Towards Mainstream Multi-level Meta-modeling

Gergely Mezei[1], Ferenc A. Somogyi[1], Zoltán Theisz[2], Dániel Urbán[1] and Sándor Bácsi[1]
[1]*Budapest University of Technology and Economics, Budapest, Hungary*
[2]*evopro Systems Engineering Ltd., Hauszmann Alajos str. 2, Budapest, Hungary*

Keywords: Meta-modeling, Multi-level Modeling, Deep Instantiation, Language Engineering.

Abstract: In recent years, a wide range of tools and methodologies have been introduced in the field of multi-level meta-modeling. One of the newest approaches is the Dynamic Multi-Layer Algebra (DMLA). DMLA incorporates a fully self-modeled textual operation language (DMLAScript) over its tuple-based model entity representation. This textual language provides effective and complex features when editing models. On the other hand, the language supports the precise injective mapping of the multi-layers. However, such complexity comes at a price. Using DMLAScript can be difficult in practice for users who are only familiar with the classical, object-oriented way of thinking, and are not confident with the multi-level approach. In this paper, we introduce the concepts of a new language above DMLAScript, which supports the more effective manipulation of concrete domain models by taking advantage of the particular process of multi-level modeling within DMLA. Although the approach is technically bound to DMLA, the concepts discussed in the paper are of general use.

1 INTRODUCTION

Meta-modeling is widely accepted as a viable technical solution to base domain-specific languages and workbenches on in order to efficiently model the selected domains in practice. In recent years, meta-model-based software engineering has also established itself in various industrial fields, such as the management of ICT systems (Rossini et al., 2015). The key enabler of this success is the ability to unambiguously model the domain concepts, the various features of a particular technology, and the relations between those concepts and features in a precise and rigorous way. Nevertheless, in some industrial projects, two level (the meta-model and the model) meta-modeling has quickly reached its limits. In such complex industrial projects, the domain requirements of the system are first specified at a high level of abstraction to give shape to the planned solution, and then the details are gradually, and sometimes recursively, being introduced. Two level meta-modeling may cope with these challenges, but only at the cost of artificially introducing accidental complexity, that is, by imposing mixed-level modeling patterns such as clabjects or power types into the design (Henderson-Sellers et al., 2013). Multi-level meta-modeling aims to achieve a better solution: by increasing the number of modeling levels, domain concepts can be defined gradually as abstraction gives way to specification, down to the instance level. The other side of the coin does not look so shiny though: the means of multi-level meta-modeling differ from classical meta-modeling, and even the exact meaning of instantiation needs to be extended or re-defined. Moreover, currently, there is still no consensus in the multi-level modeling community on how to turn research results into practical applications. An important milestone on this road was the so called Bicycle Challenge (MULTI, 2018) that aimed to condense reoccurring problems in practical models into an easily understandable, standard modeling challenge. The results show a current lack of mainstream in multi-level modeling, which is unfortunate, because it has been well-known for more than a decade that the two-levels approach will eventually reach its limit.

In this paper, we would like to contribute to bridging the gap between classical meta-modeling and the future of multi-level meta-modeling. Our research focuses on creating a general purpose multi-layer modeling framework around Dynamic Multi-Layer Algebra (DMLA) (Urbán et al., 2018) that is easily understandable by the modelers and can be effectively applied to practical domain problems in the industry. DMLA has a well-defined formal basis with the unique feature of being self-described, and it also provides what we call fluid metamodeling, which means

Mezei, G., Somogyi, F., Theisz, Z., Urbán, D. and Bácsi, S.
Towards Mainstream Multi-level Meta-modeling.
DOI: 10.5220/0007580404810488
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 481-488
ISBN: 978-989-758-358-2

that it is not required to instantiate all entities of a model at once. Models in DMLA are stored in tuples, referencing each other, and thus, forming an entity graph. Since tuples are cumbersome to produce consistently, and big quantities are needed for practical models, we have created a scripting language called DMLAScript, which enables to manipulate tuples indirectly via a domain-specific language (DSL) when building domain models. Although DMLAScript fulfilled what we aimed at originally, we had to realize that it is still challenging to be used by domain experts in practice due to its technical preciseness and verbosity. Therefore, we decided to work on a high level scripting language, the Modular DMLA Scripting Language (MDSL), that is designed to be more compact, in addition to being easier and faster to use by domain engineers.

This paper briefly reviews related work in Section 2 and introduces DMLA in Section 3. Next, in Section 4, we discuss our motivation behind MDSL, along with the requirements (Section 5) we expect it to satisfy in the future. In Section 6, we demonstrate by a case study how MDSL can accelerate domain modeling in DMLA, compared to DMLAScript. Finally, in Section 7, we conclude by paving the way for our future research, aiming for MDSL to succeed by showcasing that multi-level meta-modeling has already matured enough for practical industrial challenges.

2 RELATED WORK

The main reason behind the quest for multi-level meta-modeling is to be able to eliminate all those artificial classes, on the meta level, whose pure raison d'etre is to serve as linguistic placeholders for establishing ontological instantiation, on the model level, among the domain objects. Hence, the crux of multi-level meta-modeling is to successfully tackle the unwanted emergence of accidental complexity (Atkinson and Kühne, 2008) by clearly separating those linguistic and ontological instantiation mechanisms from each other. The most well-known solution attempt, the Orthogonal Classification Architecture (OCA) (Atkinson et al., 2009), applies the term ontological in this particular regard when it refers to the concepts which exist only in the technical domain being modelled.

Consequently, in OCA, the term linguistic thus applies to the technical modeling nomenclature of any selected modeling technology, for example, to the way how meta-meta-level elements (in M3) defined by the Meta Object Facility (MOF) are used to build domain models (in M1) through meta-model interpretation (in M2). The OCA architecture is rather intuitive and therefore there exists a few successful multi-level meta-modeling frameworks of this kind, its most notable implementations being Melanie (Atkinson and Gerbig, 2012), MetaDepth (de Lara and Guerra, 2010) and DeepJava (Kühne and Schreiber, 2007). Analyzing them independently from the prespective of their concrete syntax, it can be easily understood that semantically each originates from the common paradigm of the potency notion (Atkinson and Kühne, 2001), and therefore, they mostly differ only by their reinterpretation and implementation thereof.

Beyond OCA, an interesting alternative approach is XModeler (Clark et al., 2015), which is built on top of a self-describing meta-model, the XMF, which supports meta-modeling facilities through higher order functions in order to process the syntax and to provide a basic executable language (XOCL), following the syntax of OCL (Warmer and Kleppe, 2003). This solution is similar to MetaDepth, which has taken advantage of the Epsilon family of languages (EOL). Both benefit and get constrained by the inherited legacy of a two-level modeling language. The obvious benefit is the apparent advantage for the modeler being able to use an Object-Oriented (OO) language for model design, but the disadvantage lies in the same fact: OO features get in the way when potency or a genuine multi-level meta-model feature must be mixed. For example, in DeepJava, linguistic instantiation is built into the language per se via Java, but ontological instantiation must be managed by extending the class declaration of Java by injecting the principles of potency notion. Here, accidental complexity reappears as the necessary handling of correct potency settings at the class definition, which the programmer must not mismanage. Also, level 0, the instance level, which was never intended to be part of the program code according to the semantics of Java, will become such by the extension, which contradicts the original goal of tackling accidental complexity. Thus, accidental complexity may still be present. Both MetaDepth and XModeler are more advanced is this regard, but their languages are still hybrids: due to the legacy of their implementation, new features are difficult to put into the language without rewriting the underlying custom-designed compiler.

In DMLA, we design our languages around an initial set of entities referred to as the Bootstrap. Every scripting language of the approach is modeled, but they also have an Xtext-based editor for easy re-adaptability. We consider this an advantage compared to the above-mentioned metamodeling approaches which rely on legacy compilers. More-

over, since our standard Bootstrap is self-described, similarly to XMF, we base the usual OO-like meta-modeling features such as types, cardinality or operation signatures on the fully meta-modeled constraint concept. We consider it as our theoretical advantage since domain constraints can be introduced into custom Bootstraps as well, for example, we can introduce regular expressions by extending standard type constraints (Theisz et al., 2017). Therefore, our DSLs must be modular by design in order to establish a structurally understandable family of extendable language constructs. In this regard, we share XModeler's strategy of XOCL, but we think that our approach will enable modularity to a higher degree in semantics, and at a lower cost in syntax when complex domain models are being built.

3 THE DYNAMIC MULTI-LAYER ALGEBRA

Dynamic Multi-Layer Algebra (DMLA, 2018) 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, consisting of a set of essential entities that can be reused in all domains. In DMLA, the model is represented as a Labeled Directed Graph, where all model elements have four labels: i) the unique ID of the element, ii) a reference to its meta element by its unique ID, iii) a list of concrete values, and iv) a list of contained attributes. Besides the 4-tuples representing the model entities, there exist functions that manipulate the model graph, thus forming the Core of DMLA, which is defined over an Abstract State Machine (ASM) (Borger and Stark, 2003). 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 extends the Core by making it more usable in practice. The Bootstrap is an initial set of modeling constructs and built-in model elements (e.g., primitive types) that are needed to adapt the abstract modeling structure to practical applications of domain models. It is also worth noting that the separation of the Core and the Bootstrap allows the creation of several different Bootstraps (defining different meta-modeling paradigms), but so far, we have created one standard Bootstrap that fits our research goals.

Instantiation in DMLA has several specialties. 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. However, unlike other modeling approaches, the rules of valid instantiation is not encoded in an external programming language (e.g. Java), instead, it is modeled by the Bootstrap. All validation formulae can be modularized by being introduced directly into the Bootstrap. This even applies to constraints, like checking type and cardinality conformance (Theisz et al., 2017). Since the validation formulae directly influence the proper semantics of instantiation, the instantiation is self-defined 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 abstract syntax tree (AST) representation as tuples, which are later translated into executable code by the framework.

In DMLA, multi-level behavior is supported by fluid metamodeling. Hence, instantiation steps are independent by design. Each entity can refer to any other entity along the meta-hierarchy, unless cross-level referencing is found to be contradictory during model validation.

Entities may have attributes referred to as *slots*, describing a part of the entity, similarly to classes having properties in object-oriented programming. The concept of slots is modeled in the Bootstrap. As an example (based on our solution for the Bicycle Challenge (Mezei et al., 2018)), an entity *Bicycle* may have slots for its *Fork*, *Seat* and *Frame* components. Each slot originates from a meta-slot defining the constraints to obey to. When instantiating the entity, all of its slots are validated against the meta-slots. This is where we check the modeled type and cardinality constraints of the slot, along with other applied constraints. In our previous example, the type constraint applied to the slot *Seat* restricts the value to be an instance of the entity *Seat*.

Besides narrowing the constraints applied to a slot, one may also *divide* a slot into several instances similarly to entities, where we can create several instances of a meta-entity. For example, the *Wheel* slot may be divided into a *FrontWheel* and a *RearWheel* slot. Obviously, the number of all instance slots must not exceed the limits set by the cardinality of their meta-slot. Note that it is also possible to omit a slot completely during instantiation if it does not contradict the given cardinality restrictions.

An important feature of DMLA is that fluid meta-modeling is supported at the slot level as well: when instantiating an entity, one can decide which of its slots are instantiated and which are cloned (copied to the instance without modification). This means that we can keep some of the slots intact while concretizing the others. To continue with our example, the entity *RaceBike* is an instance of *Bicycle*. *RaceBike*

clones the definitions of the *Fork* and *Seat* slots, but it concretizes the *Frame* slot by narrowing its type constraint from *FrameComponent* to *RaceFrameComponent*. Note that this behavior reflects our way of thinking: one can gradually tighten the constraints for certain parts of the concept, without imposing any obligations on other parts of the model which are only approximately known at this time.

4 LANGUAGES OF DMLA

The 4-tuples in DMLA are the native representation of the modeled entities. Although 4-tuples enable the formal precision of meta-modeling in DMLA, producing them at scale is more than a menial task, thus, in practical modeling scenarios, their direct usage is technically impossible to be carried out. Hence, in order to create models in DMLA, we introduced a scripting language, the so called DMLAScript, which is a low-level external DSL that is used for automating 4-tuple generation at medium scale. It is worth mentioning that although DMLAScript is an external DSL with an Xtext-based workbench implementation, models written in DMLAScript are compiled to 4-tuples, and thus, the model is interpreted as genuine DMLA entities.

Moreover, we defined the standard Bootstrap in DMLAScript in order to create a self-described meta-modeling environment. Illustrating our concept design by drawing parallels with the classical theory of programming languages (see Figure 1), the 4-tuple level can be considered as the *binary code* representation of the entities in DMLA, while DMLAScript is the *assembly* language used for their production. Thus, DMLAScript must be as precise in expressing meta-level relations as DMLA itself, which has resulted in a DSL that is sometimes too verbose for managing practical domain models. Nevertheless, DMLAScript is such by design, that is, its mandate clearly stems from our explicit goal of being able to show how to define the standard DMLA Bootstrap recursively within itself, which required a DSL with very detailed and precise language constructs. However, after the standard Bootstrap had been successfully finalized in DMLAScript, we turned our attention to its direct application to practical modeling challenges, where we had to realize that DMLAScript was not suitable for these tasks. We had concluded that similarly to classical programming languages, a high-level programming language was needed to be created before modelers unfamiliar with the nitty-gritty details of DMLA and multi-level modeling could use our framework for their daily modeling tasks. Hence, by analyzing the shortcomings of DMLAScript in this regard, we started designing a new DSL around DMLA, the so called Modular DMLA Scripting Language (MDSL), which incorporates classical modeling concepts in a multi-level wrapping. Similarly to DMLAScript, we are also planning to integrate MDSL into DMLA using an Xtext-based workbench.

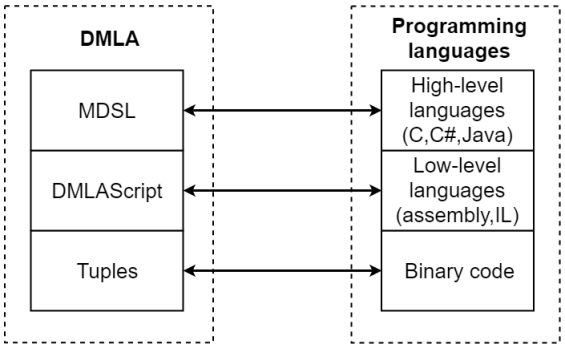

Figure 1: Layers of representation in DMLA.

To sum it up, DMLAScript would remain the low-level language of DMLA, while MDSL would become the high-level language that is to be used mainly by domain modelers. Nevertheless, MDSL would still be layered on top of DMLAScript. Hence, our goal is not to replace DMLAScript with MDSL, but to make advanced multi-level modeling concepts look familiar to domain modelers when building domain models in practice. In other words, we are trying to establish a modeling language where classical meta-modeling can be done without introducing any accidental complexity, whether it surfaces within the model or inside the language. However, whenever more precision is needed, DMLAScript can still be taken advantage of by directly injecting DMLAScript snippets into MDSL code.

5 GOALS AND REQUIREMENTS OF MDSL

As everybody knows, one of the the biggest challenges of any practical modeling technology is how to animate a potential pool of domain modelers in order to get real acceptance of its novel technological offer. We had to realize during our research studies that for DMLAScript alone it would be rather hard to attract enough attention of modeling practitioners so that it could succeed in the quite competitive market of available modeling frameworks targeted to real industrial challenges. Therefore, we do think that there is currently indeed a gap in our proposed meta-modeling

ecosystem where we are still weak on the language side although DMLA itself is firmly grounded and stable now. Hence, we believe that a language like MDSL is to be there so that mainstream DMLA domain modelers will be able to cope with their practical domain models of industrial relevance, routinely on a daily basis. Consequently, we also do believe that MDSL must mimic as many state-of-the-art OO modeling concepts and usage patterns, on the language level, as possible without ever losing grip of DMLA's formal modeling precision. In order to reach this goal of a balanced compromise in language design around DMLA, we do think that DMLAScript and MDSL must fulfill orthogonal roles. Namely, DMLAScript shall remain DMLA's main modeling language for academic research by maintaining its ability to introduce, on the language level, any novel meta-modeling concepts directly into the bootstraps and thus to precisely define their mathematical formalism as tuples in their fully reflective multi-level modeling environment. However, MDSL shall streamline, on the language level, all those sets of well-researched practical DMLA meta-modeling concepts that may effectively contribute to the mainstream modelers' daily routine and thus it will optimize bootstrap validation by partially denouncing full reflectivity of the meta-model for a faster validation execution. In order to achieve this global goal, we have set the following technical requirements on the concrete syntax of MDSL:

1. *Structuralism and modularity.* The syntax of MDSL shall showcase both structuralism and modularity in order to satisfy all practical needs of domain modelers. Nevertheless, this duality must always be balanced by language constructs with semantically equivalent alternatives. Namely, the modeler must be able to select the best trade-off based on the complexity and the quantity of the entities being modeled.
2. *Instantiation and inheritance.* The syntax of MDSL shall clearly distinguish between instantiation and inheritance as closely as possible to the classical understanding of state-of-the-art object-oriented principles. In order to satisfy this requirement, we will precisely define and map the concept of inheritance onto DMLA.
3. *Customizable constraints.* DMLA syntax for modularity shall always be based on precisely meta-modeled constraints in Bootstraps, and although their syntax may vary it shall always be as orthogonal as possible to other similar language constructs. This is the enabler to introduce new features into the language at ease such as ownership handling, two-way navigation, version handling etc. Relationships that require master-slave dynamics (owner and owned) are more difficult, due to their needed verbosity, to be expressed in DMLAScript, thus, MDSL shall support them on the language level.

In order to impose the aforementioned requirements on MDSL, but also not to make the modeling process inflexible or inconvenient, we are working on an editor with smart refactoring mechanisms. The main refactoring requirements and the corresponding editor features are as follows:

1. *Explicit omission or specification of cloned slots.* MDSL shall support the explicit omission of slots so that domain modelers could specify which slots have to be omitted. On the other hand, domain modelers should also be able to specify the inherited slots. The editor shall support copying of certain DMLA items automatically into the current entity from its instantiated or specialized meta-entity.
2. *Relocation of features.* It can be uncomfortable to relocate features along the metahierarchy upwards, or downwards. Updating the references to meta elements where the child elements are defined at should be automatically maintained.
3. *Mandatory in-between placeholders.* There may be entities that must be instantiated and/or specialized in large numbers across meta-levels. It can be inconvenient to manually copy these entities onto their appropriate meta-levels. The editor shall provide support for extending selected entities towards upper and lower meta-levels.

6 CASE STUDY

In this section, we explain some of the planned features of MDSL through a practical example. By designing MDSL, our primary goal is to simplify the currently still laborious creation of multi-level models in DMLA, and thus, to make DMLA available as a viable modeling option for practitioners without demanding special multi-level modeling skills. We demonstrate the main language features of MDSL, in comparison to DMLAScript, on a case study borrowed from the Bicycle Challenge (MULTI, 2018), for which the full solution (described in DMLAScript) is also available (Mezei et al., 2018). For the sake of clarity, we have slightly simplified the model. For example, we focus on the entity structure and thus operations are omitted. Moreover, only those parts of the entities are displayed that are necessary to illustrate the mechanisms of MDSL.

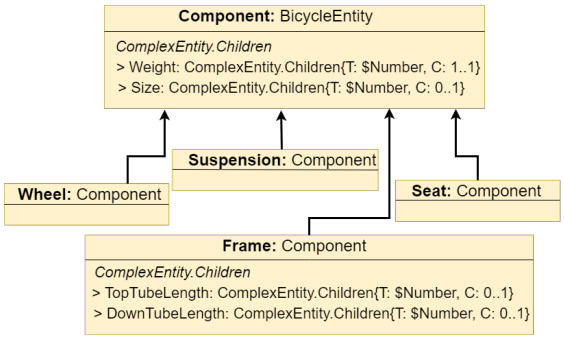

Figure 2: The case study fragment of the Bicycle challenge.

Figure 2 depicts the simplified example. The *Component* entity is our starting point of the example. The *ComplexEntity.Children* slot originates from the upper part of the meta hierarchy, defined in the Bootstrap. *Children* enables the instantiation of custom slots, like *Weight* and *Size*, by dividing the original slot into several instances as discussed in Section 3. We could have chosen to omit the *Children* slot, but here we clone it so that entities instantiated from *Component* will also be able to define their own custom slots. The *type* and *cardinality* constraints can be expressed using a more compact syntax compared to DMLAScript. Note that the syntax (and concepts) of the language are still subject to change until we have fulfilled all our requirements outlined in Section 4.

```
Component:+ BicycleEntity {
  $ComplexEntity.Children;

  [type = $Number]
  [card = [1..1]]
  Weight: $ComplexEntity.Children;

  [type = $Number]
  [card = [0..1]]
  Size: $ComplexEntity.Children;
}
```

To show the difference between DMLAScript and MDSL, the original definition of the *Component* entity in DMLAScript must be taken into account. It is more precise, but this level of preciseness is not usually needed when designing domains. As it can be seen, this script contains much more technical details. In this example, we are going to showcase only a few features of MDSL that make writing scripts in a more concise form easier. Omission of the slots is not a feature in DMLAScript, only cloning (explicitly listing every slot to be kept) is allowed. Thus, in some cases, this results in longer scripts. The type and cardinality constraints in DMLAScript (*@T:...*, *@C...*) are very verbose and contain code that is needed for the inner functions of the bootstrap to work, like the *IsOverwritable* and *IsPermanent* slots. These are not present in MDSL, instead, the default values are used when code is generated from the MDSL script. Moreover, the minimum and maximum values of the cardinality is expressed in a more concise form when using MDSL, by giving a range between the two numbers. It is worth noting that the editor requirements of MDSL (like feature relocation) are still work in progress and are not present in this example.

```
Component: BicycleEntity
{
  ComplexEntity.Children;

  @T: ComplexEntity.Children.T =
  Type: ComplexEntity.Children.T.T
  {
    Type.IsOverwritable;
    Type.IsPermanent;
    slot Type:
      ComplexEntity.Children.T.T.T=$Number;
    slot IsInclusive:
      ComplexEntity.Children.T.T.IsIncl=false;
  };
  @C: ComplexEntity.Children.C =
  Card: ComplexEntity.Children.C.Card {
    Cardinality.IsOverwritable;
    Cardinality.IsPermanent;
    slot Min:
      ComplexEntity.Children.C.Card.Min=1;
    slot Max:
      ComplexEntity.Children.C.Card.Max=1;
  };
  slot Weight: ComplexEntity.Children;

  @T: ComplexEntity.Children.T =
  Type: ComplexEntity.Children.T.T
  {
    Type.IsOverwritable;
    Type.IsPermanent;
    slot Type:
      ComplexEntity.Children.T.T.T=$Number;
    slot IsInclusive:
      ComplexEntity.Children.T.T.IsIncl=false;
  };
  @C: ComplexEntity.Children.C =
  Card: ComplexEntity.Children.C.Card {
    Cardinality.IsOverwritable;
    Cardinality.IsPermanent;
    slot Min:
      ComplexEntity.Children.C.Card.Min=1;
    slot Max:
      ComplexEntity.Children.C.Card.Max=1;
  };
  slot Size: ComplexEntity.Children;
}
```

In MDSL, slots can be explicitly kept (cloned) or omitted. The *Wheel*, *Suspension*, and *Seat* entities all instantiate *Component*. They all clone the same slots: *Weight* and *Size*, while omitting *Children*. In the case of *Seat*, we explicitly omit the listed slots, while in the

other entities, we list the slots that are cloned. This showcases two alternative ways of entity definition in MDSL: when using the ":-" operator (omit mode), we have to define the list of omitted slots, all other slots are cloned. In contrast, when using the ":+" operator (clone mode), we have to define the list of cloned slots, all other slots are omitted. Note that according to the semantics of DMLA, only slots with optional cardinality can be omitted (like *Children* in our example), otherwise, the cardinality constraint would be violated. Allowing both modes makes MDSL more balanced and flexible. In DMLAScript, our only option is to list all the slots that we want to keep.

The constraints of the slots of *Component* is defined using an annotation-like syntax, but – like many other concepts in MDSL – there is an alternative syntax we can use, which is illustrated in the *Frame* entity. Both are significantly shorter than their DMLAScript counterpart.

```
Wheel:+ Component {
  Component.Weight;
  Component.Size;
}

Suspension:+ Component {
  Component.Weight;
  Component.Size;
}

Seat:- Component {
  $ComplexEntity.Children;
}

Frame:+ Component {
  Component.Weight;
  Component.Size;
  $ComplexEntity.Children;

  TopTubeLength: $ComplexEntity.Children {
    type = $Number;
    card = [1..1];
  }

  DownTubeLength: $ComplexEntity.Children {
    type = $Number;
    card = [1..1];
  }
}
```

Unlike other components, the *Frame* entity introduces two new slots besides keeping (cloning) *Weight*, *Size* and *Children*. In the example, we use the ":+" operator (clone mode) and list every slot. Obviously, we could have used the ":-" operator (omit mode), and have enumerated no slots if we had wanted to achieve the same effect. The two new slot concretizations are defined the same way in both modes.

Although the illustrated example is simple, we believe that it demonstrates that MDSL is a more compact and more intuitive language than the current version of DMLAScript, therefore, it can be used easier by domain experts when building domain models. As the language is still under construction, its concrete syntax is not finalized yet. Therefore, we have not carried out any a public test yet, but plan to do so. Until then, we have tested the language features by re-implementing the complete Bicycle challenge in MDSL (without the operations). The efficiency gain is remarkable: we experienced a serious boost in speed and compactness. Preliminary results show that as opposed to the ˜1500 lines of code required to solve the challenge using DMLAScript, we only needed ˜700 lines of code using the new language. As a caveat, the line numbers do not include operation definitions, which are not yet supported in MDSL. There are other factors to be considered (like the expressiveness of the language), but we do believe these results are promising for the future.

7 CONCLUSIONS

In recent decades, meta-modeling techniques have been successfully used in many industrial projects, but their usage is still limited by the lack of support for the step-wise refinement of requirements in an organic way. Multi-level modeling aims to solve this issue, but there is still no consensus in how multi-level modeling should work. A few years ago, our research group was formed to create a new multi-layer modeling approach that is formal, flexible, and self-describing. This new approach is now referred to as the Dynamic Multi-Layer Algebra (DMLA). From the beginning, DMLA had always been meant to be self-validated and modular. Once we had introduced modeled operations and fully modeled the validation, we achieved our original goal. However, we had also realized that the cost of precision is payed by verbosity and complexity. Trying to balance the odds, we are currently working on a new language, the Modular DMLA Scripting Language (MDSL) that is meant to be a viable, easy-to-use solution for multi-level domain engineering, unifying the advantages of OOP-like design and multi-level modeling. As we are at the beginning of our research programme, in this paper, we have compared MDSL to our original scripting language, DMLAScript to show the relevance of our concerns. However, our research goal is to analyze the plethora of meta modeling languages commonly used in research and industry and adapt our language(s) to the precise requirements set by the community. Therefore, in order to share our vision, we have both described the main steps of our previ-

ous research and presented the requirements for our new modeling language. Even though the MDSL language is not complete yet, we have presented a simplified case study to illustrate the foreseen language mechanisms. Currently, we are busily working on an Xtext-based implementation of MDSL and eagerly waiting for case studies in order to validate the language though practical surveys with modeling practitioners. According to our preliminary investigations, modeling in MDSL is about twice as compact as in DMLAScript, while at the same time it is much easier to explain, read, produce or maintain the models. Due to the different abstraction levels, we expect similar ratios in more complex case studies as well. Obviously, MDSL has its own limits: it can be used for domain engineering, but not for manipulating the underlying Bootstrap.

Finally, as one of our main requirements imposed on MDSL, we are also working on how to map classical OOP-like features such as interfaces or inheritance to MDSL in order to increase its productivity even further. An additional unforeseen benefit of introducing MDSL has been that it has redirected our focus back to DMLAScript again. It seems that there are some DMLAScript features we can further optimize towards practicality. Thus, we are also planning to carry out surveys with domain modelers and developers in order to classify the most salient features of MDSL and DMLAScript so that in the future the language aspect of feature modularity could characterize all the DSLs around DMLA.

ACKNOWLEDGEMENT

The project was funded by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

REFERENCES

Atkinson, C. and Gerbig, R. (2012). Melanie: Multi-level modeling and ontology engineering environment. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, pages 7:1–7:2, New York, NY, USA. ACM.

Atkinson, C., Kennel, B., and Gutheil, M. (2009). A flexible infrastructure for multilevel language engineering. *IEEE Transactions on Software Engineering*, 35:742–755.

Atkinson, C. and Kühne, T. (2001). The essence of multilevel metamodeling. In Gogolla, M. and Kobryn, C., editors, *ŁUML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 19–33, Berlin, Heidelberg. Springer Berlin Heidelberg.

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

Borger, E. and Stark, R. F. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin, Heidelberg.

Clark, T., Sammut, P., and Willans, J. S. (2015). Superlanguages: Developing languages and applications with XMF (second edition). *CoRR*, abs/1506.03363.

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.

DMLA (2018). https://www.aut.bme.hu/pages/research/vmts/dmla.

Henderson-Sellers, B., Clark, T., and Gonzalez-Perez, C. (2013). On the search for a level-agnostic modelling language. In Salinesi, C., Norrie, M. C., and Pastor, Ó., editors, *Advanced Information Systems Engineering*, pages 240–255, 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.

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

Rossini, A., de Lara, J., Guerra, E., and Nikolov, N. (2015). A comparison of two-level and multi-level modelling for cloud-based applications. In Taentzer, G. and Bordeleau, F., editors, *Modelling Foundations and Applications*, pages 18–32, Cham. Springer International Publishing.

Theisz, Z., Urbán, D., and Mezei, G. (2017). Constraint modularization within multi-level meta-modeling. In Damaševičius, R. and Mikašytė, V., editors, *Information and Software Technologies*, pages 292–302, Cham. Springer International Publishing.

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.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition.