Comparing LML and FMML^x

A contribution to the MULTI Collaborative Comparison challenge

Arne Lange*, Ulrich Frank†, Colin Atkinson*, Daniel Töpel†,
* *University of Mannheim*, Mannheim, Germany
† *University of Duisburg-Essen*, Essen, Germany

***Abstract*—This paper contributes to the series of comparisons called for by the MULTI Collaboration Challenge by comparing the LML and FMML^x multi-level modeling (MLM) languages as well as their associated constraint languages. The two languages are particularly suitable for comparison because they are among the most mature MLM approaches, with rich language concepts and powerful modeling tools. Besides obvious similarities, they have a number of differences, some of which only become apparent through closer analysis. The paper applies the well-known challenge in full but adds a small number of further requirements to highlight the special features of the two approaches. Based on an analysis of the requirements, the solutions are presented and then analyzed by comparison. The analysis shows that there is a considerable overlap in the modeling strategies adopted by the two approaches, but each has specific features that allow particularly concise and elegant solutions in some cases, and the use of workarounds in others.**

***Index Terms*—Multi-level modeling, Challenge, LML, FMML^x**

I. Introduction

This paper is a contribution to the *MULTI Collaborative Challenge* first published for the 2021 MULTI workshop. Like the *MULTI Challenge*, its goal is to foster a better understanding of current approaches to multi-level modeling (MLM). To this end, it takes an approach that differs in two ways from the MULTI Challenge, which is limited to the presentation of solutions created with a single approach. First, the MULTI Collaborative Challenge requires a comparison of two selected approaches and a thorough analysis of the corresponding solutions. Second, it requires close cooperation between the developers of the selected approaches. This not only avoids misunderstandings but also emphasizes the commitment to strengthening the community.

The two multi-level languages compared in this paper are the Level-agnostic Modeling Language (LML) and the Flexible Multi-Level Modeling and Execution Language (FMML^x)[1]. The two approaches that have developed around these languages are well-suited for comparison because, on the one hand, they are similar and incorporate many of the same language design choices, and on the other hand, they have numerous features that differ with respect to concepts and terminology. Some differences are obvious and easy to understand while others are more subtle. Both approaches are supplemented by powerful constraint languages and mature modeling tools. While both tools support the design and maintenance of multi-level models, they are based on different design philosophies.

The paper is structured according to the required outline in the Challenge. First, we give an overview of the two approaches. Subsequently, we present the requirements that characterize the challenge. Against this background, each of the two solutions is then introduced step by step and then evaluated against the requirements. Finally, we discuss options for cross-fertilizing and mutually evolving the two approaches.

II. Modeling Approaches

Since both approaches have been extensively described in various publications already, in this section we only provide brief overviews of their main features and principles that should be sufficient to follow the subsequent presentation of the solutions.

A. LML and Melanee

The LML is built around the principle of separating ontological classification, which captures the instance-of relationships existing in the modeled domain, from linguistic classification, which captures how model elements are represented syntactically. This separation is captured in the form of the Orthogonal Classification Architecture (OCA) in which a linguistic metamodel spans a space (i.e., linguistic level) containing multiple ontological classification levels.

LML's approach to MLM can be characterized as deep and strict. LML is deep because it supports deep characterization (the ability to define attributes or methods across multiple levels [3]) through a mechanism referred to as "deep instantiation". This is supported by the notion of potency, a non-negative integer value associated with clabjects that control the ability of their instances to have instances, over an arbitrary number of levels. If a clabject's potency value is "0" then that clabject cannot have any direct instances (although it can have indirect instances through generalization/inheritance). Usually, such clabjects are individuals, i.e. objects that represent the most concrete things in a domain. In some cases, such clabjects are also abstract clabjects but they have to be part of a generalization set and must not be a leaf in that hierarchy [13]. Attributes and methods have similar vitality properties, called

[1]Note that an early version of the language was called "Flexible Meta-Modeling and Execution Language" [9]

979-8-3503-2498-3/23/$31.00 ©2023 IEEE
DOI 10.1109/MODELS-C59198.2023.00108

durability and *mutabiltity*, that control a clabject's intension in the classification hierarchy.

LML is strict because every clabject occupies a level, and the kinds of relationships that cross levels are limited. In particular, if a clabject, x, is a direct instance of a clabject at level M_n, x must be at level M_{n-1} [1]. Not all clabjects have to have ontological types, however. Clabjects at the most abstract level are by definition not ontological instances of any other types, and so-called linguistic extensions [7], which are clabjects introduced as types at levels below the most abstract level, do not have an ontological type.

The LML is supported by a variant of OCL, called Deep OCL (DOCL), which has been enhanced with features to support deep modeling. In particular, DOCL is aware of the two classification dimensions in the OCA and provides features to support both linguistic and ontological introspection. It is also aware of the multiple levels that can exist in the ontological dimension and provides features to allow constraints to span specified ranges of levels.

Both languages are supported by the Melanee multi-level modeling tool developed at the University of Mannheim [2].

B. The FMML^x and the XModeler^{ML}

The FMML^x comes with a corresponding modeling and execution environment, the XModeler^{ML} , and guidelines for MLM. Together with the FMML^x , the latter form a comprehensive multi-level modeling method.

The FMML^x and the XModeler^{ML} resulted from the long-term, still ongoing project *Language Engineering for Multi-Level Modeling* (LE4MM, https://le4mm.org) (for a more comprehensive account of the project's history see [11]).

On the one hand, this approach is based on extensive research on language engineering which has led to the realization of a comprehensive language engineering environment, the XModeler, which is based on the reflexive "golden braid" metamodel XCore [6], [5]. On the other hand, the approach was driven by work on the development of domain-specific languages (DSML) and corresponding tools, especially in the area of enterprise modeling and enterprise systems.

In contrast, the common representation of models and programs enabled by the XModeler^{ML} obviates the need for mutual synchronization and empowers users to change parts of the system they work with, by modifying a model designed in a DSML they are familiar with.

1) The FMML^x : While XCore allows for multiple classification levels (every class created with XCore inherits from the XCore class `Class`), it does not directly support explicit levels or deferred instantiation. The FMML^x is defined as a monotonic extension of XCore. All classes specified with the FMML^x are objects, since `Class` inherits from `Object`. Every object in a FMML^x model is assigned a level, where L0 ("L" is the abbreviation of level) represents pure objects at the bottom level. In addition, properties of a class, that is, attributes, operations and associations, can be defined as *intrinsic*, which means they are subject to deferred instantiation. Intrinsic properties require the specification of the intended classification level. Associations are possible between classes at different levels. For a comprehensive description of the FMML^x and its metamodel see [8].

The FMML^x features a default concrete syntax, which among other things includes specific representations of levels and intrinsic properties. The core concepts of the FMML^x as well as its default notation are shown in the diagram in Fig. 2.

2) The XModeler^{ML} : The XModeler^{ML} builds on the XModeler and extends it by implementing the FMML^x and by providing various specific components, such as an FMML^x diagram editor, a workspace (console), an object browser, a concrete syntax designer and a facility to design custom tools. Since the XModeler^{ML} features a common representation of models and programs, every FMML^x model within the XModeler^{ML} is executable and allows for user interaction. The XModeler^{ML} gives its users the choice of three kinds of representing models: within the diagram editor, the object browser, or through a custom GUI. The XModeler^{ML} can be downloaded from the LE4MM webpage (https://le4mm.org).

A preliminary version of the modeling method that guides the use of the FMML^x is described in [10]. Certain guidelines of the method will be referred to below in the presentation of the FMML^x solution.

III. Requirements

The requirements used for the comparison fully comply with the MULTI Collaboration Challenge to support the comparison with further approaches but have been extended with four additional requirements that are designed to highlight specific differences between the two approaches. Table I shows all of the challenge requirements including the four additional requirements (No. 14 to 17). To better integrate them with the existing requirements, we also generalized one of the existing requirements slightly. More specifically, we changed requirement R-11 so that it subsumes the original requirement by stating that the S400 phone model has between 4 GB and 8 GB of RAM.

IV. Description of the Solutions

The presentations of the solutions follow a common structure that starts with a description of higher level classes that capture general information about the domain and continues with a description of more specific domain knowledge to finally outline the state of objects representing particular exemplars. While the presentations refrain from explaining trivial design decisions and describing obvious class properties, they provide the rationale for design decisions that are more demanding. As outlined above, the LML uses potencies, where the FMML^x makes use of explicit levels to specify classes and the instantiation levels of intrinsic properties. This difference needs to be recognized when comparing the descriptions of the two solutions.

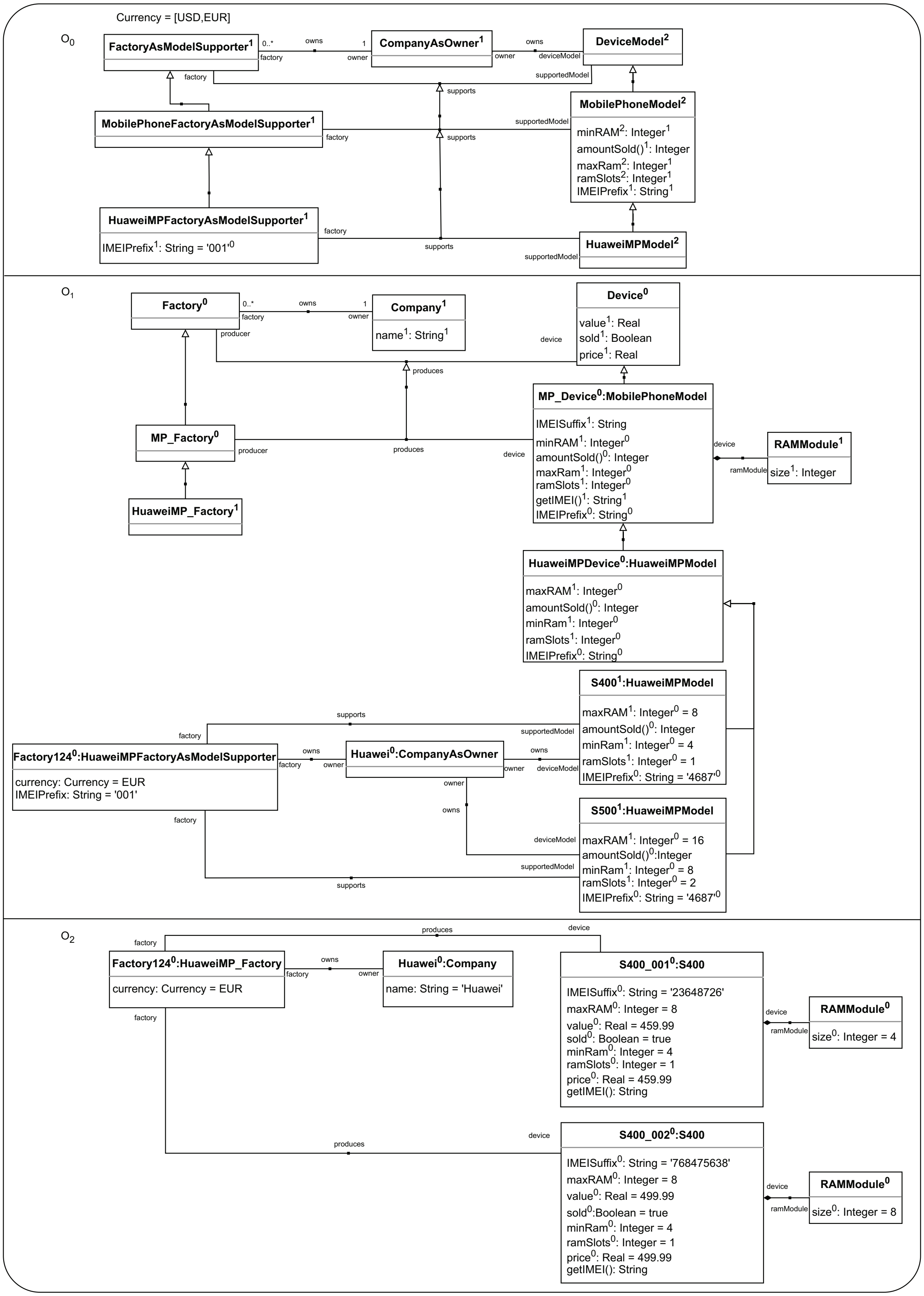

Fig. 1. LML solution

TABLE I
LIST OF REQUIREMENTS

ID	Description
R-1	A company has (a) a name, (b) owns factories, (c) owns device models
R-2	Huawei is a (a) company that (b) owns Factory124 and (c) owns mobile phone models S400 and S500
R-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
R-4	A device conforms to a device model
R-5	A device model captures what is universal about the devices it describes
R-6	A mobile phone model (a) allows specific RAM size options and (b) is a device model
R-7	A mobile phone device (a) conforms to a mobile phone model, (b) has an IMEI and (c) has a RAM size
R-8	A mobile phone factory supports mobile phone models only
R-9	A Huawei mobile 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'
R-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)
R-11	S400 (a) is a mobile phone model and (b) has between 4GB (default) and 8GB of RAM
R-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
R-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
R-N14	Particular phones may vary in terms of their memory size. Possible memory sizes are defined with the corresponding phone model. The respective definition comprises the minimum RAM size (which is the default) and the maximum memory size, which can be accomplished by adding RAM modules that are of certain sizes. The S500 phone model has a minimum memory size of 8 GB (the default) and a maximum of 16 GB.
R-N15	For accounting purposes, every device has a certain value which is expressed in the currency that was defined for the corresponding factory.
R-N16	Top management expects the number of devices that were produced per device model in a given year to be reported on demand. It also expects the corresponding accumulated value expressed in the currency defined for the corresponding factory. Furthermore, for each factory, the average value of a device across all device models is to be computed.
R-N17	All mobile phones are required to have a unique IMEI identifier which is composed of an IMEI prefix that is common to a particular mobile phone model, and an IMEI suffix that is unique to each phone.

A. LML solution

The LML solution (Fig 1) consists of three levels which are numbered from O_0 (the most abstract level) to O_2 (the most concrete level). The level O_0 hold the clabjects **DeviceModel**, **CompanyAsOwner**, and **FactoryAsModelSupporter**. The **CompanyAsOwner** clabject is connected to the **FactoryAsModelSupporter** via the **owns** connection. The **supports** connection connects the **FactoryAsModelSupporter** and **DeviceModel** clabjects (R-3(b)). The **CompanyAsOwner** is connected to the **DeviceModel** also via an **owns** connection (R-1(b,c)). In order to specialize the **supports** connection, so that only Huawei factories support Huawei mobile phone models, the **FactoryAsModelSupporter** is subclassed **MobilePhoneFactoryAsModelSupporter** and **HuaweiMPFactoryAsModelSupporter** (with the attribute **IMEIPrefix** (R-N17&R-9) set to '001' and mutability '0') while the **DeviceModel** is subclassed with **MobilePhoneModel** (with the attribute **IMEIPrefix**) and **HuaweiMPModel** respectively. This takes care of requirement R-8.

The next level, O_1, has **Company** with a **name** attribute and connects to **Factory** via the **owns** connection (R-1(a)(b)).

Huawei as an instance of **CompanyAsOwner** owns the **Factory124**, the **S400**, and the **S500** device models (R-2). It also accommodates the S400 and S500 device models (R-10).

To ensure that the model in the LML solution is well-formed with respect to the dual-level representations of **Huawei** and **Factory124**, two constraints are needed: Constraints 1 and 2 ensure that any **CompanyAsOwner** instance that is linked via **owns** to a **FactoryAsModelSupporter** instance has a name-corresponding pair of a **Company** instance that is linked via **owns** to a **Factory** instance, and vice versa. The constraints are very similar in nature, the only difference is the direction for checking the existence-implication of the connections. Checking them in both directions ensures equivalence between the **owns** connections at level O_1 and at level O_2.

```
context Company(2_2)
inv: let companyName = self.#name# in Clabject → select(
     clabject|clabject.isDeepOclTypeOf(CompanyAsOwner)) →
     select(companyAsOwner|companyAsOwner.#getPotency()# =
     0) → select(companyAsOwner|companyAsOwner.#name# =
     companyName).factory → collect(#name#) → includesAll(
     self.factory → collect(#name#))
```

Constraint 1. Linking Company to CompanyAsOwner

```
context CompanyAsOwner(1_1)
inv: let companyAsOwnerName = self.#name# in Clabject →
     select(clabject|clabject.isDeepOclTypeOf(Company)) →
     select(company|company.#getPotency()# = 0) → select(
     company|company.#name# = companyAsOwnerName).factory →
      collect(#name#) → includesAll(self.factory → collect
     (#name#))
```

Constraint 2. Linking CompanyAsOwner to Company

The **Factory** clabject is connected to the **Device** clabject via the **produces** relationship (R-3(a)).

Constraint 3 ensures that every factory only produces the devices that conform to the device models that it supports R-3(c). For every device produced by a particular O_2-level factory, its type must be among the models that are supported by the O_1-level representation (**FactoryAsModelSupporter** instance) of that factory (with which it shares the same name).

```
context Factory(2,2)
inv: let factoryName:String = self.#name# in
        let factoryTypeRole = Clabject → select(clabject|
     clabject.isDeepOclTypeOf(FactoryAsModelSupporter) →
     select(clabject|clabject.#getPotency()# = 0) → select(
     clabject|clabject.#name# = factoryName) in
        self.device → forAll(device | factoryTypeRole.
     supportedModel → includes(device.#getDirectTypes()# →
      first()))
```

Constraint 3. Factory supported devices

The **Device** clabject has subclasses which are indirect instances of **DeviceModel** and, therefore, conform to device models (R-4 and R-5). Constraint 4 returns the full IMEI string of the phone (R-7). In that expression, we first navigate to the direct type, the S400 phone model, and then to the factory it is produced in where the attribute **IMEIPrefix** is located. This value is then concatenated with the **IMEIPrefif** attribute of the particular device model, which is, in turn, concatenated with the **IMEISuffix** attribute.

```
context MP_Device::getIMEI():String (2,2)
body: self.#getDirectType()#.factory.IMEIPrefix.concat(
 self.#getDirectType()#.IMEIPrefix.concat(self.
 IMEISuffix))
```

Constraint 4. Body constraint getIMEI()

Constraint 5 checks three things regarding requirements R-N14&R-6. First, the number of RAM slots has to be greater or equal to the number of RAM modules installed in the device. The second part checks if the **maxRam** attribute value defined for the model is greater or equal to the sum of the installed RAM modules size. The last check is similar to the **maxRAM** check but checks that the **minRAM** attribute value is smaller or equal to the sum of the size of the installed RAM modules.

```
context MP_Device(2_2)
inv: self.RAMSlots >= ramModule → size() and maxRam >=
     ramModule.size → sum() and minRam <= ramModule.size →
     sum()
```

Constraint 5. Mobile Phone Model RAM option

The **Factory124** is at both levels O_1 and O_2 because at the former it supports device models, i.e., being an (indirect) instance of **FactoryAsModelSupporter**, and at the latter being the producer of the S400 device instances and itself being an instance of **Factory** (R-10).

Constraint 6 is defined in **MobilePhoneModel** and calculates the amount of sold devices. We, therefore, get all of the instances of **MobilePhoneModel** which have a potency value of ‘0’. These are the particular devices that can be sold to customers. The second part of the select statement determines if the actual device is sold or not. After the set of devices is curated we just get the size of the collection and can return the amount of sold devices (R-N16).

```
context MobilePhoneModel::amountSold():Integer (0,1)
    body: self.deepInstances() → select(clabject|clabject
     .#getPotency()# = 0 and clabject.sold = true) → size()
```

Constraint 6. Body definition for the amount of sold devices computation

Constraint 7 establishes that all potent instances of **HuaweiMPModel** must specialize **HuaweiMPDevice** so that their instances are guaranteed to have the features specified in **HuaweiMPDevice**.

```
context HuaweiMPModel(1_1)
inv: if self.#getPotency()# = 0
     then true
     else self.#getSuperTypes()# → collect(#name#)
       → includes("HuaweiMPDevice")
     endif
```

Constraint 7. Linking devices with models

The last level, O_2, contains the instances of the S400 devices that are produced by the **Factory124**, which is in turn owned by the **Huawei** company (R-10 and R-12 and R-13).

The overarching user-defined enumeration type **Currency** is tailored to the requirement R-N15.

B. FMML^x solution

The following presentation of the model developed with the FMML^x is focussed on essential aspects. It leaves out a few details that are not relevant for understanding and evaluating the solution. The complete model can be downloaded at https://le4mm.org/multi-23/. In addition, this page offers also a screencast that demonstrates the use and execution of the model within the XModeler^{ML} .

1) Focus on Generic Domain Knowledge: We follow the general design principle that known knowledge should be specified at the highest possible level within the scope of a given project [10]. At first, we identified four concepts at this level: a concept that represents companies of any kind, a concept that comprises various kinds of devices, in particular mobile devices and mobile phones, and a concept to represent various kinds of RAM Modules. Furthermore, there is a need for a concept that covers factories where devices of any kind are produced, and, more specifically, mobile phones. These concepts are represented in the classes **GenericDevice**, **GenericFactory**, and **RAM**, all at L2, and the class **Company** at L1.

The association **owns** between the classes **Company** at L1 and the **GenericDevice** serves to express that a company (at L0) owns device models, represented by a class at L1 (R-1). The **GenericDevice** represents general knowledge about all kinds of device models and particular devices in the domain. It defines the attribute **imei** as well as the intrinsic constraint **correctIMEI**, both applied at L0. The constraint checks whether a particular phone's IMEI is unique R-N17.

```
context GenericDevice, L0
@Constraint correctIMEI
    self.imei.hasPrefix(self.of().imeiPrefix) andthen self.
     of().allInstances()→excluding(self)→forAll(other |
     not self.imei.equals(other.imei))
fail
    "This IMEI is not valid."
end
```

The class **GenericDevice** also includes the intrinsic attribute **value**, which is specified with the class **MonetaryValue**, which in turn uses the class **Currency** to represent amounts of money (R-N15).

Since this knowledge applies to device models, the corresponding attributes are to be instantiated at L1. The intrinsic attribute **dateProduced** on the other hand serves the description of particular device exemplars. Therefore, it is to be instantiated only at L0. The attribute **refCurrency** of the class **Currency** within the **Company** serves to define a reference currency with every particular company R-N15. A company may own device models and factories, which is represented by the two associations **owns** and **ownsFactory**, where the latter is instantiated with an object at L1 on the side of **GenericDevice** R-1.

A further class at L2, **RAM**, serves the specification of RAM types. The association **hasRam** between the classes **RAM** and **GenericMobileDevice**, both at L2, serve to express that a mobile phone model allows its instances to vary with respect to memory size R-N14. Note that **minRam** in concretizations of **GenericMobileDevice** defines the RAM size every phone is equipped with as a default. The rules that restrict possible configuration options are defined in the constraint **properRAM**:

```
context GenericMobileDevice, L0
@Constraint properRAM
self.getRAMs()→iterate(ram sum = 0 |
     sum + ram.of().sizeGB) + self.of().minRAM <= self.of()
     .maxRAM
fail
    "Actual RAM must not exceed maximum RAM."
end
```

The diagram in Fig. 2 illustrates how the XModeler^{ML} depicts the violation of a constraint within an object.

Two more associations as well as three further constraints are used to complete the level of generic domain knowledge. The constraint **correctCurrency** of the class **GenericDevice**, which applies to L1, checks whether the currency used for expressing the default value of a phone defined with its model (at L1) complies with the reference currency defined with the corresponding factory R-N15.

The constraint **properModel** within the class **GenericFactory** is to make sure that a factory may produce only devices of device models that are supported by this company. Hence, this constraint facilitates the definition of the dependency of the association **produces** from the association **supports** R-3.

2) Focus on Specific Domain Knowledge: The classes at this level are concretized from classes at the generic domain knowledge level. Hence, the upper-level classes can be regarded as the specification of a DSML for modeling device models within a manufacturing context.

Device models are described as concretizations of the class **GenericDeviceModel** (R-4). Requirement R-6 is accounted for by linking a specific mobile phone model to the RAM module types, it may be extended by. Constraint **properProduction** of the class **MobilePhoneFactory** is to ensure that a mobile phone factory must produce mobile phones only R-8. The value of a phone changes with its memory configuration. The **totalValue()** defined with the class **GenericMobileDevice** computes this value for each exemplar.

The phone model S400 is represented by the class **S400**. Its default memory size of 4 GB is represented in the slot value of the attribute **minRAM**. Possible extensions of the memory size of its instances R-11, R-N14 are represented by the association **hasRAM** that allows creating links between an object representing a particular S400 device and RAM modules of any size, and the constraint **ProperRAM** that was already defined with **GenericDeviceModel**.

The class **ReportManager** serves the implementation of operations that compute reports for managerial purposes R-N16. The operation **averageDeviceValue**, for instance, calculates the average value of a device per device model and year:

```
context ReportManager, L0
@Operation avgDeviceValue():Auxiliary::MonetaryValue
  try
    let factory = Factories::GenericFactory.allInstances()
     →collect(F |
                  F.allInstances())→flatten→select(f |
                  f.name.toString().equals(self.
     factoryInFocus)).asSeq().at(0) then
        result = Auxiliary::MonetaryValue(0,Auxiliary::
     Currency("NUL","NUL",1000.0));
        devices = factory.getProdDevices()→select(device |
                          device.dateProduced.toString().
     subString(7,11).equals(self.queryYear.toString()))
    in @For device in devices do
        result := device.totalValue().add(result)
      end;
      if devices.size() = 0
      then "DIV/0"
      else result.mul(1 / devices.size())
      end
    end
  catch(e)
    "An error occured: " + e.message
  end
end
```

Year, selected device model, and factory are represented by the slots related to the attributes **queryYear**, **deviceInFocus**, **factoryInFocus**. The XModeler^{ML} allows the modification of slots by double-clicking them. Since device and factory are represented as strings, two operations, **changeDeviceModel** and **changeFactory** were added to make sure that the strings entered by a user are valid identifiers of device and factory objects.

3) Focus on Particular Exemplars: The classes introduced so far constitute a complete conceptual model in the sense that they specify all conceptual knowledge. The further instantiation of this model serves as the representation of particular instances.

The company Huawei is represented by the object **Huawei**, which, together with the links **owns** to **S400** and **S500** and the link **owns** to **Factory124** satisfies R-2. The navigable links **ownsFactory** and **produces** enable the company to "keep track of mobile phones it produced" R-9. The constraint **properModel** defined with the generic domain class **GenericFactory** makes sure that the company produces only devices, whose models it owns R-9.

The particular phones described in R-12 and R-13 are represented by the objects s400_001 and s400_002. The constraint **properIMEI** defined with the classMobilePhoneFactory ensures that IMEIs of these phones are valid R-N17.

C. Comparison of Solutions

Table II gives a comparative overview of both solutions. For each of the two solutions, an "s" indicates that the respective requirement is satisfied. Those cases where both solutions are characterized by a largely corresponding approach are indicated by "corresponding" in the comment column. Different approaches are marked as "different". Note, to compare both solutions in Fig. 1 and Fig. 2, it is required to "translate" potencies into explicit classes in the FMML^{x} solution, and vice versa.

V. Discussion

In this section, we compare the two approaches by discussing their difference and similarities at various levels of abstraction and detail. As requested in the Challenge, which calls for an emphasis on fundamental issues, we start by discussing underlying principles and language architectures and gradually introduce more concrete concerns related to the specific ways the approaches model the domain and fulfill the requirements.

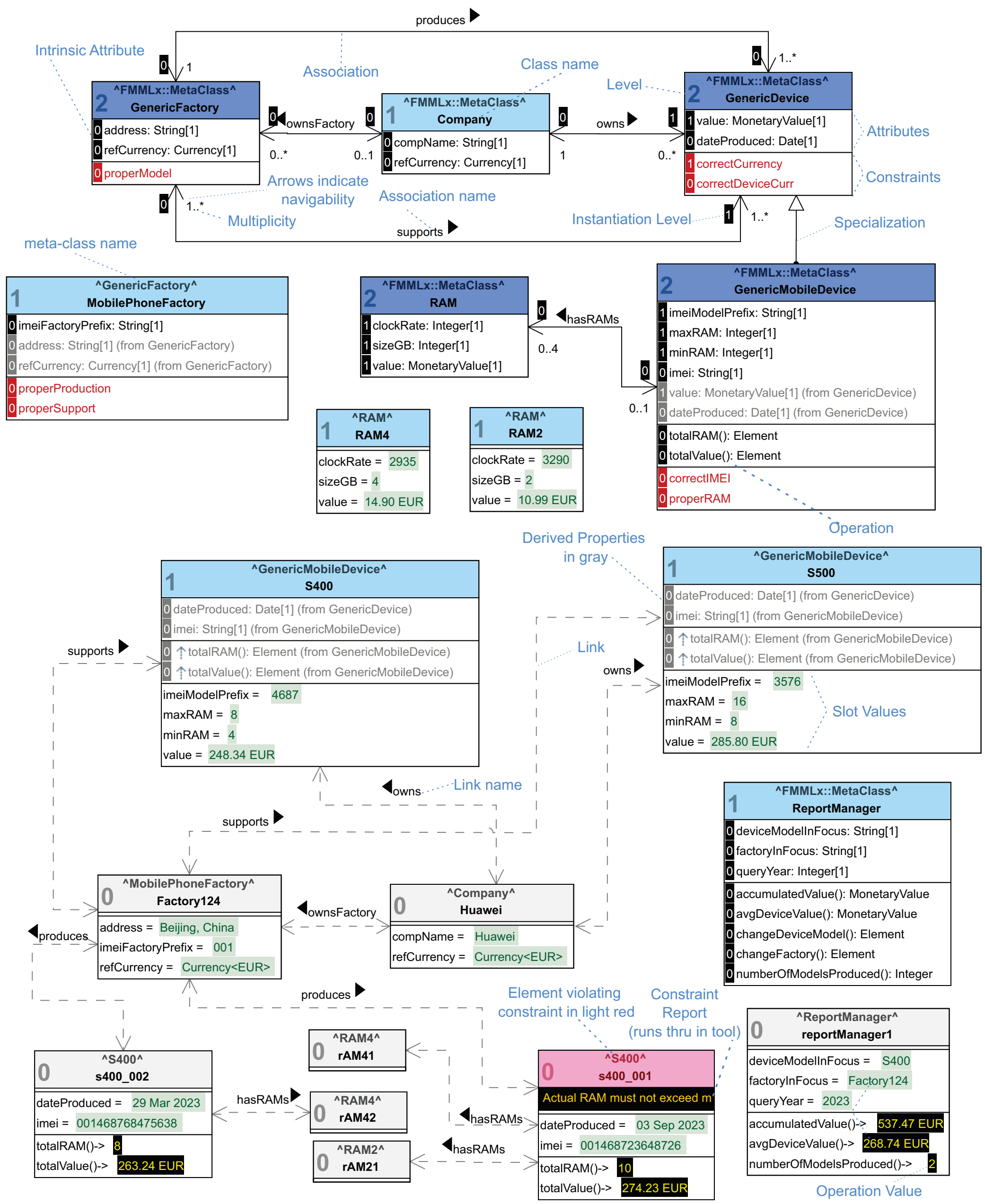

Fig. 2. FMML[x] solution

A. Language Architecture

At first sight, the underlying architectures of the LML and the FMML[x] appear to be fundamentally different. The LML architecture is based on the OCA which explicitly distinguishes ontological versus linguistic classification and organizes them into separate dimensions, while the FMML[x] architecture is based on a "golden braid" (meta)model stack. However, both approaches fulfill the same basic purpose of building the modeling features around a small, reflective core from which all model elements are derived. In the case of LML, this is achieved by orthogonality (i.e., by placing the common core in a linguistic metamodel that "spans" all domain model elements) while in FMML[x] this is achieved by meta-circularity (i.e., placing the common core in a self-describing (meta)model at the top of a model stack from which all domain model elements are derived).

Although the FMML[x] core does not explicitly refer to ontology, therefore, it follows a common idea of ontologies in philosophy, that is, it describes the basic concepts used to describe all things – and classes of things – in the world

ID	LML	FMML^x	Comment
R-1	s	s	*different* LML uses classes at more than one level.
R-2	s	s	*different* – while the classHuawei is represented twice in the LML solution as a consequence of its inherent strictness constraint, it is represented without redundancy in the FMML^x solution. The LML solution uses a constraint to connect them because they represent the same real-world entity.
R-3	s	s	*different* – while both approaches correspond with respect to (a) and (b), they are different with respect to (c). The LML makes use of multiple different associations that are all named "supports" and "produces" between different classes, where the FMML^x solution requires only the intrinsic associations **`supports`** and **`produces`** between the two classes **`GenericFactory`** and **`GenericDevice`** at L2 that is "instantiated" into a link with the same name which connects the object **`Huawei`** at L0 and the classes **`S400`** and **`S500`** at L1. The LML solution needs more associations to represent **`supports`** relationships at lower levels but needs fewer constraints because this language concept allows to specialize the **`supports`** relationship. Different from the FMML^x , the notation of the LML does not clearly distinguish between associations and links. To express the fact that only those devices can be produced by a factory that supports the corresponding device model, both approaches make use of constraints.
R-4	s	s	*different* LML uses classes at more than one level.
R-5	s	s	*different* – while both solutions define the requested properties in one class each (**`MobilePhoneModel`** and **`GenericMobileDevice`** respectively), the LML solution repeats the corresponding specification in the lower level classes **`MP_Device`** and **`HuaweiMPDevice`**. Device models describe devices by being the types of the latter.
R-6	s	s	*corresponding* – both approaches define RAM size options through an association with a class representing RAMs. The field representing the incarnations of attributes like **`minRAM`** are shown together with the datatype in the LML solution, which is not the case with the FMML^x solution.
R-7	s	s	*different* – both approaches define the required properties. However, these are represented in one class only in the FMML^x solution (**`GenericMobile`** at L2), where they are distributed/repeated in four different classes at two levels in the LML solution. Also, the specific RAM size of a device is computed by the operation **`totalRAM`** defined in the class **`GenericMobileDevice`**.
R-8	s	s	*different* – both solutions are built on associations that express support relationships between factories and mobile phone models. The FMML^x solution does that only once with the association **`supports`** between the classes **`GenericFactory`** and **`GenericDevice`**. The constraint **`properProduction`** in the class **`MobilePhoneFactory`** serves to make sure that a particular mobile phone factory may produce mobile phones only. The LML solution uses multiple classes for this purpose but does without an additional constraint (3)
R-9	s	s	*different* – within the FMML^x solution, the constraint **`properSupport`** ensures (a), which is not required by the LML solution (b) the FMML^x solution introduces a specific class for that purpose (**`ReportManager`**), and uses an attribute to define the IMEI prefix. The LML solution defined *IMEIPrefix* in *Factory124*
R-10	s	s	*different* – while there is only one occurrence of the factory object in the FMML^x solution, the LML solution requires two.
R-11	s	s	*corresponding*
R-12	s	s	*corresponding*
R-13	s	s	*corresponding*
R-N14	s	s	*corresponding* – see also R-6
R-N15	s	s	*different* – while the FMML^x solution defines the value attribute with the class **`GenericDevice`** at L2, the LML solution uses the class **`Device`** for that purpose. Also, the FMML^x solution uses a specific class, **`MonetaryValue`** to represent values, whereas the LML solution uses the datatype **`Real`** only.
R-N16	s	s	*different* – while the FMML^x solution introduces a dedicated management class (**`ReportManager`**) here, where the LML solution defined the method *amountSold()* in **`MobilePhoneModel`** at level O_0 which when invoked returns the amount of devices sold.
R-N17	s	s	*corresponding* – both approaches use definitions of prefixes for the Huawei company and for mobile phone models. The LML solution uses the *getIMEI* method to return the concatenated string of **`IMEIPrefix`** and **`IMEISuffix`**

TABLE II
COMPARISON OF THE TWO APPROACHES WITH RESPECT TO REQUIREMENTS

(e.g., [4], [12]). These basic concepts are applied to describe any class of things (concepts). Accordingly, all classes in a FMML^x model inherit from the metaclass **`Class`**, which defines generic class properties, attributes as well as associations, and, to cover functional aspects, operations. To specify and implement operations as well as constraints, it makes use of the XOCL. Constraints specified with the XOCL may span multiple levels [14].

The linguistic metamodel of the LML defines clabjects as an abstraction over classes and objects, XCore, and the FMML^x respectively distinguish between the metaclasses **`Class`** and **`Object`**. This distinction is used to separately define characteristic properties of objects (with each class being an object), e.g., that they have state or can execute methods.

B. Core Modeling Features

Again, the differences between the core modeling features used to express models are largely superficial and are rooted in terminology rather than semantics. Both approaches are oriented towards the well-known UML syntax (both abstract and concrete) to represent the entities, relationships, and properties existing in the domain of interest. Whereas the LML favors the term "clabject" to emphasize the fact that entities in a domain are represented in a way that unifies their type and instances roles, FMML^x favors the more traditional terms of "class" and "object". However, since the core FMML^x metamodel declares all classes to also be objects, the effect is the same. All classes in FMML^x correspond to clabjects, although as with LML some may only play the role of an instance while others may only play the role of a type.

In terms of modeling relationships, both languages use an approach that essentially unifies the notion of associations/links from the UML. The only significant difference is that LML allows such connections to enter into inheritance relationships while FMML^x defines specialization as a concept on its own. Both languages essentially use a similar notation to represent attributes and slots. However, while the FMML^x retains the separation between attributes or slots (i.e., a property is represented either as an attribute or a slot) LML uses the dual field approach [3], where properties can sometimes be both.

Finally, both languages support the specification of operations. For this purpose, the FMML^x uses the XOCL, which is a complete programming language. DOCL is not a programming language (or action language in MDD terminology) but can describe the behavior of operations and execute them via "body" constraint expressions.

C. Levels

One of the aspects of MLM where FMML^x and LML differ more significantly is how they define and relate levels. LML takes a more traditional (i.e. UML-like) approach by defining levels in terms of the classic "instance-of" relationship between classes and objects (as found in the UML and object-oriented programming languages). This also means that LML enforces a traditional separation between classification (i.e., instance-of relationships) and specialization (ie., subclass relationships). In contrast, FMML^x supports a more generic and flexible approach for defining levels, referred to as concretization, which combines instantiation and inheritance (not: specialization). In LML, therefore, the clabjects at one level are instances of the clabjects at the level above, while in FMML^x they are concretizations of the level above. Specialization is available in the FMML^x and restricted to classes at the same level.

As well as defining levels in different ways, the two languages also differ slightly in the way they allow clabjects at different levels to be related. Again, LML takes the more traditional, UML-like approach whereby all instances of a clabject must reside at the immediate level below, all specialization relationships must be confined to a single level, and no connections (i.e., associations/links) can cross levels. FMML^x also adopts the first two rules (but for concretizations rather than instances in the first case), but allows connections to cross level boundaries.

D. Deep Characterisation

Another area where the two approaches differ somewhat is in their approaches to deep characterization – that is, the ability for clabjects to control not only the features of their immediate instances/concretizations but also of deeper instances/concretizations created by further refinement steps. LML primarily uses three vitality properties associated with model elements to perform deep characterization - potency, which governs how many levels below a clabject may have instances, durability, which governs over how many instantiation steps a clabject must have a slot/attribute (i.e., a field) and mutability, which governs over how many levels the value of a field can vary. In contrast, FMML^x primarily uses the notion of "intrinsicness" to perform deep characterization. Instrinsicness essentially defines the level at which a property is to be instantiated, or, in the case of an operation, executed.

E. Characterisation Proximity

The final aspect of MLM where LML and FMML^x differ is in their approach to what we here call "characterization proximity". Although the difference is simple to understand, it is the origin of probably the largest visual difference between LML and FMML^x models which is the number of model elements they contain.

LML follows a **proximate characterization** strategy in which the required properties of the elements at a particular level (except the top level) are "fully" characterized by the elements above. This means that in order to know what constraints apply to model elements at a level x, modelers only need to look at level x+1. This in turn means that it is not necessary to deliver the whole multi-level model whenever it is necessary to provide stakeholders with a characterized ontological level of a domain model. However, since the properties of level x+1 may well be, and often are, controlled by level x+2, and so on, this means that multi-level models employing proximate characterization often contain redundant information.

FMML^x , on the other hand, follows a **concise characterization** strategy where only the minimal necessary model elements are included in a multi-level model to characterize all levels. All levels (except the top) are fully characterized in FMML^x , but not necessarily at the level immediately above. If users of the FMML^x want to save themselves the trouble of navigating to the upper-level ancestors of the class they focus on, the XModeler^{ML} provides filters that allow fading in properties that were defined further up the hierarchy. In the diagram, these are displayed in grey and supplemented with specific details.

Table III shows the impact of these different approaches to characterization proximity on the size of the two solutions in terms of the number of model elements. This shows that the LML model includes far more classes/clabjects, attributes, associations, and slot values above the bottom level because of the redundancy needed to achieve proximate classification. FMML^x requires far fewer model elements because much more of the domain knowledge can be abstracted to higher levels and described only once. The table also shows that the LML and FMML^x solutions require the same number of constraints, although this involves making use of the language feature that allows for specializing connections (see the `supports` connection).

F. Productivity and Comprehensibility

We were not able to determine which approach offers advantages in terms of modeling productivity. However, given

TABLE III
SELECTED METRICS

Description	LML	FMML^x
no. of classes	19	11
no. of attributes	23	15
no. of slot values above L0	12	9
no. of associations	12	5
no. of constraints	7	7

the current level of modeling support provided by each tool, we suspect that the creation of LML models usually requires greater effort. This is because of the redundant model elements required by LML to support proximate classification. The XModeler^{ML} on the other hand can insert some of this information automatically if desired (e.g., the light grey attributes in GeneralMobileDeive S400 and S500) if a user desires proximate classification, or can leave it out if a user desires concise classification.

The relative understandability of LML and FMML^x models is likely to depend on the prior background of the modeler. Modelers who are familiar with the UML are likely to find LML models easier to understand, at least initially, since apart from potency, the level content in an LML model follows the rules a UML modeler is likely to expect. However, a person whose perspective is not confined to the UML view of the world may find FMML^x models easier to understand since they are usually concise and involve fewer new modeling constructs like the vitality properties.

G. Executability and Behaviour Description

One of the largest differences between the LML and FMML^x modeling environments (i.e., Melanee and XModeler^{ML}) is their support for executability and behavior specification. XModeler^{ML} is actually a fully blown programming environment, so all the code written in the FMML^x solution is executable. In the XModeler^{ML} , there is no distinction between programs and models, they share the same representation. This is not the case with Melanee which does not provide the same support for execution. Like OCL the DOCL constraint language used in the LML solution is mostly declarative and DOCL constraints cannot make any changes to the instance of the classes to which they are applied. Nevertheless, using OCL-like "body" and "derive" specifications it is possible to define the behavior that operations are required to have (and execute them) as well as setting values to attributes. The implementations of the specified behavior are deferred to other technologies, however, such as programming language.

VI. CONCLUSION

Working on the collaborative challenge was beneficial for both participating groups. Even though both groups were aware of each other's approaches, cooperation during the creation of the solutions led to a number of questions that could only be clarified in joint discussions. On the one hand, this resulted in new insights for both groups, and on the other hand, it led to suggestions concerning the further development of both approaches. In the case of FMML^x the work on the challenge has indicated the need for two complementary language concepts. For example, LML's ability to represent relationships between associations/links has proved so useful that it is planned to extend FMML^x with a corresponding concept. In addition, it has become apparent in the discussions that, even if they are largely equivalent to explicit levels, potencies offer advantages in some cases. Therefore, it is being considered to offer potencies in FMML^x as well. The collaboration has also led to a desire to promote the integration of the languages. To this end, we will consider two alternatives: the specification of an exchange format based on a common metamodel, and the implementation of the LML in the XModeler^{ML} , which would then allow us to work on one common multi-level model using two different languages.

REFERENCES

[1] Colin Atkinson. Meta-modeling for distributed object environments. In *Enterprise Distributed Object Computing*, pages 90–101. IEEE, October 1997.

[2] Colin Atkinson and Ralph Gerbig. Flexible deep modeling with melanee. In *Modellierung 2016 - Workshopband*, volume 255, pages 117–121, Bonn, 2016. Köllen.

[3] Colin Atkinson and Thomas Kühne. The Essence of Multilevel Metamodeling. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, Lecture Notes in Computer Science, pages 19–33. Springer Berlin Heidelberg, 2001.

[4] Mario Bunge. *Treatise on Basic Philosophy: Volume 3: Ontology I: The Furniture of the World*. Reidel, Dordrecht, 1977.

[5] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language Driven Development*. Ceteva, 2 edition, 2008.

[6] Tony Clark, Paul Sammut, and James Willans. *Superlanguages: developing languages and applications with XMF*. Ceteva, 2008.

[7] Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In Jan Vitek, editor, *Objects, Models, Components, Patterns*, pages 1–20, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] Ulrich Frank. The flexible modelling and execution language (FMML^x) – version 2.0: Analysis of requirements and technical terminology.

[9] Ulrich Frank. Multilevel modeling: Toward a new paradigm of conceptual modeling and information systems design. *Business and Information Systems Engineering*, 6(6):319–337, 2014.

[10] Ulrich Frank. Prolegomena of a multi-level modeling method illustrated with the FMML^x . In *Proceedings of the 24th ACM/IEEE International Conference on Modell Driven Engineering Languages and Systems: Companion Proceedings*. IEEE, 2021.

[11] Ulrich Frank and Tony Clark. Language engineering for multi-level modeling (le4mm): A long-term project to promote the integrated development of languages, models and code. In Jaime Font, Lorena Arcega, José-Fabián Reyes-Román, and Giovanni Giachetti, editors, *Proceedings of the Research Projects Exhibition at the 35th International Conference on Advanced Information Systems Engineering (CAiSE 2023)*, CEUR, pages 97–104. CEUR-WS.org, 2023.

[12] Reinhardt Grossmann. *The Categorical Structure of the World*. Indiana University Press, Bloomington, 1983.

[13] A. Lange and C. Atkinson. On the rules for inheritance in lml. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 113–118, Los Alamitos, CA, USA, sep 2019. IEEE Computer Society.

[14] Tony Clark and Ulrich Frank. Multi-level constraints. In Regina Hebig and Thorsten Berger, editors, *Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018*, volume 2245 of *CEUR Workshop Proceedings*, pages 103–117. CEUR-WS.org, 2018.