2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)

Multi-level modeling with DMLA - A contribution to the MULTI Process challenge

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

***Abstract*—Multi-level modeling has been growing in popularity in recent years, with more and more approaches evolving and maturing over time. In order to facilitate the growth of multi-level modeling, and to identify common problems in this research field, the multi-level community issued the so-called Process challenge. The challenge contains a case description (problem domain) to which the participants should propose their solutions. This paper describes our solution for the challenge using our multi-level modeling framework, the Dynamic Multi-Layer Algebra. We present our solution starting from the more abstract domain-agnostic concepts, then presenting the more concrete domain-specific concepts, while explicitly addressing the requirements in the case description. While we have managed to solve almost every requirement of the challenge, some of them were difficult for us to solve in an elegant way. We also discuss these problems in further detail.**

***Index Terms*—multi-level, meta-modeling, process challenge**

I. Introduction

Multi-level modeling is a software development paradigm aimed at tackling complex problem domains while avoiding accidental complexity that comes from describing such domains with a methodology not suited for describing them [1], [2]. Multi-level modeling approaches have been improving rapidly over the last few years, there are many such approaches available now [3]–[6]. There are numerous ways to categorize these approaches, like making a distinction between level-adjuvant and level-blind approaches [7]. In 2018, the multi-level community issued the so-called Bicycle Challenge, which was an upgrade of the one that had been purposefully created for the MULTI 2017 workshop in order to suscitate community thinking [8]. In fact, the challenges aimed to advance multi-level modeling by proposing a relevant problem domain where many interesting problems emerged that multi-level modeling approaches must deal with. The challenges were also aimed at demonstrating the advantages of multi-level modeling over other methodologies like object-oriented programming in the case of complex problem domains. In 2019, the multi-level community issued a new challenge, the so-called Process Challenge [9]. This paper describes our contribution to this challenge, using our level-blind multi-level modeling framework called DMLA [10].

The paper is organized as requested by the Process Challenge description, which is as follows. In the next subsection (Section I-A), we elaborate the main features and specialties of DMLA. Section II contains our interpretation of the case discussion, also including some of those points we considered may be too ambiguous. Section III presents our actual solution to the challenge, it discusses every important concept in detail and also enumerates the solved challenge requirements. Section IV goes further into the particularities of our solution, touching upon some difficult-to-solve requirements, and addressing the mandatory and recommended discussion aspects prescribed in the challenge description. Finally, Section V concludes the paper, while outlining some of our future work.

A. DMLA

Dynamic Multi-Layer Algebra (DMLA) [11], [12] is our multi-layer modeling framework that consists of two parts: (i) the Core containing the formal definition of modeling structures and its management functions; (ii) the Bootstrap having a set of essential reusable entities of any modeled domains in DMLA.

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) [13]. The states of the state machine represent the snapshots of dynamically evolving models, while transitions (e.g. deleting a node) stand for modifications between those states. The Bootstrap [14] is an initial set of modeling constructs and built-in model elements, which are needed to adapt the abstract modeling structure of DMLA to practical applications. The Bootstrap itself can be modified, so even the semantics of valid instantiation can be easily re-defined if needed. From the domain engineer's point of view, this freedom may seem unnecessary and uncomfortable at first glance, but the inner details of the Bootstrap are hidden, while the external interface, i.e. the modeling paradigm offered to domain engineers is clear and simple to use. However, the separation of the Core and the Bootstrap concepts and the self-modeled instantiation make it also possible to introduce (or emulate) notions from different modeling paradigms, i.e., the potency notion.

Instantiation in DMLA means gradual constraining (refinement) and thus 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 in 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 will refer to the constructs defined by the default Bootstrap whenever discussing the details of DMLA unless mentioned otherwise explicitly.

DMLA is a level-blind approach: even though each entity has a meta-entity, levels are not explicitly modeled. 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 nature of the Bootstrap, it is possible to create a level-adjuvant Bootstrap, but currently we decided to do 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. Similarly 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 the entity, all slots are to be validated against their meta-slots. This is how DMLA checks the constraints applied on the meta-slots. These include, but are not limited to type and cardinality constraints. It is also possible to create domain-specific constraints or attach complex ones 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 it 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, supported by fluid metamodeling at the entity level also 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.

At this point it is worth mentioning that the fact that the entity structure and its constraints are determined solely by the meta-entity makes it hard to support inheritance, since in the case of inheritance, some of the structural constraints are defined in the base entity, not by the meta-entity. We are working on this feature, but currently, DMLA does not support inheritance although we can emulate it using instantiation as described later in the paper.

During some practical modeling projects with DMLA, we had realized that large number of entities and their complex relations are more than challenging to be defined merely by directly producing and manipulating 4-tuples. For example, the Bootstrap and our model solution to the Bicycle Challenge [15] consist of 13091 entities altogether. In order to simplify creating models, we introduced a scripting language, DMLAScript, with an Xtext-based [16] workbench into the DMLA framework. DMLAScript is a domain-independent external DSL language to help automate 4-tuple production. Thus, the modeler only has to deal with creating scripts for models in DMLAScript.

II. Case analysis

Before starting to solve the Process Challenge in DMLA, we have analyzed the requirements in order to understand them better and thus to provide a solution that fits the expectations. During this analysis, we tried to understand the details of the domain and the relations between the elements. In this section, we present our conclusions by beginning with two bigger statements and then continuing with the smaller points.

In our interpretation, the challenge describes not one, but two semi-separated domains: the task definition and the process flow. The aim of the first one is to define and refine tasks, while the second one uses these tasks in creating processes. This is similar to the design of a graph rewriting system, where we can first define the rewriting rules and then we build a control flow based on these rules [17]. However, this also means that the tasks (and task types) have two different views: in the task domain, they define a task with all of the structural parts e.g. artifact and actor rules; while in the process flow domain, they have only flow-related attributes, such as the next element in the flow. Artifact rules are not needed in constructing processes, while flow related attributes are useless when talking about tasks as a separate entity. Moreover, we interpreted that task types can be used in more than one process types with different connections in the flow. In order to model these requirements, we concluded that we need a wrapper as the representation of a task (type) in a process (type) definition. The wrapper refers to the underlying task (type) and describes flow specific behavior in the process. It is important to mention that we came up with the solution of having a wrapper around task (types) not because of the limitations of our tool, but because we believe that this solution better fits the challenge description, also, this led to a slightly more complex solution. Thus, the solution would work without wrappers, in fact, it would even be simpler.

We also had several discussions about the relation between task types and tasks in general. The challenge refers to this relation as instantiation and at first, we have accepted that. However, when we began digging deeper into the realm of the Process Challenge, certain situations started to look rather ambiguous. Let us see a concrete example: take a specific task type Requirement Analysis (*RA*) and a specific task (*ra01*) realizing *RA*. If we take a look solely at the structure of *ra01*, namely what slots it has, we can notice that it is determined by the *Task* concept, not by *RA*. *RA* does not takes part in determining the structure of *ra01* at all: it does not add, delete or modify any slots. Instead, *RA* adds custom validation logic related to the slots. From this sense, *RA* acts more as a specification that we have to follow rather than the type definition of *ra01*. More precisely, *ra01* does not depend structurally on *RA*, their structures are independent of each other. Moreover, *RA* is the instance of *TaskType* and thus, it does not have a beginning or end date, contrary to *ra01* that does have these properties. This means that there are two different kinds of relations in the challenge referred to as 'instantiation': one is more about structural compliance (we refer to this as 'structural instantiation'), while the other is more like instance specification (validation) (what we refer to as 'specifying instantiation'). Note that similar relations can be identified between process types and processes.

Finally, in the following, we discuss how we interpreted certain smaller points in the case description which we thought to be ambiguous:

- Since it was not clear whether and how we should differentiate between the logical meaning of different splits and joins (gateways), we have not added any custom validation logic to them. The different elements can be used in modeling the process, but their behavior is not differentiated. It is worth mentioning that the behavior can be extended by defining it via custom operations.
- We interpreted that if a particular set of actors are authorized to perform a task, then we do not have to check if their types are eligible, that is, if there actually are eligible actor types assigned to the task. Essentially, we deemed P6 to be a stronger constraint than P5. However, making an alternative solution (like checking both constraints simultaneously) would not be an issue, as this was merely a choice we thought would make the most sense in this particular domain.
- Regarding the used and produced artifacts of a task, we deemed them to be valid if there were at least one of every artifact type specified by the related task type. This also means that we did not model the cardinality of these used and produced artifacts, as the case description did not mention it explicitly.
- Regarding tested artifacts and their test reports in the software engineering domain, we interpreted that a test report can belong to more than one tested artifact. This would make little difference to our validation logic, as this was also a decision that made the most sense domain-wise.
- Although the challenge description (in Section 2.2) does not mention, according to requirement S4, we need to model relations between artifacts. Even more, we need named relations (e.g., written in).
- We believe that requirements S2 and S13 contradict each other with regards to the performer of the *Test Case Design* task. Since we thought it highly unlikely that a senior analyst can be a developer or tester too, we went with the description in S13, meaning that a *Test Case Design* must be performed by a senior analyst.

III. Model presentation

In this section, we present our solution for the Process Challenge. The presentation is separated by domain-agnostic and domain-specific concepts in the model. This separation is also marked in the uploaded solution in the form of comments [10].

A. Domain-agnostic concepts

1) Processes and Tasks: Processes and tasks form the core of the challenge that every other part builds upon. Based on the case analysis presented in Section II, we separated the two domains. Accordingly, task types have a wrapper when they are used in process types. The wrapper is responsible for acting as a bridge between the two domains and model the flow related properties of the given task type. Therefore, in our solution, task types are reusable concepts between different process types. For example, a task type called *Requirements Analysis* can be used in multiple process types (e.g. in different companies, or even different process types in the same company). The structure of task wrappers is rather simple: they only have a reference to the task and another reference to the task type wrapper. These two references are used in process definitions to validate themselves against their related process type.

To sum up, we essentially have four connected concepts regarding tasks: i) the *TaskType* entity describes the reusable aspects of a task type, ii) the *TaskTypeWrapper* entity wraps a particular task type in a *ProcessType* by adding flow related properties to it, iii) the *Task* entity symbolizes concrete tasks that must conform to a particular task type, while iv) *TaskWrapper* is used to connect a particular task to a particular task type wrapper.

When extending this setup with processes, there are two more concepts introduced: i) the *ProcessType* entity contains a linked-list with the elements (task type wrappers and gateways) of the flow it defines, and ii) the *Process* entity containing task wrappers. Note that in *Processes*, we do not have gateway entities as explained later.

ProcessType definitions consist of task type wrappers and gateways describing the logical flow. Flow elements are modeled by the *ListItem* entity, which realizes a linked list via its *NextElements* slot. *ListItem* is instantiated to create the *TaskTypeWrapper* and *Gateway* entities. There are numerous instances of *Gateways*, modeling different behaviors, such as the *Sequencing* or *AndSplit* entities. These instances refine the cardinality of the next elements of the gateway, for example, sequencing only allows exactly one next element. This solves requirements P1 and P2. However, it is also worth mentioning that gateways are differentiated based on their type, and can be further extended with other concepts like supporting branching and parallel threads. The initial and final task types (P3) are validated via our operation language, as part of the validation of *ProcessType*. Thus, they do not have to be explicitly defined, only certain requirements have to be met. These are as follows: i) there must be exactly one initial task type, where an inital task type is a list item with no previous element, ii) there must be at least one final task type, which is a list element with no next element.

As for the *Processes*, the structure is much simpler: we only allow *TaskWrappers* in the list (P11). The reasoning for the omission of *Gateways* here is that they are present in *ProcessType*, and we deemed it redundant for them to be here as the flow cannot be altered, but can only be filled out with concrete tasks. Obviously, the mapping (conformance) between *Tasks* in a *Process*, and *TaskTypes* in the respective *ProcessType* have to be accounted for. This validation can be found in *Process*. Moreover, *TaskWrappers* refer to a *TaskTypeWrapper*, as when we validate them, we have to check if every task type used in the process type is exclusively mapped by a task in the associated process.

2) Different kinds of instantiation: Thus far, we did not talk about the relationship between *Tasks* and *TaskTypes*, and between *Processes* and *ProcessTypes*. As discussed earlier, the key idea in our solution is that we have two different kinds of instantiations. As DMLA does not support more than one meta-entity for a given entity, we have decided to use the meta-instance relation whenever structural instantiation is needed, and use custom validation operations to model specifying instantiation (described in Section II). From now on, in the context of the DMLA solution, unless explicitly mentioned otherwise, we use the term *instantiation* only for structural instantiation.

Let us demonstrate our solution through an example. Let us assume that *RequirementsAnalysis* is an instance of *TaskType*, while *AcmeReqAn* is an instance of *Task*. However, we also have to support specifying instantiation, since an important requirement is that particular *Task* instances (like *AcmeReqAn*) have to fulfill certain requirements imposed on them by their respective *TaskType* instance (like *RequirementsAnalysis*). Therefore, a reference (slot) has to be assigned to every *Task*, pointing to a concrete instance of *TaskType* that it must conform to. This relationship is not a structural one, instead, we have a certain validation logic to be implemented, based on the particular *TaskType*. This validation logic is of course implemented on the meta (task + task type) level, so it only needs to be written once, instead for every particular task and task type.

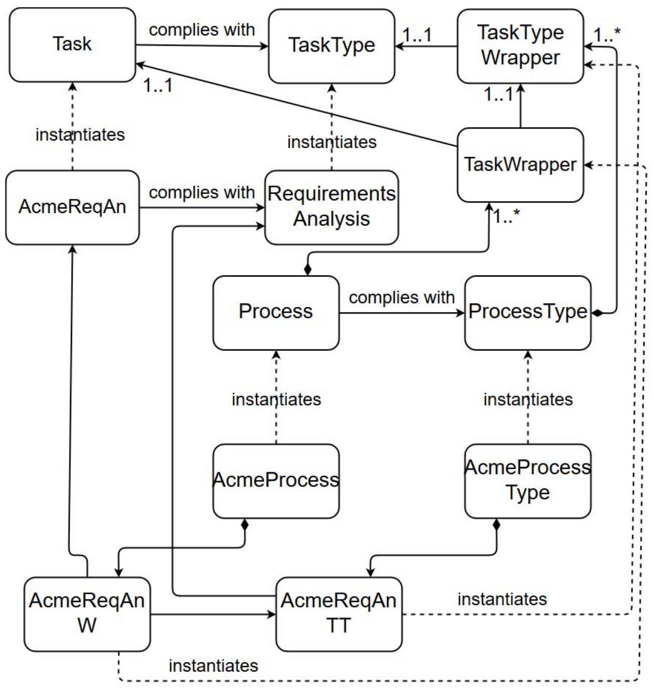

Fig. 1. Relations between tasks, task types, processes, and process types.

Due to a similar reasoning, the relationship between *Process* and *ProcessType* is solved in a similar way since concrete *Processes* are referring to concrete *ProcessTypes* while retaining their structure. Also, multiple *Processes* can refer to the same *ProcessType*, fulfilling requirement P10 which demands that each process type should be able to be enacted multiple times. Figure 1 summarizes the relationship between tasks, task types, processes, and process types. Please note that the figure is only a visualization created for the easier understanding of this paper, and is not a formal visualization of DMLA. The *complies with* relationship refers to the specifying instantiation discussed before.

3) Entities detailed: Next, let us analyze the inner workings of *TaskTypes* and *Tasks* in more detail. In the *TaskType* entity, we introduced the mandatory (1..1 cardinality) *Creator* slot (which is a reference to an *Actor*) to solve requirement P4. This would probably be a cross-level reference in a level-adjuvant solution, however, we have no problems due to level-blind nature of DMLA described in Section I-A. Moreover, we have the *EligibleActorTypes* and *EligibleActors* slots that are required to solve P5 and P6. Namely, the former slot contains the *ActorTypes* that are eligible to perform a task, while the latter is an *Actor* list that are explicitly (and in our interpretation, exclusively) allowed to perform a task. We further elaborate on this later in this section. Similarly, the used and produced artifact types are defined in their respective slots (P7). Task types also have an *ExpectedDuration* (P8), which has the type *DateTime*, a custom-defined entity with custom validation regarding valid date times. They also contain the *IsCritical* flag (boolean typed mandatory slot), which describes if the respective task is critical according to P9. Each *Task* has a begin and an end date (P12), the associated artifacts used and produced, and the list of performing actors (P13).

Actors have a *Name* and the list of *ActorTypes* they have, contained in a slot with a cardinality of 1..*, thus, an actor can have multiple types, fulfilling requirement P15. An *ActorType* can specialize other *ActorTypes* in the *SpecializedActorTypes* slot (P18). This is optional and is taken into account during validation. During the validation of a *Task*, the list of *PerformingActors* is checked against the *EligibleActorTypes*, or the *EligibleActors*. If the type of every actor (or the actors themselves, in the latter case) match (including specialized types), then the task is valid in this regard (P17).

We also introduced a *SeniorActorType*, which instantiates *ActorType*. Since instantiation in DMLA is a refining relationship, this is more akin to specialization well-known from object-oriented methodologies. We can fulfill the P9 requirement (where critical tasks must be performed by a senior actor) by checking if at least one type of every performing actor instantiates *SeniorActorType*. *Artifacts* and *ArtifactTypes* are very similar, and even more simple, since we do not have to deal with specialized types or senior entities. Every *Artifact* can also have multiple types (P16), and the validation in *Task* follows the same logic as in the case of actor types and performing actors. The relationships between actors and actor types along with domain-specific examples are represented in Figure 2. The *specializes* relationship represents the slot containing the specialized actor types, as part of the requirement of the challenge. The unnamed relationships between *Actor* and *ActorType*, and between *AnalystJoe* and *SeniorAnalyst* represent the typing slot. This is not a very elegant solution, and will be discussed later in the paper. Artifacts will be demonstrated later, after we talk about the domain-specific extensions we made regarding them. Again, the figure is only intended to visualize our solution in order to make it easier to understand in this paper.

Finally, we have *ProcessEntity*, whose only purpose is to fulfill the requirements of P19, namely, that artifacts, actors and all types (process, task, artifact, and actor types) are given a set of alternative names. The aforementioned entity contains an *AlternativeNames* slot (which can either be optional or not, we chose the latter), and every representing entity mentioned in the requirement simply instantiates *ProcessEntity*.

B. The software engineering domain

The concepts introduced so far were all general, meaning that they are the basic building blocks in the domains of task and process modeling. Demonstrating the advantage of multi-level modeling, these domain-agnostic concepts can be easily

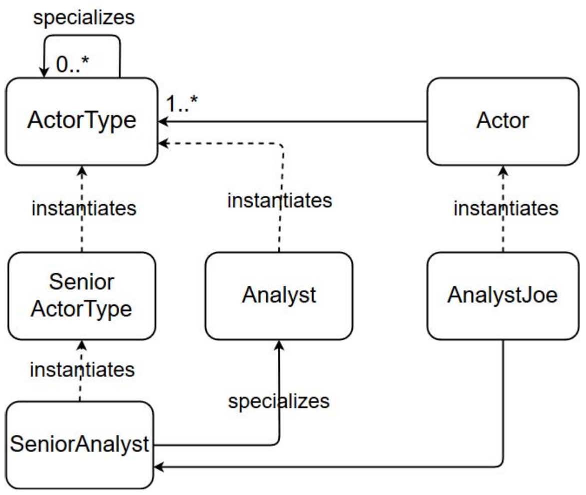

Fig. 2. Relations between actors and their types.

built upon when we are defining a particular domain, like software engineering described in the case description.

First, we introduced the task types described in the challenge (belonging to the fictional Acme Software Development Company). A *RequirementsAnalysis* is an instance of *TaskType*, and has its values filled: i) its *Creator* is *BobBrown*, who also created every task type in this domain (S11), ii) the *ExpectedDuration* is 14 days, iii) the *IsCritical* flag is set to false (meaning it is not a critical task), iv) the *EligibleActorTypes* slot contains only the *Analyst* actor type (meaning that only analysts are allowed to perform it - S1), and v) the *ProducedArtifacts* slot is set to contain *RequirementsSpecification*, which is a concrete *Artifact* (S1). Please note that previously we used *RequirementsAnalysis* in our example for easier understanding, although in the model, we consider it to be an entity specific to the software engineering domain. We also created the other task types similarly, like *TestCaseDesign*, *Coding*, or *Testing*, thus, requirements S2, S3, and S8 are addressed this way. Regarding *Coding* being able to reference one or more programming languages (S3), we introduced the *ProgrammingLanguage ArtifactType*, and used that in the *UsedArtifactTypes* of the *TaskType*. The expected duration of testing can also be set using a simple slot value, fulfilling requirement S12. Additionally, *TestCaseDesign* is a critical task (flag set to true), and its *EligibleActorTypes* slot contains only *SeniorAnalyst*, partially fulfilling requirement S13. Please note the contradiction between S2 and S13 we found and described in Section II.

After we introduced the *TaskTypes* described by the challenge, we proceeded to create *TaskTypeWrapper* instance from every one of them. We also created *Gateway* instances. For example, *AcmeReqAnTT* references *RequirementsAnalysis*, while containing *AcmeAndSplit* as its next element. We built the *AcmeSoftwareEngineeringProcessType* this way. On the *Process* level, we created *AcmeSoftwareEngineeringProcess01*, which refers to the previously described *ProcessType*, and contains a list of *TaskWrappers* referring to their respective *TaskTypeWrappers*. For example, *AcmeReqAnW* (the task wrapper) refers to *AcmeReqAnTT* (the task type wrapper), while *AcmeReqAn* (the task) refers to *RequirementsAnalysis* (the task type). The validation logic of *Process* ensures that every task type in the respective process type has a conforming task in the process, as we have discussed before.

We introduced numerous *ActorTypes* and *Actors* in the software engineering domain, based on the case description. For example, *AnnSmith* is a *Developer*, *BobBrown* is an *Analyst* and *Tester* (S11), or *AnalystJoe* is a *SeniorAnalyst*, which also specialized the *Analyst* type, similarly to *SeniorManager* specializing *Manager*.

The artifact types (e.g. *RequirementsSpecification*, *TestCase*) are all instantiated from *ArtifactType*. We introduced *SoftwareEngineeringArtifact*, instantiated from *Artifact*, that contains the slots *ResponsibleActor* and *VersionNumber* demanded by requirement S10. We did not implement software engineering artifacts as an *ArtifactType*, as they are structurally different from normal artifacts, due to the addition of the two new slots. Thus, if we make *SoftwareEngineeringArtifact* an *ArtifactType*, we cannot use instantiation (due to the types found in a slot) to reinforce this requirement. *CodeArtifact* further instantiates *SoftwareEngineeringArtifact*, containing a *WrittenIn* (for requirement S4, namely, code being able to reference the languages it was written in) a *LinesOfCode* slots. The latter was added to demonstrate that we can add further slots if we solve the problem this way. The validation logic inside *CodeArtifact* checks the correct type of the artifact itself (has to be *Code*), and the type of the *WrittenIn* slot. *COBOLCode* adds another level to this instantiation chain, instantiating *CodeArtifact*. It has a custom validation logic demanding that COBOL code must be written (in our interpretation, at least partially) in COBOL (S6). To demonstrate its simplicity, the validation of the *COBOLCode* entity can be found in Algorithm 1.

Finally, at the bottom of the instantiation chain, there is *AcmeCode01*, which instantiates *COBOLCode*, is written in *COBOL*, and is of artifact type *Code*. *COBOL* itself is an *Artifact* with the type *ProgrammingLanguage*. Figure 3 depicts the relations between artifacts and artifact types, including the extensions introduced in the software engineering domain. This figure is also an informal visualization, similarly to previous figures. The *writtenin* relationship represents the slot used to describe a code artifact being written in a programming language, while the unnamed relationships represent typing, similarly to Figure 2.

```
operation Bool COBOLCodeAlphaValidation(instance)
progLanguages =
        GetValues(instance,CodeArtifact.WrittenIn)
isCOBOLFound = false
for all $pl \in progLanguages$ do
    if pl = $COBOL then
        isCOBOLFound = true
    end if
    if not isCOBOLFound then
        Log("COBOL code must be written in COBOL")
        return false
    end if
    return false
end for
return true
```

Algorithm 1: Validation logic of the COBOLCode entity.

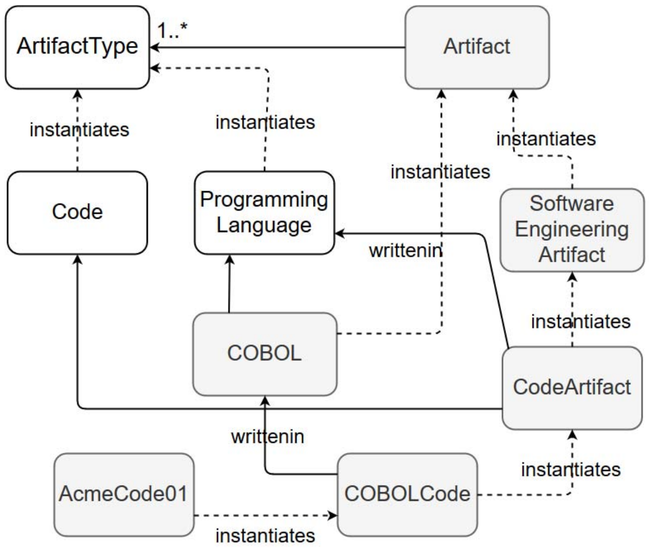

Fig. 3. Relations between artifacts and their types.

Next, we introduced the *CodingTask* entity that instantiates *Task*, extending it with domain-specific validation logic. We believe this is a good example demonstrating the advantage of the multi-level nature of the solution. The validation logic checks if the task uses *COBOL* as a used artifact and then proceeds to check if the produced code derives from *COBOlCode* (S5). It is worth noting that while we could have solved this in a more generic way (checking for every programming language), we chose to avoid that due to the added complexity of the solution, and it not being a requirement. It also checks that if *COBOL* is used, then only *AnnSmith* is allowed to perform the task (S7). *AcmeCoding* is the concrete instance of *CodingTask* in our solution.

The *TestingTask* entity extends the general concept of *Task* with domain-specific logic. It includes an optional list of *TestingAssociations*, where each element creates a link between a *TestedArtifact* and its *AssociatedTestReport*. The type (*TestReport*), the usage of the artifact in the task itself, along with the test report being produced by the task are all checked and validated. This extension was needed in order to address requirement S9. *AcmeTesting* is the concrete instance of *TestingTask* in our solution, and of course, instances of *TestingAssociations* also had to be created to connect the test cases used by the task with the test reports produced by the task.

IV. Discussion

In this section, we further discuss our solution regarding requirements that could not be addressed, along with those that could be addressed, but we thought our solution could be improved. Moreover, although most of the mandatory and recommended discussion aspects mentioned in the challenge were already discussed previously, we explicitly highlight them in this section.

A. Difficult-to-solve requirements

We identified the main issues that hindered us in achieving the most elegant solution for certain requirements. In the following, we discuss these.

1) Lack of inheritance-like relationships: The main issue we identified while working on the challenge was that we have difficulties supporting inheritance-like relationships. The main reason behind it is that in our interpretation, instantiation is a strict, clearly restrictive relation compared to inheritance, which is additive by nature. Instantiation is restrictive, since all features of an entity must have an appropriate meta-feature in the corresponding meta-entity. In contrast, in the case of inheritance, subclasses must conform to the rules of the base class and thus the base class definition can only be extended, not restricted. 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). For example, we solved requirement P18 (types specializing other types) this way, but our solution is a bit artificial. We believe that while object-oriented features are nice to have in a multi-level environment, it is worth to examine the mechanism in more detail in order to see how they can be adapted to this field. Our work around can be introduced into a new Bootstrap and we can also hide the details in an appropriate workbench, but the inner workings of DMLA is not meant be able to explicitly support inheritance. We are working on the formal differentiation between inheritance and instantiation relationships by introducing the so-called t-number, which could further aid us in this regard.

2) Lack of interface-like relationships: When we defined the types of an actor or artifact (P15 and P16), we used a slot containing a list of actor or artifact types. This is not an elegant solution, and is an unusual approach to typing, since it lacks built-in semantics and integrity. While we could specify the validation logic (and thus, the aforementioned semantics and integrity) ourselves, it would be more elegant if we could explicitly define the types an actor or artifact has in a more constraining way. We plan to introduce the concept of contracts, that will be a similar concept to interfaces from the object-oriented world, albeit with some differences (e.g., contracts will be able to contain validating operations as well). By having such a construct, P15 and P16 could be modeled in a more elegant way. Moreover, the alternative names of certain entities (P19) could also be handled better as a contract.

3) Lack of two-way navigation: Right now, DMLA does not support two-way navigation in slots. Due to this feature missing, it would have been difficult for us to solve parts of requirements P9 and S13, namely, that the artifacts produced by critical tasks have to be validated by a validation task. Our solution would have included the introduction of a *ValidationTask* entity, from which every validation task (like *TestDesignReview*) would have been instantiated from. However, we would have needed a navigational link from *Artifacts* to the *Tasks* that they are used or produced by. We could have included this in the form of a slot, but the manual (two-way) maintenance and validation of such a construct would have been complex and very error prone. Moreover, if we had wanted to fulfill these requirements in an elegant way, we would have also needed to check if the validation task that uses the produced artifact of a critical task was after the critical task, sequentially. Overall, the complexity of the solution and the workarounds we would have had to use made us decide not to solve these requirements for the challenge, as – by using our current framework – it would have introduced the same accidental complexity that MLM aims to avoid.

4) Lack of explicit specifying instantiation: As we have discussed in Sections II and III, there were two kinds of instantiation we differentiated between: i) structural instantiation referred to the concept of instantiation in DMLA, where the meta-entity describes the structure of the instance, and ii) specifying instantiation referred to the concept of a validating relationship between two elements. We solved the second one by using slots (references) added to the validated entity (tasks and processes) pointing to the specification (task types and process types). While the solution works, it could be argued that the concept of a specifying instantiation could be explicitly introduced into DMLA. This feature could also be introduced into a new Bootstrap, similarly to some of the previous aspects.

B. Mandatory discussion aspects

- **Basic modeling constructs.** Models in DMLA consist of entities, which may have slots describing the details of entities. Slots can contain primitive values or reference to other entities. The type, the cardinality and all other rules of a slot are modelled by attaching constraints to the slot, which are evaluated by using the slot-metaslot reference. Validation and operational logic are also modeled by their AST built up from modeling entities. These are further elaborated on in Section I-A and in related work [12].
- **Levels.** DMLA is a level-blind approach. Moreover, as we have discussed in Section I-A, DMLA does not require to instantiate all entities of a model at once. We call this style of modeling 'fluid meta-modeling'.
- **Cross-level relationships.** Due to DMLA being a level-blind approach, cross-level relationships are not an issue. Each entity can refer to any other entity along the meta-hierarchy unless the reference contradicts the validation logic.

- **Cross-level constraints.** Similarly to cross-level relationships, cross-level constraints are not an issue for DMLA. These are solved by operations defined on every level, and calling these validations throughout the meta hierarchy. We have seen many examples for this during our presentation in Section III.
- **Integrity mechanisms.** Every time, the model changes, a manual re-validation is applied. This validation checks every constraint of the entire model. Incremental validation is not possible in DMLA as of yet.
- **Abstraction.** Domain-agnostic and domain-specific concepts in the code are separated. We discussed this separation of concepts more deeply in Section III. We can also see the separation marked by comments in the uploaded solution [10].
- **Deep characterization.** We can assign values to slots on any level below their definition, as long as the slot is not omitted in the instantiation chain, and the cardinality of the slot allows it. The value can also be overwritten anytime, but it is validated against the constraints of the slot. As an additional limitation, slots without constraints are handled as final (fully concretized) instances and cannot be instantiated furthermore.
- **Reuse.** The multi-level nature of the solution clearly separates domain-agnostic and domain-specific concepts in the code. Thus, it is possible to extend the solution by using the domain-agnostic concepts, and instantiating domain-specific concepts from them. However, this may not be trivial in the case of dynamic abstractions. Again, the separation was discussed in Section III and can be seen in the uploaded solution.

C. *Recommended discussion aspects*

- **Comparison with related work.** DMLA is level-blind (with added specialties). We highlighted the main differences compared to level-adjuvant approaches (which we believe to be in a majority in the literature right now) previously, mainly regarding cross-level references and constraining. To elaborate, since we have no strict levels in DMLA, it means that any entity can reference any other entity. Cross-level constraints behave in a similar way. Furthermore, constraining is modeled, which means that we do not use an external language (like some OCL variant), instead, we use our own operation language which is modeled. This has the advantage of greater extensibility, at the cost of requiring more effort to use and introduce. Of course, the down-side compared to level-adjuvant approaches is that it is easier to construct invalid models, which the environment has to take into account. DMLA is strictly self-validated, according to its rules defined in the Bootstrap. One of our intentions with the strict self-validation mechanisms is to alleviate the fact that it is easier to construct invalid models due to the level-blind nature of the framework. Deep characterization is also different in DMLA, compared to some other approaches, since it does not use the concept of the potency notion. We further talked about instantiation and deep characterization in Sections I-A and III. Moreover, we talked about object-oriented features like interfaces and inheritance previously in this section. DMLA does not explicitly support them, although it can simulate them. Other frameworks like Melanee [4] or XModeler [6] explicitly support inheritance-like relationships.
- **Underlying formalism.** As explained in Section I-A, DMLA is based on Abstract State Machines. Both the 4-tuples structure of the entities and the basic functions querying/manipulating these structures are defined by the ASM definition. ASM grants the basics of an implementation independent virtual machine running the models. Currently we have a Java-based ASM implementation, but we are working on a new, more general and dynamic GraalVM-based [18] implementation as well.
- **Model verification.** DMLA is a self-validated multi-level modeling framework. The nature of validation was briefly touched upon in Section I-A, and is further detailed in related work [11].

V. Conclusions

In this paper, we presented our solution for the 2019 Process Challenge issued by the multi-level modeling community, using our level-blind multi-level modeling framework, the Dynamic Multi-Layer Algebra (DMLA). Before elaborating the details of our solution, we discussed how we interpreted the challenge. We believe that our thoughts on the interpretation are not specific to DMLA, but are of more generic use. We started the presentation of our solution with the basic building blocks (domain-agnostic concepts), and continued with the concepts of the software engineering domain described by the challenge description (domain-specific concepts). We highlighted an important advantage of multi-level modeling during our presentation, which is the separation of domain-agnostic and domain-specific concepts.

We believe the challenge was very useful to the multi-level modeling community, as it highlighted some interesting problems. While we managed to propose a solution to almost every requirement issued by the challenge, there were some criteria that we could not solve as perfectly as we would have liked. The main weaknesses we found in our approach were the lack of explicit support for specifying instantiation, OO-like features (inheritance and interfaces), and two-way navigation. In the future, we aim to work on further analyzing these problems regarding DMLA, in addition to our earlier plans, like the visualization of models, or making the underlying structure of DMLA more efficient by transforming it into a Graal-based implementation. We are also working on making DMLA more accessible to domain experts by supporting common library functions (like generic operations) in the Bootstrap, or making it easier to write code by introducing a new language that is easier to use and supports more advanced quality-of-life features than the current one [19].

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, "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.

[2] T. Kühne and C. Atkinson, "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

[3] 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

[4] 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/

[5] J. de Lara and E. Guerra, "Deep meta-modelling with metadepth," in *Objects, Models, Components, Patterns*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–20.

[6] 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.

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

[8] MULTI, "https://www.wi-inf.uni-duisburg-essen.de/multi2018/wp-content/uploads/2018/03/multi2018-bicyclechallenge.pdf."

[9] MULTI2019, "https://www.wi-inf.uni-duisburg-essen.de/multi2019/wp-content/uploads/2019/05/multi_process_modeling_challenge.pdf."

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

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

[12] 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

[13] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, 1st ed. Springer-Verlag New York, Inc., 2003.

[14] 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.

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

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

[17] L. Lengyel, T. Levendovszky, G. Mezei, and H. Charaf, "Control flow support in metamodel-based model transformation frameworks," *EUROCON 2005 - The International Conference on "Computer as a Tool"*, vol. 1, pp. 595–598, 2005.

[18] D. Palatinszky, D. Urbán, and D. Dunaev, "Introduction to multi-layer metamodeling with Graal and Truffle," in *Proceedings of the Automation and Applied Computer Science Workshop 2019 : AACS'19*, D. Dunaev and I. Vajk, Eds., 06 2019, pp. 119–128.

[19] G. Mezei, F. A. Somogyi, Z. Theisz, D. Urbán, and S. Bácsi, "Towards mainstream multi-level meta-modeling," in *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, INSTICC. SciTePress, 2019, pp. 483–490.