# By multi-layer to multi-level modeling

Zoltán Theisz[1] Sándor Bácsi[2] Gergely Mezei[2] Ferenc A. Somogyi[2] Dániel Palatinszky[2]
*evopro Systems Engineering Ltd.[1]*
*Department of Automation and Applied Informatics,*
*Budapest University of Technology and Economics[2]*
Budapest, Hungary
zoltan.theisz@evopro.hu[1]
{sandor.bacsi, gergely.mezei, somogyi.ferenc, palatinszky.daniel}@aut.bme.hu[2]

***Abstract*—Multi-level modeling has a well-defined and commonly agreed on aim of avoiding any kind of accidental complexity by the introduction of meta-levels. However, the actual means of achieving this goal are left to the discretion of the particular approach to decide on. Potency notion-based clabjects provide a suitable trade-off in this regard, hence other approaches tend to imitate these characteristics. Dynamic Multi-Layer Algebra (DMLA) is such an alternative formal modeling technique that has already been tested successfully against multi-level challenges such as the MULTI 2018 workshop's Bicycle Challenge. Although DMLA has proved its merit, it has still been lacking the capability of integrating object-oriented features such as inheritance into its formalism. Therefore, in this paper, we showcase one potential way of incorporating inheritance with abstract entities into DMLA without having to relinquish any of its formal precision such as self-validation or self-description. The paper both describes our technical solution and illustrates it through a model excerpt borrowed from the Bicycle Challenge.**

***Index Terms*—Meta-modeling, Multi-level Modeling, Deep Instantiation, Inheritance, Abstract Class, Potency Notion**

## I. Introduction

Multi-level modeling is the common term widely used to cover a plethora of meta-model based modeling approaches with the shared goal to avoid the emergence of any kinds of accidental complexity [1]. Accidental complexity may be caused by not having applied meta-levels adequately either during the model building process or in the resulted domain models. In order to achieve this ambitious goal, the various multi-level modeling approaches have successfully introduced their particular understanding of the level concept. Moreover, they combined levels with some well-established state-of-the-art model building methodology. In many contemporary multi-level modeling paradigms, the traditionally distinct concepts of class and object have been succinctly fused into an inseparable entity named clabject [2]. One of the major characteristics of popular multi-level modeling methodologies is the capability of successfully blending inter-level relations between clabjects such as instantiation with intra-level relations among clabjects such as inheritance. We believe that the particular way of combining instantiation and inheritance in modeling is a clear characterization of any modeling paradigm whether it can be classified as multi-level or not.

In our research group, we started investigating the possibility of creating a self-contained meta-modeling theory some years ago. We aimed to critically reconsider the fundamentals of state-of-the-art multi-level modeling approaches and to build a new framework that could be applied as a multi-level modeling approach without being anchored in any underlying traditional modeling technique. The work of our research resulted in Dynamic Multi-Layer Algebra (DMLA) [3], which is a self-validating meta-modeling formalism relying on gradual model constraining through its interpretation of the classical instantiation relation. Since our main research goal was to create a self-validating meta-modeling framework, where instantiation is only a means to reach that goal, we named the approach *multi-layer* instead of *multi-level*. Nevertheless, we had also tested the applicability of DMLA through multi-level modeling requirements. During this evaluation process, we had to realize that DMLA was not expressive enough by its native features as a good candidate for a proper multi-level framework. In fact, we had to realize that our abstraction handling mechanisms were sometimes too restrictive for practical multi-level use-cases due to the exclusive reliance on gradual constraining via instantiation. We classified DMLA's most frequently applied modeling patterns by their direct application to the Bicycle Challenge [4]. Here, it became obvious that two of the most important multi-level features missing from DMLA were (i) the correct handling of the inheritance relation and (ii) the distinction between abstract and concrete entities. Although we were able to come up with a list of the necessary DMLA patterns in order to solve the challenge completely, we had seen an ever-increasing gap between multi-layer and multi-level expectations vis-a-vis model design as well.

In this paper, we intend to demonstrate that selected multi-level modeling features can indeed be supported by DMLA. Our target of such a feature demonstration is the incorporation of abstract classes and explicit inheritance into DMLA's standard Bootstrap. The key to the solution is a new notion referred to as tuple-number, or t-number for short. Based on the notion of t-number, the validation mechanisms of DMLA can be extended and thus the goal is achievable. For the sake of straightforward demonstration, we will rely on a model excerpt borrowed from the Bicycle Challenge in order to explain the proposed technical solution in DMLA.

The paper is organized as follows: in Section II we elaborate on our motivation and introduce the technical challenge inspired by our DMLA pattern-based solution of the Bicycle Challenge. Then, Section III is dedicated to discuss the related

work in multi-level modeling. In Section IV, we shortly describe DMLA and also introduce our multi-level representation of inheritance via abstract entities. Next, in Section V, we describe our algorithm that can properly differentiate DMLA entities which are needed only for purposes of multi-level inheritance from those which represent genuine DMLA instantiation through gradual constraining. Here, we also explain the most relevant technical details of the algorithm, both on the conceptual level and in pseudo code. Then, in Section VI, the motivating example is presented again, however this time extended with the proposed solution. Finally, in Section VII, we conclude the paper and also describe our future research plans of applying similar strategies on other multi-level notions.

## II. Background

In this section, we present the background and motivation behind our new notion. Before digging deeper into the field, let us take a short look on instantiation and inheritance. The pivotal point of multi-level modeling is the instantiation relation that connects the object to its class: the object represents an instance, that is, an element of the set represented by the class. Hence, some features of the class must be concretized or constrained in order for the object(s) to implement the instantiation relation to the class. Therefore, instantiation is a vertical relation. On the contrary, inheritance acts among the classes on the same abstraction level as a horizontal relation. Also, note that inheritance is additive by their nature: the base class definition can only be extended and cannot be restricted by the inheriting classes (otherwise it would violate the Liskov substitution principle). Therefore, inheritance tend to combine features of the base class(es) the target class inherits from.

The MULTI Process Challenge [5] is the latest expectation set of requirements a multi-level modeling framework must satisfy. Being a commonly agreed challenge of the multi-level community, it is a good starting point of our discussion. Some parts of the challenge are worth mentioning in order to support our motivation. For example, it is described in the challenge that there are different kinds of gateways, i.e. and-split, or-split, etc. Another requirement is to have initial and final task types, which are specialized from the general task type. Such requirements show that one of the main purposes of relying on OOP-styled inheritance with abstract classes is to build a taxonomy of the contained features. In the case of gateways and process types, these features may become attributes and/or operations of the corresponding clabjects. Without the notion of t-number introduced in this paper, it is difficult for DMLA to satisfy such requirements by its native multi-layer instantiation formalism.

In general, popular multi-level modeling approaches cover instantiation mainly by the concept of the potency notion, where an integer number stands for the meta-level the clabject is placed on [6]. When two clabjects are connected via instantiation, the potency of the clabject in the role of the class, must be greater by one than the one with the role of the object. Also, concrete objects must always have potency value zero, that is, they must be already fully concretized as they cannot be further instantiated. However, when it comes to inheritance, potency notion based approaches rely on standard OOP-defined inheritance. Thus, the potency number of the in-relation clabjects must be the same, while the rest of the inheritance semantics is equivalent to usual MOF/UML interpretation. One of the best practical implementations of the two relations in consistent harmony can be found in DeepJava [7].

Although, in many modeling domains, the above-mentioned hybrid handling of both relations is fully satisfactory, there are many practical domains, where a model must contain both fully abstract classes and fully concrete objects at the same time. In those situations, one must not assume that the abstract class is there only because it has not been concretized yet. In some cases, the abstract class purposefully stands for purely abstract features of later concrete objects. For example, one may think of an emerging feature within the abstract class, which represents some computation on a group of objects and thus may contain derived attribute(s) which must be computed on the corresponding features of concrete objects. Those abstract classes should not exist in the real world of the technical domain. However, the abstract classes can play a more than essential role to control the behavior of the concrete objects. A very good example of that feature are the query functions of the Bicycle Challenge which are used for example, to calculate the average price for a bicycle model type, e.g. Mountain bike.

When modelers build taxonomies on a particular abstraction level, they usually prefer to rely on abstract classes as types in order to introduce and/or reuse existing clabject features. The requirements on the task types of the process challenge are good examples of this phenomenon. Obviously, concrete classes can be further instantiated into concrete objects as the modeling goes towards the real domain objects. Hence, if we follow any of the inheritance/instantiation paths from the concrete objects upwards through their meta classes, there will be eventually clabjects that are fully abstract and they are directly related only to each other via inheritance. Hence, the relationship paths from the very abstract clabjects downwards to their fully concrete objects contain a potentially alternating sequence of inheritance and instantiation relations.

As we mentioned, multi-level modeling aims to eliminate any form of accidental complexity. Therefore, managing abstraction through inheritance must also comply with this principle. Although we can easily model many multi-level modeling notions in DMLA by modeling patterns, as we demonstrated in our solution of the Bicycle Challenge [4], we had been still lacking standalone entities representing multi-level notions (e.g. potency notion) in the standard Bootstrap.

With inheritance in place, DMLA entities which represent abstract classes can be used as grouping constructs for other entities which share some common features at a particular abstraction level. Although the grouping facility of the abstract classes is very appealing in the beginning, one will have to face the consequences rather quickly. Namely, the inheritance/instantiation graph of such a multi-level model in

DMLA will not be balanced. As an illustration, let us have a quick look at a fragment of the Bicycle Challenge (Figure 1). Note that we have simplified and slightly modified the model for the sake of clarity. Only those parts of the entities are kept that are necessary to illustrate the phenomenon.

*BicycleEntity* is the root of the tree. *Component*, on the left hand side, is a fully abstract class. It does have inheriting classes such as *Wheel* and *Frame*, and it gets through further instantiation via *Frame*. However, *Component* itself does not have any fully concretized objects. Nevertheless, it does have an operation implementing the emergent feature, called *CountComponents*, which counts all the concrete *Component* instances such as for example *RaceFrame*. The query function operates on the meta-level of *Component*.

If we follow the entities on the right hand side, we can notice that *BicycleEntity* is eventually fully instatiated as we go on. Hence, taking into account the whole model, we will have an unbalanced tree of inheritence and instantiation relations: one level of instantiation on the left, three levels of instantiation on the right. The real technical challenge for us is how to precisely define inheritance in a multi-level consistent way so that DMLA's model validation mechanism can correctly distinguish inheritance and instantiation.

In order to put our technical solution into the right perspective, we had to study state-of-the-art multi-level modeling research considering abstract clabjects and inheritance. Hence, in the sequel, in Section III, we survey the various potency based multi-level modeling approaches regarding their handling of abstract clabjects in models, then, in Section IV, we introduce our particular way of formalizing similar semantics in DMLA's self-validation framework.

## III. Related work

Deep instantiation is usually achieved via the potency notion [6]. The potency value is a number assigned to entities, features and relationships and it represents the number of levels the element has to pass before a value can be assigned to it. As we already discussed in Section I, this approach uses the concept of clabjects, which means that every entity has a class (type) facet, and an object (value) facet. This makes it possible to achieve deep instantiation, while also keeping the process constrained, i.e. we cannot assign values on levels where it is not allowed to.

Potency notion has proved itself to be a sound technique that is employed in a number of multi-level modeling approaches, for example, both Melanee [8] and XModeler [9] use it in order to realize deep instantiation. XModeler is slightly different in the sense that the potency values describe the level the given element must get a value assigned to, instead of the number of levels thereafter the same assignment should take place. In Melanee, it is possible that an unlimited number of instances can be created and a feature can be passed over an unlimited number of instantiation levels. Hence, there is no canonical definition of the potency notion yet, thus, in practice, variations may occur, like the ones observed between Melanee and XModeler. However, this does not alter the meaning of the core concept of the potency notion.

Here, it is also important to highlight a categorization which can help to better understand the main contribution of this paper. Levels have been present in software modeling since the introduction of UML. Kühne argues that well-defined levels "safeguard against ill-formed models" [10], which means that it is easier to create valid, well-formed models. Atkinson and Kühne compare this concept to strong-typed programming languages, while approaches that "ignore the fact that there are levels" are akin to weakly-typed languages [11]. They call the former *level-adjuvant*, the latter *level-blind*. For example, Melanee and XModeler support the principles of the level-adjuvant style of modeling.

Compared to those approaches, in DMLA, we do not require instantiating every element on all layers of the model. Therefore, it can be easier to map the characteristics of the domain step-by-step since we are not forced to group elements into levels. Since we do not have to specify entire layers of the domain at once, the model is described progressively, where certain parts of it can be left ambiguous for the time being.

Taking a closer look at the literature on potency notion again, it also reveals a number of open questions. For example, Kühne has highlighted several problems surrounding classic potency [12]: intermediate abstract class inconsistency, potency zero ambiguity and implicit subclass anomaly. The notion of order is also used. Order corresponds to the maximum depth of the type-of relationships originating from a modeling element. The "type-of" relationship includes both "direct type-of" and "indirect type-of". Kühne claims that a much deeper distinction between order and potency can be used, paving the way for a new approach. The main insight is to consider order as referring to the depth of classification, while potency as referring to the depth of instantiation. Our work, presented in this paper, proposes a different method to solve a similar problem, in order to support the modelers facing unbalanced level trees of entities in level-blind environments.

Beyond potency, there are other approaches in the literature which introduce multi-level modeling over existing modeling paradigms. One of the most notable is MultEcore [13], which extends the standard modeling features of the Eclipse Modeling Framework in order to provide functionality to create multi-level hierarchies. MultEcore is an alternative solution to the potency notion and it introduces multiple levels by allowing any two adjacent levels in that hierarchy to be represented as an Ecore metamodel and an XMI instance. Obviously, MultEcore has both the notions of inheritance and abstract class inherited form Ecore. Compared to MultEcore, DMLA is self-contained as it uses its own modeling formalism and operation language for model manipulation instead of relying on another modeling paradigm.

## IV. DMLA in a nutshell

### A. Basics

Dynamic Multi-Layer Algebra (DMLA) [14], [15] is our multi-layer modeling framework that consists of two parts:

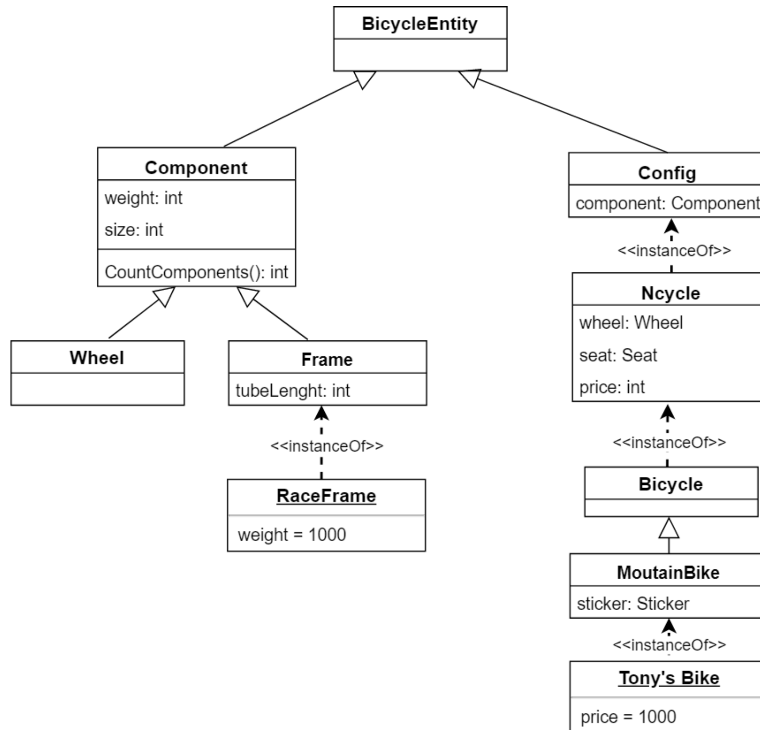


Fig. 1. Unbalanced tree of entities from the Bicycle Challenge

(i) the Core containing the formal definition of modeling structures and its management functions; (ii) the Bootstrap having a set of paradigm-specific reusable entities for modeled domains.

According to the Core, each entity is defined by a 4-tuple (unique ID, meta-reference, attributes and concrete values). Besides these tuples, the Core also defines basic functions to manipulate the model graph, for example, to create new model entities or query existing ones. These definitions form the Core of DMLA, which is defined over an Abstract State Machine (ASM) [16]. The states of the state machine represent the snapshots of dynamically evolving models, while transitions (e.g. deleting a node) stand for modifications between them.

The Bootstrap [17] is an initial set of modeling constructs and built-in model elements which are needed to customize the standard modeling structure of DMLA to specific domain applications. The Bootstrap itself can be modified, so even the semantics of valid instantiation can be easily re-defined if needed. From the domain modeler's point of view, this unabridged freedom may seem unnecessary and even uncomfortable at first glance, but one should keep in mind that the internal modeling details of the Bootstrap remain conveniently hidden, while the external interface that the particular modeling paradigm offers to the domain experts is clear and straightforward to use. However, the separation of the Core and the Bootstrap concepts and the self-modeled instantiation make it also possible to introduce various multi-level modeling notions, e.g. the potency, or the t-number notion.

Practical DMLA models are written in a scripting language, the so-called DMLAScript, instead of being produced as sets of 4-tuples. DMLAScript is equipped with an Xtext-based [18] workbench that automates efficient 4-tuple production. Domain modelers carry out all their model building tasks in DMLAScript, which makes domain modeling in DMLA really fast.

### *B. Instantiation*

Instantiation semantics of DMLA means gradual model constraining and thus it has several peculiarities. Whenever a model entity claims another entity as its meta, the framework automatically validates if there is indeed a valid instantiation between the two entities. However, unlike other modeling approaches, the rules of valid instantiation are not encoded in an external programming language (e.g. Java), they are instead modeled by the Bootstrap. Therefore, both the main validation logic and also the constraints used by the validation, like checking type and cardinality conformance must be precisely modeled within the Bootstrap. Moreover, since the modeled validation logic is not predefined by some inherent instantiation semantics of DMLA, the instantiation itself is Bootstrap-dependent. The operations needed for encoding the concrete validation logic are modeled by their abstract syntax tree (AST) representation of 4-tuples in the Bootstrap. Note that from now on, we always refer to the constructs defined by the standard Bootstrap whenever we discuss any details of DMLA unless it is mentioned otherwise explicitly.

DMLA is a *fluid style* modeling approach, that is, although each entity has a meta-entity, levels are not explicitly modeled in advance. Each modeled entity can refer to any other entity

along the meta-hierarchy, unless cross-level referencing is found to be contradictory to the validation rules. We use the term fluid metamodeling to characterize this freedom in referencing between the entities, as the references are flowing freely between the levels. Note that due to the self-modeled feature of the Bootstrap, it is possible to create a level-adjuvant Bootstrap, but up till now we have preferred level-blind fluid metamodeling instead.

Entities may have attributes referred to as slots, describing a part of the entity similarly to classes having properties in Object-Oriented Programming. Also, similar to the relation between an entity and its meta-entity, each slot originates from a meta-slot defining the constraints it must take into consideration. When instantiating an entity, all its slots are to be validated against their meta-slots. This is how DMLA checks the constraints applied on the meta-slots. These constraints include, but are not limited to type and cardinality constraints. It is also possible to create domain-specific slot constraints, or attach complex constraints validating a certain configuration of several slots.

Besides gradually narrowing the constraints imposed on a slot, DMLA enables the division of a slot into several instances thus fragmenting a general concept into several, more specific ones. For example, a general purpose meta-slot *Components* can be instantiated to *Display*, *CPU* and *HardDrive*. Moreover, it is also possible to omit a slot completely during the instantiation if that does not contradict the cardinality constraint.

Another important feature of DMLA is that, when an entity is being instantiated, one can decide which of the slots are being instantiated and which are merely cloned, that is, being copied to the instance without any modifications. It means that one can keep some of the slots intact while the others are being concretized. This feature with the support of fluid meta-modeling makes it possible to build structures composed of parts from different abstraction levels. For example, a preliminary car concept can have a fully concretized engine, while also having a highly abstract description of wheels and chassis. Later on, one can concretize abstract details and thus create a concrete car specification. This behavior clearly reflects DMLA's way of modeling: one can gradually tighten the constraints on certain parts of the model without having to impose any unrelated obligations on other parts of the model which may not be known at that time.

In DMLA, inheritance between the entities is emulated by instantiation. Although instantiation and inheritance relations look very similar on the surface, for example both are having grouping and substitution goals, they are significantly different as discussed before. Instantiation is natively supported by DMLA, but inheritance is not. Despite the differences, we can simulate inheritance by keeping a general usage slot with infinite cardinality in entities (and instantiating this slot whenever a new feature is to be added) and clone all other slots (to obey the rules of the base class).

Nevertheless, we cannot distinguish entities which are needed for the purpose of handling abstraction from those which are used for actual instantiation. This was the main incentive to explicitly introduce the concept of the t-number.

### C. Domain engineering

From the domain engineer's point of view, the most important part of DMLA are those elements, which can be directly used in creating the domain. A usual entry point of domain definition is the *ComplexEntity* defined in the Bootstrap. *ComplexEntity* has a slot called *Children*. The cardinality of the slot *Children* allows any number of instances (0..*) of any practically available type.

Since in DMLA, all slots must have an origin, it is not possible to add new features to an entity, unless the meta-entity has an appropriate meta-slot. The slot *Children* is the usual way to allow to extend entities by new features. In these cases, *Children* is divided into a slot representing the new feature, while the original, general-purpose *Children* slot may also be kept. If we omit the slot, we can deny the introduction of other features in the instances of the given entity.

In DMLA, validation is based on three types of formulae: alpha, beta and gamma. Alpha type formulae have been designed to validate an entity against its instances, by simply checking if the instantiation relation can be verified between the meta and instance entities. During validation, the framework iterates over all entities of the model and invokes alpha type validation on every entity – meta-entity pair. In contrast, beta type formulae are in-context checks: they are used when an entity has to be validated against multiple related entities. For example, cardinality-like constraints are evaluated by beta formulae due to their underlying one-to-many relation. Gamma formulae are used when validation cannot be applied locally, but requires checking for a global conditions, e.g. the uniqueness of an identifier.

All entities have these formulae, which are very permissive at the root of the instantiation hierarchy, but they are getting more and more restrictive as the abstraction level of the entity is being decreased. Validation and thus the definition of valid instantiation relies on these formulae.

## V. The T-number notion

The notion of t-number consist of two components: a slot and an algorithm. The slot contains an integer number, the t-number, whose value is validated by the algorithm implemented as an alpha validation formula. In this section, we define the requirements against the t-number, then we introduce the entity *TBase* (the entry point for the notion), and finally we present the validation algorithm itself.

### A. Requirements

The concept behind the t-number is quite simple: real instantiation increases the value by one, while abstract entities reset the value to zero. The numbering starts from zero at the root of the hierarchy unlike in the case of several other notions, e.g. order [12]. Numbering from the root allows the modeler to measure the maximum depth of "type-of" relationship chains and differentiate abstract entities in DMLA. Moreover, we do

not need to maintain the value of t-number at higher layers when adding a new instantiation to the bottom.

We specified the following requirements for t-number:

1) If classification is used, the t-number is set to zero.
   a) Existing slots are cloned and
   b) No slot is omitted and
   c) No slot is created, except if its meta is ComplexEntity.Children.
2) If 1) is not met, it is a real instantiation, thus, t-number is increased by one.

Note that dividing the slot *ComplexEntity.Children* (choosing *Children* as a meta-slot) does not reset the value of the t-number. This behaviour keeps the characteristics of the classification relationship, similarly to the classical inheritance, when one may introduce new features in the inheritance chain.

By defining these requirements, we identified what we need in order to distinguish vertical instantiation from classification instantiation. As next, we elaborate the details of our implementation.

## B. Implementation

In this section, we describe our algorithm of properly distinguishing entities which are needed for the purpose of abstraction from those which are used for instantiation.

We introduced a specific entity (*TBase*), which serves as the new entry point of the domain definition. *TBase* instantiates *ComplexEntity* and clones slot *Children*. *TBase* contains two important features (i) a specific slot (*TNumber*) for storing the value of the t-number; and (ii) an alpha formulae to validate the rules of the t-number (*TNumberAlphaValidation*). Figure 2 shows the structure of *TBase*. The type constraint on *TNumber* grants that only number values are accepted, and the cardinality is restricted to 1..1 (the slot is mandatory and unique).

**TBase: ComplexEntity**

ComplexEntity.Children
>TNumber: ComplexEntity.Children {T:$Number, C:1..1}
>TNumberAlphaValidation: Base.AlphaValidation{Op:$Bool:(ID)}

Fig. 2. Entity TBase.

We used alpha formulae to validate an entity against its instances, by simply checking the correctness of the t-number in every entity considering the rules of t-number. During validation, the framework iterates over all entities of the model and invokes alpha validation on every entity and its meta-entity. The mechanism enforces that all entities must obey to all t-number requirements along their meta hierarchy. In the following, we describe the mechanism of *TNumberAlphaValidation*. Algorithm 1 shows the pseudo code of t-number alpha validation.

The algorithm consists of four major parts: (i) Querying the basic elements for the algorithm (Ln. 2-4),(ii) Checking the slots of the particular entity against the slots contained

```
1:  operation Bool TNumberAlphaValidation(instance)
2:  meta = Meta(instance)
3:  tnumber = GetValue(instance, $TNumber)
4:  metanumber= GetValue(meta, $TNumber)
5:  for all slot s ∈ slots do
6:    metaslot = Meta(s)
7:    if not Contains(meta,s) then
8:      if not DerivesFrom($TNumber,s) and not
        (metaslot = $ComplexEntity.Children) then
9:        if tnumber = metanumber +1 then
10:         return true
11:       end if
12:     else
13:       return false
14:     end if
15:   end if
16: end for
17: for all meta_slot m ∈ meta_slots do
18:   if not Contains(instance,m) then
19:     if not DerivesFrom($TNumber,m) then
20:       if tnumber = metanumber +1 then
21:         return true
22:       end if
23:     else
24:       return false
25:     end if
26:   end if
27: end for
28: if tnumber = 0 then
29:   return true
30: end if
31: return false
```

**Algorithm 1:** Pseudo code of t-number algorithm

by the meta (Ln. 5-16), (iii) Checking the slots in the meta against the instance slots (Ln. 17-27) and (iv) Validating the classification relationship among the instance and its meta-entity (Ln. 28-31).

In *TNumberAlphaValidation*, as first step of the algorithm, the basic building blocks are queried, which are needed to construct the validation logic for t-number. We query the meta of the particular entity by calling the *Meta* built-in operation on *instance* (Ln. 2). We also query the t-number value of the instance (Ln. 3) and the meta-entity (Ln. 4).

The second part of the algorithm iterates through the slots of the particular entity and checks whether a new slot is introduced compared to the slots of the meta-entity by calling *Contains* (Ln. 7). Note that new slots are not really new, but created by dividing an existing meta-slot into several parts using the slot division feature of DMLA. Although the t-number itself is also stored in a specific slot, obviously it is not included in the validation logic (Ln. 8).

According to the specification, the division of the slot *ComplexEntity.Children* (choosing *Children* as a meta-slot) does not reset the value of the t-number (Ln. 9). If a new

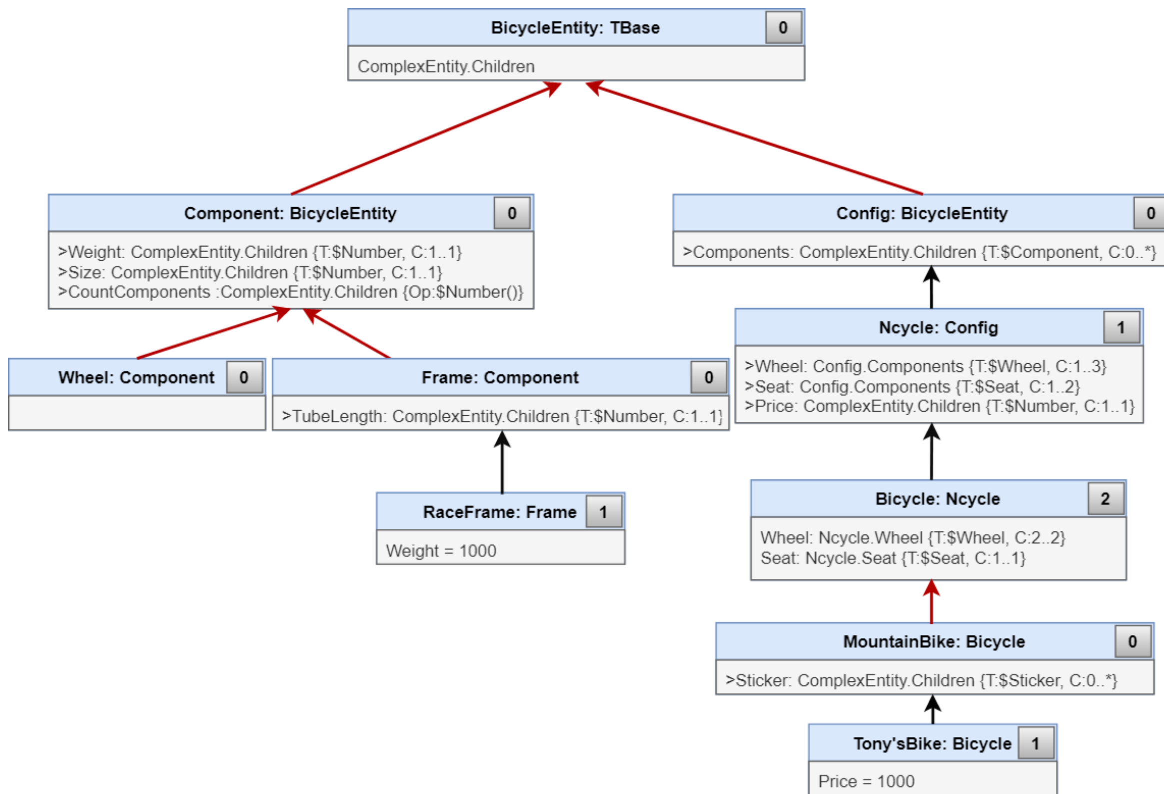


Fig. 3. Solution of the motivating example by using the t-number

slot is introduced (considering the aforementioned rules), the t-number must be higher with one than the value of the meta-entity, since the entity is concretized (instantiated).

The third part of the algorithm is responsible for looking for omitted slots in the particular entity by iterating through the slots of the meta (Ln. 17). If a slot of the meta is not contained by the *instance* (Ln. 18), the value of the t-number must be increased by one, as slot omission is a way to concretize entities. Note that the specific t-number slot is not included in the validation (Ln. 19), similarly to the previous part.

If none of the aforementioned conditions fulfill the rules of the instantiation relation, the fourth part is evaluated. This part simply checks the validity of the classification by comparing the value of t-number to zero.

## VI. Demonstration of t-number

In this section, we explain the working principles of the t-number through the practical example presented earlier in Section II. Note that although the example is taken from the Bicycle Challenge [19], it is simplified and slightly modified for the sake of clarity. Figure 3 depicts the elements of our solution including the t-number.

The slots are shown embedded into the entities. Meta-slot relationships are represented with *Slot:MetaSlot* labels. The different constraints (e.g. type or cardinality) are expressed between curly brackets. The ">" symbol stands for the introduction of a new slot. Red arrows represent the inheritance relation, while the black arrows stand for the instantiation. Note that we must not use the UML convention of arrows for typing (dotted/dashed) and inheritance (empty triangle arrow head), because in DMLA instantiation has a different semantics and inheritance is a mere multi-level notion implemented in DMLA. Numbers in the upper-right corner of the entities represent the t-number values.

The *BicycleEntity* is our starting point. The t-number of *BicycleEntity* is zero, since we copy slot *Children* from *TBase* and no slot is omitted. As discussed before, the slot *ComplexEntity.Children* enables the creation of custom slots, like *Weight* and *Size*. Besides, *Component* has an operation *CountComponents* as well, which can be used to count all entities instantiated directly or indirectly from *Component*. However, since these slots divide *ComplexEntity.Children*, the t-number of *Component* and *Config* remains zero. Note that this also means that they serve as abstract entities.

The t-number of entity *Wheel* is zero, because it clones all of the slots from *Component*, no slots are created or omitted. In contrast, *Frame* introduces *TubeLength*, but *TubeLength* is also originated from *Children*, thus the t-number remains zero. This is not the case with *RaceFrame*, which instantiates *Frame* by filling out the slot *Weight* with a concrete value. Accordingly, the T-number of *RaceFrame* is set to one.

On the right branch of the meta hierarchy, both *Ncycle* (the common meta entity of all cycles, such as bicycles, unicycles, etc.) and *Bicycle* increases the value of t-number, because they contain slots which are not originated from

*Children*. For example, the slot *Wheel* is originated from *Config.Components*. By following the chain, we may notice that *MountainBike* resets the t-number to zero, since it only introduces a slot (*Sticker*), which is originated from *Children*. This showcases an important feature of t-number: one may easily distinguish abstract entities along instantiation chains. At the end of the right branch, *Tony's bike* instantiates *Bicycle*, thus, t-number is increased again.

Although the illustrated example is simple, we believe that it demonstrates that t-number can properly distinguish entities which are needed for the purpose of inheritance from those which are used for instantiation. Thus, we have presented one potential way of incorporating inheritance with abstract entities into DMLA without having to renounce any of DMLA's formal aspect such as self-validation or self-description.

## VII. Conclusions and future work

In this paper, we presented the concept of the *t-number*, which has been introduced in our fluid style multi-layer modeling framework. Before elaborating the details of its idea, we discussed briefly how one could properly interpret the level abstraction in multi-level modeling. The integration of inheritance and instantiation in DMLA requires that the extended Bootstrap is still self-validating after the introduction of inheritance. Therefore, we also described DMLA in enough details regarding the technical possibilities. Finally, we detailed how we modeled the t-number and demonstrated its uses on a model excerpt borrowed from the Bicycle Challenge.

We did that because we do believe that our way of thinking is not only applicable to DMLA, but it can also be taken over by more generic level-blind approaches. However, we only claim that formal integration of inheritance and instantiation in a level-blind manner can be implemented via the t-number only in our fluid style multi-layer formalism where instantiation has a particular semantics. Nevertheless, since DMLA can now properly mimic one of the most important characteristics of the potency notion, we do believe that DMLA will be able to effectively contribute to the future evolution of multi-level modeling concepts such as order and potency.

Regarding our future work, one of our main goals is to create a specific research bootstrap for multi-level modeling notions where multi-level ideas can be experimented with. We aim to achieve this goal by specifying the formal semantics of the notions in DMLA. Before that, we will extend the standard Bootstrap with state-of-the-art OOP modeling concepts which UML modelers are familiar with and apply on a daily basis. We do that because we think that if broader support for multi-level and OOP modeling features are available for professional modelers in practical tool-chains, they will easier accept the no-accidental-complexity principle, which may revolutionize the modeling industry for the very best interest of the multi-level research community.

## Acknowledgment

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. FIEK_16-1-2016-0007 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 (FIEK_16) funding scheme.

## References

[1] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," *Software & Systems Modeling*, vol. 7, no. 3, pp. 345–359, Jul 2008. [Online]. Available: https://doi.org/10.1007/s10270-007-0061-0

[2] C. Atkinson and T. Kuhne, "Meta-level independent modelling," in *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, 2000.

[3] DMLA, "https://www.aut.bme.hu/pages/research/vmts/dmla."

[4] G. Mezei, Z. Theisz, D. Urbán, and S. Bácsi, "The bicycle challenge in dmla, where validation means correct modeling," in *Proceedings of MODELS 2018 Workshops*, 2018, pp. 643–652. [Online]. Available: http://ceur-ws.org/Vol-2245/multi_paper_2.pdf

[5] MULTI, "https://www.wi-inf.uni-duisburg-essen.de/multi2019/wp-content/uploads/2019/05/multi_process_modeling_challenge.pdf."

[6] C. Atkinson and T. Kühne, "The essence of multilevel metamodeling," in *łUML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, M. Gogolla and C. Kobryn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 19–33.

[7] T. Kühne and D. Schreiber, "Can programming be liberated from the two-level style: Multi-level programming with deepjava," *SIGPLAN Not.*, vol. 42, no. 10, pp. 229–244, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1297105.1297044

[8] C. Atkinson and R. Gerbig, "Flexible deep modeling with melanee," in *Modellierung 2016 - Workshopband : Tagung vom 02. März - 04. März 2016 Karlsruhe, MOD 2016*, vol. 255. Bonn: Köllen, 2016, pp. 117–121. [Online]. Available: http://ub-madoc.bib.uni-mannheim.de/40981/

[9] T. Clark and J. Willans, "Software language engineering with xmf and xmodeler," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, vol. 2, 01 2012, pp. 311–340.

[10] T. Kühne, "A story of levels," in *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, 2018*, 2018, pp. 673–682.

[11] C. Atkinson, R. Gerbig, and T. Kühne, "Comparing multi-level modeling approaches," in *CEUR Workshop Proceedings*, vol. 1286, 09 2014.

[12] T. Kühne, "Exploring potency," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '18. New York, NY, USA: ACM, 2018, pp. 2–12. [Online]. Available: http://doi.acm.org/10.1145/3239372.3239411

[13] F. Macías, A. Rutle, V. Stolz, R. Rodríguez-Echeverría, and U. Wolter, "An approach to flexible multilevel modelling," *Enterprise Modelling and Information Systems Architectures*, vol. 13, pp. 10:1–10:35, 2018. [Online]. Available: https://doi.org/10.18417/emisa.13.10

[14] D. Urbán, Z. Theisz, and G. Mezei, "Self-describing operations for multi-level meta-modeling," in *Proc. of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, INSTICC. SciTePress, 2018, pp. 519–527.

[15] D. Urbán, G. Mezei, and Z. Theisz, "Formalism for static aspects of dynamic metamodeling," *Periodica Polytechnica Electrical Engineering and Computer Science*, vol. 61, no. 1, pp. 34–47, 2017. [Online]. Available: https://pp.bme.hu/eecs/article/view/9547

[16] R. Boerger, Egon; Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[17] G. Mezei, Z. Theisz, D. Urbán, S. Bácsi, F. A. Somogyi, and D. Palatinszky, "A bootstrap for self-describing, self-validating multi-layer metamodeling," in *Proceedings of the Automation and Applied Computer Science Workshop 2019 : AACS'19*, D. Dunaev and I. Vajk, Eds., 06 2019, pp. 28–38.

[18] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*, 2nd ed. Packt Publishing, 2016.

[19] MULTI, "https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/," 2018.