FMML^x and DLM

A Contribution to the MULTI Collaborative Comparison Challenge

Thomas Kühne
thomas.kuehne@ecs.vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Pierre Maier
pierre.maier@uni-due.de
University of Duisburg-Essen
Essen, Germany

ABSTRACT

This paper is a response to the MULTI 2022 Collaborative Comparison Challenge [23]. We compare FMML^x- and DLM-based solutions. We first present each approach and solution separately, and then discuss trade-offs of both the solutions and the approaches.

CCS CONCEPTS

• **Software and its engineering** → **Software design engineering**; • **Computing methodologies** → **Modeling methodologies**.

KEYWORDS

MLM, modeling challenge, FMMLx, DLM

ACM Reference Format:
Thomas Kühne and Pierre Maier. 2024. FMML^x and DLM: A Contribution to the MULTI Collaborative Comparison Challenge. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3688212

1 INTRODUCTION

Multi-level modeling (MLM) languages share similarities but also differ drastically with respect to certain aspects. The "MULTI Collaborative Comparison Challenge" aims at supporting the appreciation and understanding of such differences by inviting solutions to a modeling challenge. Respective collaborations are meant to result in a deepened understanding of the employed approaches [23].

This paper is a contribution to the challenge, presenting solutions using the FMML^x [10, 11] and DLM [16, 17]. A solution to this challenge with the FMML^x was already presented at MULTI 2023 [21]. The FMML^x solution in this paper is different in some regards in part due to the use of new FMML^x constructs that were not available in 2023 (see Section 2.1).

In this paper, we first provide brief characterizations of FMML^x and DLM in Section 2 and then present respective solutions to the domain challenge in Section 3. Subsequently, we analyze the key commonalities and differences between the approaches, providing a discussion of the respective pros and cons in Section 4 and conclude with Section 5.

2 MODELING APPROACHES

In this section, we characterize the two MLM approaches we used to model our solutions.

2.1 FMML^X and XModeler^{ML}

The FMML^x is an object-oriented multi-level language; its core modeling concepts are class and object. A class can have associations, attributes, operations, and constraints. Collectively, these form the *properties* or alternatively *features* of a class. The FMML^x is based on a meta-circular language architecture; all classes are also objects and consequently have a state and are executable.

The meta-circularity of the FMML^x allows for the specification of an arbitrary number of modeling levels. Objects can instantiate properties from higher-level classes but can also, if the object possesses a class facet, inherit them. We refer to this combination of instantiation and inheritance of properties as *concretization* [13]. We call each direct or indirect concretion of a class A its *descendant* and class A inversely the *ancestor*. Each object in the FMML^x is assigned a level value that reflects its concretization depth potential. Note that the concretization relationship of classes between modeling levels does not exclude the specification of generalization/specialization relationships within the same modeling level. A special case of concretization applies between L1 classes and L0 objects. L0 objects cannot possess class facets and are therefore pure instances (rather than concretions) of L1 classes.

Each property of a class is assigned a target level which specifies the level at which the property is instantiated. This is referred to as the *instantiation level* of a property and supports *deferred instantiation*, since properties need not be instantiated at the immediate level below their specification but can be instantiated further down descendant chains [10, 12]. Associations may have association ends with different target levels, enabling the specification of cross-level associations and links.

The FMML^x is executable; operations and constraints have executable bodies which are specified using the executable object constraint language (XOCL), a variant of the object constraint language (OCL) [6, 7]. XModeler^{ML}, an executable modeling environment that supports the FMML^x, provides a default notation that resembles basic UML notation of classes. The XModeler^{ML} can be downloaded at https://www.wi-inf.uni-due.de/LE4MM/. The FMML^x and XModeler^{ML} are described in more detail in [5–7, 10, 11].

Since the preceding MULTI challenge participation in 2023, the FMML^x has been extended with new language constructs. Among them are a more comprehensive version of contingent-level classes and contingent-level properties (a first version was discussed in [13]), association types, and association dependencies.

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09…$15.00
https://doi.org/10.1145/3652620.3688212

Association types serve as a means to specify properties of associations. An association type constrains the classes between which associations may be created. It can also restrict the multiplicities of associations belonging to one association type. Association types moreover support a custom graphical notation of respective associations and links as illustrated in Figure 1.

The specification of an *association dependency* restricts the set of valid links of the dependent association. If association AB, between classes A and B, is dependent on an association CD, between classes C and D, links of AB may only be created if their respective ancestors are linked via a CD link. Association AB may only be defined as dependent on CD if C and D are either ancestors of or are identical to A or B. Objects with links of AB must reside on the same or on a lower modeling level than objects with links of CD.

2.2 DLM

Domain Level Modeling (DLM) originates from work on the orthogonal classification architecture (OCA) [3] that also gave rise to Melanee/LML [20] and is based on the notion of a "clabject", i.e., a unification of "class" and "object" [2]. DLM supports an unbounded number of classification levels and uses an order-alignment level segregation principle [15]. Its deep instantiation approach is based on characterizing classification potency [1, 14] and supports the separation of domain-induced *classification clusters* [16, p. 551]. An implementation of a formalization of DLM well-formedness rules exists as a ConceptBase implementation [17].

Like FMML^X, DLM supports deep characterization via *feature potency* [1, 2], and allows connections [8] to be deep and cross level-boundaries. DLM connections may also connect elements from different orthogonal ontological classification dimensions [16].

DLM is currently not associated with any particular constraint language. In this paper, DOCL, a variant of OCL designed to support multi-level models is used [19]. DLM does not specify an execution language either, but its well-formedness rules are designed to support execution, e.g., not break client expectations [22].

3 CHALLENGE SOLUTIONS

Table 1 summarizes how the challenge requirements [23] were addressed by each solution. Note that the table only captures the summary points 1)–13) of the challenge and therefore does not cover the fact that the challenge description elsewhere restricts mobile phone factories to produce mobile phone devices only. Our solutions nevertheless implement the latter requirement.

3.1 FMML^X Solution

The FMML^X solution model is shown in Figure 1. It distinguishes between three layers of domain knowledge, which are separated by layout (top, middle, bottom area) in Figure 1.

Generic Domain Knowledge. We identified three core concepts from the challenge requirements: companies, factories, and devices. L2 class Factory, L1 class Company, and L3 class DeviceModel serve to represent these domain concepts at the top level respectively. The level of each class follows from the required concretizations that have be to performed. For example, L0 Factory124 is an instance of L1 MobilePhoneFactory which is a concretion of L2 Factory.

```
Context MobilePhoneFactory, L1
@Constraint properSupport
    self.supportedMobilePhone→forAll(device |
     device.company = self.company)
fail
    "Supported device model is not owned by company!"
end
```

Constraint F1: Proper Support

Two association types are defined: producesAssociationType and supportsAssociationType. Associations of both types must associate a direct descendant of Factory with a direct descendant of DeviceModel. The association types also specify a custom graphical notation of association and links which is why supports associations and links are pink/purple in Figure 1.

According to the challenge description, a company may specify an IMEI prefix. The FMML^X solution assumes this information to be optional which is why the attribute imeiPrefix has a multiplicity of [0..1]. Since mobile phone factories depend on the presence of a prefix value, the operation isImeiConstrained() in Factory checks whether an IMEI prefix exists. This operation is used in MobilePhoneModel to ensure that the IMEI of a mobile phone device begins with the company's IMEI prefix.

Specific Domain Knowledge. Specific domain knowledge refers to types of factories and types of device models. In the FMML^X solution, the L1 class MobilePhoneFactory and the L2 class MobilePhoneModel represent the respective types mentioned in the challenge description. The produces association and the supports association between MobilePhoneFactory and MobilePhoneModel conform to the previously described association types.

The challenge description requires that a Huawei mobile phone factory may only support Huawei mobile phone models (see requirement 9) (a)). The DLM solution introduces the classes Huawei Mobile Phone Factory and Huawei Mobile Phone Model and connects both via a supports association (which is a restricted version of the supports association between Factory and DeviceModel) to fulfill this requirement. In contrast, the FMML^X solution forgoes such dedicated classes for Huawei-owned factories and mobile phone models. It treats a Huawei mobile phone factory as any mobile phone factory that is being owned by the company Huawei. This is one of the reasons why the FMML^X requires fewer modeling elements than the DLM solution. To satisfy requirement 9) (a), constraint F1 ensures the validity of support links. The identifiers supportedMobilePhone and company used in the constraint correspond to the user-defined identifiers of objects referenced per link, which are not visible in the diagram.

To ensure that a factory may only produce devices whose device models it supports (see requirement 3) (c)), the association produces is made dependent on the supports association. In Figure 1, this is indicated by the expression *depends on supports*, which follows the produces name of the association. In the 2023 solution [21], a dedicated constraint had to be used. In this version, it is sufficient to define the dependency; no custom constraint must be written by the modeler.

To satisfy requirement 9) (c), constraint F2 checks whether the IMEI number of a mobile phone device begins with the IMEI prefix of its producing company.

Requirement	**FMML^{x}**	**DLM**
1) A company has (a) a name, (b) owns factories, (c) owns device models	*Company* has (a) a *companyName* attribute and two *owns* associations, one to (b) *Factory* and one to (c) *DeviceModel*	*Company* (C_1) has (a) a *name* attribute and two *owns* associations, one to (b) *Factory* and one to (c) *DeviceModel*
2) Huawei is a (a) company that (b) owns Factory124 and (c) owns mobile phone models S400 and S500	(a) *Huawei* is an instance of *Company*, (b) has a link to *Factory124* and (c) has two links to the mobile phone models *S400* and *S500*	(a) *Huawei* is an instance of *Company* (b) has an *owns* link to *Factory124*, and (c) has two *owns* links to the Huawei phone models *S400 & S500*
3) A factory (a) produces devices, (b) supports a list of device models, (c) can only produce devices that conform to (are of) supported device models	Two association types between *Factory* and *DeviceModel* enable (a) production associations and (b) support associations. (c) An association dependency between *produces* and *supports* restricts the production of devices to supported device models.	(a) *Factory* (F_1) has a *produces* association to *Device*, and (b) has a *supports* association to *Device Model.* (c) Alignment of production of devices with supported models is realized by Constraint D2
4) A device conforms to a device model	Devices are instances of device models	Devices are instances of device models
5) A device model captures what is universal about the devices it describes	L1 device models serve as types for L0 devices	Device models (P_1) are types for devices (P_0)
6) A mobile phone model (a) allows specific RAM size options and (b) is a device model	(a) *MobilePhoneModel* has an attribute *allowedRamSizeInGB* which is populated by mobile phone models on L1. (b) *MobilePhoneModel* is a concretion of *DeviceModel*	(a) *Mobile Phone Model* declares *RAMoptions* which is populated by mobile phone models, (b) *Mobile Phone Model* specializes *Device Model*
7) A mobile phone device (a) conforms to a mobile phone model, (b) has an IMEI and (c) has a RAM size	(a) L0 Mobile phone devices are instances of L1 mobile phone models which are concretions of L2 *MobilePhoneModel. MobilePhoneModel* includes the attributes (b) *imei* and (c) *ramSizeInGB*	(a) Mobile phone devices are instances of mobile phone models which are in turn subtypes of *Mobile Phone Device*, the latter mandates an (b) IMEI number and a (c) RAM size via features
8) A mobile phone factory supports mobile phone models only	Association *supports* between *MobilePhoneFactory* and *MobilePhoneModel* has no generalization that needs restricting	Association specialization is used to restrict the support of *Mobile Phone Factory* instances to *Mobile Phone Model* instances
9) A Huawei mobile phone factory (a) supports Huawei mobile phone models only, (b) keeps track of mobile phone devices it produced, and (c) constrains the IMEI of the mobile phone devices produced by the factory to start with '001'	(a) The constraint *properSupport* restricts support of mobile phone factories to mobile phone models owned by the same company. (b) A mobile phone factory returns the number of produced phones via *numberOfProducedMobilePhone().* (c) *Company* has attribute *imeiPrefix. Huawei* defines it as "001". Constraint *properImei* checks whether mobile phone device IMEIs start with the IMEI prefix of the linked company object.	(a) Association specialization is used to restrict the support of *Huawei Mobile Phone Factory* instances to *Huawei Mobile Phone Model* instances, (b) *Huawei Mobile Phone Factory* has a respective *produces* association, (c) *Huawei* (C_0) defines the prefix "001" which is accessed in *IMEI* in *Huawei Mobile Phone Device*
10) Factory124 (a) is a factory, (b) supports Huawei S400 and S500 mobile phone models, and (c) produced two S400 devices (S400_001, S400_002)	(a) L0 *Factory124* is a descendant of L2 *Factory*, (b) has *supports* links to *S400* and *S500* (c) and *produces* links to *s400_001* and *s400_002*	(a) *Factory124* (F_0) is an indirect instance of *Factory* (F_1), has (b) *supports* links to *S400 & S500* and (c) *produces* links to *S400_001 & S400_002*
11) S400 (a) is a mobile phone model and (b) has either 4GB or 8GB of RAM	(a) *S400* is a concretion of *MobilePhoneModel* and defines a sequence of ram options containing the values 4 and 8.	(a) *S400* is an indirect instance of *Mobile Phone Model* and (b) defines a *RAMoptions* list containing the 4GB and 8GB options
12) S400_001 (a) is a mobile phone device, (b) conforms to the S400 model, (c) has 4GBs of RAM, and (d) has '001468723648726' as its IMEI	(a) L0 *S400_001* is a descendant of L2 *MobilePhoneModel* and (b) an instance of L1 *S400.* It specifies the values for ram in *ramSizeInGB* and for IMEI in *imei.*	*S400_001* is (a) an indirect instance of *Mobile Phone Device*, (b) an instance of *S400*, (c) chooses *option 1* (4GB) of *RAMoptions*, and (d) specifies the IMEI suffix following "001" via its *id* value
13) S400_002 (a) is a mobile phone device, (b) conforms to the S400 model, (c) has 8GBs of RAM, and (d) has '0018768768475638' as its IMEI	(a) L0 *S400_002* is a descendant of L2 *MobilePhoneModel* and (b) an instance of L1 *S400.* It specifies the values for ram in *ramSizeInGB* and for IMEI in *imei*	*S400_002* is (a) an indirect instance of *Mobile Phone Device*, (b) an instance of *S400*, (c) chooses *option 2* (8GB) of *RAMoptions*, and (d) specifies the IMEI suffix following "001" via its *id* value

Table 1: Requirements of the challenge and their realization in FMML^{x} and DLM

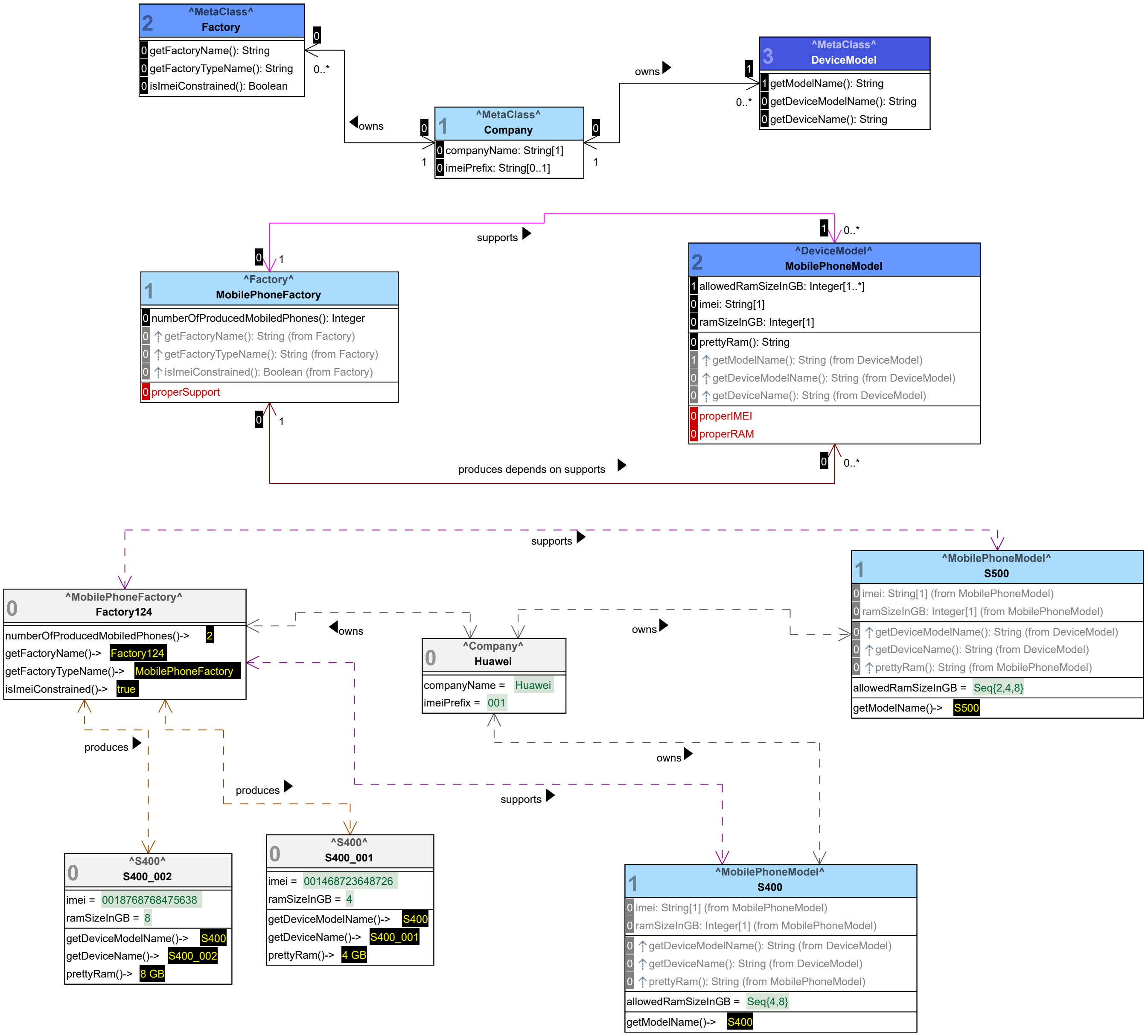

Figure 1: FMML^x solution model

```
Context MobilePhoneModel, L2
@Constraint properIMEI
    self.producingMobilePhoneFactory.isImeiConstrained() ⇒ self.
     imei→truncate(self.producingMobilePhoneFactory.company.
     imeiPrefix→size()) = self.producingMobilePhoneFactory.
     company.imeiPrefix
fail
   "The IMEI does not start with the company's prefix!"
end
```

Constraint F2: Proper IMEI

Valid RAM sizes are ensured via a sequence of RAM value options that must be specified by each mobile phone model (cf. attribute allowedRamSizeInGB with multiplicity [1..*]) and Constraint F3 which checks that a mobile phone's specified RAM size of a mobile phone device is a member of the allowed RAM values of its model.

```
Context MobilePhoneModel, L2
@Constraint properRAM
    self→of().allowedRamSizeInGB→exists(ramSize |
     ramSize = self.ramSizeInGB)
fail
    "RAM size does not correspond to mobile-phone model
     specification!"
end
```

Constraint F3: Proper RAM

Particular Exemplars. One may designate the term “exemplar” to those elements of a domain that are deemed to constitute a particular modeling scenario, rather than belonging to a generic domain characterization. Here, the company Huawei, its factory Factory124, which produces devices S400_001 and S400_002, and supports models S400 and S500 can be considered to be such exemplars. Modeling these is straightforward since their domain types are already specified on higher modeling levels.

In contrast to the MULTI 2023 solution [21], and to the DLM solution, the exemplars are enhanced with a domain-specific graphical notation (see Figure 2). This alternative visualization is intended to improve the readability of the model and illustrates the use of the XModeler^{ML}’s *Concrete Syntax Wizard*. The FMML^{X} meta model specifies that each object has a name; “DeviceModel”, for example, is the name of L3 DeviceModel. To improve the readability of the domain-specific graphical notation, it is sometimes useful to access the name of an object or its type. This allows, e.g., to display the name “MobilePhoneFactory” and “Factory124” in Figure 2. To enable this, the FMML^{X} solutions adds operations to DeviceModel and Factory that return the name of respective descendants. The implementation for these operations is straightforward. Here, we only present one example which retrieves the direct ancestor of self via `of()` and returns the value of the name attribute:

```
Context DeviceModel, L3
    @Operation getDeviceModelName():XCore::String
    self→of().name
end
```

3.2 DLM Solution

The DLM solution model, shown in Figure 3, is structured in three main parts, so-called classification dimensions: “C” (Company), “F” (Factory), and “P” (Product), each featuring its own level hierarchy (e.g., P_0–P_2 within in the “P” dimension). The underlying concept is that a domain scenario often contains so-called *classification clusters* – in the example they are formed by company-, factory-, and product concerns – which give rise to separate classification hierarchies, each with its individual hierarchy depth [16]. Note that unlike the product hierarchy, the company and factory hierarchies only require two levels each.

DLM supports *orthogonal ontological classification* [16], i.e., allows classification dimensions to overlap in the sense that a single element can be classified simultaneously by more than one dimension. Since the domain scenario of the challenge does not feature any overlapping classification, the solution uses multiple classification hierarchies for their organizational effect only. Each of the three classification hierarchies “C”, “F”, and “P” enforces local well-formedness rules on the elements within a hierarchy but allows unrestricted inter-hierarchy connections to other hierarchies [16].

Although the restrictions on IMEIs and available RAM sizes are commonly realized via constraints (cf. constraints F2 & F3), the DLM solution opts to use a “correct-by-construction” approach: P_1-level element *Huawei Mobile Phone Device* redefines the *IMEI* signature declaration with an operation *IMEI* that constructs the IMEI value for Huawei mobile phone devices by prefixing their (IMEI-) ids with a value (“001”) obtained from C_0-level element *Huawei*[1]. Likewise, a mobile phone device specifies its RAM size via selecting one of the valid options available from its corresponding mobile phone model. In Figure 3 the mobile phone model *S400* makes the options “4GB” and “8GB” available. The actual RAM size of a mobile phone device (e.g., *S400_001*) is then determined by evaluating operation *RAM*, defined in *Huawei Mobile Phone Device*, which references the RAM option value of *S400_001* (*option = 1*) and uses it to index the available RAM options defined in *S400.* In summary, instances at the bottom level select one of many valid options which leads to overall properties whose form meets the requirements by construction.

Note that clabject potency and feature potency values default to the level of their enclosing clabject and are only explicitly specified if they are needed to restrict the instantiation depth of a clabject or feature. This is the case for *RAMoptions* at level P_2 which would otherwise be interpreted as a deep field whose value assignments would occur at level P_0.

It is of note that in terms of the domain’s exemplars (cf. Section 3.1) the DLM solution exactly coincides with the FMML^{X} solution regarding the names of modeling elements and their relationships. The differences between the solutions comprise

1. a naming difference between P_2 Mobile Phone Model & L2 MobilePhone. The latter FMML^{X} concept is the direct ancestor of mobile phone models, hence the corresponding DLM name (cf. Section 4.1).
2. the presence of ownership-related concepts (Huawei Mobile Phone Factory & Huawei Mobile Phone Model), which has already been explained in Section 3.1 (also see Section 4.7).
3. the presence of three additional supertypes (Huawei Mobile Phone Device & Mobile Phone Device & Device).
4. higher level placements of Factory & DeviceModel in the FMML^{X} solution.

Regarding (3), these supertypes are technically not required and could be replaced with respective deep feature declarations in corresponding P_2 elements. They have been included, despite causing the need for constraint D1, because

- they support natural domain relationships such as the produces association between Factory and Device (which would have otherwise required a FactoryType concept with an higher-order association to DeviceModel),
- they allow the restriction of association end domains (between Mobile Phone Factory and Mobile Phone Device) through the use of “association inheritance” [25], thus forgoing the need for a respective constraint (cf. Section 4.7).

The aforementioned constraint needs to ensure that *Huawei Mobile Phone Model* instances are subtypes of *Huawei Mobile Phone Device* (cf. constraint D1). If that were not enforced then the second-order instances of *Huawei Mobile Phone Model*, i.e., actual Huawei mobile phone devices, would not have to conform to the stipulations made in the specialization hierarchy that has *Huawei Mobile Phone Device* at its bottom. See Section 4.7 for a further discussion.

```
context Huawei Mobile Phone Model
inv: self.#getSuperTypes()# → collect(#name#)
        → includes("Huawei Mobile Phone Device")
```

Constraint D1: Linking devices with models

---

[1] The 2022 solution stored this prefix at Huawei Mobile Phone Factory, not exploiting the fact that the prefix must be the same for all Huawei-owned factories [18].

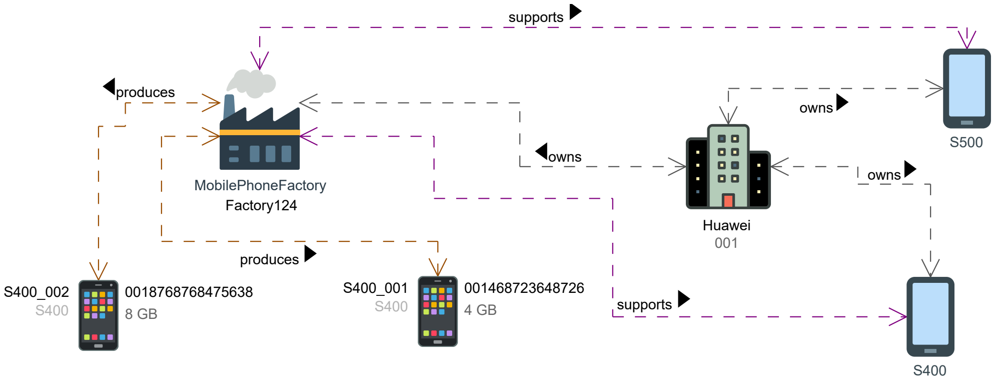

Figure 2: Domain-specific graphical notation in FMML^x solution

Regarding (4), the DLM solution has no need for a type for MobilePhoneFactory, such as L2 Factory (which would have been named FactoryType in the DLM model). Instead, a generalization of MobilePhoneFactory in the form of P_1 Factory was used. Likewise, the DLM solution simply generalizes Mobile Phone Model to Device Model at P_2 rather than introducing a type for it (cf. L3 DeviceModel in the FMML^x solution which would have had to be named DeviceModelType in a DLM model). See Section 4.6 for the motivation of the aforementioned higher-level elements in the FMML^x solution.

Since DLM does not support association dependencies, requirement 3) (c) (see Table 1) had to be addressed via constraint D2.

```
context Factory
inv: self.produces → forAll(device |
  self.supports → includes(device.#getDirectTypes()#→first()))
```

Constraint D2: Factory supported devices

4 DISCUSSION

In this section, we compare the two solutions, highlighting similarities and differences of both the solutions and approaches, and discuss the respective trade-offs.

4.1 Clabjects

Both approaches support elements that combine an instance and a (deep) type facet, i.e., modeling elements that have been dubbed “clabjects”. Clabjects with classifier roles have different meanings in the approaches, though. FMML^x classes have descendants where the difference between immediate (first-order) descendants and more remote (higher-order) descendants is de-emphasized due to the dual nature of the concretization between FMML^x classes. In contrast, DLM distinguishes between direct (first-order) instances, indirect instances (which are first-order as well, but are classified by more specific (sub-) types), and deep (higher-order) instances, i.e., instances of instances, etc. DLM clabjects therefore retain traditional class-instance relationships that can be mapped to “membership”-relationships in the domain, whereas the underpinning of FMML^x classes challenges the utility of this traditional approach.

These different class roles have implications on class naming conventions. DLM clabject names should always describe their first-order instances, e.g., the name of clabject Factory at F_1 describes Factory124 at F_0 as a “factory” (see Figure 3). In FMML^x, the modeler has the freedom to use a class name to reference any of the descendant levels. In Figure 1, L2 class MobilePhoneModel describes its immediate descendants (S400 & S500) but L2 class Factory describes its second-order descendant (Factory124). If the former described its immediate descendant (MobilePhoneFactory), it would have to be named FactoryType. Note that an FMML^x user may impose any convention on themselves and also adopt the DLM naming scheme but this may not always work out when a concretization is meant to simultaneously represent classification and generalization.

4.2 Level Concept

Both FMML^x and DLM require elements to be manually assigned to levels, i.e., each element has a “level” value that associates it with a particular level within the hierarchy. As an exception to this rule, FMML^x features so-called “contingent-level classes” allowing such classes to specify multiple levels [13]. Level membership in FMML^x is visualized via color-coding of the name compartments, along with an explicit level number at the left-hand side of the name compartment. DLM has no codified presentation specification but common presentation conventions include the use of horizontally dashed lines to indicate level boundaries or the use of different color-shaded backgrounds to separate levels. In both cases, the areas between levels are often labeled with a letter that features the level number as an index (cf. C_1 in Figure 3).

In both approaches, generalization relationships, i.e., pure generalization relationships in the case of FMML^x, are intra-level relationships. Since in both approaches inter-level relationships always have an element of classification, the insertion of a new intermediate level is never possible without having significant ramifications on existing elements.

Both approaches allow associations to cross level boundaries. However, only FMML^x supports association ends with different concretization depths in the style of dual potencies [24].

DLM has a homogeneous level hierarchy, while the L1→L0 level boundary in FMML^x, unlike any other level boundary, only allows instance-of relationships, i.e., excludes concretization relationships. FMML^x levels are *order-synchronized*, i.e., an element’s level corresponds to its concretization depth potential, whereas DLM levels are *order-aligned* [15]. Due to these different level-segregation

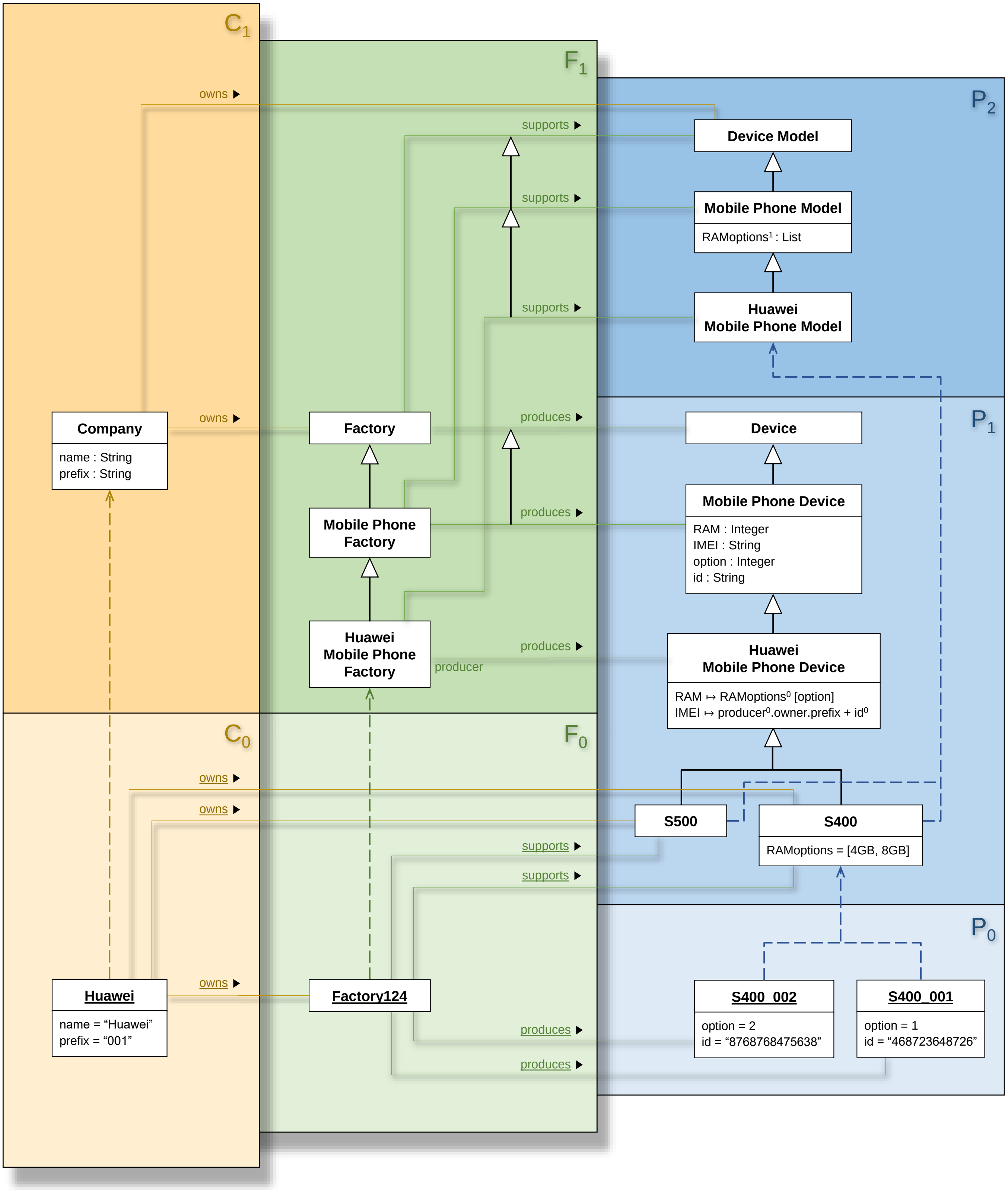

Figure 3: DLM solution model

principles, concepts like Factory may end up at different levels in comparable FMML^X and DLM models. For instance, L2 class Factory in Figure 1 is modeled as a level one element in Figure 3, satisfying the traditional desire to claim that Factory124 is an (indirect) instance of Factory. Conversely, L3 class DeviceModel in Figure 1 could have been modeled as a level two element (cf. GenericDevice in [21]), were it not for an association type (not shown in Figure 1).

4.3 Language-supported Sanity Checks

Models can be complex, in particular larger models of inherently complex domains that especially benefit from the use of MLM technology. Particularly, but not exclusively, models that have multiple authors can be prone to contain conceptual fallacies or exhibit inconsistencies of various kinds [4, 9].

Many languages specify integrity constraints which are enforced by tools that help to ensure basic levels of soundness, such as that instances actually conform to their types or definitions are cycle-free. Due to the aforementioned complexity of models or sometimes simply due to the inexperience of modelers to work with multi-level hierarchies, it is desirable to provide modelers with feedback on their models that goes beyond such traditional checks. For instance, it is possible to identify *anti-patterns* for conceptual models that are not intended to uncover questionable but potentially still correct model fragments, but rather detect model fragments that violate fundamental well-formedness expectations [4].

To that end, DLM leverages its strict distinction between classification and generalization in order to alert users when their models make mutually incompatible claims. For instance, the historical Wikidata problem of Tim Berners-Lee ending up being categorized as a Profession [4] – based on the simultaneous claims that Tim Berners-Lee is an instance of Scientist, Scientist is an instance of Profession, and Scientist “isA” (here meaning “is a subtype of”) Profession – cannot occur within a DLM model since the respective network of relationships does not satisfy DLM well-formedness rules [16, 17]. In the above scenario – now removed from Wikidata which nevertheless still contains numerous analogous or otherwise inconsistent claims [9] – Tim Berners-Lee is claimed to be a second-order instance of Profession (via Scientist as an instance of Profession) and an indirect first-order instance of Profession (via Scientist as a specialization of Profession), which is a logical contradiction. This is just a simple example, the DLM well-formedness rules cover a wide range of potential problems that exceeds the range covered by the three anti-patterns defined in [4].

The sanity checking of some aspects of conceptual model soundness are only possible because DLM differentiates between transitive specialization relationships and intransitive classification relationships, and requires models to have a sound set-theoretic interpretation. In contrast, FMML^X does not support these kinds of sanity checks since the inter-level relationship, concretization, lacks the specificity to support the detection of such modeling flaws.

Since concretization is a mixture of instantiation and specialization, the “descendant-of” relationship can be considered to be transitive. Sometimes, this aligns with intuition, e.g., in case of Factory124 and Factory where it is possible to claim that the former is an indirect “instance” of the latter. However, sometimes the same kind of reasoning does not yield the desired result, e.g., regarding the chain of relationships between S400_1 and DeviceModel, which equally allows one to consider the former to be an indirect “instance” of the latter. Note that choice of class names is important; the FMML^X solution model adopts terminology from the challenge description and is not meant to suggest counter-intuitive inferences such as the one above. Consequently, one one should refrain from reading the transitive descendant hierarchy of an FMML^X model with a traditional understanding of intransitive classification and indirect instance-of relationships as presented here.

4.4 Separation of Modeling Concerns

As mentioned before, multi-level models can be complex and thus challenging to manage. The respective complexity can sometimes be partially harnessed by structuring mechanisms, or even just strategies without formal support that support navigability and readability of multi-level models. For example, the requirements of the challenge (cf. Table 1) may be interpreted as implying the interrelated subdomains of companies, factories, and products.

Such a division of the domain into subdomains can be exploited in DLM to structure the solution accordingly with respective classification dimensions (cf. Figure 3). DLM therefore effectively supports a separation of modeling concerns [16]. While the primary use for separate classification dimensions is to cleanly handle overlapping classifications, the same approach can also be used for structuring domains that do not require any form of multiple classification, as is the case with the challenge domain.

The FMML^X supports separation of concerns by two means. First, XModeler^{ML} allows for the specification of *views*. Views are user-defined dissections of diagram content. Such dissections do not imply any semantics or well-formedness restrictions. Views are separated via layout, similar to the three layers of domain knowledge used for the description of the FMML^X solution. Second, XModeler^{ML} allows for the specification of multiple diagrams for one model, with “diagram” being understood as a visual representation of a model. Modeling elements can be specified in one diagram and used in another.

4.5 Deep Characterization

Both approaches allow classes to define features of higher-order instances. They both attach non-negative integers to features to control at which level the feature will be instantiated.

The difference between the respective mechanisms – *deferred instantiation* (FMML^X) versus *deep instantiation* (DLM) respectively – is essentially the difference between an absolute target-level specification (FMML^X) and a relative target-level specification (DLM). FMML^X features, whose target level is lower than the immediate level below, are thought of as being inherited by concretions (as the absolute target level remains unchanged upon concretization) while features with a target level matching the level below are instantiated at that level below. Hence the level heterogeneity of FMML^X (see Section 4.2) is mirrored by a heterogeneous treatment of features. In contrast, DLM features are always instantiated by an instance, in analogy to how a UML attribute is instantiated into a UML slot. DLM does not distinguish between attributes and slots; an attribute corresponds to a potency-one feature and a slot corresponds to a potency-zero feature, and feature potencies decrease exactly by one upon each instantiation.

DLM feature potency values default to the level of the enclosing clabject, i.e., in the absence of any explicit user-assigned feature potency values, the respective meaning is equivalent to a target level specification of zero in FMML^x. This blurs the difference between the two level-targeting styles as in neither case manual adaptation would be required if the element were assigned a different level but the features were still intended to instantiate at level zero. Partly for this reason, it is difficult to assess which style – absolute vs relative specifications – is more robust against change in practice. Whether deferred instantiation and deep instantiation differ with respect to intuitiveness and model understanding is difficult to assess and would have to be established through empirical studies.

4.6 Relationships between Associations

Relationships between associations in the solutions are used for the following purposes:

(1) restricting the domain of association ends.
(2) making links contingent on the presence of other links.
(3) supporting a custom concrete syntax.

Regarding (1), the challenge description restricts mobile phone factories to produce mobile phone devices only. The DLM solution explicitly models the fact that factories in general may produce any device in general and therefore needs to restrict the kinds of devices a mobile phone factory may produce to mobile phone devices only. It accomplishes that by using association inheritance as known from the UML [25] (see how association produces between Mobile Phone Factory and Mobile Phone Device specializes association produces between Factory and Device in Figure 3). The FMML^x solution accomplishes the same by i) not using a produces association between Factory and DeviceModel in the first place, and ii) using an association type to restrict produces links to allow connecting mobile phone factories with mobile phone devices only.

Regarding (2), requirement 3) (c) (cf. Table 1) restricts produces links between a particular factory and the devices it produced to those where corresponding supports links exist between the particular factory (here Factory124) and the device model of the devices (here S400). In other words, the validity of certain links is made contingent on the presence of other links. Specifically, the links do not need to be related, e.g., via a deep association hierarchy. The FMML^x solution uses a dependency relationship between the produces and supports associations between MobilePhoneFactory and MobilePhoneModel (see the "produces depends on supports" association in Figure 1 which textually indicates the presence of the dependency relationship). Association dependencies were recently added to the FMML^x since the kind of dependencies between links that occurs between produces and support links had been observed to be a commonly occurring pattern in FMML^x models. DLM has no such dedicated support yet which is why the DLM solution has to employ constraint D2 to enforce requirement 3) (c).

Regarding (3) above, the FMML^x solution uses association types for the specification of a custom graphical notation of produces and supports associations and links.

Association types could have also been used to specify the multiplicities of produces and supports associations, but those were not stipulated by the challenge requirements. The respective DLM solution would have relied on deep connections [8] for this purpose.

4.7 Need for Constraints

Textual constraints, as expressed in OCL, for instance, are sometimes necessary to realize integrity conditions of models. They should be regarded as a last resort, however, since

(1) visual counterparts are readily identifiable in a diagram.
(2) equivalent language constructs are easier to use and,
(3) are likely to be more robust against model changes.
(4) textual constraints are more error prone to write, and
(5) they are not as amenable to a reader of the model as standard notation is.

In total, the FMML^x solution uses three constraints (constraint F1–F3). The last two, regarding the validity of IMEIs and RAM configurations, should not be counted in a tally against the DLM solution, though, because the FMML^x could have used the same "correct-by-construction" approach of the DLM solution as well.

This leaves constraint F1 which has no equivalent constraint in the DLM solution, since the latter employs association inheritance between the two lower supports associations in Figure 3.

In turn, the DLM constraint to realize requirement 3) (c) (cf. constraint D2), is more concisely and robustly replaced by the FMML^x association dependency (see Section 4.6). Note that constraint D2 uses a hard-coded "Huawei Mobile Phone Device" string literal to perform a test. Tools are unlikely to pick up on such model element name dependencies within constraints when they ideally should alert users in case the respective model element is renamed.

The second, and last, DLM constraint is required because DLM currently lacks a "powertype" relationship between a generalization and its "powertype" (cf. constraint D1). It would have been possible to avoid the need for this constraint by lifting all the stipulations made by P_1-supertypes to the corresponding P_2-elements, making them deep features, but the resulting model would not have been as accessible and it would not have been as easy to see the correspondence to the requirements.

4.8 Executability

The FMML^x is a monotonic extension of XCore, which is part of the executable meta-modeling facility (XMF) and therefore readily supports model execution [5–7]. The XModeler^{ML} includes a just-in-time compiler that supports the instantiation of models and the execution of the respective model instances. The executable object constraint language (XOCL) is used to specify constraints and operations in the XModeler^{ML} [6, 7].

DLM has no associated execution language yet but is designed with execution and constraint evaluation in mind. Operations are features, for instance, and specialization relationships in DLM should obey the Liskov substitution principle [22].

4.9 Modeling Notation

Both approaches have a graphical notation that resembles the concrete syntax of the UML.

Unlike DLM constraints, FMML^x constraints are explicitly featured in diagrams in the form of class features, e.g., see feature properSupport in MobilePhoneFactory in Figure 1.

Additionally, the XModeler^{ML} includes a *Concrete Syntax Wizard* that allows users to specify a domain-specific graphical notation. The Concrete Syntax Wizard supports accessing values of objects so that objects can be supplied with a value-dependent notation (cf. Figure 2). Within the XModeler$^{\mathrm{ML}}$, it is then possible to dynamically switch between the domain-specific graphical notation and the standard FMML$^{\mathrm{x}}$ notation. FMML$^{\mathrm{x}}$ association types have no visual presentation and therefore do not appear in FMML$^{\mathrm{x}}$ diagrams.

The DLM solution uses explicit visual *instance-of* relationships to connect ontologically-typed instances to their types. Alternatively, it would have been possible to use a *"typed element name"* approach, e.g., to use "Huawei : Company" in the name compartment of Huawei at C_0. For this solution, using visual relationships was deemed to yield better trade offs. The colored backgrounds in Figure 3 are in part supported by the ConceptBase implementation of DLM [17, Fig. 5]. This implementation does not support the vertically stacked shaded backgrounds within classification concerns of Figure 3; these are currently manually created, but could technically be enforced to coincide with the level values of the elements populating them.

5 CONCLUSION

We presented two solutions to the MULTI *Collaborative Comparison Challenge* modeled using the MLM approaches FMML$^{\mathrm{x}}$ and DLM. The comparison is particularly instructive because the approaches use distinctively different level concepts. As expected, the bottom-level elements in the solutions are effectively identical with the exception of minor realization differences. The solutions furthermore share similarities regarding higher levels, but the use of association types pushed some FMML$^{\mathrm{x}}$ elements up a level, and the different underlying level segregation principles led to some naming differences as well. Some further discrepancies between the solutions were caused by different stylistic choices – e.g., how to ensure value integrity and whether or not to explicitly represent core domain generalizations – but some were the result of different underlying modeling philosophies. DLM aims to support the construction of ontologically correct models, i.e., encourages modelers to establish a direct mapping between natural domain concepts and corresponding model elements. It requires a sound set-theoretic interpretation of the models to exist so that it can provide modelers with feedback in case they create contradictory model fragments. FMML$^{\mathrm{x}}$, on the other hand, follows a pragmatic/constructivistic approach. Models created with the FMML$^{\mathrm{x}}$ are not meant to represent truthful representations of a given domain. Rather, FMML$^{\mathrm{x}}$ aims at supporting a purpose-driven linguistic reconstruction of a domain.

Interestingly, the layout choices for the FMML$^{\mathrm{x}}$ diagram emphasize layers of domain specificity, while the DLM diagram emphasizes the cohesion of concepts at the same logical classification level. It might be useful to support both alternatives via respective views or complementing diagrams.

In terms of lessons learned, the challenge has reinforced the utility a native powertype relationship would have in DLM and that incorporating the equivalent of an association dependency construct would be very useful. The challenge furthermore emphasized the need to reflect upon interpretations of transitivity within FMML$^{\mathrm{x}}$ models and mechanisms to prevent counter-intuitive model interpretations.

REFERENCES

[1] Colin Atkinson and Thomas Kühne. 2001. The Essence of Multilevel Metamodeling. In *UML 2001 — The Unified Modeling Language.*, Martin Gogolla and Cris Kobryn (Eds.). Springer, Heidelberg, Germany, 19–33.

[2] Colin Atkinson and Thomas Kühne. 2002. Rearchitecting the UML Infrastructure. *ACM Trans. Model. Comput. Simul.* 12, 4 (Oct. 2002), 290–321.

[3] Colin Atkinson and Thomas Kühne. 2003. Model-Driven Development: A Metamodeling Foundation. *IEEE Software* 20, 5 (Sept. 2003), 36–41.

[4] Freddy Brasileiro, João Paulo A. Almeida, Victorio A. Carvalho, and Giancarlo Guizzardi. 2016. Applying a Multi-Level Modeling Theory to Assess Taxonomic Hierarchies in Wikidata. In *Proceedings of WWW'16.* ACM, NY, USA, 975–-980.

[5] Tony Clark. 2024. Executable Multi-Level Modelling: Establishing Foundations, Methods and Tools. In *Informing Possible Future Worlds: Essays in Honour of Ulrich Frank*, Stefan Strecker and Jürgen Jung (Eds.). Logos, Berlin, 157–172.

[6] Tony Clark, Paul Sammut, and James Willans. 2008. *Applied Metamodelling; A Foundation for Language Driven Development* (2 ed.). Ceteva, Sheffield. https://eprints.mdx.ac.uk/id/eprint/6060

[7] Tony Clark, Paul Sammut, and James Willans. 2008. *Superlanguages: Developing Languages and Applications with XMF.* Ceteva, Sheffield.

[8] Thomas Kühne Colin Atkinson, Ralph Gerbig. 2015. A unifying approach to connections for multi-level modeling. In *Proceedings of MODELS'15.* IEEE Computer Society, NY, USA, 216–225.

[9] A. Dadalto, J. P. Almeida, C. Fonseca, and G. Guizzardi. 2021. Type or Individual? Evidence of Large-Scale Conceptual Disarray in Wikidata. In *ER 2021.* Springer-Verlag, Heidelberg, 367–377.

[10] Ulrich Frank. 2014. Multilevel Modeling: Toward a New Paradigm of Conceptual Modeling and Information Systems Design. *Business and Information Systems Engineering* 6, 6 (2014), 319–337.

[11] Ulrich Frank. 2018. *The Flexible Multi-Level Modelling and Execution Language (FMMLx), Version 2.0: Analysis of Requirements and Technical Terminology.* Ph. D. Dissertation. Universität Duisburg-Essen, Essen.

[12] Ulrich Frank, Pierre Maier, and Daniel Töpel. 2023. Modeling Facets of a Warehouse with the FMMLx: A Contribution to the MULTI Warehouse Challenge. In *MODELS 2023 Companion (MODELS-C).* IEEE, NJ, USA, 659–668.

[13] Ulrich Frank and Daniel Töpel. 2020. Contingent Level Classes: Motivation, Conceptualization, Modeling Guidelines, and Implications for Model Management. In *MODELS '20: Companion Proceedings*, Esther Guerra and Ludovico Iovino (Eds.). ACM, NY, USA, 1–10.

[14] Thomas Kühne. 2018. Exploring potency. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems.* ACM, NY, USA, 2–12. https://doi.org/10.1145/3239372.3239411

[15] Thomas Kühne. 2018. A story of levels. In *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, 2018.* CEUR, http://ceur-ws.org/Vol-2245/, 673–682.

[16] Thomas Kühne. 2022. Multi-dimensional multi-level modeling. *Software and Systems Modeling* 21, 2 (2022), 543–559.

[17] Thomas Kühne and Manfred A. Jeusfeld. 2023. Sanity-Checking Multiple Levels of Classification. In *Conceptual Modeling.* Springer, Cham, 162–180.

[18] Thomas Kühne and Arne Lange. 2022. Melanee and DLM: a contribution to the MULTI collaborative comparison challenge. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Montreal, Quebec, Canada) *(MODELS '22).* Association for Computing Machinery, New York, NY, USA, 434–443. https://doi.org/10.1145/3550356.3561571

[19] Arne Lange. 2016. *dACL: the deep constraint and action language for static and dynamic semantic definition in Melanee.* Master's thesis. University of Mannheim. http://ub-madoc.bib.uni-mannheim.de/43490/

[20] Arne Lange and Colin Atkinson. 2022. Multi-level modeling with LML. *International Journal of Conceptual Modeling* 17 (June 2022), 1–36. https://doi.org/10.18417/emisa.17.6

[21] Arne Lange, Ulrich Frank, Colin Atkinson, and Daniel Töpel. 2023. Comparing LML and FMMLx: A Contribution to the MULTI Collaborative Comparison Challenge. In *MULTI 2023.* IEEE, NJ, USA, 669–678.

[22] Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841. https://doi.org/10.1145/197320.197383

[23] Gergely Mezei, Thomas Kühne, Victorio Carvalho, and Bernd Neumayr. 2021. The MULTI Collaborative Comparison Challenge. MULTI 2021 Call for Papers. https://jku-win-dke.github.io/MULTI2022/MULTI2021_Challenge.pdf

[24] Bernd Neumayr, Christoph G. Schuetz, Manfred A. Jeusfeld, and Michael Schrefl. 2016. Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic. *Software & Systems Modeling* 17, 1 (2016), 1–36.

[25] OMG 2017. *Unified Modeling Language Specification, Version 2.5.1.* OMG. OMG document formal/17-12-05.