Towards a Visualization of Multi-level Metamodeling Techniques

Sándor Bácsi and Gergely Mezei
Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Magyar tudósok krt. 2., H-1117 Budapest, Hungary

Keywords: Multi-level Modeling, Metamodeling, Dynamic Instantiation, Visual Language, Visualization.

Abstract: In the recent decade, a wide range of tools and methodologies have been introduced in the field of multi-level metamodeling. One of the newest approaches is the Dynamic Multi-Layer Algebra (DMLA). DMLA incorporates a fully self-modeled textual operation language above the tuple-based model entity representation. This textual language simplifies editing the models, but it has its drawbacks especially in following evolving requirements. In this paper, we introduce a visualization concept, which can support the more effective manipulation of the particular model and facilitate the process of multi-level metamodeling within 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. Nevertheless, the most relevant design decisions must be made at this early stage of the development leaving concrete implementation later.

In the early phase of the development, the software model is flexible because of the high-level specification. As the project evolves, the requirements may change, thus new constraints and properties may reduce the original flexibility of the software model. In many domain problems, it would be useful to support this evolutionary nature of software development by applying model-based solutions.

Mainstream metamodel based methods and techniques have been applied in real industrial cases several times, for example in the field of IoT and management of telecommunication systems. It turned out that these approaches may have their drawbacks, since several problem settings cannot be solved expressively and flexibly with classic metamodeling techniques. Most of the current metamodel based solutions have difficulties with supporting the evolution of the domain-model during the lifecycle.

One of the new branches of metamodeling focus on multi-level metamodels. By increasing the number of modeling levels, we can obtain an environment, where the definition of domain concepts are composed of fine graded steps simplifying thus the customization of the domain. However, there is no consensus in the literature on the exact meaning of the methodology and there are no widely accepted principles, which makes practical application hard.

In the recent years, several new approaches and methodologies have been introduced or suggested by researchers to support the process of multi-level metamodeling. One of these approaches is the Dynamic Multi-Layer Algebra (DMLA), which is a flexible, self-validation and formal multi-level modeling framework. In DMLA, all model elements are stored as 4-tuples and all operations are applied on these tuples. DMLA incorporates a fully self-modeled textual operation language (DMLAScript) above the 4-tuple representation, which provides a user-friendly interface to reach and manipulate the tuples. Due to the initial concepts and the textual representation, DMLAScript has its drawbacks in the effective manipulation of the domain model, especially when focusing on realistic, evolutionary model editing. In this paper, we point out these drawbacks and introduce a visualization concept, which can facilitate the process of multi-level metamodeling on a higher abstraction level. Although the concept is related to DMLA, the main principles are more general and can be beneficial for any multi-level visualization environment.

The paper is organized as follows: Section 2 presents the background and the related work. Section 3 introduces the DMLA approach in order to give

Bácsi, S. and Mezei, G.
Towards a Visualization of Multi-level Metamodeling Techniques.
DOI: 10.5220/0006915603550362
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 355-362
ISBN: 978-989-758-320-9
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

clear and precise understanding of the main concepts. Section 4 elaborates our method, while concluding remarks are outlined in Section 5.

2 RELATED WORK

In this section, we summarize the most relevant approaches in the field of multi-level metamodeling and we give a general overview on advantages of visual languages in different problem domains. We also summarize existing visualization approaches regarding multi-level metamodeling.

2.1 Multi-level Metamodeling Approaches

OMG's Meta-Object Facility (OMG, 2005) (MOF) is often referred to as the de-facto standard for implementing metamodel based solutions. MOF provides a four-layer architecture, which can be satisfying for most of the problems. However, alternative as multi-level solutions pointed out (Atkinson et al., 2014), it is not always enough neither in flexibility nor focusing on the preciosity. To overcome the weaknesses of MOF, n-level metamodeling has gained increasing popularity in the last decade. Here, it is worth to explicitly differentiate between linguistic and ontological metamodeling based upon separate linguistic and ontological instance-of relations. Several approaches also differentiate between shallow and deep instantiation. Shallow instantiation means that the domain information is available exactly one modeling layer below its definition. In opposition to shallow instantiation, deep instantiation means that the domain information may be used at layers below as well according to its parametrization.

One of the most relevant deep instantiation techniques is the so called potency notion (Atkinson and Kühne, 2001). Here, a potency is assigned to each model entities. The assigned potency value represents the number of model levels the certain element can get through before reaching its fully instantiated state. Melanee (Atkinson and Gerbig, 2016), (Gerbig et al., 2016) is a deep modeling tool based upon the concept of orthogonal classification and potency notion.

Another remarkable deep modeling methodology is the Lazy Initialization Multilayered Modeling Framework (Golra and Dagnat, 2011). This framework provides an object oriented modeling language, which is based on lazy instantiation. Lazy initialization can facilitate to manage the instance-of relationships between the layers and their classification.

The (metaDepth (de Lara and Guerra, 2010) and XModeler (XModeler, 2014)), are also multi-level metamodeling approaches. Besides the modeling structure, they provide an operation language as well, which allows the creation of operations within the multi-level metamodeling workbench. In metaDepth, both Java and EOL (EOL, 2007) can be used to specify actions and constraints. XModeler provides an executable programming language, XOCL (Clark and Willans, 2012), which is based on OCL.

2.2 The Benefits of Visualization

Domain-specific languages (DSLs) are specialized modeling languages that allow the effective management of the behavior and the structure of software programs and systems in a specific domain. DSLs provide an improved abstraction of the problem that allows the rapid development of the targeted domain. Visual DSLs, compared to textual DSLs, can further improve the expressiveness and the usability of the model.

There are several advantages of using a visual DSL. The richness of the visual representation can simplify the modeling process and increase flexibility, thus visual DSLs can be intuitively usable even for complex language constructs. Visual DSLs also provide high-level abstractions to manipulate the structure and the functioning of the given model. On the other hand, visualization can facilitate to understand the concepts and the main relations in the targeted domain. It can be easier to explain the main characteristics of a domain problem by using visual notations.

There are visual DSLs e.g., (Simon et al., 2017), (Bottoni and Ceriani, 2015) which can ensure that users without technical skills can be more closely involved in the targeted domain. In other domains (Wienands and Golm, 2009), it can be beneficial to use the combination of a textual and a visual DSL. The visual form of the well-known UML diagrams can facilitate the customization of the modeling elements and interactions in a more flexible way, especially in design-time.

To sum up, a visual DSL can be beneficial to represent and manage existing abstractions. Each problem domain requires a different visual representation to meet the requirements of the targeted domain, therefore it is essential to choose the most appropriate representational concepts in design-time. Obviously, not all domain problems can be solved expressively and flexibly using visual DSLs. Hence, it is worth considering the use of visual DSLs depending on the characteristics of the targeted domain.

Recently, the demand for an appropriate visualization has arisen also in the field of multi-level metamodeling. XModeler (Clark et al., 2015) provides a visual editor to support the process of multi-level modeling. As a concrete syntax, there is a named box notation in XModeler's visual editor, which represents the class of the given entity. Hence, it is possible to navigate among instances and class declarations. XModeler has a color node notation as well in order to be able to distinguish the levels of type abstraction. The header of each class has a specified background color to express the appropriate level of type abstraction.

Melanee provides a diagrammatic workbench to facilitate the process of deep-modeling. The Melanee workbench uses clabject (Gerbig et al., 2016) notations and provides a wide range of tools to build a deep-model. The main advantage of the Melanee workbench is the dynamic display of the currently usable types. The deep-model can be edited by using both the textual and the visual editor.

3 DMLA IN A NUTSHELL

Dynamic Multi-Layer Algebra (DMLA) (DMLA, 2015) is a multi-level modeling framework based on the Abstract State Machines (ASM) (Egon Brger, 2003) formalism. DMLA consists of two major parts: (i) the *Core*, a formal definition of the modeling structure and its management functions; (ii) the *Boostrap*, an initial set of pre-defined modeling entities.

Each model is represented as a Labeled Directed Graph, where all model elements have four labels. Table 1 shows the labels with their short descriptions.

Table 1: 4-tuple representation.

Label	Description
X_{ID}	globally unique ID of model element
X_{Meta}	a reference to the meta of the element
X_{Values}	a list of concrete values
$X_{Attributes}$	a list of contained attributes

Besides the 4-tuples representing the model entities, there exist functions to manipulate the model graph, for example, to create new model entities. These definitions form the *Core* of DMLA, which is specified over an Abstract State Machine (ASM). Thus, in DMLA, the states of the state machine are snapshots of the dynamically evolving models, while transitions (e.g. deleting a node) represent modification actions between those states.

The elements of the *Boostrap* can facilitate the adaptation of DMLA's modeling structure to existing 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 modeling entities of the current bootstrap (Fig.1) can be categorized into four groups: (i) basic types (blue boxes) providing a basic structure for multi-level metamodeling, (ii) built-in types (purple boxes) representing the primitive types available in DMLA, (iii) entities facilitating the introduction of operations into DMLA (green boxes), and (iv) validation related entities (red boxes).

For the sake of compactness, we give only a short overview of the modeling structure here. For more details check the DMLA website (DMLA, 2015). Basic entities are the basic building blocks of the modeling process. The most important among them is the *Base* entity, which is at the root of the meta hierarchy. Hence, all other model entities are instantiated from *Base* entity. *ConstraintContainers* (and the Constraints contained) are used to customize instantiation validation. At this point, we should discuss the validation mechanism: 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. The validation formulae can be modularized by introducing them directly into the Bootstrap. Since these formulae directly influence the actual semantics of instantiation, every model validation gets modularized and DMLAs instantiation becomes effectively self-defined by the model per se. The *SlotDef* entity is a direct instantiation of Base. It is used to define slots, which represent substitutable properties, in syntactically similar manner to class members in OO languages. Slots can contain *ConstraintContainers*, which grants them the capability to attach constraints to the containment relations defined by the slot. Base defines that entities can have slots, thus any direct or indirect instance of Base will inherit this behavior. The *Entity* entity is another direct instance of *Base*. *Entity* is used as the common meta of all primitive and user-defined types. *Entity* has two instances: *Primitive* (for primitive types) and *ComplexEntity* (for custom types).

The core entities needed to represent the universes of ASM in the bootstrap are: *Bool*, *Number* and *String*. All these types refer to sets of values in the corresponding universe. For example, we create entity *Bool* so that it could be used to represent Boolean type expressions. Built-in types are relied on when a slot is filled by a concrete value and that value is not a reference to another model entity, however it is

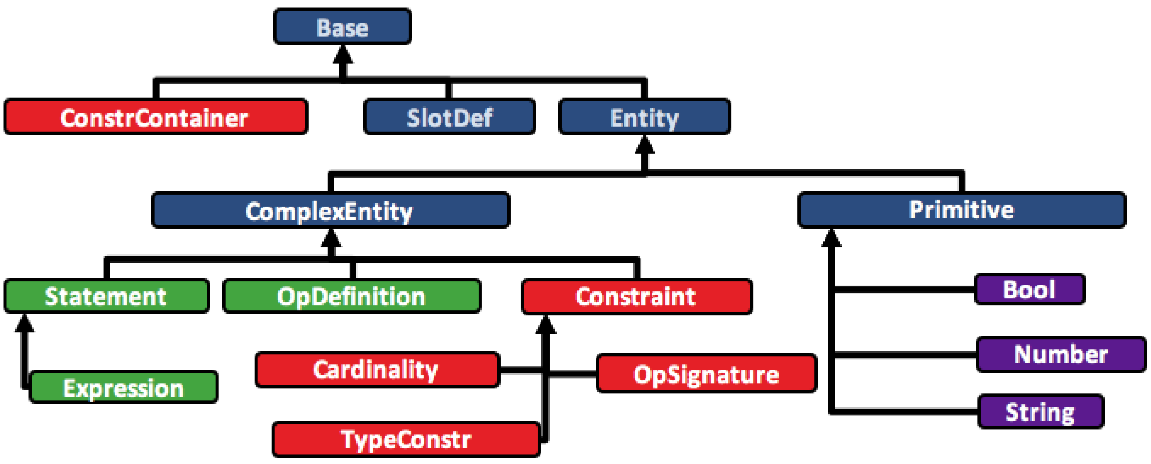

Figure 1: The elements of the current Bootsrap.

a primitive, atomic value. All built-in types are instances of *Primitive*.

Operations are responsible for changing the behavior of the entities in their sub-branches of the meta-tree. In other words, new entities within the model may provide their own specialized definition of valid instantiation, provided they do not contradict the standard validation rules imposed by *Base*.

DMLA incorporates a fully self-modeled textual operation language above DMLA's 4-tuple representation, DMLAScript, which has its drawbacks in the manipulation of the operations within DMLA. In this paper, we introduce a visualization concept, which can facilitate the process of multi-level metamodeling in DMLA.

4 TOWARDS VISUALIZATION

In this section, we summarize the most relevant concepts regarding the visual workbench for DMLA. In the first subsection, we show the justification of the visualization, while the main concepts are outlined in the latter subsections.

4.1 The Justification of the Visualization

As the number of multi-level metamodeling approaches is increasing, there is growing debate in the literature about the exact meaning of the multi-level metamodeling methodology and there are no widely accepted principles. Hence, there is no de-facto standard for multi-level metamodeling and every approach has different capabilities. This is the main reason why the multi-level committee has created the so called Bicycle challenge (MULTI 2017, 2017) to investigate the strengths and drawbacks of the different multi-level approaches. The challenge is also intended as a basis for demonstrating the benefits of multi-level modeling. Recently, we have resolved the Bicycle challenge, to investigate the capabilities of DMLA. While working on this challenge with DMLAScript, we have realized several drawbacks of the practical application of DMLAScript. Different scenarios show different problems regarding the use of the textual DMLAScript. In the following, we summarize the relevant scenarios and drawbacks which we have investigated.

The drawbacks can be categorized into two groups: (i) The management of the relations among entities; (ii) Refactoring and editing of certain entities. Group (i) contains drawbacks which affect the management and the manipulation of the relations among entities. Group (ii) contains drawbacks regarding the effective editing of certain entities.

4.1.1 Managing the Relations Among Entities

Here, we present the drawbacks regarding the management of the relations among entities:

- *Lack of fragmentation:* In a textual description document of the model, it can be inconvenient to trace relations between modeling entities based upon specification or the containment because of the lack of fragmentation and structuralism.
- *Navigation through meta-hierarchy:* In design-time, it can be essential to navigate up or down on the relevant part of the hierarchy in order to get more information about a certain entity. In a textual description, it is not possible navigating through the meta-hierarchy in a seamless way, therefore the modeling process can be uncomfortable in design-time.

- *Semantics based grouping:* Having met with a large number of modeling entities can be confusing and impenetrable. It would be advantageous to have a precise view of the relevant entities in design-time.

A visual DSL can provide a set of features to solve the problems mentioned in this section. The *Lack of fragmentation* and the *Navigation through meta-hierarchy* can be easily resolved, because a well-established concrete syntax of a visual DSL can provide structuralism and facilitate the seamless navigation through the meta-hierarchy. Different views and filters can display the expected set of modeling elements, thus *Semantics based grouping* can be resolved by applying appropriate views.

4.1.2 Refactoring and Editing Certain Entities

Here, we present the drawbacks regarding the refactoring and editing of certain entities:

- *Relocating the features:* In design-time, it can be important to move some parts (e.g. the children of a certain entity.) up or down on the hierarchy. It can be uncomfortable to update references to meta elements the children elements are defined at.
- *Keep flexibility:* It is easy to forget to keep the ability to add any kind of children to entities especially in a deep meta-hierarchy.
- *Mandatory in-between placeholders:* There can be elements, which must be cloned (or instantiated) by a large number of meta-levels. It can be inconvenient to manually copy these elements to the appropriate meta-levels. It would be useful to have a feature to extend particular elements to upper and lower levels.

Automation and refactoring can be used both in a textual and visual DSL, thus all of these drawbacks can be resolved. Nevertheless, these issues are rather editor dependent problem settings.

4.2 The Visualization of the Meta-hierarchy

Our first approach in creating the visual language to edit DMLA models was to create a simple demonstration for the visualization of the meta-hierarchy by constructing a tree of the model showing the instance-of hierarchy. We created a visual DSL using the Eclipse Modeling Framework (EMF, 2016) and Sirius (Sirius, 2018). With the help of the modeling framework of the EMF project we built the metamodel of the visual DSL. Sirius allows to effectively manipulate a certain part of the model graphically in order to focus on what really matters. Sirius can simplify the graphical representations and highlight the elements of interest. These are the main reasons why we decided to use Sirius, because these features can facilitate the more effective manipulation of the particular model. Fig.3 shows the relevant part of the Bicycle challenge.In this simple demonstration, we used a rectangular box notation for each entity. The meta-entity can be set for each entity (*Entity:MetaEntity*). Besides the header label, the instantiation relationship is visualized by an arrow as well. Slots of the entities are represented as *Slot:MetaSlot*. Note that the metaslot is also important, since it determines the validation rules of the slot.

Although it looks beneficial to visualize the whole meta-hierarchy, it has its limitations and drawbacks. Due to a large number of model entities, the depth of the instance tree can be large, thus the diagram may become over-complicated and it can make the modeling process harder in design-time. On the other hand, there are containment relationships in the model as well, which should also be visualized in order to get a better overview of the model. By realizing the shortcomings of our first attempt, we re-built our visual language from scratch.

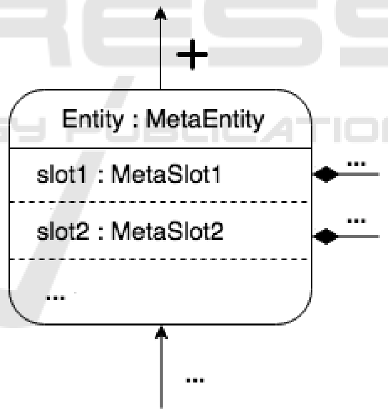

Figure 2: Expansible visual pivot point.

4.3 Combining Meta-, and Containment Hierarchy Views

We have created a new visualization concept which supports both the meta-hierarchy view and containment view in a more flexible and expressive way. As it is not usable to display the whole model in one instance tree, visual pivot points are needed in order to make the diagram extensible vertically by the meta-hierarchy and horizontally by the ownership. Hence, it is possible to navigate only on the relevant part of the model by traversing the appropriate upper and

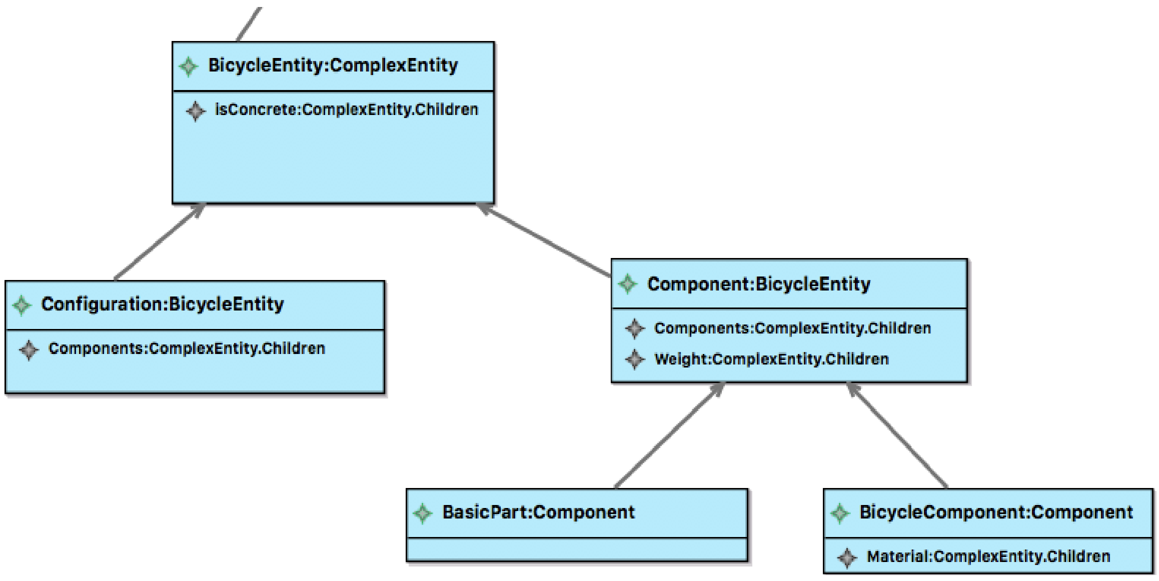
BicycleEntity:ComplexEntity
isConcrete:ComplexEntity.Children
Configuration:BicycleEntity
Components:ComplexEntity.Children
Component:BicycleEntity
Components:ComplexEntity.Children
Weight:ComplexEntity.Children
BasicPart:Component
BicycleComponent:Component
Material:ComplexEntity.Children

Figure 3: The visualization of the meta-hierarchy.

lower levels and displaying entities by containment relationships. Fig.2 illustrates the concept of combining meta-hierarchy and containment views. We believe that both *Lack of fragmentation* and *Navigation through meta-hierarchy* drawbacks can be resolved with the combination of meta-hierarchy view and containment view.

4.4 Package View

In order to make the visual representation more modularized and structured, a new view is required. Instead of following the structural connections as in case of instantiation and containment relations, the package view reflects a logical grouping among model elements. The following advantageous points can facilitate the modeling process by applying a package view:

- The package view can be used to to categorize particular entities in the model, therefore they can be easily maintained and used.
- Built-in bootstrap entities and user-defined entities can be distinguished by applying the appropriate concrete syntax (i.e. using different color notations)
- Logically related modeling entities can be encapsulated in a named package. Each package is easy to understand, and the interface between packages can be simple, clear, and well-defined.
- The relevant entities can be displayed easily, non-relevant entities can be hidden.

To sum up, all of the aforementioned points strengthen the need for package view. This is the reason why we support the introduction of package view in our visualization concept. The *Semantics based grouping* drawback can be easily resolved by using an appropriate package view.

4.5 Testing the Approach

In this subsection, we present a simple example to give a better overview of the advantages of our visualization concept. Let us follow the process of designing a domain step-by-step.

1. *A bicycle is built of components. A component is a bicycle entity.* – We create an entity called *Component* by instantiating the previously created *BicycleEntity* entity.
2. *There are several types of components, e.g.* Frame *or* Wheel. –We set our pivot to *Component* and instantiate it to create the two entities.
3. *The XZ32 frame is a concrete frame* – We instantiate *Frame* and create the new entity. Fig. 4 shows the actual state of the process after the third step.
4. *A* Frame *has a color.* – We add a new slot to *Frame*, but *Component* has no (meta)slots to be instantiated to create the color slot. We have two options: we can add a universal slot to *Component* granting the ability to add any kind of slots to its instances (including color), or we can add a specific Color slot to *Component*. In the latter case, we say that all components can have colors. Now, we choose this option. Note that we also need to add the Color slot to *Wheel* and to *XZ32*. These

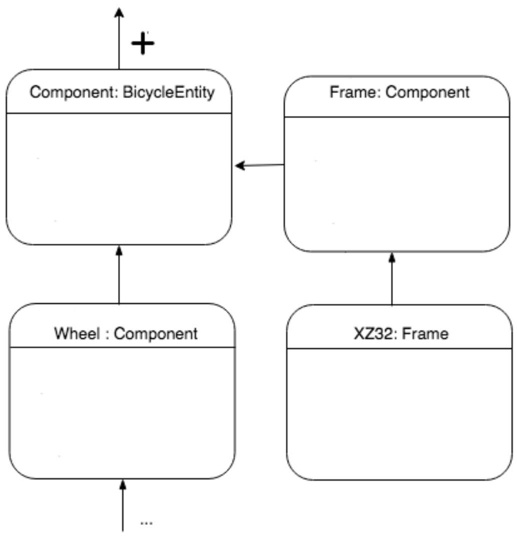

Figure 4: The actual state after adding XZ32 frame.

additions are applied automatically by the framework as we used the features *Keep flexibility* and *Relocating the features*. We can also specify that *XZ32* frame is always painted to red thus setting the concrete value of the slot. Fig. 5 shows the actual state of the process after the fourth step.

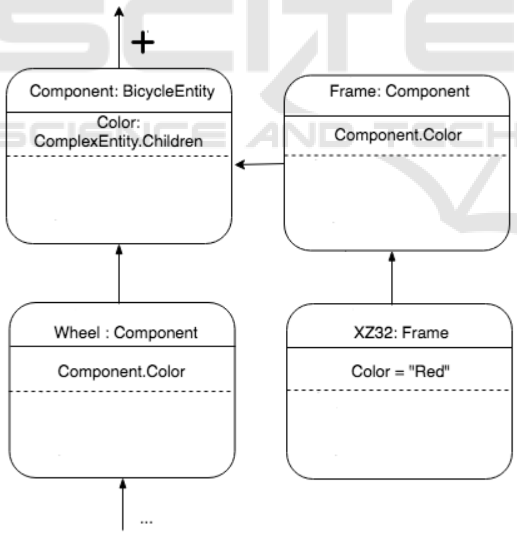

Figure 5: The actual state after adding Color slot.

5. *The weight of the* XZ32 *frame is 10kg* – We need to create a new slot in *XZ32* and similarly to the previous case, we need to create its metaslot and in this case also its meta-metaslot (in *Component*). Note that *Frame* is a mandatory in-between placeholder in this case. Adding the Weight slot to it is only required because our validation is strict and does not allow any element not enabled by its meta-element.

It is worth to mention that in design-time, it can be convenient to move the *Color* and *Weight* attributes up or down on the hierarchy automatically by the decisions of the user. Our visualization approach can effectively simplify the process of defining and moving slots. Fig. 6 shows the end result of the process.

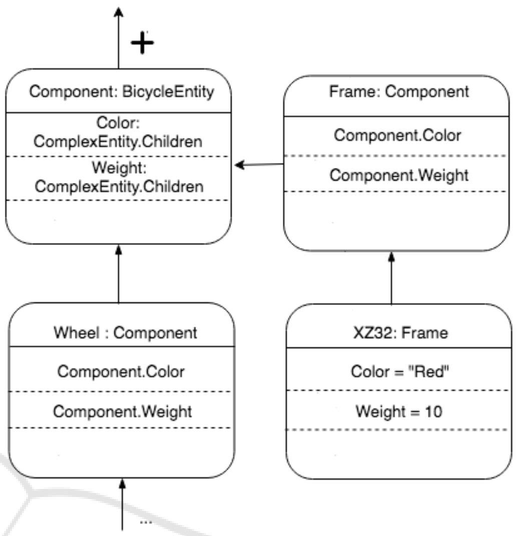

Figure 6: The end result of the process.

5 CONCLUSIONS

Model-driven development has become a feasible option to create and maintain complex systems. However, static modeling solutions are not always sufficient any longer in the modern era of industrial applications. On the other hand, it is not always acceptable to create a language definition first and create models conforming to it afterward. Models and even the modeling languages evolve continuously. Thus, the demand for dynamic modeling techniques became a natural tendency in many fields. It is one of the challenges in supporting this trend is to facilitate the process of dynamic modeling with the appropriate visualization techniques. The richness of the visual representation can simplify the modeling process and increase flexibility.

Managing the complexity of the dynamic modeling requires breaking down a domain problem into simpler parts. We believe that our visualization approach allows the user to easily manipulate sub-parts of a model in order to focus on what really matters. The approach automatically simplifies several key processes and graphically highlight the elements of interest while editing a certain domain in design-time.

In this paper, we classified and identified the most relevant difficulties that can occur in a particular multi-level modeling design process. We also introduced visualization solutions, which can support the evolutionary nature of domain models and facilitate the dynamic editing of certain models. Our visualization approach can provide a flexible and expressive way of multi-level modeling within DMLA. Although the presented concept is related to DMLA, we believe that the main principles are much more general and can be beneficial for any multi-level visualization problem settings.

In the future, we aim to finish the implementation based upon the new concepts we introduced in this paper. We plan to create more detailed case studies to prove the utility of our visualization concept. We plan to investigate our visualization concept in the context of other relevant multi-level metamodeling approaches by creating a comparison between the proposed method and relevant existing methods in a visual way. The comparison would provide a better overview of the real advantages of our visualization approach.

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. The research has been supported by the European Union, co-financed by the European Social Fund. (EFOP-3.6.2-16-2017-00013).

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., Gerbig, R., and Khne, T. (2014). Comparing multi-level modeling approaches. In *CEUR Workshop Proceedings*, volume 1286.

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.

Bottoni, P. and Ceriani, M. (2015). Sparql playground: A block programming tool to experiment with sparql. In *VOILA@ISWC*.

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

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.

DMLA (2015). Dmla website. aut.bme.hu/Pages/Research/VMTS/DMLA. Accessed: 2018-04-20.

Egon Brger, R. S. (2003). *Abstract State Machines*. Springer-Verlag Berlin Heidelberg.

EMF (2016). Emf. Accessed: 2018-04-20.

EOL (2007). Epsilon object language. http://www.eclipse.org/epsilon/. Accessed: 2018-04-20.

Gerbig, R., Atkinson, C., de Lara, J., and Guerra, E. (2016). A feature-based comparison of melanee and metadepth. In *MULTI 2016 : Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016) Saint-Malo, France, October 4, 2016*, volume 1722, pages 25–34, Aachen. RWTH.

Golra, F. R. and Dagnat, F. (2011). The lazy initialization multilayered modeling framework: Nier track. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 924–927.

MULTI 2017 (2017). Bicycle challenge multi 2017. https://www.wi-inf.uni-duisburg-essen.de/MULTI2017/. Accessed: 2018-04-20.

OMG (2005). Omg metaobject facility. http://www.omg.org/mof/. Accessed: 2018-04-20.

Simon, G., Palatinszky, D., and Mezei, G. (2017). Demonstration of using a domain-specific visual modeler for building semantic queries. In *Joint Proceedings of the 2nd RDF Stream Processing (RSP 2017) and the Querying the Web of Data (QuWeDa 2017) Workshops co-located with 14th ESWC 2017 (ESWC 2017), Portoroz, Slovenia, May 28th - to - 29th, 2017.*, pages 51–54.

Sirius (2018). Sirius. https://www.eclipse.org/sirius/. Accessed: 2018-04-20.

Wienands, C. and Golm, M. (2009). Anatomy of a visual domain-specific language project in an industrial context. In Schürr, A. and Selic, B., editors, *Model Driven Engineering Languages and Systems*, pages 453–467, Berlin, Heidelberg. Springer Berlin Heidelberg.

XModeler (2014). Xmodeler website. https://www.wi-inf.uni-duisburg-essen.de/LE4MM/xmodeler-ml/. Accessed: 2018-04-20.