Playground for multi-level modeling constructs

Ferenc A. Somogyi¹ · Gergely Mezei¹ · Zoltán Theisz¹ · Sándor Bácsi¹ · Dániel Palatinszky¹

Received: 24 May 2020 / Revised: 14 May 2021 / Accepted: 3 June 2021 / Published online: 9 August 2021
© The Author(s) 2021

Abstract

In recent years, multi-level modeling has become more and more popular. It is mainly due to the fact that multi-level modeling aims to reduce or even totally eliminate any accidental complexity inadvertently created as by-product in traditional model design. Moreover, besides reducing model complexity, multi-level modeling also improves on general comprehension of models. The key enablers of multi-level modeling are the concepts of clabjects and deep instantiation. The latter is often governed by the potency notion, of which many different interpretations and variations emerged over the years. However, there exist also some approaches that disregard the potency notion. Thus, multi-level modeling approaches tend to take advantage of different theoretical and practical backgrounds. In this paper, we propose a unifying framework, the Multi-Level Modeling Playground (MLMP), which is a validating modeling environment for multi-level modeling research. The MLMP environment is based on our multi-layer modeling framework (the Dynamic Multi-Layer Algebra), which provides useful mechanisms to validate different multi-level modeling constructs. Since beyond the structure also the well-formedness rules of the modeling constructs can be specified, our proposed MLMP environment delivers several practical benefits: i) well-formedness is always verified, ii) multi-level constructs can be experimented with independently of any concrete tool chains, and iii) relationships (i.e., correlations or exclusions) between different multi-level constructs can be easily investigated in practice. Also, the capability of the environment is demonstrated via complete examples inspired by state-of-the-art research literature.

Keywords Multi-level modeling · Potency notion · Clabject · Level-blind · Scientific experimentation · Modular playground

Communicated by Adrian Rutle and Manuel Wimmer.

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013), Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications. Project no. FIEK16-1-2016-0007 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Centre for Higher Education and Industrial Cooperation – Research infrastructure development (FIEK16) funding scheme.

✉ Gergely Mezei
gergely.mezei@aut.bme.hu

Ferenc A. Somogyi
somogyi.ferenc@aut.bme.hu

Zoltán Theisz
zoltan.theisz@aut.bme.hu

Sándor Bácsi
sandor.bacsi@aut.bme.hu

Dániel Palatinszky
daniel.palatinszky@aut.bme.hu

1 Introduction

Multi-level modeling is a modeling paradigm which aims to solve the shortcomings of classic (meta-)modeling approaches such as OMG's four-level Meta-Object Facility (MOF) [47]. Multi-level modeling allows for an unlimited number of instantiation levels as opposed to the four levels prescribed by OMG. One of the main goals of multi-level modeling is to reduce accidental complexity, which refers to parts of the solution needed only to express its multi-level nature, instead of describing the domain in question [9]. A good example of this is the application of the Item Description [17] pattern to describe multiple domain levels in object-oriented languages. The main idea behind the pattern is that objects play the role of classes, and as such, type-related information can be encoded in them. This means that classification is replaced by association, which — in addition to objects representing

¹ Budapest University of Technology and Economics, Muegyetem rkp. 3, Budapest 1111, Hungary

class information — leads to accidental complexity. Compared to classic modeling approaches, multi-level models are often reduced in size, are more compact, and describe the target domain more accurately. Therefore, many problem domains can benefit significantly from using multi-level modeling.

One of the most prevalent enablers of multi-level modeling are the notions of clabject and potency. They were introduced to handle challenges caused by instantiation spanning across multiple levels (referred to as deep instantiation). Over the years, several variations of the potency notion emerged, each of them adding to, modifying, or improving the original concept. While there have been recent advances made to unify multi-level modeling concepts [22,32], there are still many different variations and ideas behind multi-level modeling that are closely or loosely related to each other. For example, some approaches do not employ the potency notion, while others use one or more of its different variations. However, it is often difficult to compare or evaluate these approaches in practice, since they are usually based on different foundations and principles, or rely on different toolchains. It is also difficult to experiment with new ideas and apply them in practice to see how they compare to other ideas. Thus, it would be beneficial to the research community to be able to better experiment with different multi-level modeling constructs. While multi-level language constructs could be experimented with in existing approaches (i.e., using EMF [60] or an arbitrary multi-level tool), there is no existing solution that explicitly focuses on experimenting with multi-level constructs and evaluating them in a practical environment.

Therefore, we propose the Multi-Level Modeling Playground (MLMP), a validating modeling environment that is capable of defining the underlying modeling structure of multi-level modeling constructs along with their related well-formedness rules. We also propose a textual language — the Multi-Level Specification Language (MLSL) — which enables the simple definition of multi-level constructs. MLMP is not intended to be a full-fledged multi-level environment, its purpose is to enable experimentation on a common basis. In order to make our solution easier to understand, MLMP is demonstrated via complete examples in which we present the specification of multi-level constructs that exist in the research literature. The playground is based on the Dynamic Multi-Layer Algebra (DMLA), which is our multi-layer modeling framework [61,63,64].

In summary, the paper makes the following main contributions: (i) It proposes the Multi-Level Modeling Playground (MLMP), supporting experimentation with multi-level modeling constructs in a modular way. (ii) It proposes the Multi-Level Specification Language (MLSL), a textual language capable of describing the underlying structure and well-formedness rules of multi-level constructs in a simple way. It is also capable of defining domain models to be exper-

imented on. (iii) It demonstrates why MLMP and MLSL are better suited for experimentation with multi-level constructs by comparing it to some alternatives.

The structure of the paper is as follows. Section 2 introduces the main concepts of multi-level modeling, along with some of the most well-known approaches and tools in this research field. In Sect. 3 we further explain our motivations behind a multi-level modeling playground and also express the main requirements to be satisfied by one such environment. In Sect. 4 we introduce DMLA and reason why it is a suitable base for MLMP. We present the main concepts of MLMP and MLSL in Sect. 5, where we also showcase our solution in practice by presenting the specification of four selected multi-level constructs. Section 6 contains a domain model described in MLSL, demonstrating the specified constructs in practice. In Sect. 7 we evaluate our playground by comparing it with some alternatives that could be used to experiment with multi-level constructs, like the Eclipse Modeling Framework [60]. We also discuss our solution by analyzing the previously mentioned requirements. Finally, Sect. 8 concludes the paper, outlining some possibilities for future work.

2 Background

In this section, we introduce the main concepts in the field of multi-level modeling. The terminology used in the rest of the paper is also clarified here. Moreover, we briefly demonstrate some of the main concepts of multi-level modeling on a simple example. Finally, we take a look at some existing approaches, highlighting differences in their foundations. For easier comparability, a summary of the approaches discussed in this Section can be found in Appendix A.

2.1 Basics of multi-level modeling

Abstraction levels have been present for a long time in modeling. OMG's Meta-Object Facility (MOF) [47] describes a four level meta-modeling architecture where elements on a level are instances of elements on the level above. In contrast, multi-level modeling approaches rely on an unlimited number of levels, with every adjacent level being in an "instance-meta" relationship.

According to Kühne, well-defined levels "safeguard against ill-formed models" [33], thus, it is easier to create valid, well-formed models. Atkinson and Kühne compare this concept to strongly-typed programming languages, while approaches that "ignore the fact that there are levels" are akin to weakly-typed languages [4]. In the literature, the former approaches are called *level-adjutant*, while the latter are called *level-blind* [4]. Level-blind approaches can still

Fig. 1 A simple multi-level model inspired by related work [34]

implicitly employ the concept of levels, although they are not explicitly present in the model per se.

In multi-level modeling, model elements usually have a type (class) and an instance (object) facet, since a model element can serve as a type for the elements on the level below, and as an instance for the elements on the level above. This makes it possible to instantiate elements at any level in the model. To reflect this behavior, model elements are often called *clabjects* in the literature [6]. Clabjects may contain attributes (terminology may differ, e.g., fields, features) and can have relationships (edges) between them, just like in the MOF architecture.

Another important concept in multi-level modeling is *deep instantiation* [7]. In classic, four-level meta-modeling, instantiation only concerns two levels: the meta level and the instance level. This is usually referred to as *shallow instantiation* [7]. In contrast, deep instantiation spans multiple levels, namely, instances can also have their own instances. Moreover, a clabject on level N might indirectly influence another one on level $N-2$, $N-3$, and so on. This is referred to as *deep characterization* [32]. One of the most prevalent approaches for governing deep instantiation (and deep characterization) is the *potency notion* [7]. Potency is a non-negative integer number assigned to every model element in a multi-level model. In short, the potency notion is responsible for governing the instantiation depth of model elements. For example, a model element with potency of 2 supports an instantiation chain of depth level equal to two. Thus, it can be instantiated once, on the level below, and then this instance can be instantiated again on a level further below.

The concepts of clabject and potency are demonstrated on a simple example in Fig. 1. The model has three levels: L2, L1, and L0. The numbers in the upper right corner of model elements represent their potency. For example, the node *ProductType* (which is a clabject) has potency of 2. Instantiation decreases potency by 1, and when potency reaches 0 (like the *taxRate* field or the *MobyDick_Book* node), the given model element cannot be instantiated anymore.

In multi-level modeling, one can distinguish between *linguistic* and *ontological* classification [40]. One of the most prevalent architectures that achieves this is the Orthogonal Classification Architecture (OCA) [3,8]. In OCA, clabjects have two orthogonal classification dimensions: the ontological dimension defined by the modeled domain, and the linguistic dimension defined by the modeling structure. OCA-based approaches often employ classic modeling constructs like edges, attributes, and inheritance.

In order to have a unified terminology in this paper, from now on, we refer to clabjects as *nodes*, to relationships between them as *edges*, and to attributes contained in clabjects as *fields*. Together, we refer to all of the above as *model elements*. Note that some approaches also model edges as clabjects, but our terminology differs in this regard.

2.2 Multi-level approaches

The first iteration of the potency notion was proposed by Atkinson and Kühne in 2001 [7]. It is often referred to as the *classic potency notion* and has received several iterations over the years [32]. According to the classic potency notion, nodes, edges, and fields are all assigned a potency value which governs their instantiation depth. The potency of inheritance edges and abstract nodes are usually 0, since they cannot be instantiated any further. This is not always the case, for example, an inheritance edge can sometimes be given a potency value of 1 to defer the actual inheritance relationship to a level below [34]. Note that when specializing (inheriting from) abstract nodes, potency can increase in the descendant. Thus, instantiating from said descendant nodes makes it possible to continue instantiation chains with an abstract node. Fields are either of a simple or a dual nature. Simple fields only receive a value once (at potency 0), while dual fields can (but may not) receive a new value every time they are instantiated. One of the earliest applications of the classic potency notion is Deep Java [34], which combines multi-level modeling with object-oriented programming.

The classic potency notion demands the instantiation of every intermediate model element between level n and 0. According to de Lara et al., clabjects being artificially instantiated is sometimes referred to as *identity instantiation* [39]. The authors suggest that it should be possible to sometimes avoid the presence of intermediate clabjects. To elaborate, there can be cases when only the model element at potency 0 is relevant, but intermediate model elements must still be present due to the semantics of classic potency. Direct instantiation — leaving out the intermediate elements — in the classic potency notion is semantically unsound. The concept of leap potency introduced by de Lara et al. focuses on this problem. When a model element is assigned a leap potency of n , the element can only be instantiated n levels below. In practice, so-called phantom elements may be

created (thus hiding intermediate clabjects instead of skipping them entirely), which is the solution the tool *MetaDepth* employs [38].

The concept of star (*) potency aims at introducing uncertainty into the model [24]. When instantiating from a model element that has star potency assigned to it, the instance can either be assigned i) star potency or ii) an arbitrary (non-negative) potency. This means that the potency of the model element is undefined, it is decided later in the instantiation chain. The tools *Melanee* [2] and *MetaDepth* [38] both support the concept of star potency [25].

Edges can be extended with level-agnostic behavior and can support deep instantiation [5,27]. Instead of having a single potency on an edge, a potency value along with multiplicity is assigned to both of its endpoints. Dual deep modeling is an approach formalized in F-Logic [50]. It is an altered version of the potency notion which enables cross-level references. This is achieved by using a source and target potency on properties or associations. The source and target potencies can be asymmetrical. Dual deep modeling is an iteration on dual deep instantiation [48], extending it with new features like multi-level cardinality constraints. *DeepRuby* is a Ruby-based implementation of dual deep instantiation [49]. According to the authors, a dynamic programming language like Ruby is an ideal base for implementing multi-level concepts inside a programming language since it does not strictly separates classes and objects.

The concept of order is often used in multi-level modeling [14,32,33]. Order is defined similarly to potency as both are non-negative numbers assigned to model elements. The order of a model element is the “maximum number of classification relationships between an element and its most remote instances” [32]. Pure instances have an order of 0. However, abstract nodes, unlike potency, have an order that is greater than 0 since their descendants can classify other nodes via inheritance. Thus, an abstract node is also capable of indirectly classifying other nodes.

Rossini et al. proposed a formalisation of fundamental concepts like deep instantiation in multi-level modeling [55]. The formalisation is based on the Diagram Predicate Framework, which is in turn based on category theory [10]. The authors also include the concepts of multi-potency and single-potency, based on previous work [54]. Multi-potency is similar to classic potency, namely, a clabject with a multi-potency of p can be instantiated p times, on every level below. Attributes cannot be assigned a multi-potency, but constraints can, which denotes that said constraint must be evaluated at all levels, similarly to how clabjects are instantiated. Single-potency works similarly to leap potency, in that clabjects and attributes with a single-potency of p must be instantiated only p levels below. Constraints with single-potency are evaluated similarly, only at p levels below.

Igamberdiev et al. differentiate between non-potency, single-potency, and multi-potency elements in their Open integrated framework for Multi-Level Modeling (OMLM) [28]. These concepts are inspired by the work of Rossini et al. [54]. A single-potency element can be instantiated once (akin to classes in UML), while a multi-potency element behaves like a model element under the classic potency notion. A non-potency element cannot be instantiated ontologically, and can appear on any level. This categorization aims to differentiate between special elements that have potency of 0 in the classic approach, like between abstract nodes and inheritance edges. This makes it possible to separately handle model elements that are not ontologically instantiated.

The *DPF Workbench* provides a graphical editor based on the Diagram Predicate Framework (DPF) [35]. It is an Eclipse-based tool for multi-level modeling with an additional focus on code generation. While the tool provides explicit support for levels, it does not rely on the notion of potency.

ML2 is a textual, multi-level conceptual language supporting a formal theory for multi-level modeling called MLT [20]. ML2 reflects MLT’s definitions in its syntax, including entities, features (references or attributes), or regularity features that can constrain features on lower levels. ML2 is unique in the sense that it aims to combine level-adjutant and level-blind approaches in its design philosophy.

In the Level-Agnostic Modeling Language — on which the tool *Melanee* [2] is based — field values are assigned a potency, which is referred to as mutability. Together with durability (which refers to the potency of a field), they are called endurance features. A further usage of endurance features is to experiment with rules regarding valid inheritance between two nodes in multi-level modeling [36].

The approach proposed by Carvalho and Almeida (referred to as *cross-level relations* [14]) aims to formally characterize the nature of classification levels and structural relations that may occur between elements on adjacent levels. The authors also argue that the potency notion is overloaded with implicit semantics regarding specialization.

MultEcore [41] is a multi-level modeling framework integrated into the Eclipse Modeling Framework (EMF) [60]. Any two adjacent levels can be handled as an Ecore model. This integrates the approach within the Eclipse ecosystem. It also introduces the concept of range potency. Range potency governs the interval in which a particular model element can be instantiated.

In a recent study, Kühne analyzed some issues with the classic potency notion that are mainly related to abstract nodes having potency of 0 [32]. The concept of *incarnation* is introduced, which refers to direct instantiation between two nodes. By making this distinction, characterization potency is defined, which is a “measure of the element’s maximum incarnation depth in all possible worlds”.

There are several other multi-level modeling tools besides the ones mentioned earlier. For example, the Flexible Meta-Modeling and Execution Language (FMML^x) [21] — the language behind the tool *XModeler* — is an approach that aims to keep some distance from the potency notion. Namely, FMML^x is not based on the OCA, instead, it uses a custom meta-model based on Xcore [16]. *XModeler* employs the concept of intrinsic features, which explicitly specify the level on which instantiation takes place, and also support operations and associations. *Nivel* is a strict potency-based approach [1]. It extends classic potency via more extensive edge support, or by using an abstraction flag instead of potency 0 for abstract nodes. *Modelverse* is a model repository that supports basic CRUD operations and multi-level modeling [46]. It is based on the classic potency notion. *Modelverse* also uses the concept of undefined potency, which is similar to star potency in the sense that it represents uncertainty. *SLICER* is another multi-level modeling framework that is aimed at modeling large ecosystems, mainly focusing on scalability [56]. *SLICER* is different from other approaches as it has no explicit levels, — therefore, it can be considered to be level-blind — and is not potency-based. Levels are derived based on the type of model elements and the relationships between them. *DeepTelos* is a level-blind tool using the notion of Most General Instances (MGI) instead of potency [30]. *DeepTelos* extends the *Telos* knowledge representation system, focusing more on knowledge representation.

A recent study by Jácome-Guerrero and de Lara proposed TOTEM (Tool for Multi-level Modeling) in an effort to unify two-level modeling with multi-level modeling [29]. The tool is an Eclipse-based implementation of the underlying theory aiming to achieve this goal. TOTEM employs a multi-level meta-model that is capable of annotating a two-level meta-model with multi-level constructs. The other main idea is to use specialization to emulate instantiation. This way, the annotated meta-model can be extended, and from that a standard two-level meta-model that contains the annotated multi-level constructs can be compiled. The instance model then conforms to the extended meta-model, while also being an instance of the compiled one. Thus, TOTEM brings multi-level modeling closer to the world of two-level modeling. While annotating multi-level constructs is similar to how our playground operates, TOTEM focuses on two-level modeling. In contrast, our playground focuses on experimenting with multi-level constructs often spanning across multiple levels. We analyze these differences further later in Sect. 7. In addition, the authors propose a feature model to categorize multi-level modeling based on different aspects (e.g., the nature of instantiation, the handling of levels or constraints), and categorize many existing approaches.

Cuadrado and de Lara [18] proposed open meta-modeling frameworks, which can be extended and configured via the

so-called meta-object protocols (MOPs) [31]. The authors identify that currently different multi-level tools implement slight variations of core multi-level modeling concepts, with the result of being incompatible with each other. Unlike our solution, the paper focuses on the adaptation and extensibility mechanisms in existing modeling frameworks. The scope of our work is slightly different: we propose a validating modeling playground where one can experiment with multi-level constructs on a common basis. Our work does not intend to provide methodologies for adaptation or extensibility mechanisms.

3 Towards a general multi-level playground

In scientific and engineering research it has always been an essential requirement to rely only on measurements and experiments that are repeatable and discernable by the research community [19,57]. Hence, there has always been an accentuated interest in having demonstrative playgrounds where researchers encode their ideas and others can reproduce the measurements, thus verifying the results [23]. These environments may focus on supporting a single approach or may act as a general framework where different approaches can be encoded or emulated, and sometimes even combined. In such a general playground, one can even solve the same problem with different approaches. Therefore, general playgrounds can act as a common ground for comparison, and can aid in finding the advantages and drawbacks of the approaches. It may also become possible to identify dependencies and other relations (e.g., mutual exclusion) between different concepts.

As Sect. 2 has shown, there exist numerous approaches with different backgrounds in the field of multi-level modeling. Having realized this, the multi-level modeling community has recently put an effort on comparing these. The need for finding the principles to compare solutions has arisen many times during the verbal discussions at the multi-level workshops and is also manifested by the published challenges [11,53] and other papers [25]. Although the demand exists, the task is challenging, since approaches tend to differ in their regards to the underlying formalisms and tool chains. However, if we take a look at them on a theoretical level and understand their concepts, it seems possible to create a universal environment that can capture the characteristics of most approaches on a common basis. Figure 2 illustrates the basic concepts of a multi-level modeling playground that can emulate certain concepts from different approaches, and even use them together in an environment that targets experimentation. In order to reach this goal, we have identified several requirements that have to be satisfied.

Concise and flexible specification A solution is needed that offers maximum flexibility in the feature description of multi-

Fig. 2 High-level concepts of a multi-level modeling playground

level language constructs. The solution also has to be concise so that only the specialties of the particular construct need to be described in a brief but comprehensive form without expressing common characteristics. In a practical modeling playground, the rules of different constructs should be given in a flexible, high-level specification which covers both structure and well-formedness rules. The specification should support the description of all the common concepts used by multi-level approaches, e.g., node, edge, field. The specification should also be able to describe specific constructs (e.g., potency) along with their semantics (e.g., potency decreases at instantiation). Moreover, the specification should grant an easy-to-use interface for researchers to realize their ideas in the playground.

Rigorous validation. Being able to specify constructs in a concise way is not enough, modeling should be driven by the semantics of the approach. Thus, well-formedness specifications should not be used as mere descriptions, but as strict validation rules, since the well-formedness of a construct should never be violated. This means that the framework behind the playground should be able to interpret the specifications and turn them into a representation that obeys the well-formedness rules of the given construct with all its peculiarities. This representation should be able to be evaluated in practice.

Domain models. The playground can only be considered successful if the specified constructs behave exactly as their well-formedness rules dictate. The rigorous validation of constructs leads to another requirement: it is essential to be able to create domain models and perform the evaluation of constructs on them, since practical domain examples may highlight certain situations where the exact behavior of constructs can be difficult to predict. Thus, the playground can only reach its potential by supporting domain models, since

they can be considered to be the experiments and measurements the playground is designed to act upon. Using domain models, one should be able to test and validate the semantics of a given construct.

Modular structure. The final goal of a general multi-level playground is reaching the capability of specifying all available multi-level constructs, in addition to being able to experiment with new ideas. It is beneficial to have an architecture where construct specifications can be enabled and disabled, thus switching the playground from emulating one construct to another. Constructs should be able to be applied independently, or they could even depend on one another. From an architectural point of view, this leads to a modular structure where construct definitions are translated to modules which can depend on each other. Modular structure also reduces redundancy if the playground enables importing concepts defined in other modules.

Thus far, we have discussed the main requirements of a general multi-level playground. In the next section, we introduce the framework we have used in implementing our Multi-Level Modeling Playground (MLMP) in practice.

4 Dynamic Multi-Layer Algebra

A few years ago, our research group realized the importance of reducing accidental complexity through multi-level modeling techniques after we had analyzed some practical model designs. Since then we have been working on a modeling framework referred to as the Dynamic Multi-Layer Algebra (DMLA) [61,63,64]. It is inspired by many core ideas of multi-level modeling, however it has been built from scratch. DMLA is a level-blind approach that is not based on OCA and does not directly use the potency notion concept, either. How-

Fig. 3 The architecture of DMLA

ever, — as this paper intends to demonstrate — its flexibility allows us to emulate levels and other multi-level concepts such as the potency notion itself.

4.1 Main concepts of DMLA

DMLA consists of two main parts. The first part — the Core — acts as a formal ground upon which we can build modeling systems following different paradigms. These modeling systems are contained in so-called Bootstraps, which are layered on top of the Core (Fig. 3).

In DMLA, the basic modeling element is referred to as *entity*. Nodes, edges, and even fields are modeled as entities. The Core defines the basic modeling structure describing the entities, along with low-level functions to access and modify them. The Abstract State Machine (ASM) [12] formalism is used to describe the Core [63]. Besides a formal structure definition, one also necessitates modeling facilities to build models. The facilities are modeled by a set of modeling entities referred to as the Bootstrap [52]. To draw parallels, the Core can be considered as the hardware of DMLA, while the Bootstrap is the operating system layered on top of it. We have created a default Bootstrap that seems to satisfy all of our current needs, but it is possible to develop another Bootstrap upon the Core motivated by performance, expressibility reasons, or by other special needs.

The Bootstrap does not only define the entity set available for domain engineers, but it also provides the validation of the models. It is worth mentioning that the validation mechanism in DMLA is not hard-wired in the framework or in the Core. Instead, it is completely modeled by the Bootstrap. Moreover, validation plays a more essential role in DMLA than in usual modeling environments, since even the definition of instantiation itself is determined by modeled validation rules, and not by a hard-wired framework definition. This means that validation can only be altered by modifying the models (Bootstrap) and not via the execution environment. This is the main capability that makes it possible to easily model different multi-level constructs in the playground, switch between their validation rules or even mix them. By modeling these rules, it becomes part of the built-in validation system and the

rules are automatically enforced from that point on. Since the actual semantics of instantiation is defined in the Bootstrap, we should mention that in this paper we use our default Bootstrap semantics whenever we discuss any particular concepts of DMLA.

In DMLA, instantiation expresses, in general, a refinement relation between two entities. The meta-entity defines a frame, which is then filled up (concretized) by the instance. The number of steps during refinement is not limited: we can use as many instantiation steps as we want in order to reach a pure instance from an abstract concept. While descending the meta hierarchy via instantiation, we can add more and more constraints (e.g., type and cardinality), thus limiting the expressiveness of the entities, but at the same time also reaching a concrete object with all of its fields set to concrete values.

In DMLA, it is common to instantiate only certain entities of a model when stepping down a level of abstraction, while other entities remain on their original abstraction level as we do not want or cannot concretize them at the moment. This interpretation of instantiation is supported by non-strict meta-modeling, meaning that entities can refer to each other independently of their abstraction level. For the sake of clarity, it is worth mentioning that in previous work (e.g., [44], [61]), we have used two more terms to characterize the instantiation in DMLA: lazy instantiation and fluid meta-modeling. Lazy instantiation refers to the fact that it is allowed to postpone decisions regarding the instantiation of entities. As relations show fluid behavior between entities on different abstraction levels (they tend to flow between levels similarly to fluids), we referred to non-strict meta-modeling as fluid meta-modeling. It is also important to mention that DMLA does not have a genuine inheritance concept defined in the Bootstrap, thus inheritance cannot be used directly when DMLA models are being built. However, inheritance can be modeled by extending the Bootstrap as we demonstrated it in previous work [61].

4.2 Modeling and validation in DMLA

The essence of DMLA lies at its flexible instantiation and validation mechanism defined by the Bootstrap [52]. Since a basic knowledge of our Bootstrap is essential to better understand the mechanisms of our playground, in the following, we elaborate on its most important concepts, including the validation mechanism defined within.

The Bootstrap consists of entity definitions. Every entity has exactly one meta-entity. The meta-entity explicitly enables or disables certain features for its instances via the validation logic attached to it. Regarding validation, DMLA is very strict. For example, it is not allowed to add a field to an entity if it is not included in the meta-entity.

Figure 4 illustrates the most important entities of the Bootstrap. The color of the boxes depicts the categorization of the entities: (i) basic building blocks—blue, (ii) constraints and validation in general—red, (iii) operation language—green, (iv) basic (primitive) types—purple. Arrows between the blocks represent an instantiation relationship between the entities. It is worth mentioning again that DMLA does not have the concept of inheritance, instantiation is used in its stead as explained later. Figure 4 represents the most important concepts, but not all of them are the focus of this paper. More details on these concepts are available in previous work [43].

At the top of the meta-hierarchy, we have the *Base* entity. All other entities are either direct or indirect instances of *Base*. *Base* enables the ability to have fields (referred to as *slots* in DMLA) and defines the basics of validation. It is important to understand that the structure and even the behavior of slots (fields) is modeled by the entity *SlotDef*. Unlike other entities, slots can contain a value, similarly to an attribute of a class having a value.

In DMLA, validation logic is defined by operations, which are contained in slots. According to the definition of *Base*, there are three designated operations in every entity, enabling the specification of three different kinds of validation logic. They are referred to as the alpha, beta, and gamma validation formulae [45,63]. Alpha formulae are designed to validate a meta-entity against its instances by simply checking if the instantiation relationship can be verified between the meta-entity and the instance. During validation, the framework iterates over every entity of the model and invokes alpha type validation on each instance - meta pair. Beta formulae are used when an entity has to be validated against multiple related entities, like when validating cardinality. Gamma formulae are used when validation cannot be applied locally, instead, it requires checking global conditions, e.g., the uniqueness of an identifier. In other words, while an alpha validation formula is used to validate a meta-to-instance relation, a beta formula checks one-to-many relations between corresponding entities, and gamma formulae are needed when global constraints must be checked. Instances may refine the formulae of their meta-entity (appending additional validation checks), but every entity is validated against all formulae of its meta-entities, along the entire hierarchy. This means that validation formulae may be extended, making them more rigorous, but cannot be relaxed.

Since slots are defined by the *SlotDef* entity, which is an instance of *Base*, their validation can also be fine-tuned. The validation formulae of *Base* ensure that the custom validation logic on the slots is evaluated whenever their container entity is validated, mirroring the fact that an entity is valid only if all of its components are also valid. Thus, when we validate an entity, we also validate all of its components (slots) recursively. In other words, this means that compliance with

validation formulae is enforced not only along the instantiation chain, but also along the containment chain.

While creating the Bootstrap, we have realized that the validation formulae of entities often tend to be similar. Thus, we have created the *Constraint* entity that allows encapsulating re-usable validation logic for certain purposes (e.g., type checking). Since *Constraint* is an (indirect) instance of *Base*, it can have slots, thus supporting parameters in constraints as well.

The Bootstrap defines several constraints for commonly occurring validation checks. For example, the type constraint is used to check the type of a slot's value. It is worth mentioning that DMLA does not rely on a pre-defined, hard-coded type system, but uses the Bootstrap to define type conformance by type constraints. DMLA has built-in primitive types (Number, Boolean and String), but (modeled) user types are also supported. The cardinality constraint is another built-in constraint, which is used to determine the valid number of instances a given entity can have. By altering the operation definition of this constraint, it is possible to support custom cardinality, e.g., non-continuous ranges as well. Type constraints are checked in alpha validation formulae, while cardinality constraints are checked in beta formulae due to their inherent behavior. Besides the built-in constraints, users can create their own (domain-specific or general purpose) constraints to have customized validation, for example filtered cardinality, advanced type checks (e.g. regular expressions), or complex domain-specific algorithms validating the composition of multiple domain features [62].

Validation is based on operations. Unlike many other modeling solutions, we do not use an external operation language (e.g., Java), instead, we have created an operation language that is completely modeled as part of the Bootstrap [64]. The operations are modeled by their abstract syntax tree (AST). Every programming statement and expression of the language has a corresponding entity and the AST is composed from the instances of these entities. For example, every concrete conditional statement refers to the *If* entity as their meta-entity.

By relying on the hierarchy of validation formulae, one can build a domain model and gradually refine its concepts by adding more and more restrictions. The Bootstrap of DMLA is a complex set of entities. This may seem overwhelming from the domain engineer's point of view, but there is a well-identified entry point for domain models: *ComplexEntity*. By using *ComplexEntity*, one receives a base point and can build the domain model without needing to know all details of the Bootstrap entities. *ComplexEntity* has a general-purpose slot (*Children*) that is the base of every slot defined in domain models, as it is the enabler of introducing new slots in domain entities. The type constraint value of *Children* is set to *Base*, which enables having values of any type in the slot.

Fig. 4 The entities of the Bootstrap

As mentioned earlier, DMLA does not support native inheritance. We use instantiation whenever inheritance is needed based on the fact that instantiation means refinement in DMLA. For example, we model the relation between *ComplexEntity* and *Entity* by instantiation, meaning that *ComplexEntity* refines the concept of *Entity*. Since DMLA is a level-blind approach, an instance can substitute its meta-entity in a similar way inheritance works in level-adjuvant approaches. Finally, the notion of T-number [61] allows us to better imitate inheritance-like instantiations, but currently, we do not use this feature in practice. It was only introduced in order to prove DMLA's compatibility with other multi-level modeling approaches regarding inheritance.

5 Modular playground for multi-level modeling

5.1 Overview of the playground

The flexibility in the interpretation of instantiation makes DMLA a good candidate for the enabler of MLMP. For example, since DMLA is level-blind, the interpretation of cross-level references and even the interpretation of levels themselves can be introduced in exactly the way the definition of the given concept requires.

Instead of writing a completely new Bootstrap for MLMP, we have decided to add a new stratum of entities responsible for the mapping between the inner structure of DMLA and the selected multi-level constructs. We have decided to reuse existing concepts in the Bootstrap (e.g., entities, slots, the basics of validation logic), and extend them with specified multi-level semantics. Figure 5 illustrates the extended architecture of DMLA, including the injected multi-level concepts.

The injected stratum lies directly below the Bootstrap. Once a construct specification becomes a part of the DMLA framework, it gets fully validated by the generic validation

Fig. 5 The architecture of DMLA injected with multi-level concepts

mechanism defined in the Bootstrap. Therefore, inserting a construct as a set of model entities can also be regarded as a kind of a meta-refinement in the sense that the well-formedness rules of the construct will further concretize the native DMLA instantiation mechanism. Since the construct specifications described are being genuinely embedded into the fabric of DMLA's multi-layer modeling, they can act directly on domain models. Thus, domain models will be automatically validated by DMLA based on the well-formedness rules attached to the corresponding constructs.

From an architectural point of view, MLMP is modular: it consists of modules and domain models. Modules contain construct definitions, while domain models contain domain model definitions and are based on one or more modules. The structure of MLMP is illustrated in Fig. 6. Module definition consists of three components, with arrows depicting dependencies between them. In contrast, domain model definition consists of only one component. Between modules and domain models, the following dependencies are allowed:

- A module can optionally depend on one or more other modules. Modules can use every concept from the module(s) they depend on.
- A domain model must depend on one or more modules. Domain models can use every concept defined in

Fig. 6 The modular structure of MLMP

the module(s), and are validated according to the well-formedness rules defined within.

- A domain model can optionally depend on one or more other domain models. Domain models can use every concept (model element) from the domain model(s) they depend on.

Before discussing the details of the playground, it is worth mentioning that in previous work [59], we have already presented the mapping of the classic potency notion in DMLA. That solution was our initial attempt at mapping multi-level concepts in DMLA, thus the current work is superior in several aspects:

- The requirements of classic potency are more precise in this paper.
- In previous work, classic potency was embedded inside DMLA in a native way, specified by DMLAScript, the textual model specification language of DMLA [42,64]. The current work offers a much more general solution and presents the mapping of classic potency (among other constructs) specified by a new language designed to support MLMP.
- The generated DMLA entities and validation formulae (which are usually of little relevance to the users of a playground) are built on similar foundations, but the details are different due to entities and validation formulae being automatically generated from the specifications inside MLMP.
- Essentially, the native mapping of classic potency can be considered to be a proof-of-concept implementation of the specification of multi-level constructs inside DMLA, while this work expands upon that.

5.2 Modules and domain models

Modules and domain models in MLMP are specified using the Multi-Level Specification Language (MLSL), which is a textual language explicitly designed to support MLMP. The following discussion can be considered to be both the introduction of the main components of module and domain model definitions, and the introduction of MLSL itself. Simple MLSL specifications are used as examples for every

discussed component. It is worth noting that MLSL does not depend on the inner structure of DMLA, however, the model elements generated from MLSL specifications are full-fledged DMLA entities, which could have been built manually. In this regard, MLSL can be considered to be syntactic sugar upon DMLA. A brief description of the syntax of MLSL can also be found in the MLMP repository¹.

5.2.1 Module definition

A module contains the definition (or a part) of a multi-level construct. As we will explain later, every structural entity and annotation within a module must have an identifier, while validation rules may also optionally have one. Every identifier in a module must be unique.

In the following, we discuss the main components of module definition as seen in Fig. 6:

- In the **structural mapping**, the underlying modeling structure of the constructs can be defined, making it possible to specify constructs based on different structures. Note that only concepts that are relevant to the theory of a given construct need to be specified. Obviously, common structural elements are not needed to be specified again when a module depends on another module.
- Annotation-like features on model elements are common in multi-level modeling, for example, the several different variations of the potency notion. In the **annotation definitions**, annotations can be defined and assigned to model elements in a modular way.
- The semantics of a construct has to be specified in a concise way, taking into account the underlying modeling structure and defined annotations. In the **validation rules**, the rules of a construct can be given in the form of constraint-like validation rules.

The **structural mapping** consists of *structural entities*. Every structural entity has an identifier and a *structural meta type*, which is either i) directly mapped to a DMLA entity, or ii) refers to another structural entity. Therefore, it is possible to build up multi-level modeling hierarchies by combining structural entities with DMLA entities. The list of built-in structural meta types along with their DMLA counterparts can be found in Table 1. Structural entities may also have *structural attributes*. Using structural attributes, it is also possible to model association or composition-like relationships between structural entities. Structural attributes must have an identifier and a structural meta type. In a module depending on other modules, it is possible to override a structural entity from the depended modules by using the same identifier. In

¹ <https://github.com/bmeaut/MLMP>

Table 1 Built-in structural meta types in MLSL

Meta type	DMLA counterpart
Entity	ComplexEntity
Slot	ComplexEntity.Children
Slot.value	Value of a slot
Slot.type	Value of the type constraint of a slot
Bool	Boolean modeled type
Float	Number modeled type
Int	Number modeled type
String	String modeled type

this case, the structural meta type and structural attributes of the original are dismissed.

The example below is the MLSL specification of the *Node* structural entity with a structural meta type of *ModelElement*, which is another structural entity defined elsewhere. It has two structural attributes: i) *fields* is an array with a structural meta-type of *Field* and ii) *isAbstract* is of structural meta type *bool*, a primitive type. The former is an array since it later enables node instances having multiple fields.

```
Node: ModelElement {
  fields: Field[];
  isAbstract: bool;
}
```

In the **annotation definitions**, the following properties of an annotation must be defined:

- The **identifier** (name) of the annotation must be unique.
- The **range** of an annotation can either be an interval of integer values (e.g., 0..*), or a set of literals (e.g., simple or dual).
- The **mutability** of an annotation determines whether the value can or cannot be different in the instance (compared to the meta element) when an instantiation occurs. For example, classic potency is mutable, since the instance can (and in this case, should) have a different value than its meta element.
- The **targeted elements** of an annotation must refer to a valid structural entity or structural attribute. These define the elements which the annotation will be assigned to.
- The **optionality** of an annotation determines whether it must always have a value or not. Later in the validation rules, it can be further specified when an optional annotation can be omitted.

Similarly to structural entities, an annotation can override another annotation with the same identifier. Overriding results in the definition of a new annotation with the old identifier. Every property of an annotation besides its identifier can be changed when overriding. Overriding can be useful

when someone would like to keep one or more of the validation rules referring to an annotation, but also wants to redefine one or more of its properties in the process, most often when depending on another module.

The MLSL example below defines the *potency* annotation which is mutable, mandatory (signified by the lack of a '?' after the identifier), and has a range of 0..*, meaning it is a non-negative integer number. The definition also overrides any other annotations with the same name in depended modules. The example specification targets two elements: the *Node* and *Edge* structural entities, which means that in domain models, potency can only be assigned to instances of *Node* and *Edge*.

```
override mutable potency in {0..*} on Node, Edge;
```

In the **validation rules**, constraint-like rules that need to be satisfied at all times are defined. An optional identifier may be assigned to a validation rule, making it possible to omit the rule in a depending module. Overriding a rule is also possible by first omitting the rule, and then defining a new one with the same identifier. A validation rule can refer to structural entities, structural attributes, and annotations along with their minimum or maximum value. The meta element of a structural entity or attribute can also be referred to. By creating validation rules that constrain the relationship between an instance and its meta element, it becomes possible to specify the semantics of valid instantiation. Validation rules support conditional statements, arithmetic and logical expressions, transitive closures (inspired by OCL [65]), and the universal and existential mathematical quantifiers. Each validation rule behaves like a restriction on the model, similarly to the invariant rules expressed by OCL [65]. The evaluation of rules is free of side effects, they cannot change any values.

The MLSL example below depicts a rule with an ID (#RP13) that constrains the level annotation of the *Arrow* structural entity. The level must be less than or equal to the sum of the *level* and *potencyMax* annotation values of the meta element. Of course, without knowing the context, *level* and *potencyMax* could be structural attributes as well.

```
#RP13: Arrow.level <= Arrow.meta.level + Arrow.meta.potencyMax;
```

5.2.2 Domain model definition

Domain model definition consists only of structural mapping, which behaves similarly to module definition. The difference is that the values of structural attributes and annotations also have to be specified. Obviously, optional annotations only have to be given a value when the validation rules demand it. Instantiation relationships between structural entities are again specified by setting their structural meta type. Domain models can depend on one or more modules and thus can use the specified multi-level constructs within. The multi-

level constructs themselves are not innate parts of domain models.

A domain model is validated according to the well-formedness rules of the depended modules. When comparing different multi-level constructs, similarly constructed domain models that depend on different modules can be used to perform a comparison between specified multi-level constructs. This mechanism makes it easier to compare different multi-level constructs on a common basis. Note that validation rules from different modules cannot be mixed, but rules can be omitted and then redefined if needed.

In the MLSL example below, *Computer* is a structural entity in a domain model with a structural meta type of *Node*. It has three annotations assigned: a potency of 0, a level of 4, and an order of 3. It also has the *isAbstract* structural attribute set to true.

```
Computer: Node {
  @potency = 0;
  @level = 4;
  @order = 3;
  isAbstract = true;
}
```

5.3 Transforming MLMP to DMLA

From the modules and domain models of MLMP, DMLA entities and validation formulae are generated. The validation rules of a construct are modeled and validated within DMLA, making use of its built-in validation mechanisms. In the following, we further discuss the transformation from MLMP to DMLA:

- *Structural mapping* DMLA entities are generated from structural entities, while slots are generated from structural attributes. The structural meta type of a structural entity is either (i) one of the built-in types listed in Table 1—in which case the meta-entity is set according to the table—or (ii) it refers to another structural entity, thus the meta-entity is set to the DMLA entity generated from the referred structural meta type. The structural meta type of a structural attribute is encoded in the type constraint value of the generated DMLA slot.
- *Annotation definitions* From annotations, slots or constraints are generated, depending on the targeted elements of the annotation. The range of an annotation defines its generated type constraint value, for example, a range of numbers is translated into a type constraint restricting the value to the modeled Number type in DMLA. If an annotation is immutable, a validation formula is generated that forbids its modification. For every targeted element mapped to a DMLA entity, a slot representing the annotation is generated within the entity. In contrast, for every targeted element mapped to a DMLA slot, a constraint

capable of holding the value of the annotation is generated on the slot. If an annotation is mandatory, validation formula is generated ensuring that the annotation must always have a value.

- *Validation rules* Alpha validation formulae are generated into context-appropriate entities. For example, if a validation rule constrains the level of a field within a node, then alpha validation formulae are generated within the node. If a validation rule is violated, the generated alpha validation formula automatically detects and reports the issue.
- *Domain model definition* Entities are generated similarly to the structural mapping of a module definition. Due to the alpha validation mechanism of DMLA being enforced on the entire meta-hierarchy, validation formulae generated from modules are automatically enforced on the appropriate domain models.

5.4 MLMP in practice

We have selected four multi-level constructs in order to illustrate the mechanisms of MLMP and MLSL in practice. After identifying their requirements (listed in Appendix C for the sake of compactness), we have specified the constructs in MLMP using MLSL. In this Section, we present only the selected constructs, but more are available on the DMLA website.²

5.4.1 Basic modeling structure

The Basic module contains a UML-like basic modeling structure that is often seen in multi-level modeling. The module is inspired by the linguistic meta-model of the OCA, but only uses a simplified set of modeling elements that are necessary to demonstrate the specified multi-level constructs. The goal of the basic module is to separate the underlying structure of a construct from the construct specification itself.

Structural mapping. The *ModelElement* structural entity is introduced so that one can handle nodes and edges in a unified way, i.e., when specifying validation rules regarding potency and level. *ModelElement* is mapped to a DMLA entity as it is of structural meta type *entity*. The concept of *Node* is defined as a structural entity with a structural meta type of *ModelElement*. Nodes contain a structural attribute (*isAbstract*) that flags their abstract nature (following the requirements of classic potency), and can also contain *fields*. The *Field* structural entity is of structural meta type *slot*. Fields have a *value* and a *type*, both of which are built-in structural meta types and refer to the value and type of a DMLA slot. Unfortunately, since *Field* is of structural meta

² <https://www.aut.bme.hu/Pages/Research/VMTS/DMLA>.

type slot (so that we can refer to its type and value in validation rules if needed), we are not able to define it as a `ModelElement`. We could have defined the fields structural attribute as “fields: slot[];” instead of “fields: Field[];”. However, this would have i) negated the possible extension of `Field`, and ii) if we introduce a new structural entity that also contains fields, we would not be able to refer to `Field`, which means we could not handle them uniformly later.

```
ModelElement: entity {}

Node: ModelElement {
  fields: Field[];
  isAbstract: bool;
}

Field: slot {
  value: slot.value;
  type: slot.type;
}
```

Edges are modeled as entities, the *Edge* structural entity is of structural meta type `ModelElement`. Edges have two endpoints (*source* and *target*), which are of structural meta type `Node`, meaning that an edge can connect exactly two nodes. The multiplicity of edges is specified by four structural attributes representing the minimum and maximum multiplicities, following UML conventions. Although inheritance has special semantics (especially in the case of multi-level modeling), it can still be considered an edge. Therefore, *Inheritance* is a structural entity with a structural meta type of *Edge*. It is not necessary to list the structural attributes of *Edge* here again. Note that while DMLA does not inherently support inheritance, we can still define it as a modeled concept inside MLMP.

```
Edge: ModelElement {
  source: Node;
  target: Node;
  sourceMin: int;
  sourceMax: int;
  targetMin: int;
  targetMax: int;
}

Inheritance: Edge { }
```

Inheritance being defined as an edge that has multiplicities can seem strange at first glance if we do not want to support multiple inheritances. An alternative solution would add the definition of *StandardEdge* with multiplicities. This way, inheritance edges would only have a source and a target (descendant and parent), while further specialized edges (*Association*, *Composition*, and so on) can have multiplicities. However, for the purposes of this paper, we have chosen the simpler option presented earlier. This case demonstrates one of the main benefits of the playground: to foster discussion about (multi-level) modeling concepts and the different ways one can model them and think about them.

```
Edge: ModelElement {
  source: Node;
  target: Node;
}

StandardEdge: Edge {
  sourceMin: int;
  sourceMax: int;
  targetMin: int;
  targetMax: int;
}

Association: StandardEdge {}
Composition: StandardEdge {}
Inheritance: Edge {}
```

Annotation definitions. The basic module defines the level annotation which is mutable with a range of `0..*`, and is assigned to every node, edge, and field, either directly or indirectly (via `ModelElement`). Note that level is also mandatory, which means that every targeted element must have a value assigned.

```
mutable level in {0..*} on ModelElement, Field;
```

Structure and annotation definitions in DMLA. Before we move on to validation rules, first we briefly demonstrate how the concepts defined in the Basic module are transformed into DMLA entities. Figure 7 illustrates the DMLA entities generated from the Basic module. Nodes, edges and inheritance edges all indirectly instantiate *ComplexEntity*, which is the entry point of the Bootstrap for entities. The *ModelElement* entity contains the *level* slot that every instance of `ModelElement` also gets. Note that there is a containment relationship between entities and slots, enabling the validation hierarchy on the containment chain as well. Slots representing fields are generated inside nodes, instantiating *ComplexEntity.Children*, which is the entry point of the Bootstrap for slots. In the figure, where there is no meta-slot indicated by ‘:’, *ComplexEntity.Children* is considered to be the meta-slot. Note that the meta-slot does not equal to the value of the type constraint in DMLA. For example, the *source* slot in *Inheritance* is the instance of the *source* slot in *Edge*, with both of them having a type constraint restricting their value to *Node*. Constraints are marked between curly braces, with type (T) and cardinality constraints indicated (C) accordingly. Note that cardinality constraints are generated to be optional (`0..1` or `0..*`) in all cases, and have to be specified in the validation rules if they are to be mandatory. The type constraint value of fields is set to *Base* so that a field can be of any type. Edge endpoints are generated as slots with their type constraint value set to *Node*.

Fig. 7 DMLA entities generated from the Basic module

Validation rules Obviously, no construct-specific validation rules are encoded in the Basic module, since that is the responsibility of modules that use this structure to define new constructs. However, rules related to the concept of level can be found here. Note that every rule has an identifier assigned to it (e.g., #B1, #B2). The identifier not only simplifies referencing the rule here, it also makes it possible to omit them in depending modules. In the code below, rules #B1–#B6 define the semantics of edge multiplicity. Namely, the minimum multiplicity values must be less than or equal to the maximum, and the multiplicity values of an inheritance edge must always be 1. Note that we could have also specified these in one rule, using the *and* keyword. To simulate the common behavior of inheritance, every field with a potency greater than zero in the target must also be present in the source (#B7). Rules #B8–10 specify certain semantics of edges: cross-level references are disabled by enforcing that the level of an edge and its endpoints must always be equal to each other, and the source and target between levels must be consistent with instantiation. The latter means that if an edge *E* connects nodes *A* and *B* on level *N*, an edge on level *N* – 1 instantiated from *E* must connect instances of *A* and *B*. However, if inheritance edges are present, they must also be taken into account as well. Thus, the transitive closure in #B9 collects every candidate starting from the source of the meta edge, traversing inheritance hierarchies. It then verifies if the meta element of the source of the edge is amongst these candidates. Rules #B11–12 specify the multi-level behavior of edges regarding multiplicity. Rule #B13 ensures that multiple inheritance is forbidden, while #B14 is responsible for disallowing self-inheritances. Finally, #B15 defines a transitive closure to ensure that inheritance cycles cannot be defined.

```
#B1: Edge.sourceMax >= Edge.sourceMin;
#B2: Edge.targetMax >= Edge.targetMin;

#B3: Inheritance.sourceMin = 1;
#B4: Inheritance.sourceMax = 1;
#B5: Inheritance.targetMin = 1;
#B6: Inheritance.targetMax = 1;

#B7: forall inh as Inheritance:
  forall f2 as Field:
    exists f1 as Field:
      f2.name = f1.name and f2.type = f1.type and
      f2.value = f1.value
  where f1 in inh.source.fields and f2.potency > 0 and
  f2 in inh.target.fields;

#B8: Edge.source.level = Edge.target.level and
      Edge.source.level = Edge.level;

#B9: Edge.source.meta in closure(Edge.meta.source) {
  Inheritance.source where Inheritance.target = Edge.meta.
  source;
};

#B10: Edge.target.meta in closure(Edge.meta.target) {
  Inheritance.source where Inheritance.target = Edge.meta.
  target;
};

#B11: forall node as Node:
  forall edge as Edge:
    sum(edge2.sourceMin) >= edge.meta.sourceMin and
    sum(edge2.sourceMax) <= edge.meta.sourceMax
  where edge.meta=edge2.meta and edge2.source=node
  where edge.source=node;

#B12: forall node as Node:
  forall edge as Edge:
    sum(edge2.targetMin) >= edge.meta.targetMin and
    sum(edge2.targetMax) <= edge.meta.targetMax
  where edge.meta=edge2.meta and edge2.target=node
  where edge.target=node;

#B13: forall i1 as Inheritance:
  ! exists i2 as Inheritance:
    i1 != i2 and i1.source = i2.source;

#B14: Inheritance.source != Inheritance.target;
#B15: !exists inh as Inheritance:
  inh.source in closure(inh.target) {
    Inheritance.target where Inheritance.source = inh.target;
  };

```

5.4.2 Classic potency

The Classic potency module depends on the Basic module and extends it with potency-based semantics. Its structural mapping is empty, since it only relies on the structural entities defined in the Basic module.

Annotation definitions The annotations defined in the Classic potency module are all mandatory and are as follows:

- The **potency** of nodes, edges, and fields. Potency is mandatory and mutable with a range of non-negative integer numbers.
- The **nature** of fields. Nature is mandatory and immutable with two possible literal values: simple and dual.

```
mutable potency in {0..*} on ModelElement, Field;
immutable nature in {simple, dual} on Field;
```

Validation rules Note that the rule identifiers seen here (i.e., #CP1) and in other modules later do not represent the numbering seen in the requirement specifications in Appendix C. The reason for this is that many of the requirements cannot be satisfied in exactly one validation rule or may not even be related to a validation rule. #CP1 states that potency must be less than or equal to level. The compiler behind MLSL automatically concludes that this rule applies to every common targeted element of level and potency (nodes, edges, and fields), and validation formulae will be generated accordingly. #CP2 states that the level of a field must always be equal to the level of its containing node. If a node is abstract, it must have a potency of 0 (#CP3). Simple fields must not contain a value if their potency is not 0 (#CP4). Dual fields require no extra constraints due to the requirements. Finally, inheritance edges must always have a potency of 0 (#CP5).

```
#CP1: potency <= level;
#CP2: Field.level = Node.level where Field in Node.fields;
#CP3: if (Node.isAbstract) {
  Node.potency = 0;
}

#CP4: if (Field.nature = simple) {
  if (Field.potency > 0) { Field.value = null; }
}

#CP5: Inheritance.potency = 0;
```

The semantics of valid instantiation between model elements are also defined. This means that the potency of the meta element must not be 0 (#I1, #F1) and that the level and potency of the instance must always be less than that of the meta element by 1 (#I2-3, #F2-3). Note that there is some redundancy present due to previously discussed problems not allowing us to handle Field and ModelElement uniformly. Also note that MLSL supports the automatic generation of null-checks whenever it is context-appropriate. A good example of this is rule #I1 where the potency of the meta element can be null (e.g., in case of a top-level domain element), and thus MLSL automatically generates a null-check to remedy this problem.

```
#I1: ModelElement.meta.potency != 0;
#I2: ModelElement.level = ModelElement.meta.level - 1;
#I3: ModelElement.potency = ModelElement.meta.potency - 1;

#F1: Field.meta.potency != 0;
#F2: Field.level = Field.meta.level - 1;
#F3: Field.potency = Field.meta.potency - 1;
```

5.4.3 Leap potency

The Leap potency module depends on the Classic potency module, and thus, indirectly on the Basic module as well. It extends the classic potency notion with leap semantics.

Annotation definitions Leap potency requires that only either classic or leap potency be present on a Node or Edge (ModelElement). Thus, classic potency is overridden to be optional, and the new leap potency annotation is also defined as optional. Moreover, compared to classic potency, leap potency cannot be assigned to fields and its minimum range is 2 instead of 1 since a leap potency of 1 is semantically equivalent to classic potency.

```
override mutable potency? in {0..*} on ModelElement, Field;
mutable leapPotency? in {2..*} on ModelElement;
```

Validation rules Most instantiation-related rules (#I1-I3, #F2-3) from the Classic potency module are omitted, as they would conflict with the new module. Rule #F1 is not omitted because the potency of the meta element of a field must still not be 0, since leap potency cannot be assigned to fields. Rule #LP1 ensures that only either classic or leap potency can be assigned to a target. The compiler behind MLSL automatically concludes that the common targeted elements of potency and leap potency are Nodes and Edges (ModelElement). #LP2 specifies that leap potency must be less than or equal to level, while #LP3 specifies that leap potency cannot be assigned to inheritance edges. Rules #N_classic and #E_classic describe the semantics of instantiation for classic potency. New leap semantics are introduced in rules #N_leap and #E_leap. They specify that upon an instantiation, the level of instance nodes, edges, and even contained fields must be lowered by the leap potency of the meta element. They also state that the potency of the instance must be 0 with the exception of fields. The rest of the rules are the same as in the Classic potency module, they do not need to be specified here again.

```
omit I1, I2, I3, F2, F3;

#LP1: (potency = null and leapPotency != null) or
      (potency != null and leapPotency = null);

#LP2: leapPotency <= level;
#LP3: Inheritance.leapPotency = null;

// Replaces I1-3
#N_classic: if (Node.meta.potency != null) {
  Node.meta.potency != 0;
  Node.level = Node.meta.level - 1;
  Node.potency = Node.meta.potency - 1;
}

// Replaces F2-3
Field.level = Field.meta.level - 1;
Field.potency = Field.meta.potency - 1;
}

// Replaces I1-3
#E_classic: if (Edge.meta.potency != null) {
  Edge.meta.potency != 0;
  Edge.level = Edge.meta.level - 1;
  Edge.potency = Edge.meta.potency - 1;
}

// New leap semantics
#N_leap: if (Node.meta.leapPotency != null) {
  Node.level = Node.meta.level - Node.meta.leapPotency;
  Node.potency = 0;

  Field.level = Field.meta.level - Node.meta.leapPotency
    where Field in Node.fields;
  Field.potency = Field.potency - Node.meta.leapPotency;
}

#E_leap: if (Edge.meta.leapPotency != null) {
  Edge.level = Edge.meta.level - Edge.meta.leapPotency;
  Edge.potency = 0;
}
```

5.4.4 Star potency

The Star potency module depends on the Classic potency module, and thus, indirectly on the Basic module as well. It extends the Classic potency module by overriding the original potency annotation with an extended range, and by overriding certain validation rules to accommodate this extension.

Annotation definitions The range of the classic potency annotation is extended with the value '-1' which represents star potency.

```
override mutable potency in {-1..*} on ModelElement, Field;
```

Validation rules. Instantiation-defining rules that cause conflicts have to be omitted and redefined to be compatible with star potency (#I3, #F3). Note that the redefined rules are assigned the same identifier as the original, which is a choice made to improve clarity and increase compatibility in case another module would depend on this one. Let us examine the case when star potency is assigned to a model element. In this

case, no additional rules are necessary, since by definition, the potency of the instance can be any value, including star potency. The rule #SP1 constrains that only those fields are allowed to have star potency that are contained in a node that also has star potency.

```
omit I3, F3;

#I3: if (ModelElement.meta.potency != -1) {
  ModelElement.potency = ModelElement.meta.potency - 1;
}

#F3: if (Field.meta.potency != -1) {
  Field.potency = Field.meta.potency - 1;
}

#SP1: if (Node.potency != -1) {
  Field.potency != -1 where Field in Node.fields;
}
```

5.4.5 Order

The Order module depends on the Classic potency module, and thus, indirectly on the Basic module as well. It extends the Classic potency module with a new annotation definition and associated rules.

Annotation definitions Order is a mandatory and mutable annotation with a range of non-negative integer numbers, and can be assigned to nodes.

```
mutable order in {0..*} on Node;
```

Validation rules The validation rules are similar to the Classic potency module. Potency must be less than or equal to order (#O1), and order must be less than or equal to level (#O2). As opposed to classic potency, abstract nodes must have an order greater than 0 (#O3). Finally, —similarly to classic potency—the order of a meta node cannot be 0 (#O4) and the order of an instance must be less than the order of its meta node by 1 (#O5).

```
#O1: potency <= order;
#O2: order <= level;
#O3: if Node.isAbstract {Node.order > 0;}
#O4: Node.meta.order != 0;
#O5: Node.order = Node.meta.order - 1;
```

5.5 Expressiveness and usability of MLSL

After seeing how MLSL can describe the details of the selected constructs, it is worth briefly discussing why we need this language. Without MLSL, the construct definitions would have to be defined in DMLAScript (the textual model specification language of DMLA), thus, these two languages are to be compared.

The scopes of MLSL and DMLAScript are different: while MLSL focuses on the high-level specification of multi-level constructs, DMLAScript is a lower-level language capable of

describing DMLA entities and other concepts. While MLSL was created to support the playground, DMLAScript has a more general purpose of describing any DMLA model. Because of this, DMLAScript is much more verbose. Using DMLAScript, we have to specify every detail explicitly, while MLSL allows us to skip parts of the specification that can be generated automatically. This makes MLSL much more suitable for the end-users of the playground, as they need not bother with low-level details. In Appendix B, we illustrate the differences using a concrete example. Even though the scope of the two languages are different, the advantages of MLSL compared to DMLAScript can be clearly seen. However, DMLAScript is still useful when one would like to fine-tune the low-level details of DMLA entities, as its expressive power is greater than that of MLSL.

6 Domain example

In this section, we demonstrate the specification of four selected constructs on a domain example inspired by the Bicycle Challenge [11]. The multi-level modeling community issued the Bicycle Challenge with the goal of identifying issues that multi-level modeling usually faces in practical scenarios. It was successfully solved by several multi-level approaches. The Bicycle Challenge is one of the best testing grounds currently available to facilitate the comparison of multi-level frameworks and their language capabilities, thus we use it to evaluate the specification of the four selected constructs. However, the example is simplified and slightly modified: only parts necessary to illustrate the particular construct are kept. In order to focus on the most salient points, we present a few MLSL snippets demonstrating the most important concepts. The goal of this demonstration is to present MLSL in practice instead of solving the Bicycle challenge in a complete or elegant way. The full domain model can be found on the DMLA website³.

6.1 Classic potency

Figure 8 illustrates classic potency on an example. Potency values are marked in the upper right corner of model elements, except for edges, where potency is given in the diamond shape at the middle of the edge. The naming of edges aim to follow the conventions of classic potency, with edges at higher levels (≥ 1) being named according to the lower level concepts they connect.

A *BicycleModelType* is defined as a composition of *ComponentModelTypes* and *BasicPartModelTypes*. Each bicycle model type has a regular *salesPrice*, which is modeled as a field in *BicycleModelType* with a potency of 2. Every direct

and indirect instance of *BicycleModelType* has a *description* represented by a dual field that can receive a value every time it is instantiated. An underline is used to indicate dual fields. Edges are also assigned a potency value, for example, the *composedOfParts* edge between *ComponentModelType* and *PartModelType* has a potency of 3. Multiplicity indicates that a *ComponentModelType* may have 0 or more *PartModelTypes*. The *ComponentModelType* node also specializes *PartModelType* on account of the composite pattern.

In the MLSL specification, the *PartModelType* structural entity is of structural meta type Node and is found at level 3 with a potency of 0. Since its potency is 0, the node cannot be instantiated. If we tried to create a structural entity with a structural meta type of *PartModelType*, the relevant validation formulae would be triggered and the model would be reported as invalid. Note that the *isAbstract* field cannot be found in Fig. 8, since it is a part of the inner mechanisms of classic potency. The *PartModelType* structural entity has two fields: *weight* and *size*. They are contained in the *fields* structural attribute which was defined as an array in the Classic potency module. The values of the fields must not be specified since they are simple fields with a potency greater than 0.

```
PartModelType: Node {
  @potency = 0;
  @level = 3;
  isAbstract = true;

  fields {
    weight: Field {
      @potency = 3;
      @level = 3;
      @nature = simple;
      type = float;
    }
    size: Field {...}
  }
}
```

The *ComponentModelType* structural entity is of structural meta type Node. There is an inheritance relationship (*componentModelTypeInheritance*) connecting it with *PartModelType*. The potency of every inheritance must be set to 0. *ComponentModelTypes* must have the same fields as *PartModelTypes* due to the inheritance between the two. Thus, the fields have to be cloned according to the rules of valid inheritance defined in the Basic module. Since the definition of *ComponentModelType* is very similar to that of *PartModelType*, it is omitted from the example. Finally, there is an edge representing the composition (*composedOfParts_L3*) connecting *ComponentModelType* and *PartModelType*. The definition of an edge triggers the validation formulae that checks for cross-level references. Moreover, the validation checks the multiplicity values of the edge, and also whether multiplicities are respected on the lower levels.

³ <https://www.aut.bme.hu/Pages/Research/VMTS/DMLA>

Fig. 8 A domain model demonstrating classic potency

```

ComponentModelType: Node { ...
}

componentModelTypeInheritance:
  Inheritance {
    @potency = 0;
    @level = 3;
    source = ComponentModelType;
    target = PartModelType;
    ...
  }

composedOfParts_L3: Edge {
  @potency = 3;
  @level = 3;
  source = ComponentModelType;
  target = PartModelType;

  sourceMin = 1;
  sourceMax = 1;
  targetMin = 0;
  targetMax = *;
}

```

Level L^3 shows that a *BicycleModel* is a composition of components: a *FrameModel* and two *WheelModels*. Multiplicity values indicate that a *BicycleModel* must have exactly one *FrameModel* and two *WheelModels*. The association *soldToCustomer* with potency 2 at L^2 between *BicycleModel* and *CustomerType* specifies that at L^1 the association *soldToCustomer* with potency 1 in turn specifies that bicycles are sold to customers. Level L^1 contains the components of a *ChallengerA2-XL* bicycle model. *ChallengerA2-XL* has a *RocketA1-XL* frame, but it does not have a *Suspension*. Level L^0 contains the pure instances of the model. For example, *Bike#134123* is a pure instance with a potency of 0 and all of its parts are concretized.

In the MLSL specification, the *Frame#134123* (of structural meta type *RocketA1-XL*) structural entity can be found at level 0 with a potency of 0. Since *RocketA1-XL* has a potency of 1, if we created a structural entity with a potency other than 0, the model would be invalid. Similar rules apply for its level. Since the simple field *weight* also has a potency of 0, its value must be specified, otherwise the model would be invalid.

```

Frame134123: RocketA1-XL {
  @potency = 0;
  @level = 0;
  isAbstract = false;

  fields {
    weight: Field {
      @potency = 0;
      @level = 0;
      @nature = simple;
      type = float;
      value = 20;
    }
    size: Field {...}
  }
}

```

6.2 Leap potency

Using leap potency, one can hide intermediate elements that can be considered to be a part of identity instantiation. Figure 9 illustrates the concepts of leap potency. For the sake of clarity and compactness, only model elements that are necessary to illustrate the mechanisms of leap potency are shown. In order to better accommodate the semantics of leap

potency, we have made some modifications to the model: L^2 contains *Customer* and *Bicycle*, while L^1 contains no instances of *Customer*. This way, introducing a new *Bicycle* instance at L^1 does not require the addition of new edges connecting customer instances to them, the leap semantics allow the *soldToCustomer* edge to instantiate directly at L^0 .

At level L^2 , *Customer* is assigned a leap potency of 2, which is marked in parentheses. This way, we can instantiate the node directly at level L^0 . Similarly, at level L^2 , the edge (*soldToCustomer*) between *BicycleModel* and *Customer* is also assigned a leap potency of 2, avoiding the possibly redundant instantiation of new edges at L^1 should a new bicycle model be introduced.

In the MLSL specification, a leap potency value is added to the *Customer* structural entity (replacing classic potency), activating the validation formulae enabling the instantiation of *SusanStorm* from *Customer*. Note that according to the validation rules, *SusanStorm* must have a potency of 0 (making it a pure instance) instead of a leap potency of 0 (leap potency has a minimum value of 2).

```

Customer: Node {
  @leapPotency = 2;
  @level = 2;
  isAbstract = false;
}

SusanStorm: Customer {
  @potency = 0;
  @level = 0;
  isAbstract = false;
}

```

A leap potency is also added to the *soldToCustomer* edge, which enables the instantiation of *soldToSusanStorm* directly from *soldToCustomer*.

```

soldToCustomer: Edge {
  @leapPotency = 2;
  @level = 2;
  source = Customer;
  target = Bicycle;
  ...
}

soldToSusanStorm:
  soldToCustomer {
    @potency = 0;
    @level = 0;
    source = SusanStorm;
    target = Bike134123;
    ...
  }

```

6.3 Star potency

Model elements that are assigned a star potency do not have a set instantiation depth and can be instantiated on any level below. Figure 10 depicts the concepts of star potency. In the *BicycleModelType* node, it may be difficult to specify the potency of *salesPrice*, especially if we start constructing the model on the highest abstraction level. Thus, in the *BicycleModelType* node, *salesPrice* has a star potency assigned (marked by a star). We specify the potency of *salesPrice* only at level L^1 , where the field is assigned a potency of 0.

In the MLSL specification, we can set the potency of the *salesPrice* field in *BicycleModelType* and *BicycleModel*

Fig. 9 A domain model fragment demonstrating leap potency

to -1, thus enabling star potency. This disables validation formulae that would trigger on classic potency. Therefore, the instance can be assigned any potency value. In *ChallengerA2-XL*, we can set the potency of *salesPrice* to 0.

The circled numbers in the upper right corner of the nodes stand for order values. The value of order decreases by 1 upon instantiation. *BicycleModelType* has an order of 3, since the maximum depth of the classification relationships originates

```
BicycleModelType:
Node {
  @potency = 3;
  @level = 3;
  isAbstract = false;

  fields {
    salesPrice: Field {
      @potency=-1;
      @level = 3;
      @nature =
        simple;
      type = int;
    }
    description: Field {...}
  }
}
```

```
BicycleModel:
BicycleModelType {

  @potency = 2;
  @level = 2;
  isAbstract = false;
  fields {
    salesPrice: Field {
      @potency=-1;
```

```
    @level = 2;
    @nature =
      simple;
    type = int;
  }
  description: Field {...}
}}
```

```
ChallengerA2-XL:
BicycleModel {

  @potency = 1;
  @level = 1;
  isAbstract = false;

  fields {
    salesPrice: Field {
      @potency=0;
      @level = 1;
      @nature =
        simple;
      type = int;
      value=2000;
    }
    description: Field {...}
  }
}}
```

6.4 Order

Figure 11 demonstrates the main concepts of order. For the sake of clarity, only model elements that are necessary to illustrate the mechanisms of order are shown at levels L^3 and L^0 .

ing from this node is 3. *PartModelType* also has order 3, since even though it does not directly classify instances, it indirectly classifies a 3-depth instantiation chain via *BasicPartModelType* and *ComponentModelType*.

In the MLSL specification, adding an order value triggers validation formulae checking that the order of the abstract node *PartModelType* must be greater than 0. It

Fig. 10 A domain model fragment demonstrating star potency

is also enforced that order must decrease by one upon instantiation. Pure instances at level L^0 (e.g., *Bike#134123*, *Frame#134123*) also have an order of 0 since they do not classify any instances.

```
PartModelType: Node {
  @potency = 0;
  @level = 3;
  @order = 3;
  isAbstract = true;
  fields { ... }
}
```

6.5 Combining different language constructs

In MLMP, we can combine different modules, and thus different constructs, as we see fit. Obviously, we may have to modify the modules in order to handle mutually exclusive annotations. For example, the modules containing order, leap and star potency are all dependent on the Classic potency module. When we use leap potency, star potency, and classic potency together, certain validation rules have to be modified to allow for only one of the three to be assigned to a model element at a time. The classic potency annotation also has to be overridden. These changes are not difficult to make and can be done in one or more modules that contain the compatible definitions.

The Bicycle challenge domain described in this section also demonstrates this aspect of the playground. Classic potency can be used as the base, leap potency can help avoid identity instantiation, while star potency is useful when we want to express uncertainty in the depth of modeling hierarchies. Finally, order can be used to differentiate between abstract nodes and pure instances both having a potency of 0. Thus, we can use these constructs on the same domain model, making it easier to experiment with them on a common basis.

7 Evaluation

In this section, we present the evaluation of MLMP via a comparison with alternative solutions. The purpose of the comparison is to discover the advantages and drawbacks of the playground compared to other solutions that can be used to experiment with multi-level constructs. We are comparing MLMP with the following solutions: i) a two-level general modeling framework that is widely known in the research community, and ii) with level-adjuvant multi-level tools in general, with one such tool used as a representative example. Note that this is not a full-scale evaluation (as MLMP is not a full-fledged modeling language), the focus is solely on experimentation with multi-level constructs. Among others, some of the main aspects discussed here are modularity, compactness of constraint definitions, efficient handling of annotation-like concepts, and domain model definition. The evaluation is conducted by specifying the classic potency notion in the alternative solutions, and then comparing it to the MLSL specification described earlier. Moreover, we discuss the modularity of the alternative solutions by analyzing how leap potency could be introduced into the particular solution.

7.1 Eclipse Modeling Framework

The Eclipse Modeling Framework [60] is a well-known modeling solution that is an implementation of OMG's EMOF standard. Various multi-level languages are also built upon EMF [2,29,41]. EMF provides a two-level meta-modeling environment for specifying meta-models and instance models. It also supports the definition of OCL [65] constraints that can be used to specify well-formedness rules on the model. In our evaluation, we used OCLinEcore⁴.

Figure 12 illustrates the underlying structure of the classic potency notion modeled in EMF. It contains the merged counterparts of the Basic and Classic potency modules from MLMP, for reasons explained later. A *Model* consists of *ModelElements* (which is an abstract class) that contain the mandatory *potency* and *level* attributes, along with the *name* of the model element. A model element can also have a *meta* element, which is required to simulate the multi-level nature of models. *Nodes*, *Edges*, and *Fields* specialize *ModelElement*, enabling multi-level features in them. The attributes inside these classes are self-explanatory with *isAbstract* representing the abstract flag in nodes, *type* and *value* referring to the type and value of field (using the *AnySimpleType* type), and *sourceMin* and similar attributes representing multiplicities on edges. *FieldNature* is an enum, used to differentiate between simple and dual fields. The *source* and *target* of an edge

⁴ <https://wiki.eclipse.org/OCL/OCLinEcore>

Fig. 11 A domain model fragment demonstrating order

Fig. 12 The structural mapping of classic potency in EMF

refer to a node. The *Inheritance* class specializes *Edge*, inheriting every attribute. The *edgesWithMeta* and similar attributes are derived attributes required by certain constraints.

In the following, we present the OCL invariants used to specify the well-formedness constraints of classic potency. Where possible, the name of the constraint refers to the identifier of the corresponding validation rule in MLSL

(e.g., CP1). The invariants in the context of the *ModelElement* class specify the relation between potency and level (CP1) and the semantics of valid instantiation (I1-I3).

```
abstract class ModelElement
{
  invariant CP1: not potency.oclIsUndefined() implies
    potency <= level;

  invariant I1: not meta.oclIsUndefined() and
    not meta.potency.oclIsUndefined() implies
    meta.potency < 0;

  invariant I2: not meta.oclIsUndefined() implies
    level = meta.level - 1;

  invariant I3: not meta.oclIsUndefined() and
    not meta.potency.oclIsUndefined() implies
    potency = meta.potency - 1;
}
```

The level of fields inside a node must be the same as the containing node (CP2) and abstract nodes must have a potency of 0 (CP3). Invariant B11_1 is responsible for checking the multi-level behavior of edges respecting multiplicities. Three other similar constraints were omitted due to reasons of compactness.

```
class Node extends ModelElement
{
  property edgesWithSource : Edge[*|1] { derived }
  {
    initial: Edge.allInstances()->select(e | e.source = self);
  }
  property edgesWithTarget : Edge[*|1] { derived } {...}

  invariant CP2: fields->forAll(f | f.level = self.level);
  invariant CP3: isAbstract implies potency = 0;
  invariant B11_1: edgesWithSource->forAll(
    e | e.edgesWithSameMeta->select(e2 | e2.source = self)->
    asSet()->collect(sourceMin)->sum() >= e.sourceMin);
  ...
}
```

Invariant B8 is responsible for forbidding cross-level references. Invariants B9 and B10 specify the well-formedness of edge source and target instances, taking into account inheritance edges. Note that using the built-in edge and inheritance relationships of EMF is not feasible since they are part of the toolkit and are not modeled relationships.

```
class Edge extends ModelElement
{
  property edgesWithSameMeta : Edge[*|1] { derived }
  {
    initial: Edge.allInstances()->select(e | e.meta = self.
      meta);
  }

  invariant B8: source.level = target.level and
    source.level = level;

  invariant B9: not meta.oclIsUndefined() implies
    meta.oclAsType(Edge).source->
    closure(s | Inheritance.allInstances()
      ->select(i | i.target = s).source)
      ->includes(source.meta);

  invariant B10: not meta.oclIsUndefined() implies
    meta.oclAsType(Edge).target->
    closure(t | Inheritance.allInstances()
      ->select(i | i.target = t).source)
      ->includes(target.meta);
}
```

Inheritance edges must have a multiplicity of 1..1 (B3-6) and a potency of 0 (CP5). Invariant B7 specifies that the descendant (source) of an inheritance must contain every field from the parent (target), based on the name, type, and value of the field, excluding fields with a potency of 0. Invariants B13-15 forbid multiple inheritance, self-inheriting, and inheritance cycles.

```
class Inheritance extends Edge
{
  invariant B3: leftMin = 1;
  invariant B4: leftMax = 1;
  invariant B5: rightMin = 1;
  invariant B6: rightMax = 1;

  invariant B7: target.fields->select(f | f.potency > 0)->
    forAll(f | source.fields->
      select(f2 | f2.name = f.name and f2.type = f.type and
        f2.value = f.value and f.potency > 0)->isEmpty() = false
    );

  invariant B13: Inheritance.allInstances()
    ->select(i | i <> self and i.source = self.source)->
    isEmpty();

  invariant B14: self.source <> self.target;

  invariant B15: target->closure(t | Inheritance.allInstances()
    )
    ->select(i | i.source = t).target->excludes(source);

  invariant CP5: potency = 0;
}
```

7.1.1 Multi-level nature

As opposed to MLMP, EMF does not inherently support multi-level modeling. This requires the introduction of basic

multi-level concepts like levels. The modeled meta relationship in the EMF solution is one such example, since one cannot use instantiation as a relationship in EMF. Demanding type conformity with the meta element should also be required (i.e., the meta element of a node can only be another node), but is omitted from our solution.

7.1.2 Modeling facilities

EMF employs traditional modeling facilities including a standard graphical notation and traditional model elements like inheritance or interfaces that can be used in the creation of multi-level concepts. Traditional modeling facilities are widely known and thus, it is most likely easier to use a traditional approach as opposed to MLMP which does not provide them due to the nature of DMLA [58]. It is worth noting that it is possible to model these commonly used model elements in MLMP (and also in EMF, e.g., see inheritance in both solutions) and then later use them when building other constructs. However, they are not inherently supported by MLMP and cannot be used in the construction of top-level modules.

7.1.3 Constraint definition

Constraints in EMF are often defined by OCL, a well-known formalism and thus, it is easier to use at first than MLSL. However, since MLSL was conceived with the goal of supporting a multi-level modeling playground, it can offer several advantages to the end user. For example, in invariants I1-3, a null-check is needed due to top-level domain entities having an empty meta relationship. In MLSL, the compiler generates these null-checks automatically based on the context. In MLSL, module entities are also excluded from validation, leading to a better separation of modules and domain models. Of course, MLSL was inspired by OCL, which is evident in the somewhat similar syntax and concepts like transitive closures.

7.1.4 Annotations

EMF does not support annotations or stereotypes that can be constrained by languages like OCL [37]. Instead, annotation-like concepts like level and potency are embedded inside an abstract class in our solution. This results in less redundancy since in MLMP we could not specify *Field* to be a *slot* and a *ModelElement* at the same time. However, this solution also heavily restricts the structure of the model, namely, specializing an abstract class disables the specialization of other classes. It is worth mentioning that EMF Facet [13] allows the extension of existing models via inheritance. OCL invariants however cannot be omitted in the extended modules, since a subclass can only strengthen an invariant but cannot weaken it. Thus, for example, when we want to extend classic potency

Fig. 13 An interface-based solution for handling annotation-like concepts

by leap potency, we still cannot redefine the necessary OCL invariants (the semantics) and also classic potency itself (the structure) if it would be required.

Figure 13 depicts an alternative, interface-based solution for introducing level and potency. This solution also has several drawbacks. Firstly, OCL invariants would have to be specified in the context of the interfaces. This could greatly complicate them as an interface implementation-check would be required whenever we refer to a feature from another interface. Secondly, it would require duplicating the *potency* and *level* attributes in every class implementing the corresponding interface. Thirdly, the optionality of an attribute would also introduce some accidental complexity, since extra null-checks would be needed on function return results. These problems are amplified exponentially when new interfaces are introduced. We believe that MLSL and its annotation support are superior in this regard, as they lead to more simplified and easier-to-read specifications with less accidental complexity.

Figure 14 introduces an alternative solution inspired by the annotation-based approach of TOTEM [29]. Instead of embedding multi-level constructs inside the basic structure of the model, the structure can be deployed inside a package. Then, concepts like potency and level can be injected into the structure via annotations (references) from outside the package (or from another meta-model). This way, we could eliminate the need for the *ModelElement* abstract class and give every node, edge, and field a name and a meta element. However, this would result in the complication or duplication of certain OCL invariants referring to a meta element. More precisely, this approach would introduce an extra layer of navigation for the invariants. For example, in some cases, additional elements (like *PotencyAnnotation*) need to be queried not only for the model element in question, but for its meta element as well. To sum up, the annotation-based solution of TOTEM works because it focuses on transform-

Fig. 14 An alternative solution for handling annotation-like concepts

ing multi-level concepts into two-level models. However, a multi-level modeling playground in EMF has a different scope: it aims to transform two-level concepts into multi-level concepts while also aiming to make it modular in terms of introducing new concepts and checking their well-formedness. Finally, note that using the standard concept of EAnnotation in EMF is not a feasible solution to our problem as its purpose is to support code generation or model serialization.

7.1.5 Modularity

EMF employs an extension mechanic for meta-models via inheritance. This enables the layered definition of certain concepts, for example we could make a Basic module similarly to MLMP, or extend classic potency with leap potency. Figure 15 depicts the extension of classic potency with leap potency. First, the *ModelElement* class is extended with the *leapPotency* attribute. The *Model* root class is also redefined. However, supertypes are not supported on edges, thus the *elements* composition has to be duplicated. The original *Node*, *Field*, and *Edge* classes still specialize the old *ModelElement* class. Therefore, these classes also have to be duplicated. The same arguments can be made for the Basic module and Classic potency module, which is the reason why we have chosen to merge the two in our EMF-based solution. When defining multi-level constructs in a modular way, due to the redefinition of concepts, MLMP performs better compared to EMF. In EMF, separation into modules is more difficult, since an extension can have a rippling effect throughout the model: in the example, extending *ModelElement* requires the extension of every class that specializes it in order for them to be able to support leap potency. Finally, if we use an annotation-based approach, introducing the concept of leap potency in an elegant way is difficult since we still cannot omit OCL invariants.

Fig. 15 Extending classic potency with leap potency

7.1.6 Domain model definition

MLMP enables sandboxing by letting users decide for each domain model which module(s) it depends on. In EMF, an instance model is typically the instance of one meta-model, but it is possible to load one or more different meta-models as additional resources. The interface-based solution would in theory be feasible here, but we have already discussed its weaknesses. We could use an extended leap potency meta-model containing duplicate features. However, the redundancy (also discussed earlier) would still be present. A solution for the complex combination of constructs would be to create a merged meta-model for every combination, duplicating the elements inside each time. Moreover, as we have discussed already, OCL invariants cannot be omitted. This can introduce conflicts or redundancy in the definition of constraints as well. While MLMP is not perfect in this regard, it still offers a more modular solution.

7.1.7 Validation

Reporting on violations of well-formedness is similar in both approaches, with the violated rules along with the violating entity being displayed. The constraining instance model editor in EMF helps ensure that the model will be well-formed at all times. The validating mechanisms of DMLA performs a well-formedness check on the generated model, including checking the model for basic (multi-level) violations that are inherently supported by DMLA. In conclusion, while there are many possibilities in better exploit the robust validation mechanism of DMLA, at this time, both approaches do a similarly adequate job of validating the model and reporting errors.

7.2 Level-adjutant multi-level solutions

In the following, we briefly compare MLMP to level-adjutant multi-level languages in general. The particular tool that we have chosen to implement our solution in is Melanee [2]. Figure 16 depicts the mapping of classic potency in Melanee.

This solution is similar to the previous one, with a few notable differences:

- Instead of classes, we have clajjects that are all assigned star potency as we do not want to constrain the maximum instantiation depth of domain models. Note that star potency is not supported by every multi-level tool.
- There is no need to implement a modeled *meta* relationship, since we can use the built-in levels and meta relationships in the tool.
- However, there is still a need to implement potency and level, since the built-in potency and level mechanisms are different from the modeled ones. For example, the built-in potency on the modeled elements is set to star potency in order to allow an unlimited number of levels in domain models. Moreover, the playground is meant to experiment with the constructs by altering their definitions. Thus, we cannot use a hard-wired implementation, we need a modeled one.
- *Inheritance* again specializes *Edge* instead of being instantiated as explained later.
- The type and value of a field uses the type *void*.

The Deep OCL dialect of Melanee extends OCL with multi-level modeling support. The logic of OCL constraints in Melanee is similar to the EMF solution, but since Melanee uses a different OCL dialect, there are some differences. Most notably, clajjects on the topmost level (that a module consists of) have to iterate through their subtypes and instances in the constraints. We can see that while the constraints are more verbose than their EMF counterparts, their expressiveness is greater due to supporting multiple levels. We have used the following DeepOCL-specific operations in our solution:

- **getInstances()** - returns all clajjects that are instances of the source clajject.
- **getSubtype()** - returns all clajjects that specialize the source clajject.
- **getBlueprint()** - returns the direct type (meta element) of the source clajject.
- **getAttributeByname(name)** - retrieves an attribute in the source clajject, specified by *name*.

```
context ModelElement:
inv CP1: self.getSubtypes().getInstances()=>forAll(i | not
i.getAttributeByName('potency').value.oclIsUndefined()
implies
i.getAttributeByName('potency').value.toInteger() <=
i.getAttributeByName('level').value.toInteger())

context ModelElement:
inv I1: self.getSubtypes().getInstances()=>forAll(i |
not i.getBlueprint().oclIsUndefined() and not
i.getBlueprint().getAttributeByName('potency')
.oclIsUndefined() implies i.getBlueprint()
.getAttributeByName('potency').value.toInteger() < 0)

context Node:
inv CP3: self.getInstances()=>forAll(i |
i.getAttributeByName('isAbstract').value.toBoolean() implies
i.getAttributeByName('potency').value.toInteger() = 0)

context Inheritance:
inv B3: self.getInstances()=>forAll(i |
i.getAttributeByName('sourceMin').value.toInteger() = 1)
```

7.2.1 Analysis

The multi-level support of Melanee and level-adjuvant languages in general simplifies the creation of a playground by not having to explicitly model basic concepts like the meta relationship. In the Deep OCL dialect of Melanee for example, the *getBlueprint* and other similar operations support the multi-level traversal of the model. However, the multi-level nature of a solution also comes with disadvantages regarding experimentation. For example, expressing multi-level concepts like classic potency that already exist in a multi-level tool can make the solution more confusing, as it can be difficult to differentiate between the built-in and modeled concepts. Level-adjuvant languages like Melanee also restrict the modular definition of constructs by adhering to strict levels. If one would try to construct a module consisting of several levels, the introduction of intermediate elements would be required in most level-adjuvant tools. Figure 17 illustrates a solution similar to that of MLMP, where *Inheritance* is instantiated from *Edge*. Here, we can see that intermediate elements like *Node* would have to be instantiated, introducing accidental complexity into the solution. We believe that the strict nature of level-adjuvant solutions is more disadvantageous here, as a playground may need more

Fig. 16 The structural mapping of classic potency in Melanee

Fig. 17 An alternative definition of Inheritance in Melanee

flexibility to better support certain multi-level concepts, like enabling cross-level references.

Similarly to EMF, Melanee offers traditional modeling facilities. This can differ in various solutions, but in this case, a similar reasoning to the analysis of the EMF-based solution can be applied. Thus, this is an advantage over MLMP and its DMLA-based foundation.

Regarding the constraint language, the Deep OCL dialect used by Melanee has multi-level support, but the invariants may be less concise in return. The reasoning from earlier regarding deep constraining languages applies here.

Modularity can also vary from solution to solution, but supporting annotations and the ability to omit and redefine constraints enables MLMP to compete with most solutions in terms of modularity. The reasoning regarding interfaces in the EMF-based solution applies here. From the viewpoint of an end user, model validation and reporting on well-formedness violations tend to be similar in most solutions.

7.3 Discussion

To finish our evaluation, there are several other minor advantages of using MLMP due to its narrow focus:

- When one refers to multiple annotations by their identifiers (e.g., potency and level) in a validation rule, MLSL

automatically calculates the common targeted elements and generates a validation rule for each one of them. Thus, the context of a rule is assigned automatically. In most alternative solutions (especially if one is based on OCL), the context has to be specified explicitly.

- MLSL supports the “*” notation which refers to an unlimited integer number value. This can be useful when specifying the multiplicities of modeled edges. EMF or Melanee does not have an explicit counterpart that can be used in domain models. In our solutions, an arbitrarily high integer number was used instead.
- MLMP supports field-like constructs in the form of general slot types and values that are implicitly mapped to their respective DMLA types. There is explicit support for the most often used primitive types (boolean, int, float, string), but field type and value can still be defined in a general way. In EMF, we have used the *AnySimpleType* (which is mapped to a general Java object) to achieve a similar result, while in the Melanee-based solution, the *void* type was used. Neither have constraints or restrictions on them, and type errors have to be dealt with explicitly in both solutions.

While it is possible to experiment with multi-level constructs using EMF, it becomes apparent that EMF is not specialized for this task and has several shortcomings. The most prevalent drawbacks are related to modularity and sandboxing, with EMF having difficulties to achieve modularity in construct specifications and thus in domain model definitions. Although EMF offers traditional modeling facilities that are easier to use, its modularity and sandboxing capabilities can be lacking for experimentation with multi-level constructs. This, along with basic concepts like multiple levels being “forced” into the solution, makes two-level solutions like EMF less ideal for experimentation. We believe that comparing MLMP to other, similar (meta-)modeling tools

that do not inherently support multi-level modeling would yield similar results as the presented comparison to EMF. There are recent efforts in bringing two-level and multi-level modeling closer to each other, which could lead to better experimentation in these particular solutions as well [26,29].

Using a level-adjuvant multi-level tool to experiment with multi-level constructs has some advantages over EMF, but also has its own drawbacks. If a particular tool inherently supports multi-level constructs that also have to be modeled to be experimented with like potency variants, it can make the solution more complicated and less comprehensible. Modularity-wise, we believe MLMP can compete with most approaches when it comes to experimentation with constructs. Constraint definition is still an active research area in multi-level modeling [15], and while Deep OCL dialects and similar languages can offer greater expressive power, MLSL offers a more focused and high-level specification.

Finally, as an alternative solution, directly embedding a given construct in a multi-level tool is certainly possible. However, the goal of a playground is to have an environment that serves as a common basis for specifying, comparing, and experimenting with multi-level constructs. Embedding a construct in a particular tool is most likely only possible for the creators of the tool, while a multi-level modeling playground should provide a high-level and unified way for researchers to specify and evaluate different constructs.

To conclude our evaluation, in the following, we discuss how MLMP satisfies the requirements set in Sect. 3.

Concise and flexible specification To support MLMP, we have created a high-level textual language. Using the module definition of MLSL i) the underlying modeling structure, ii) associated annotations, and iii) the well-formedness rules of a multi-level construct can be specified. MLSL focuses on the essential characteristics of multi-level constructs, low-level details are not needed to be covered. During the specification of constructs, one can rely on other constructs facilitated by the modularized architecture. However, it is worth mentioning that the approach cannot be considered to be completely compositional, as adding a new annotation often requires changing existing ones. This could be improved in the future, for example, it might be possible to take ideas from feature-oriented programming [51]. MLSL also offers several practical shortcuts when specifying validation rules. In order to demonstrate the capabilities of MLSL, we have specified several multi-level constructs. For the sake of brevity, we have described only a few selected constructs in the paper, while the complete list of mappings are available on the DMLA website⁵. Based on our experiences, our conjecture is that MLSL is capable of capturing the essential characteristics of most multi-level constructs in a compact way. Future work is planned to further strengthen this claim.

Rigorous validation. It is important to have a convenient way to define the well-formedness of multi-level constructs. However, enforcing well-formedness all the time is just as important. Many modeling solutions use an external language to specify well-formedness rules. DMLA is different, as its built-in operation language makes it possible to describe the validation by a set of model entities. The well-formedness rules of multi-level constructs are translated into strict, fully modeled validation formulae. From this point on, these formulae are handled as part of the built-in validation system. Since DMLA validates its entities rigorously and continuously, any well-formedness violations are identified and reported immediately. This solution has several important characteristics: (i) validation is managed by DMLA, thus we can be certain that any violation of rules will be reported automatically; (ii) the built-in validation mechanism enforces validation rules along the entire meta-hierarchy including the Bootstrap and the domain models; (iii) entities generated from the modules and domains of MLMP can even be used independently, as full-fledged DMLA models.

Domain examples. MLMP supports the definition of domain models, making it possible to evaluate multi-level constructs in practical scenarios. We have shown how MLSL is capable of describing a domain model example inspired by the Bicycle Challenge. By providing an easily accessible testing environment, researchers can fine-tune and customize their constructs based on the results of validation on a domain example. In the example, we have discussed how a given construct is evaluated on a domain model, leading to violation reports on its well-formedness rules.

Modular structure. MLMP has a modular architecture consisting of modules and domain models. This leads to reduced redundancy and facilitates the re-usability of constructs. Moreover, module definition itself is separated along structure, annotation, and validation rule definitions. Thus, MLMP also makes it possible to interchange the underlying modeling structure that a particular construct relies upon, enabling experimentation with different variations. This can be achieved by defining modules that contain only the underlying structure of multi-level constructs (like our Basic module seen earlier). One can also define specific extensions for different constructs, and thus for example able to extend model elements with stereotypes or tags, or even add support for uncommon features such as n-ary edges. Validation rules can also be omitted and redefined (overridden). In this section, we have analyzed how MLMP competes with alternative solutions regarding modularity when experimenting with multi-level constructs.

Combining constructs This is not a previously mentioned requirement but a consequent feature of our playground. In MLMP, one can combine different multi-level constructs. This enables extensive experiments where one can identify correlations between different constructs. By mixing and

⁵ <https://www.aut.bme.hu/Pages/Research/VMTS/DMLA>

matching different multi-level constructs on the same domain model, one can experiment with practical modeling scenarios and specific problems. We have seen in our domain example how different constructs can be combined, which can lead to the discovery of synergies and conflicts between them. Nevertheless, combining constructs opens up many possibilities in the world of multi-level modeling. One such example is (asynchronous) shared modeling between researchers on the same domain model using different constructs. Future research should further develop and confirm our initial findings regarding these claims by combining and analyzing a larger set of existing multi-level constructs.

8 Conclusion

There exist many multi-level modeling approaches in the literature, based on different underlying theories, formalisms, or toolchains. In order to make it easier to discuss, compare, and experiment with different ideas, we proposed the Multi-Level Modeling Playground (MLMP). MLMP is not meant to be a full-fledged multi-level modeling language, its purpose is to enable experimentation with multi-level constructs on a common basis, without needing to rely on different toolchains. Using MLMP, researchers can realize their ideas in practice without having to focus on low-level, technical details. MLMP is built on our level-blind multi-layer modeling framework, the Dynamic Multi-Layer Algebra (DMLA), which provides the underlying modeling structure and performs the task of validating the models. In order to make it easier to specify constructs, we also proposed a textual domain-specific language, the Multi-Level Specification Language (MLSL). MLSL is akin to syntactic sugar upon DMLA as the entities generated from MLSL specifications are considered to be a part of full-fledged DMLA models containing the specified multi-level semantics. Once the specification of a construct written in MLSL is automatically transformed into DMLA entities, it becomes part of DMLA and it is validated accordingly. Researchers can not only specify multi-level constructs in a modular way, they can also create domain models in which they can experiment with the constructs. We evaluated the playground by comparing it to alternatives in experimenting with multi-level constructs: the Eclipse Modeling Framework, and level-adjuvant multi-level solutions in general, using Melanee as a representative example. Some of the main advantages of our solution based on the evaluation are i) high-level and concise specification, and ii) modularity in defining specifications and applying multi-level constructs to domain models. The main disadvantage of our solution is the lack of familiar modeling facilities like the graphical representation of models. Also, we do not claim that in its current, initial state, MLMP is capable of describing all multi-level constructs in the literature. How-

ever, initial results seem to suggest that in time — and with feedback from the research community — it could eventually achieve this goal.

8.1 Future work

Currently, our most important goal is to invite researchers of the multi-level community to experiment with MLMP to improve its mechanisms. For example, after analyzing existing approaches in the literature, we have identified four properties of annotations thus far, but it is possible that more could be added in the future. We also plan to combine and then analyze more multi-level constructs in order to support our conjecture that MLMP has sufficient flexibility for describing them. This would also lead to our vision of (asynchronous) shared modeling, in which researchers can edit the same domain model using different constructs. The combination of constructs may also need hints on how and when a particular construct is efficient. Thus, design patterns on construct creation and usage could also be established. The modularity of MLMP could be expanded by allowing rules from different modules to be mixed when applying them to a domain model. Redundancy could also be further eliminated by introducing constraints inside functions in the validation rules. Alternatively, ideas from feature-oriented programming could be employed to make the approach more compositional when defining new annotations based on existing ones.

Involving researchers to experiment with the playground could also be supported by categorizing MLMP using the feature model created by the authors of TOTEM [29]. While MLMP is not a full-fledged multi-level language, in the future, we could still categorize its capabilities based on whether it can simulate the features described in the feature model. Finally, we should mention that we continuously plan to improve the infrastructure and usability of MLMP, for example, by eventually adding a visual modeling workbench, in order to make it more easily accessible. Moreover, we also plan to improve the performance of validation in general as currently it does not scale well enough to support industrial size models. However, from MLMP's point of view, this is not a severe limitation as performance has not been considered as the main concern of our proposed modeling environment.

Acknowledgements We would like to thank the research community — especially Thomas Kühne and Juan de Lara — for providing insights and feedback on the requirements of selected constructs. We would also like to thank the reviewers whose insightful comments led to a much improved paper.

Funding Open access funding provided by Budapest University of Technology and Economics.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adap-

tation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

A Summary of discussed multi-level approaches

B Comparison of MLSL and DMLAScript

The following examples are inspired by the native mapping of classic potency from previous work [59]. Naming can differ, but the represented concepts have similar roles to their MLSL counterparts.

The code below defines the structure of *PotencyEntity* in DMLAScript, which fills a similar role to *ModelElement* from the Basic module in MLMP. To make the code more compact, “ComplexEntity” is shortened to “CE”. The first three lines specify that *CE.Children*, along with beta and gamma validation operations are cloned, which means that they can later be used by instances. The slots *Potency* and *Level* are introduced with a type constraint of *Number* and a cardinality constraint of 1..1.

Name	Description	Nature
characterization potency [32]	an iteration on classic potency, focusing on abstractness-related problems	approach
classic potency [7]	the original potency notion	approach
classic potency with edges [5]	extended support for handling edges	approach
cross-level relations [14]	level-adjuvant, not based on potency	approach
Cuadrado and de Lara [14]	provides adaptation and extensibility for multi-level tools	approach
DeepJava [34]	Java with multi-level programming	tool
DeepRuby [49]	implementation of dual deep instantiation by extending the Ruby language	tool
DeepTelos [30]	knowledge representation system employing multi-level modeling	tool
DPF Workbench [35]	based on the DPF	tool
dual deep instantiation [48] / modeling [50]	uses dual potency on feature endpoints	approach
endurance properties [36]	assigns potency to field values	approach
FMML ^x [21]	modeling language using intrinsic features that also support operations	approach
formalisation by Rossini et al. [55]	multi-level modeling based on the DPF	approach
leap potency [39]	aims to remedy identity instantiation	approach
Melanee [2]	supports star potency and endurance properties	tool
MetaDepth [38]	supports leap potency and star potency	tool
ML2 [20]	provides a text-based language for conceptual multi-level modeling based on MLT	tool
Modelverse (AToMPM) [46]	model repository that supports classic potency-based multi-level modeling	tool
MultiEcore [41]	potency-based tool based on EMF	tool
Nivel [1]	uses a modified version of the classic potency notion	tool
non-potency elements [28]	an iteration on classic potency, removing certain elements from the potency notion	approach
OMLM [28]	potency-based framework that uses non-potency elements	tool
order [32,33]	represents “the maximum number of classification relationships between an element and its most remote instances”	approach
SLICER [56]	level-blind; scalability is the main focus	tool
star (*) potency [24]	introduces uncertainty in the model	approach
TOTEM [29]	focuses on closing the gap between multi-level and two-level modeling	tool
XModeler [21]	based on FMML ^x	tool

```
PotencyEntity: CE {
  CE.Children;
  Base.BetaValidation;
  Base.GammaValidation;
  @T: CE.Children.T =
  Type: CE.Children.T.T
  {
    Type.IsOverwritable;
    Type.IsPermanent;
    slot Type: CE.Children.T.T.T
      = $Number;
    slot IsInclusive: CE.Children
      .T.T.IsIncl = false;
  };
  @C: CE.Children.C =
  Card : CE.Children.C.Card {
    Cardinality.IsOverwritable;
    Cardinality.IsPermanent;
    slot Min: CE.Children.C.Card.
      Min = 1;
    slot Max: CE.Children.C.Card.
      Max = 1;
  };
  slot Potency: CE.Children;
```

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

```
@Base.AlphaValidation.OpSig
@Base.AlphaValidation.T
@Base.AlphaValidation.C
slot AlphaValidation: Base.
  AlphaValidation =
  operation Bool ID::
    PotencyEntityAlpha(ID
      instance)
  {
    ID meta=call $Meta(instance);
    Object potency = call
      $GetRelevantAttributeValue
      (instance, $PE.Potency);
    Object metaPotency = call
      $GetRelevantAttributeValue
      (meta, $PE.Potency);
    Object level = call
      $GetRelevantAttributeValue
      (instance, $PE.Level);
    Object metaLevel = call
      $GetRelevantAttributeValue
      (meta, $PE.Level);

    if(level==null || potency==
      null) return true;

    if(potency!=null && potency<0)
    {
      call $Log('potency<0');
      return false;
    }
  }
```

```
if(level<0) {
  call $Log('level<0');
  return false;
}

if(potency!=null &&
  potency>level){
  call $Log('potency>level');
  return false;
}

if(metaPotency==null ||
  metaLevel==null)
  return true;

if(metaPotency==0){
  call $Log('metaPotency=0');
  return false;
}

if(potency!=null &&
  potency!=(metaPotency-1)){
  call $Log('potency!=meta-1');
  return false;
}

if(level!=(metaLevel-1)){
  call $Log('level!=meta-1');
  return false;
}
return true;
}
```

The MSL specification of ModelElement includes the two relevant annotations. The low-level details that are present in DMLAScript are generated from the MSL specification. For example, the first three lines of the DMLAScript example, along with the potency and level slots are generated inside the ModelElement entity, with appropriate type and cardinality constraints.

```
ModelElement: entity {}

mutable level in {0..*} on ModelElement, Field;
mutable potency in {0..*} on ModelElement, Field;
```

The alpha validation operation of PotencyEntity is responsible for level and potency-related well-formedness constraints. To make the code more compact, “PotencyEntity” is shortened to “PE”, and log messages are also shortened. The null-check at the beginning implicitly checks if the entity is a module entity (meaning it is not part of a domain model), in which case validation is forbidden. The rest of the operation definition is more straight-forward, representing the requirements of classic potency. Log operations are also called to improve feedback on well-formedness violations.

In the MSL specification, level and potency being a positive number is implicitly encoded by their range. Log messages are automatically displayed for every validation rule upon violation. Instantiation-related rules (#I1-3) and #CP1 are similar to the conditional statements in DMLAScript. MSL automatically determines the context of the validation logic, namely, that the alpha validation operation of the ModelElement entity is where the logic will be generated into.

```
mutable level in {0..*} on ModelElement, Field;
mutable potency in {0..*} on ModelElement, Field;
```

```
#CP1: potency <= level;
#I1: ModelElement.meta.potency != 0;
#I2: ModelElement.level = ModelElement.meta.level - 1;
#I3: ModelElement.potency = ModelElement.meta.potency - 1;
```

C Requirements of selected multi-level constructs

C.1 Classic potency

- CP1 “As its name implies, the level of a model element is an integer representing the model level in which the element resides.” [7] Therefore, every model element (node, edge, field) must be assigned a level n , which reflects that said model element is on level n .
- CP2 The level of a model element must be greater than or equal to 0.
- CP3 The level of a field must be equal to the level of the containing node.
- CP4 Levels are separated by the instantiation relationship. The instance of a model element is on the level exactly one less than that of the model element.
- CP5 A cross-level reference is an edge that runs between two nodes that are on different levels. Cross-level references are not allowed. Strictly speaking, this requirement relates to “strict meta-modeling” frameworks like the OCA [8].
- CP6 “The potency of a model element is an integer that defines the depth to which a model element can be instantiated.” [7] Therefore, every model element must have potency assigned.
- CP7 “...potency of a model element cannot be greater than its level ...” [7]
- CP8 The potency of a model element must be greater than or equal to 0.
- CP9 “The act of instantiating a model element obviously reduces its potency by one.” [7] Thus, the instance of a model element has a potency value that is exactly one less than that of the model element.
- CP10 “Elements whose potency is 0 cannot be instantiated, regardless of their level number.” [7]
- CP11 “A model element which has an *isAbstract* slot containing the value true has a potency of 0.” [7]. Abstract nodes must have a potency of 0 and thus cannot be instantiated.
- CP12 “A field with potency 0 represents a slot, and therefore must have a value. However, a field of higher potency may or may not have a value, depending on whether it is a simple field or a dual field. ...A *simple* field is a field which does not have a value unless it has potency 0...A *dual* field is a field which has a value even for potencies greater than zero.” [7]
- CP13 When instantiating a node, only fields with a potency greater than 0 appear in the instance node (with potencies reduced by exactly 1).
- CP14 “Another good example of a model element with potency 0 is the inheritance relationship between two

elements, as it can not be further instantiated.” [7] Thus, inheritance is a special edge with a potency of 0.

- CP15 When inheriting between two nodes, the descendant must inherit every field of the parent, which has a potency greater than 0. This requirement is our interpretation, as we believe it better reflects and restricts the semantics of inheritance.

C.2 Leap potency

- LP1 “Clabjects and references with a leap potency of n can be instantiated exactly n meta-levels below, but not at intermediate meta-levels.” [39] This is conceptually equivalent to skipping the instantiation of the model element at the intermediate levels. Only nodes and edges can have leap potency assigned.
- LP2 Classic potency and leap potency cannot be used in conjunction on a model element, as their semantics would overlap.
- LP3 Similarly to classic potency, the leap potency of a model element must be less than or equal to its level.
- LP4 When instantiating a node or edge, the leap potency of the instance must be 0.

C.3 Star potency

- SP1 “If an isonym i is instance of a clabject c and c has a potency of “*” then i can have “*” or any non-negative number as potency. “*” is represented in the model by the integer value -1 because the potency is of type integer.” [24] Therefore, the potency value must either be a non-negative number (similarly to classic potency) or -1, which represents star potency. Moreover, after an instantiation, the potency of the instance can be of any value.
- SP2 Star potency can be assigned to nodes, edges, and fields.
- SP3 Star potency cannot be used in conjunction with classic potency on the same model element.
- SP4 “If the corresponding Feature of its type t has a “*” potency the Feature in i can have a non-negative or “*” potency.” [24] This means that star potency features are only allowed in clabjects that have a star potency themselves.

C.4 Order

- O1 “The order of a modeling element can be regarded as a measure of its set-theoretic classification power, since it corresponds to the maximum depth of the “type-of” relationships originating from an element.” [32] In our interpretation, order can only be assigned to nodes.

- O2 The order of a node must be less than or equal to its level. Note that this depends on the particular solution as it could be “order=level” as well.[32]
- O3 The order of a node must be greater than or equal to its potency.[32]
- O4 The instance of a node has an order value that is exactly one less than that of the node.
- O5 Nodes with order 0 cannot be instantiated.
- O6 “Abstract classes are restricted to having indirect instances (through their subclasses).”[32] Therefore, the order of an abstract node must be greater than 0.
- O7 “Pure instances, typically representing domain particulars, have order zero, since they do not classify any instances.”[32] Therefore, a pure instance with an order of 0 must have a potency value of 0.
- O8 In an inheritance relation, the order of the related types must be the same, i.e., the supertype (parent) and subtype (descendant) must have the same order.

References

1. Asikainen, T., Männistö, T.: Nivel: a metamodeling language with a formal semantics. *Software & Systems Modeling* **8**(4), 521–549 (2009). <https://doi.org/10.1007/s10270-008-0103-2>
2. Atkinson, C., Gerbig, R.: Flexible deep modeling with melanee. In: *Modellierung 2016 - Workshopband : Tagung vom 02. März - 04. März 2016 Karlsruhe, MOD 2016*, vol. 255, pp. 117–121. Köllen, Bonn (2016). <http://ub-madoc.bib.uni-mannheim.de/40981/>
3. Atkinson, C., Gerbig, R., Kennel, B.: Symbiotic general-purpose and domain-specific languages. In: *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, p. 1269–1272. IEEE Press (2012)
4. Atkinson, C., Gerbig, R., Kühne, T.: Comparing multi-level modeling approaches. In: *CEUR Workshop Proceedings*, vol. 1286 (2014)
5. Atkinson, C., Gerbig, R., Kühne, T.: A unifying approach to connections for multi-level modeling. In: *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems, MODELS '15*, p. 216–225. IEEE Press (2015). <https://doi.org/10.1109/MODELS.2015.7338252>
6. Atkinson, C., Kühne, T.: Meta-level independent modelling. In: *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, pp. 1–4 (2000)
7. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pp. 19–33. Springer-Verlag, Berlin, Heidelberg (2001). <http://dl.acm.org/citation.cfm?id=647245.719475>
8. Atkinson, C., Kühne, T.: Rearchitecting the uml infrastructure. *ACM Transactions on Modeling and Computer Simulation* **12**(4), 290–321 (2002). <https://doi.org/10.1145/643120.643123>
9. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. *Software & Systems Modeling* **7**(3), 345–359 (2008). <https://doi.org/10.1007/s10270-007-0061-0>
10. Barr, M., Wells, C.: *Category Theory for Computing Science*. Prentice-Hall Inc, USA (1990)
11. BicycleChallenge: <https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/wp-content/uploads/2018/03/multi2018-bicyclechallenge.pdf> (2018)
12. Börger, E., Stärk, R.: *Abstract State Machines: A Method for High-Level System Design and Analysis*, 1st edn. Springer-Verlag, New York, Inc (2003)
13. Bruneliere, H., Langer, P.: You need to extend your models? EMF Facet vs. EMF Profiles. In: *EclipseCon 2012 - Modeling Symposium*. Reston, VA, United States (2012). <https://hal.inria.fr/hal-00688390>
14. Carvalho, V.A., Almeida, J.P.A.: Toward a well-founded theory for multi-level conceptual modeling. *Software & Systems Modeling* **17**(1), 205–231 (2018). <https://doi.org/10.1007/s10270-016-0538-9>
15. Clark, T., Frank, U.: Multi-level constraints. In: R. Hebig, T. Berger (eds.) *Proceedings of MODELS 2018 Workshops, CEUR Workshop Proceedings*, vol. 2245, pp. 103–117. CEUR-WS.org (2018). http://ceur-ws.org/Vol-2245/ocl_paper_2.pdf
16. Clark, T., Sammut, P., Willans, J.: *Applied metamodeling: A foundation for language driven development* (third edition) (2015)
17. Coad, P.: *Object-oriented patterns*. *Communications of the ACM* **35**(9), 152–159 (1992)
18. Cuadrado, J.S., de Lara, J.: Open meta-modelling frameworks via meta-object protocols. *Journal of Systems and Software* **145**, 1–24 (2018). <https://doi.org/10.1016/j.jss.2018.07.023>
19. Ellison, A.M.: Repeatability and transparency in ecological research. *Ecology* **91**(9), 2536–2539 (2010). <https://doi.org/10.1890/09-0032.1>
20. Fonseca, C.M., Almeida, J.P.A., Guizzardi, G., Carvalho, V.A.: Multi-level conceptual modeling: From a formal theory to a well-founded language. In: Trujillo, J.C., Davis, K.C., Du, X., Li, Z., Ling, T.W., Li, G., Lee, M.L. (eds.) *Conceptual Modeling*, pp. 409–423. Springer International Publishing (2018)
21. Frank, U.: Multilevel modeling: Toward a new paradigm of conceptual modeling and information systems design. *Business & Information Systems Engineering* **6**(6), 319–337 (2014). <https://doi.org/10.1007/s12599-014-0350-4>
22. Frank, U.: Toward a unified conception of multi-level modelling: advanced requirements. In: R. Hebig, T. Berger (eds.) *Proceedings of the 5th International Workshop on Multi-Level Modeling (MULTI) 2018*, pp. 718–727. Copenhagen (2018). http://ceur-ws.org/Vol-2245/multi_paper_10.pdf
23. Gay, G., Menzies, T., Cukic, B., Turhan, B.: How to build repeatable experiments. In: *Proceedings of the 5th International Conference on Predictor Models in Software Engineering, PROMISE '09*. Association for Computing Machinery, New York, NY, USA (2009). 10.1145/1540438.1540460
24. Gerbig, R.: The level-agnostic modeling language: Language specification and tool implementation. Master's thesis, University of Mannheim, Mannheim (2011). <https://madoc.bib.uni-mannheim.de/37153/>
25. Gerbig, R., Atkinson, C., de Lara, J., Guerra, E.: A feature-based comparison of melanee and metadepth. In: *Proc. 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., pp. 25–34 (2016). <http://ceur-ws.org/Vol-1722/p2.pdf>
26. Gogolla, M.: Experimenting with multi-level models in a two-level modeling tool. In: C. Atkinson, G. Grossmann, T. Kühne, J. de Lara (eds.) *Proceedings of the 2nd International Workshop on Multi-Level Modelling, CEUR Workshop Proceedings*, vol. 1505, pp. 3–12. CEUR-WS.org (2015). <http://ceur-ws.org/Vol-1505/p1.pdf>
27. Gutheil, M., Kennel, B., Atkinson, C.: A systematic approach to connectors in a multi-level modeling environment. In: *Proc. of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS '08*, p. 843–857. Springer-Verlag (2008). https://doi.org/10.1007/978-3-540-87875-9_58

28. Igamberdiev, M., Grossmann, G., Selway, M., Stumptner, M.: An integrated multi-level modeling approach for industrial-scale data interoperability. *Software and Systems Modeling* **17**(1), 269–294 (2018). <https://doi.org/10.1007/s10270-016-0520-6>
29. Jácome-Guerrero, S.P., de Lara, J.: TOTEM: Reconciling multi-level modelling with standard two-level modelling. *Computer Standards & Interfaces* **69**, 103,390 (2020). <https://doi.org/10.1016/j.csi.2019.103390>
30. Jeusfeld, M.A., Neumayr, B.: Deeptelos: Multi-level modeling with most general instances. In: Comyn-Wattiau, L., Tanaka, K., Song, I.Y., Yamamoto, S., Saeki, M. (eds.) *Conceptual Modeling*, pp. 198–211. Springer International Publishing (2016)
31. Kiczales, G., Rivieres, J.D.: *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA (1991)
32. Kühne, T.: Exploring potency. In: Proc. of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '18, pp. 2–12. ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3239372.3239411>
33. Kühne, T.: A story of levels. In: Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, 2018, pp. 673–682 (2018). http://ceur-ws.org/Vol-2245/multi_paper_5.pdf
34. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style: Multi-level programming with deepjava. *SIGPLAN Not.* **42**(10), 229–244 (2007). <https://doi.org/10.1145/1297105.1297044>
35. Lamo, Y., Wang, X., Mantz, F., Bech, Ø., Sandven, A., Rutle, A.: Dpf workbench: a multi-level language workbench for mde. *Proceedings of the Estonian Academy of Sciences* **62**(1), 3–15 (2013)
36. Lange, A., Atkinson, C.: On the rules for inheritance in lml. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 113–118 (2019). <https://doi.org/10.1109/MODELS-C.2019.00021>
37. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: Emf profiles: A lightweight extension approach for emf models. *Journal of Object Technology* **11**(1), 8:1-29 (2012). <https://doi.org/10.5381/jot.2012.11.1.a8>
38. de Lara, J., Guerra, E.: Deep meta-modelling with metadepth. In: *Objects, Models, Components, Patterns*, pp. 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
39. de Lara, J., Guerra, E., Cobos, R., Moreno-Llorena, J.: Extending Deep Meta-Modelling for Practical Model-Driven Engineering. *The Computer Journal* **57**(1), 36–58 (2012). <https://doi.org/10.1093/comjnl/bxs144>
40. de Lara, J., Guerra, E., Cuadrado, J.S.: When and how to use multi-level modelling. *ACM Transactions on Software Engineering and Methodology* **24**(2), 12:1-12:46 (2014). <https://doi.org/10.1145/2685615>
41. Macías, F., Rutle, A., Stolz, V., Rodríguez-Echeverría, R., Wolter, U.: An approach to flexible multilevel modelling. *Enterprise Modelling and Information Systems Architectures* **13**, 10:1-10:35 (2018). <https://doi.org/10.18417/emisa.13.10>
42. Mezei, G., Somogyi, F.A., Theisz, Z., Urbán, D., Bácsi, S.: Towards mainstream multi-level meta-modeling. In: S. Hammoudi, L.F. Pires, B. Selic (eds.) *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019*, Prague, Czech Republic, February 20–22, 2019, pp. 481–488. SciTePress (2019). <https://doi.org/10.5220/0007580404830490>
43. Mezei, G., Somogyi, F.A., Theisz, Z., Urbán, D., Bácsi, S., Palatinzsky, D.: A bootstrap for self-describing, self-validating multi-layer metamodeling. In: *Proceedings of the Automation and Applied Computer Science Workshop*, pp. 28–38 (2019)
44. Mezei, G., Theisz, Z., Bácsi, S., Somogyi, F.A., Palatinzsky, D.: Towards flexible, rigorous refinement in metamodeling. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 455–459. IEEE Computer Society, Los Alamitos, CA, USA (2019). <https://doi.org/10.1109/MODELS-C.2019.00073>. <https://doi.ieeecomputersociety.org/10.1109/MODELS-C.2019.00073>
45. Mezei, G., Theisz, Z., Urbán, D., Bácsi, S.: The bicycle challenge in dmla, where validation means correct modeling. In: R. Hebig, T. Berger (eds.) *Proceedings of MODELS 2018 Workshops: 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*, Copenhagen, Denmark, October, 14, 2018, *CEUR Workshop Proceedings*, vol. 2245, pp. 643–652. CEUR-WS.org (2018). http://ceur-ws.org/Vol-2245/multi_paper_2.pdf
46. Mierlo, S.V., Barroca, B., Vangheluwe, H., Syriani, E., Kühne, T.: Multi-level modelling in the modelverse. In: C. Atkinson, G. Grossmann, T. Kühne, J. de Lara (eds.) *Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014)*, Valencia, Spain, September 28, 2014, *CEUR Workshop Proceedings*, vol. 1286, pp. 83–92. CEUR-WS.org (2014). <http://ceur-ws.org/Vol-1286/p9.pdf>
47. OMG: MetaObject Facility. <http://www.omg.org/mof/> (2005). Accessed:2020-03-10
48. Neumayr, B., Jeusfeld, M.A., Schrefl, M., Schütz, C.: Dual deep instantiation and its conceptbase implementation. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) *Advanced Information Systems Engineering*, pp. 503–517. Springer International Publishing (2014)
49. Neumayr, B., Schuetz, C.G., Horner, C., Schrefl, M.: Deepruby: Extending ruby with dual deep instantiation. In: *Proceedings of MODELS 2017 Satellite Event: Workshops, CEUR Workshop Proceedings*, vol. 2019, pp. 252–260. CEUR-WS.org (2017). http://ceur-ws.org/Vol-2019/multi_7.pdf
50. Neumayr, B., Schuetz, C.G., Jeusfeld, M.A., Schrefl, M.: Dual deep modeling: multi-level modeling with dual potencies and its formalization in f-logic. *Software & Systems Modeling* **17**(1), 233–268 (2018). <https://doi.org/10.1007/s10270-016-0519-z>
51. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: Akşit, M., Matsuoka, S. (eds.) *ECOOP'97 – Object-Oriented Programming*, pp. 419–443. Springer, Berlin Heidelberg, Berlin, Heidelberg (1997)
52. Prinz, A., Mezei, G.: The art of bootstrapping. In: S. Hammoudi, L.F. Pires, B. Selic (eds.) *Model-Driven Engineering and Software Development - 7th International Conference, MODELSWARD 2019*, Prague, Czech Republic, February 20–22, 2019, *Revised Selected Papers, Communications in Computer and Information Science*, vol. 1161, pp. 182–200. Springer (2019). https://doi.org/10.1007/978-3-030-37873-8_8
53. ProcessChallenge: https://www.wi-inf.uni-duisburg-essen.de/MULTI2019/wp-content/uploads/2019/05/multi_process_modeling_challenge.pdf (2019)
54. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Lamo, Y.: A graph transformation-based semantics for deep metamodelling. In: Schürr, A., Varró, D., Varró, G. (eds.) *Applications of Graph Transformations with Industrial Relevance*, pp. 19–34. Springer, Berlin Heidelberg, Berlin, Heidelberg (2012)
55. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Wolter, U.: A formalisation of deep metamodelling. *Formal Aspects of Computing* **26**(6), 1115–1152 (2014). <https://doi.org/10.1007/s00165-014-0307-x>
56. Selway, M., Stumptner, M., Mayer, W., Jordan, A., Grossmann, G., Schrefl, M.: A conceptual framework for large-scale ecosystem interoperability and industrial product lifecycles. *Data & Knowledge Engineering* **109**, 85–111 (2017). <https://doi.org/10.1016/j.datak.2017.03.006>
57. Simons, D.J.: The value of direct replication. *Perspectives on Psychological Science* **9**(1), 76–80 (2014). <https://doi.org/10.1177/1745691613514755>

58. Somogyi, F.A., Mezei, G., Urbán, D., Theisz, Z., Bácsi, S., Palatinszky, D.: Multi-level modeling with DMLA - A contribution to the MULTI process challenge. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15–20, 2019, pp. 119–127. IEEE (2019). <https://doi.org/10.1109/MODELS-C.2019.00022>
59. Somogyi, F.A., Theisz, Z., Bácsi, S., Mezei, G., Palatinszky, D.: Multi-level modeling without classical modeling facilities. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODEL-SWARD., pp. 393–400. INSTICC, SciTePress (2020). <https://doi.org/10.5220/0008973503930400>
60. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
61. Theisz, Z., Bácsi, S., Mezei, G., Somogyi, F.A., Palatinszky, D.: By multi-layer to multi-level modeling. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 134–141 (2019). <https://doi.org/10.1109/MODELS-C.2019.00024>
62. Theisz, Z., Urbán, D., Mezei, G.: Constraint modularization within multi-level meta-modeling. In: Damaševičius, R., Mikašytė, V. (eds.) Information and Software Technologies, pp. 292–302. Springer International Publishing, Cham (2017)
63. Urbán, D., Mezei, G., Theisz, Z.: Formalism for static aspects of dynamic metamodeling. Periodica Polytechnica Electrical Engineering and Computer Science **61**(1), 34–47 ((2017)). <https://doi.org/10.3311/PPee.9547>. <https://pp.bme.hu/eecs/article/view/9547>
64. Urbán, D., Theisz, Z., Mezei, G.: Self-describing operations for multi-level meta-modeling. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODEL-SWARD., pp. 519–527. INSTICC, SciTePress (2018). <https://doi.org/10.5220/0006656105190527>
65. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley Longman Pub. Co., Inc, Boston, MA, USA (2003)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ferenc Somogyi has received his master's degree in 2016 from Budapest University of Technology and Economics. He is currently working on his PhD. His main research topics are compiler and programming language design, model-based software development, and multi-level modeling. His current research interest focuses on enhancing model-driven approaches with text-based modeling techniques.

Gergely Mezei has received his master's degree in 2004 and his PhD in 2008 from Budapest University of Technology and Economics. Now, he is an associate professor at the same university. His research focuses mainly on compilers for domain-specific languages, software modeling, model visualization, model transformations and multi-level modeling. He is the leader of the DMLA research group.

Zoltán Theisz received his master's degree in computer science in 1994 and his postgraduate specialization in nuclear reactor engineering in 2019 from Budapest University of Technology and Economics. He started his professional career in the telecom industry, working at Ericsson in different technical positions such as telecom expert, software architect, modeling expert and research fellow. During these years, he got deeply involved in various research projects targeting model-driven soft-

ware engineering, meta-model design, ontology engineering and semantic services orchestration in ICT and IIoT network management. Later he joined evopro group Ltd. where he has been designing and developing automation solutions combining state-of-the-art industrial control, heterogeneous parallel computing and customized mobile clients. His current research interest focuses on the practical application of multi-level meta-modeling for industrial control in big physics and the nuclear industry.

Sándor Bácsi has received his master's degree in 2019 from Budapest University of Technology and Economics. Now, he is a PhD student at the same university. His main topics of interest are model-driven software engineering including multi-level modeling, model visualization and graph transformation.

Dániel Palatinszky has received his master's degree in 2020 from Budapest University of Technology and Economics. He is currently working as a full-time back-end developer at Qualysoft and Betsson Group while he is a part-time programming teacher as well. He is also a member of the DMLA research group at the university he graduated from where he mainly focuses on language development and interpreters.