper slot of the operation that was refined. In

the previous example, at line 64, the meta of *RoadBicycle.CheckFrameWrapperSlot* is *Bicycle.CheckFrameWrapperSlot*. However, as it is not needed to manually declare the operation wrapper slots, it is no longer possible to express in the code exactly which operation one would like to refine. For this reason, we have introduced the keyword *override*. Similarly to slots, having to explicitly declare the name of the overridden operation would be redundant (if it follows the default naming convention). Thus, it is the parser that automatically finds the original operation to be overridden.

5.2.3 Summary

The steps presented in this section can significantly reduce redundancy in the language. Listing 2 illustrates the code we obtain after applying every step. Note that the lines of code needed to define the same model have decreased from 62 to 35, which is roughly a 40% reduction. According to our preliminary test, in terms of larger and more complicated models, this reduction would be even more spectacular.

```

1 entity ComplexEntity: Base
2 {
3   @Type: Base
4   @Cardinality: 0..*
5   slot Fields;
6 }
7 entity NCycle: ComplexEntity {
8   @Type: Wheel
9   @Cardinality: 1..*
10  slot Wheels:
11    ComplexEntity.Fields;
12
13   @Type: Frame
14   @Cardinality: 1..1
15   slot Frame:ComplexEntity.Fields;
16 }
17 entity UniCycle: NCycle {
18   @Cardinality: 1..1
19   slot Wheels;
20 }
21 entity Bicycle: NCycle {
22   @Cardinality: 2..2
23   slot Wheels;
24
25   operation void CheckFrame
26     (Frame frame) {...}
27 }
28 entity RoadBicycle: Bicycle {
29   override operation
30     void CheckFrame
31       (CarbonFrame lowerHalf,
32        GoldFrame upperHalf) {...}
33 }
```

Listing 2: The reduced example model defined in D#.

6 CONCLUSION

Dynamic Multi-Layer Algebra (DMLA) is a multi-layer modeling approach aiming at providing an efficient and flexible solution for multi-level modeling. In this paper, we have briefly introduced D#, a domain specific language that we use for defining DMLA domain models. We have also presented the peculiarities and the non-conventional difficulties of the parsing process of DMLA. One of the main challenges was making a trade-off between the conciseness and expressive power of the scripting language. By omitting redundant information, the language becomes much easier and more comfortable to use. In turn, this raises the need for the parser to be able to handle numerous problematic cases automatically. When parsing slots, slots defined in the meta entities must be automatically cloned by the parser. In some cases, automatically finding a slot's appropriate meta slot is also required. In the case of operations, a container slot should automatically be generated by the parser and the operation should be wrapped in it. Overriding an operation in DMLA means further refining a meta operation. When an operation refines another, it should be marked with the *override* keyword, which signals the parser to automatically find the operation that it refines. This refinement may be semantically similar to object-oriented languages, or completely different depending on the Bootstrap. The steps efficiently reduced the redundancy in D# scripts, but sometimes at the cost of making the parser more complicated and slightly less efficient.

In terms of future work, there exist many promising aspects that could be explored upon. Analyzing the efficiency of the parser and measuring the loss of performance introduced with the reduction of the language remains our top priority. Extending the parser to support incremental parsing (Ghezzi and Mandrioli, 1979) would also be desirable. This means that instead of fully parsing scripts every time, only the difference should be processed when compared to the previous state. Dynamically changing the structure of models using operations should also be supported.

ACKNOWLEDGEMENT

The work presented in this paper has been carried out in the frame of project no. 2019-1.1.1-PIACI-KFI-2019-00263, which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2019-1.1. funding scheme.

REFERENCES

- Atkinson, C. and Kühne, T. (2000). Meta-level independent modelling. In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, pages 1–4.
- Atkinson, C. and Kühne, T. (2001). The essence of multi-level metamodelling. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, 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 Frank, U. (2020). Multi-level modelling with the fmmlx and the xmodelerml. In Bork, D., Karagiannis, D., and Mayr, H. C., editors, *Modellierung 2020*, pages 191–192, Bonn. Gesellschaft für Informatik e.V.
- Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.
- Ghezzi, C. and Mandrioli, D. (1979). Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70.
- Lange, A. and Atkinson, C. (2018). Multi-level modeling with melanee a contribution to the multi 2018 challenge.
- Mezei, G., Somogyi, F. A., Theisz, Z., Urbán, D., Bácsi, S., and Palatinszky, D. (2019). A bootstrap for self-describing, self-validating multi-layer metamodelling. In *Proceedings of the Automation and Applied Computer Science Workshop*, pages 28–38.
- 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: 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*, volume 2245 of *CEUR Workshop Proceedings*, pages 643–652.
- MOF (2005). OMG: MetaObject Facility. <http://www.omg.org/mof/>. Accessed:2021-11-07.
- Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition.
- Somogyi, F. A., Mezei, G., Urbán, D., Theisz, Z., Bácsi, S., and Palatinszky, D. (2019). 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 2019*, pages 119–127. IEEE.