Peculiarities of Language Engineering in Multi-Level Environments or: Design by Elimination

A Contribution to the Further Development of Multi-Level Modeling Methods

Ulrich Frank
ulrich.frank@uni-due.de
Universität Duisburg-Essen

Tony Clark
tony.clark@aston.ac.uk
Aston University, Birmingham

ABSTRACT

Multi-level modeling (MLM) facilitates the design of modeling languages because foundational language concepts (defined with "linguistic" metamodels) can be reused on any classification level and consequently frees the developer from the burden of re-specifying these concepts each time a new language is designed. This strength of MLM can be used profitably in teaching since it enables students to specify languages with relatively little effort with associated tool support. However, MLM introduces new features that require existing methods to be extended with user support and which introduce verification challenges. This paper describes these challenges with respect to teaching modeling languages and outlines approaches to address them.

KEYWORDS

software system architecture, DSML, language integration

ACM Reference Format:
Ulrich Frank and Tony Clark. 2022. Peculiarities of Language Engineering in Multi-Level Environments or: Design by Elimination : A Contribution to the Further Development of Multi-Level Modeling Methods. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3550356.3561544

1 INTRODUCTION

Our work on multi-level language architectures is motivated by serious obstacles we encountered with the design of domain-specific modeling languages (DSMLs) and the implementation of corresponding tools. The lack of abstraction that is characteristic of MOF-like architectures forced us to represent knowledge redundantly which is a threat to reuse, adaptability and integrity [5, 11]. In addition, it was not possible to instantiate models within a model editor, because all classes, or metaclasses respectively, had to be implemented as objects on M0.

Definition of a multi-level modeling language, the FMML^x [8], and its implementation in the XModeler [6], called XModeler^{ML} [12] addresses most of the issues noted above. However, further development of the language and the tool raised new challenges (e.g., [12, 13]).

A few years ago we introduced multi-level modeling into a Master's course on advanced conceptual modeling. It did not come as a surprise that many students struggled with the new paradigm. To support them achieve an effective understanding of multi-level modeling we developed a specific design method [10] and provided numerous example models[1]. Nevertheless, it remains a challenge to convince students of the merits of the approach. Our experience to date indicates that problems highlighting limitations of traditional non-MLM approaches is an effective motivation for ambitious students.

Recently we realized that multi-level modeling is suited to introductory modelling courses at undergraduate level where we currently teach general-purpose languages such as the ERM, DFD, state charts and the UML. To promote students' understanding of modeling languages, they receive simplified versions of corresponding metamodels. To illustrate the instantiation of models from metamodels, elements of metamodels are linked to the concept they are instantiated from. Nevertheless, many students have a hard time understanding this relationship. Also students appear to struggle with model relationships, e.g., a UML object model and its corresponding instantiations.

Since the XModeler^{ML} supports the integration and execution of models at any level, including L0[2], into one multi-level diagram, it gives students the opportunity to modify a model and see the immediate effect on corresponding instances and allows students to check whether a specific instance model is a valid instantiation: in case it is not, the XModeler^{ML} will prevent them from creating it. Preliminary experience using this approach indicates that it improves students' understanding of models and their relationships and it seems obvious to extend the approach to metamodels and their instantiations where students would have the opportunity to interact with objects, models and the corresponding metamodels at the same time. This turned out to be a profound challenge, not only for integrating metamodels of general-purpose modeling languages (GPMLs) with their instantiations, but also with respect to language design within multi-level architectures in general.

In the following we will at first illustrate the benefits multi-level modeling offers for teaching the use of traditional GPML and then analyse the challenge that results from using a multi-level language architecture for this purpose. We demonstrate that this

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9467-3/22/10…$15.00
https://doi.org/10.1145/3550356.3561544

[1] see https://www.wi-inf.uni-duisburg-essen.de/LE4MM/
[2] in the context of multi-level models, we refer to levels with the letter "L" to emphasize that the notion of level in multi-level architectures is different from that in MOF-like architectures.

challenge needs to be addressed when designing multi-level DSMLs, too. Finally, we will discuss key approaches to the challenge and outline an implementation.

2 MULTI-LEVEL ARCHITECTURES FOR TEACHING GENERAL-PURPOSE MODELING LANGUAGES

Modeling tools that are based on a multi-level language architecture support simultaneous work on a model and its instance within one diagram. This opens up exciting opportunities to support teaching of traditional GPMLs.

2.1 Potential Benefits

GPMLs usually feature a relatively small set of concepts that are relevant for teaching introductory level courses. This is the case, e.g., for ERM and for DFDs, but also for the UML as long as basic concepts are concerned. Our experience with teaching various GPMLs in a business informatics Bachelor's program indicates that students often have a problem understanding the purpose of a metamodel and relating models to a corresponding metamodel. In addition, some students even struggle with understanding the relationship between a model such as a UML object model and corresponding instantiations. A MLM tool like the XModeler^{ML} is suited to support students develop knowledge and skills over multiple levels of abstraction because metamodels, models and instances can co-exist within the same editor.

The following two examples illustrate this feature for the ERM and DFDs. Constructing a metamodel with the XModeler^{ML} helps students understand language design. As soon as a metaclass is added to a diagram, it will appear in the palette, where it can be selected for creating a corresponding model element. Note that we refrain from the definition of the concrete syntax, since the didactic focus is on abstract syntax and semantics. Fig. 1 shows a metamodel of DFDs that was created with the XModeler^{ML} . As soon as the metamodel is specified, students can create DFDs. In addition, they may change the metamodel during or after the design of DFDs. For example, the metamodel in Fig. 1 was extended by the attribute `critical` of the class `Function`, which results in adding a corresponding slot to all instances of `Function`. Instantiating a metamodel supports students with checking whether it is complete. Apparently, this is not the case for the metamodel in Fig. 1, because it allows, among other things, the existence of functions in a DFD that are not connected to a data flow.

The design and use of a metamodel that represents the ERM is shown in Fig. 2. Note how constraints can be added and tested immediately where issues are displayed on the diagram where constraints fail to be satisfied. While there is no need to introduce the idea of MLM, the students should be aware of different classification levels which are shown as numbers in diagram elements. In addition, they should be told that every class, no matter whether it is on M1 or M2, implicitly includes the attribute `name`. Note that we excluded ternary relationships for didactic reasons.

The next example relates to a UML-like language and shows how a model and its instantiation can be represented in one diagram. In addition, the XModeler^{ML} allows the methods defined for a class to be implemented and executed so that students can quickly build a small application that is integrated with the object model it is based on and can interact with the objects they create through the diagram or through additional text browsers (see Fig. 3). Modifications of the metamodel have an immediate effect on corresponding models.

2.2 Limitations

The examples shown above demonstrate how a MLM tool can be used to promote students understanding of traditional GPMLs by representing two levels simultaneously. It would be nice to further enrich the diagrams. This is especially the case for the ERM and the UML-like diagrams, because DFDs are usually not shown at M0. That would require the ERM diagram to be supplemented with objects at M0 that are instantiated from the data model. In addition, the UML-like diagram would have to be supplemented by a metamodel of object models. Unfortunately, both extensions are not trivial.

Without further measures the data model shown in Fig. 2 does not support instantiation because instances of `EntityAttribute` and `RelationshipAttribute` cannot be further instantiated. To enable the instantiation of attributes, one could add an attribute like `value` to the `Attribute`. In order to express that it is to be instantiated only at M0, it would have to be marked as intrinsic with the instantiation level 0. But that would not be enough. In addition, it would be required to somehow establish links between a slot value at level 0 and the corresponding attribute specification at level 1. This could be achieved by adding an intrinsic association between `EntityType` and `RelationshipType`, which would to be instantiated at level 0. A constraint, specified either with `EntityType` or `EntityAttribute` would then have to ensure that instances of instances of `EntityType` may be linked to instances of instances of `EntityAttribute`, if their respective classes are linked on level 1. Handling relationships would require an even greater effort. First, relationship attributes and their slots would need to be handled the same way as entity attributes. In addition, there would be need to make sure that objects at level 0 are properly linked — according to the corresponding relationships defined at level 1. Furthermore, the instantiation mechanism in the tool would have to be adapted to handle the multiplicities defined with instances of `Link` adequately. Apparently, such an approach to enable the representation of objects on three levels would not only require a significant effort. It would also fail to serve the primary objective, that is, to help students with developing a better understanding of defining and using modeling languages. Instead, it would likely contribute to students' confusion.

The object model in Fig. 3 includes objects at level 0, but does not include a metamodel. Adding a metamodel would require an effort comparible to extending the metamodel of the ERM, which is, therefore, not an option. Furthermore, it would be strange to add a metamodel, since there is already a metamodel that allows to create UML-like object models. The concepts defined with the metamodel of the FMML^{x} , which in turn is an extension of XCore [6, p. 78] are available at every level above 0. This corresponds to the idea of a "linguistic metamodel" [4]. We will refer to it as "foundational metamodel" in the following. Therefore, they do not have to be specified anew. This represents a great advantage of multi-level language

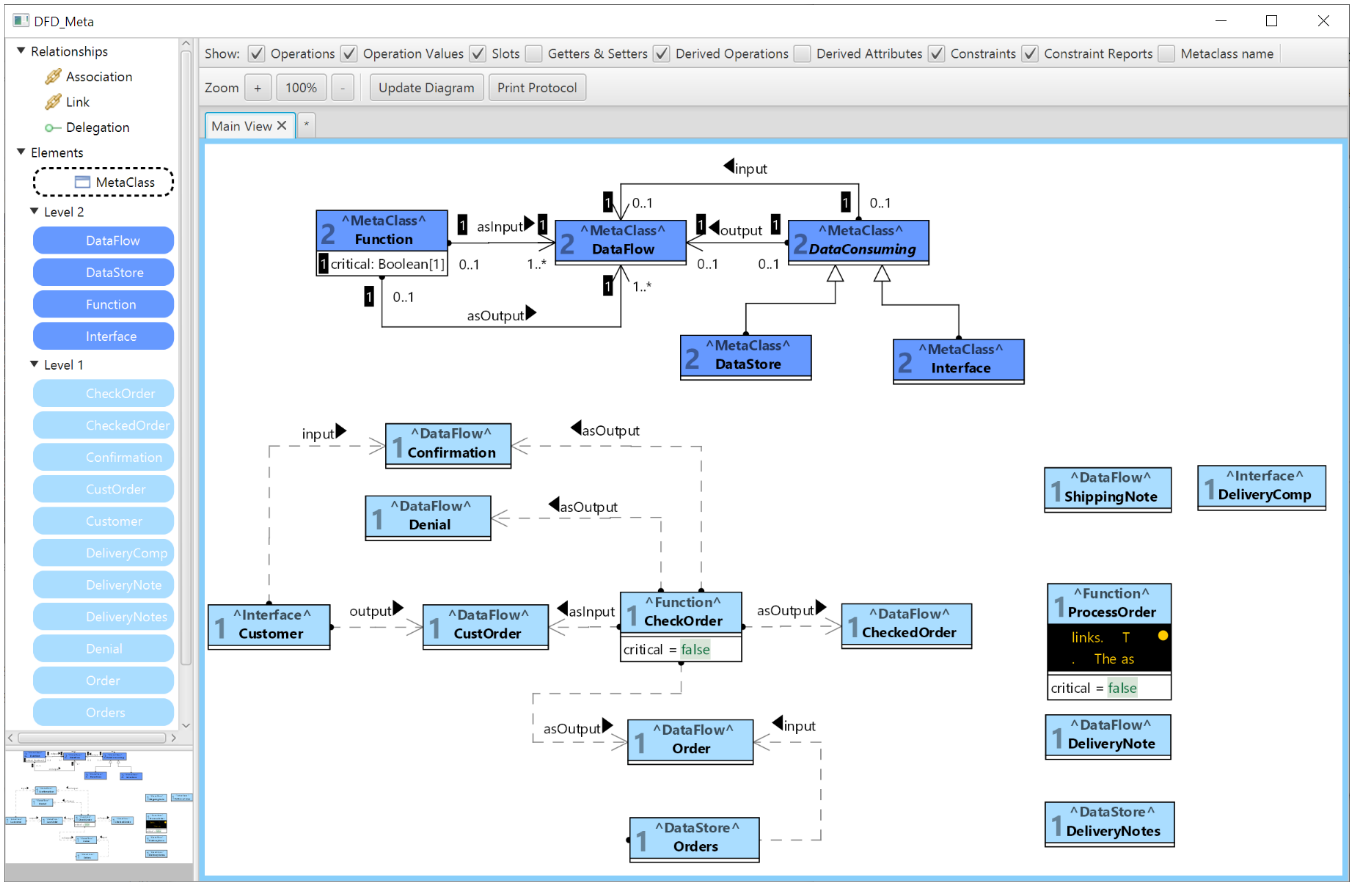


Figure 1: Metamodel of DFD and example instantiation

architectures that follow the idea of "Orthogonal Classification Architectures" [3]. Probably everyone who has developed modeling languages within a MOF like architecture will have suffered from the need to specify concepts like associations or attributes anew when they were needed for a modeling language, even though they had been defined with the corresponding meta-modeling language. However, even though the FMML^x enables the construction of UML-like object models that are sufficient for teaching at the undergraduate level, it includes concepts that are inappropriate for objects models, such as intrinsic features that allow for deep instantiation. These concepts are not only suited to confuse students, they would also allow the construction of object models that are not consistent with the idea of traditional UML-like objects models.

Before we present strategies that utilize the power of ubiquitous language concepts without the threat to compromise model integrity, we will look at peculiarities of designing DSMLs within multi-level language architectures.

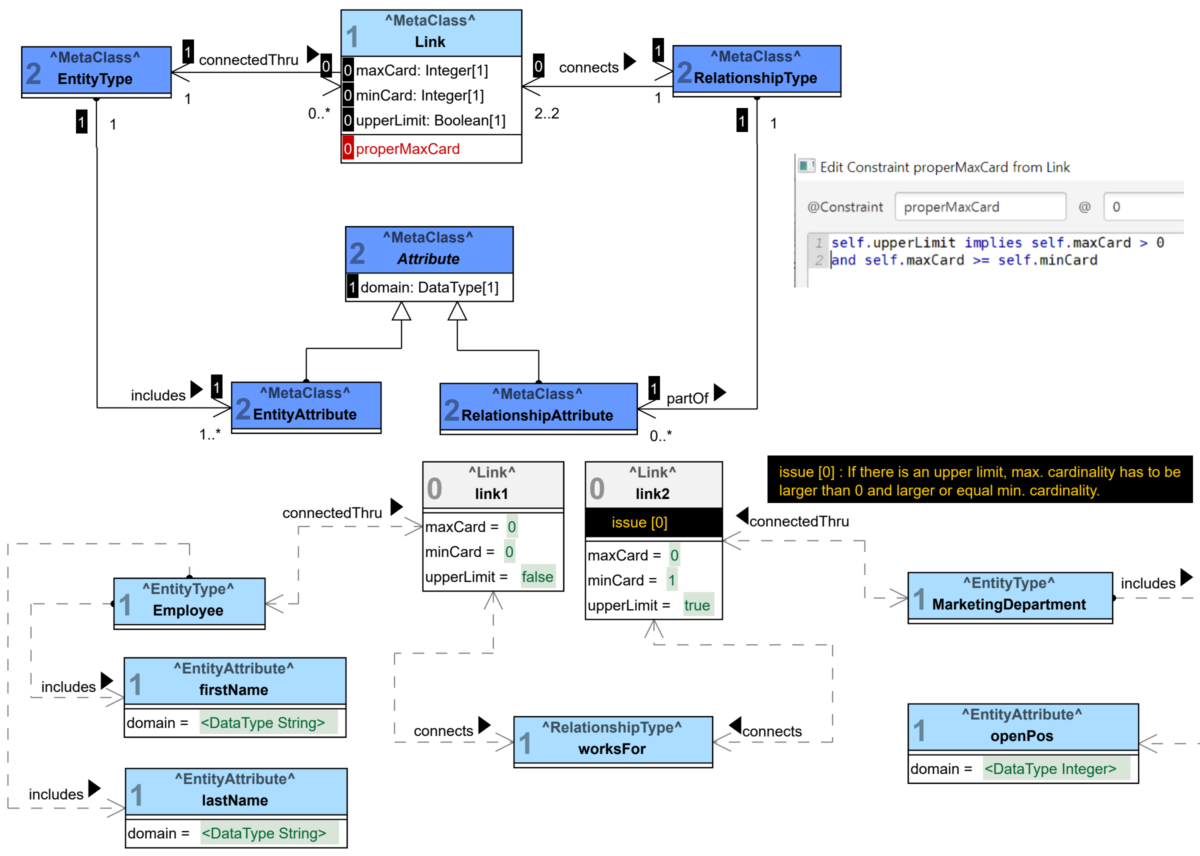
3 FOCUS ON DOMAIN-SPECIFIC LANGUAGES

The specification of traditional GPMLs is not the essential purpose of multi-level languages. This is the case, too, for promoting teaching of traditional GPMLs. Therefore, one could argue that these problems are of minor relevance. However, as we shall see, similar problems occur with the design of DSMLs, which for us is one of the most important use cases of multi-level modeling.

3.1 Multi-level DSMLs

The design and implementation of DSMLs within traditional MOF-like language architectures leads to serious and frustrating problems [5, 11]. It is not possible to express all knowledge one has about a domain with the metamodel that serves the specification of a DSML. For example, a DSML to specify product types is restricted to properties of types and cannot express properties that characterize particular product instances such as a serial number. Also, the specification of a DSML always has to start from scratch, usually with the basic concepts provided by a general-purpose metamodel. That leads to unnecessary effort and jeopardizes the integrity of a language specification. In addition, as already mentioned above, the need to define concepts required for a DSML anew, even though they already exist in the meta-modeling language, is annoying.

Therefore, multi-level modeling has made our lives as language designers clearly easier. Since the first version of the FMML^x was implemented with the XModeler we have developed various DSMLs which offered clear and substantial benefits over previous DSMLs that were designed with a slightly extended MOF architecture. The design of a DSML did not have to start from scratch with basic



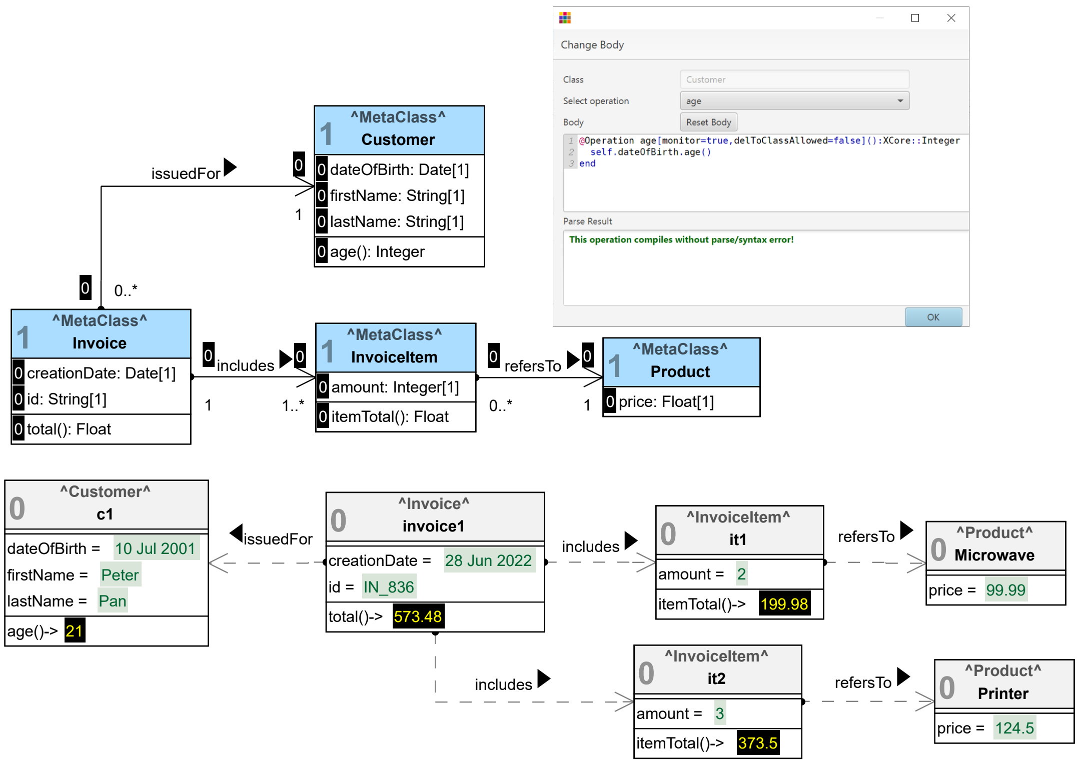
```
self.upperLimit implies self.maxCard > 0
and self.maxCard >= self.minCard
```

Figure 2: ERM meta model and example data model

concepts such as “Class” and “Attribute”. Instead, a DSML could be specified with a more general DSML, similar to the evolution of technical languages. It was possible to specify features that applied to instances on L0 with metamodels on L2 or above. We could instantiate and execute models at any level within one diagram editor. As a consequence, applications could be integrated with the DSMLs and models they were constructed with at runtime. And of course, we were relieved from the burden to specify certain features such as associations or attributes again and again.

Our enthusiasm for these advantages was so great that we initially ignored some drawbacks – also because they do not pose a significant problem for experienced users. However, they can hardly be ignored because they are suited to compromise the idea of a DSML. A DSML should support its users with the creation of models that are in line with a corresponding domain language. In particular, it should prevent users to a large degree from creating nonsensical models. Unfortunately, this is not always the case for DSMLs developed within a multi-level language architecture, which is due to the ambivalent effects of ubiquitous language concepts. While they are extremely beneficial for the specification of DSMLs for which they are required, their implicit inclusion in other DSMLs creates a threat to the integrity of these languages and the models created with them.

The example in Fig. 4 shows a screenshot of a part of a multi-level model of the domain IT management. Different from traditional approaches to DSML design, a DSML does not have to be designed from scratch. Instead, a more general DSML can be used for this purpose, e.g. a particular printer model is specified with a general concept of printer at a higher level. It is also possible to express knowledge about certain characteristics of objects at L0, e.g. the address of a particular server or the serial number of a peripheral device. Thus, the DSML provides support to efficiently design more specific models. However, as the diagram in Fig. 4 illustrates, the language architecture also allows the creation of models that obviously deviate from the domain knowledge represented by the DSML, but that would be still be valid, because they represent correct instantiations of the metamodel of the FMML^X that is available on each level. While the DSML – in this case at level 2 or above – does not define a possible specialization relationship between **ERP1** and **Person**, such a specialization can be added nevertheless, because XCore, and the FMML^X respectively, defines that every class can have a superclass (it allows, in fact, for multiple superclasses). Similarly, associations can be added no matter how bizarre they are, like the association “partOf” between the classes **P1** and **Scanner**.



Class: Customer
Select operation: age
Body

```
@Operation age[monitor=true,delToClassAllowed=false]():XCore::Integer
  self.dateOfBirth.age()
end
```

This operation compiles without parse/syntax error!

Figure 3: Class diagram and corresponding instantiation

3.2 Adaptability and Integrity: A Trade-Off

The above examples of corrupting a multi-level model may seem absurd and of little practical relevance. Nevertheless, it seems inappropriate that a DSML would allow for this kind of corruption. There is, however, a difference between traditional GPMLs and multi-level DSMLs. A GPML like the ERM or the UML are based on a certain language specification that is widely accepted or even standardized. Therefore, the creation of models that do not conform with the language specification should be prevented by a modeling tool. This can be different with a multi-level DSML. It intentionally blurs the distinction between model and modeling language. Every class above level 1 is part of a model that is concretized[3] from higher level classes and, at the same time, represents a language to create models at a lower level. In other words: higher level concepts, that is, languages, are subject to change, too. Modelers can reuse higher level concepts and adapt them to their needs, if required. This kind of adaptability represents a clear advantage of multi-level DSMLs. On the one hand, they benefit from economies of scale, since higher level concepts can be (re-) used in a larger scope. On the other hand, they can still tailor given concepts and add new ones, in order to make a language more efficient for specific use cases.

Unless some kind of reference knowledge representation of a domain exists (which is very unlikely), it is hardly possible to entirely exclude modifications that are beside the point. Therefore, the demand for adaptability implies enabling the definition of additional associations, attributes, specialization relationships, etc. It should also allow for deleting parts of (meta) models that do not fit a specific domain. However, as soon as a multi-level model serves as a reference, e.g., to support the integration of application systems within an organizational information system, or even as a standard, integrity, that is, compliance with a reference model is likely to become a higher priority than adaptability. The higher the level of a language in such a hierarchy, the more invariant it should be. First, it will serve more parties/systems as a reference, second the modification of higher levels is likely to complicate maintenance of a multi-level model.

Against this background, we arrive at the following preliminary interim result. While the proper use of given GPMLs will usually require to prevent modifications of language concepts, multi-level DSMLs demand for a more differentiated approach that aims at an appropriate trade-off between adaptability and integrity.

4 APPROACHES TO COPE WITH UBIQUITOUS LANGUAGE CONCEPTS

As we have seen above, the benefits of ubiquitous language concepts are offset by significant challenges. They require appropriate measures to avoid or at least mitigate serious drawbacks. Before we present three prototypical approaches, we will at first consider a few essential properties of multi-level language architectures as they are enabled by a meta language like the FMML^X .

[3]We adopted the term from [17] to stress the difference to "regular" instantiation.

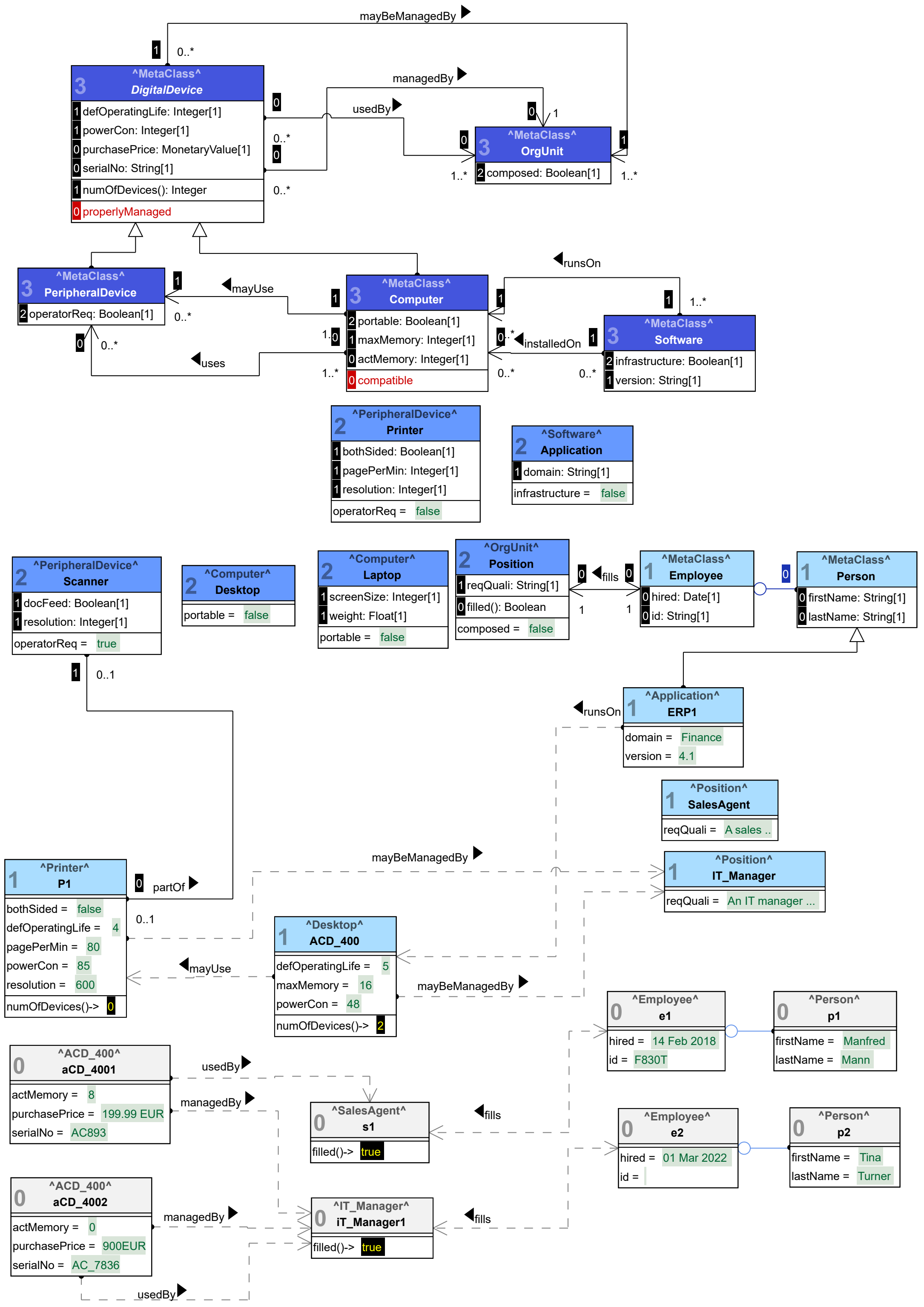


Figure 4: Multi-level model of IT management

4.1 General Considerations

Apart from monotonic extensions, changing the foundational metamodel of a multi-level language hierarchy represents a delicate concern. By the very nature of a multi-level language hierarchy, the concepts of the metamodel are valid at all levels. Removing a certain concept at a lower level would create a serious problem. For example, every class in a multi-level model created with the FMML^x has an attribute **`name`**, which is defined with the class **`NamedElement`** in the metamodel. Apart from the fact that the XModeler^{ML} does not allow this attribute to be removed from a class, doing so would produce a contradiction. If the attribute was removed from a class c, it would obviously be the case that c does not include such an attribute. However, at the same time, it would be valid to conclude that it has this attribute, because of the general statement backed by the metamodel: "Every class has an attribute **`name`**." Accordingly, it would not be feasible to delete other elements of the FMML^x metamodel see [9], such as intrinsic features (defined through the attributes **`isIntrinsic`** and **`instLevel`** with classes that represent properties of classes, e.g., **`Attribute`** or **`CompiledOperation`**).

Deleting unwanted concepts directly in the metamodel is not a good idea either. It would result in destroying the entire tool, because the construction of all objects the tool is built from reflect the structure of the metamodel. The following options avoid the obstacles of corrupting a given multi-level model by using less radical measures. Note that the term "elimination" is not meant as deletion from the system, but rather as making concepts unavailable to users.

4.2 Elimination by Hiding

To avoid the inadequate use of concepts defined with the foundational metamodel, they can be faded out at the level of the user interface. This is the case for the FMML^x diagram editor to prevent the use of multiple inheritance which would be allowed by the metamodel. It defines a view on the metamodel that is sufficient for a certain language. For example, a view provided for the specification of a modeling language that is restricted to static abstraction would fade out operations. Similarly, access to constraints, intrinsic features or specialization could be hidden from the user.

This kind of elimination by hiding is in principle a suitable approach to "tailor" a foundational metamodel to specific requirements. It does not require changes to the language architecture and therefore does not jeopardize its integrity. However, its suitability depends on use case and users. As long as the use case is restricted to editing diagrams only, and users do not request alternative approaches to interact with a model, elimination by hiding is satisfactory. In this case, the view would be defined for the diagram editor only. The effort required to adapt a diagram editor accordingly depends on its architecture, that is, the more possible views were anticipated with the design of a tool and represented by adequate abstractions, the easier will it be to fade out certain concepts.

However, this approach reaches its limits, as soon as a diagram editor is not the only way to access a model. In the FMML^x , it is possible to interact with a model through object browsers or through the console. While it is conceivable to design object browsers that can be adapted to specific views dynamically (which is not the case for the current implementation of the FMML^x), restricting access through the console is hardly an option.

4.3 Elimination by Constraints

Constraints represent a more effective approach. A *language definition* is a multi-level model at level L2 or higher together with the constraints that apply to instances of the language definition, i.e. the *models* written in the language. In general, these constraints should not contradict the foundational meta-model since they serve to restrict the given range of possible instantiations.

The XModeler^{ML} implements a model as a *package* which can contain elements at any level. Since `Package` is a specialization of `Class`, a package can have instances, which themselves are packages. This meta-package relationship leads to an XCore language constraint require that packages only contain instances of elements defined in their meta-package, and which satisfy the constraints defined on the meta-package. The meta-package together with its constraints form a *language definition.*

The following examples show constraints that are attached to the package **`ERM`**, which represents a specification of the ERM in Fig. 2. The first two constraints prevent the use of operations and specialization:

```
context ERM
  @Constraint noSpecialization
     not(contents→exists(c |
       c.isKindOf(Class) implies c.parents→size = 1))
  end
  @Constraint noSpecialization
     not(contents→exists(c |
       c.isKindOf(Class) implies c.operations→size = 0))
  end
```

As defined above, `ERM` is a meta-package whose instances are models (contained in packages). The contents of a model may contain many different types of elements, however we define any *classes* to have exactly one superclass (represented by the implicit superclass **`Class`** in XCore) and no operations.

The final example shows how to prevent the use of intrinsic features:

```
context ERM
  @Constraint noIntrinsics
    not(contents→exists(c |
      c.isKindOf(Class) and c.hasIntrinsics())
  end
```

In addition to preventing the use of certain language features in general, it may also be required to constrain the value range allowed by a concept or to prevent its use with selected classes only (see Subsection 4.6).

4.4 Elimination by *Freezing*

Sometimes we require that elements are not changed, deleted or supplemented by further elements. This is especially the case for languages that need to comply with certain standards. *Freezing* a certain state is a special case of elimination by constraints. Sometimes all classes of a specific level need to be defined as invariant, which may also be required to prevent additional classes. An example of this case in shown in Fig. 2 that represents a minimal metamodel of the ERM. In other cases, classes at different levels may have to be defined as invariant. If a higher level DSML for the domain of IT management should serve as a common reference, more generic classes and their properties should not be deleted. Depending on the intended use cases, it may be possible to make monotonic extensions. In addition to classes at a certain level (in the example in Fig. 4, e.g., all classes at L3), further classes at other levels could be regarded as sufficiently generic to protect them against changes. This could be the case for the class **Person**, even though it is at L1 only.

Freezing can be implemented by XModeler^{ML} in the context of language definitions through the use of multi-level constraints defined at the package level. To understand how this works, consider a model element `e`. The semantics of `e` is defined in terms of the constraints provided by its classifier `m.of()`. If `m` is semantically correct then each `m.of().constraints` must hold for `m`. In a MLM situation, there may also be constraints defined by `m.of().of()` and so on up the classification chain. In order to apply to `m` such multi-level constraints must define the levels over which they span. For example, a constraint at `m.of().of()` may state that it applies only to elements 2 classification levels below, one classification level below or *all* classification levels below.

Freezing can be implemented using multi-level constraints on meta-packages that apply to *all* classification levels below the meta-package. For example, the following constraint applies ERM as a meta-package and requires that all of its classes that are defined at level 2 have frozen instances. This is achieved by the *instances* of classes at level 2 (which themselves are classes at level 1) are limited to having `Object` as a super-class. The effect is that no level 1 class that conforms to the `ERM` language can extend the features defined by the language.

```
context ERM
  @Constraint FreezeClasses
    contents→forall(c |
      c.level = 2 implies
        c.allInstances→forall(sc |
          sc.parents = {Object}))
  end
```

Once we allow constraints that range over multiple levels of classification (via the `allInstances` property) we can state conditions that range both *up* and *down* the type hierarchy. Another example permits new attributes to be added:

```
context ERM
  @Constraint FreezeClassesButAllowAttributes
    contents→forall(c |
      c.level = 2 implies
        c.allInstances→forall(sc |
          sc.parents.getAllOperations()→isEmpty))
  end
```

4.5 Further Issues

It will usually be possible to compromise a multi-level model by introducing formally valid, but nonsensical features like a bizarre specialization, or absurd attributes and associations. While such modifications cannot entirely be prevented, because that would not only require complete knowledge about the present and future of a domain, but also a standardized terminology, it is conceivable to at least exclude modifications that are known as being inadequate. This should often be possible at the highest level of abstraction. With respect to the model in Fig. 4, one could define that no class within the concretization subtree[4] of, e.g., the class **Computer** can be specialized from a class within the concretization subtree of the class **Software**, et vice versa; or that none of the classes at level 1 within the concretization tree of classes at L3 can be specialized from **Person**, or the other way around. This knowledge could then be represented by corresponding constraints.

While the ubiquitous availability of the concepts defined with the foundational metamodel will often be a relief, those concepts will sometimes not be sufficient. There are, e.g., modeling language where we know that there will be a certain kind of association, e.g., at L1, but we do not know the specific multiplicities. While we can express this knowledge at the highest level where it applies through intrinsinc associations, the foundational metamodel of the FMML^{x} does not allow deferring the definition of multiplicities to a lower level. Adaptations like this one cannot be addressed by the prototypical approaches presented above.

4.6 Implications on Design Method and Tools

Multi-level DSMLs are often used by domain experts who lack the technical knowledge to express constraints. This will often be the case, too, for students that attend introductory level courses on conceptual modeling. In order to enable these users to "tailor" a foundational metamodel to the language of their choice, specific support is required.

At first, methods for designing modeling languages or multi-level models in particular, e.g., [10, 16], would have to be adapted to include aspects of "design by elimination". For that purpose, an additional phase like "language configuration" could be added. It would be aimed at selecting those concepts of a given foundational metamodel that are required to satisfy the requirements of the model to be designed. It should be supported by guidelines, a given list of concepts, and specific tool support (see below). In addition, the phase that is dedicated to (meta) model design would have to be modified. This is especially relevant for the specification of traditional GPML. As the example of the ERM in Fig. 5 indicates, metamodels that reflect the idea of "design by elimination" are not only different from traditional metamodels, they even seem to be insufficient. Therefore, it is important to explain the idea that every specific (meta) model is virtually extended with the concepts defined with the foundational metamodel. In addition, the method should guide users with the identification of further constraints. To this end, they could be presented with questions like "is it conceivable that classes of the kind x are specialized from classes of the kind y?", or "are there any constraints that apply to general concepts like attributes or associations?". The latter could, e.g., relate to multiplicities (see metamodel of ERM below) or to possible types of attributes. Also, such a method would demand for checking whether parts of a (meta) model should be protected against various forms of change, such as adding further elements, or modifying/deleting existing ones.

Second, there is need for tool support. Users are presented with a list of language concepts provided by the foundational metamodel. Then they simply select the ones that are required (or that should be excluded) for the purpose they have in mind. In the simplest case,

[4]We use the this term to denote the set of all direct and indirect concretizations (see [11, p. 454]) of a class.

these would lead to the activation of corresponding constraints. More sophisticated tools would feature a dynamic adaptation of the GUI. Third, and this is especially relevant for teaching modeling languages, the resulting metamodel should be presented to users in a comprehensible form. Subsequently, a minimal metamodel is designed, which is a model consisting only of those concepts required in addition to the ubiquitous concepts of the foundational metamodel. The user would then have to decide whether the metamodel is complete and must not be changed.

Fig. 5 shows a metamodel of the ERM that was created by "design by elimination". It consists of two classes at L2 only. Attributes and associations are implicitly defined with the foundational metamodel. The proper use of these concepts requires additional constraints, e.g. that associations are allowed only between instances of **EntityType** and **RelationshipType**:

```
context ERM
  @Constraint EntityRelationshipConstraint
    contents→forall(a |
      a.isKindOf(Association) implies
         {a.end1.class,a.end2.class} = {EntityType,RelationshipType
      })
  end
```

or that instances of **RelationshipType** have to be associated with two and exactly two instances of **EntityType** (this is the case for the version of the ERM we use for teaching purposes).

```
context ERM
  @Constraint RelationshipType2EntityType
    contents→forall(r |
      r.isKindOf(RelationshipType)
        contents→exists(a1,a2 |
          a1.isKindOf(Association) and
          a2.isKindOf(Association) and
          {a.end1,a.end2}→subsetOf({a}) and
          {a.end1,a.end2}.class→subsetOf({EntityType}) and
          contents→forall(a |
            a.isKindOf(Association) and
            a.end1 = r or a.end2 = r
            implies a = a1 or a = a2))
  end
```

Furthermore, multiplicities of associations assigned to instances of **EntityType** have to be 1,1. The corresponding implementation is suited to show the interplay of metamodel, model and model instances within one diagram. Nevertheless, it is hardly satisfactory for teaching purposes. Therefore, the reduced metamodel needs to be supplemented by those parts of the foundational metamodel selected for a specific language. Fig. 5 illustrates the design process. After a user selected the required concepts, a corresponding view on the FMML^X metamodel is generated. Note that the view on the metamodel is simplified. The metaclass **Class** is level contingent, indicated by a "C". Also, the diagram is not integrated with the diagram editor. Instead, the view on the metamodel is generated as SVG and shown together with the minimal metamodel as demonstrated in Fig. 5.

4.7 Related Work

De Lara et al. [7, 14] developed approaches to customize metamodels. To that end, they propose two "control mechanisms". "Modifiers" serve marking elements of a language that must not be extended. That corresponds to the idea of "freezing" language concepts. Constraints are to "ensure a certain extensibility degree" [7, p. 43], which means to control the use of elements of the foundational metamodel at a specific level, such as allowing or permitting adding further attributes. This mechanism corresponds widely to "elimination by constraints". While our approach is clearly similar to [7], it differs from that in a number of ways. It also accounts for "elimination by hiding" (which is a minor difference only). Different from MetaDepth, the XModeler^{ML} features a diagram editor for the graphical representation of models. Our work has a specific focus on using multi-level modeling for teaching modeling languages that were originally specified within MOF-like architectures. To that end, we propose how to present metamodels of traditional modeling languages that are based on an orthogonal classification architecture.

The work of Atkinson et al. to support changes ("emendations") of multi-level models [2] does not share the specific motivation of our work, but includes mechanisms that could be used for our purpose, too. For example, "value mutability" that is offered by Melanie [1] could be used to change multiplicities defined at a higher level, which can be helpful for customizing a language within a multi-level hierarchy. It seems, however, that it does not allow for "eliminating" parts of the foundational metamodel.

5 CONCLUSIONS AND FUTURE RESEARCH

The potential threat of concepts implicitly provided by a foundational metamodel is probably not a serious problem for experienced language designers and modelers who are familiar with multi-level language architectures. However, the use of multi-level modeling is not limited to advanced modelers or language engineers and can be beneficial for a wide range of users. This includes the use of specific DSMLs and the use of models written in these languages. Corresponding users do not have to be familiar with foundational aspects, but should nevertheless be protected against unintentionally damaging a model. As we have shown, multi-level language hierarchies also provide a promising foundation for teaching the use of traditional GPMLs or the design and implementation of DSMLs. We believe that this is an important use case to promote the adoption of multi-level modeling.

In order to use the power of multi-level modeling for demonstrating various levels of abstraction, we presented a preliminary approach to superimpose a generated view on a foundational metamodel with a minimal metamodel of a specific language. The "configuration" of a metamodel (which is actually not changed, only its use is restricted) should work for most modeling languages, including DSMLs. Currently, additional constraints that represent specific aspects of a language, have to be specified manually. We plan to study more languages in order to find patterns that support automatic constraint generation of certain kinds.

The current version of the XModeler^{ML} does not include an editor for graphical notations. An earlier version can no longer be used as a result of a major overhaul of the diagram editor. As soon as the new version of the notation editor is available, we will extend the use of multi-level modeling for teaching purposes, both on the graduate and undergraduate level. Building on initial studies [15], we plan to conduct further empirical investigations on the effect methods and tools have on teaching and learning conceptual modeling.

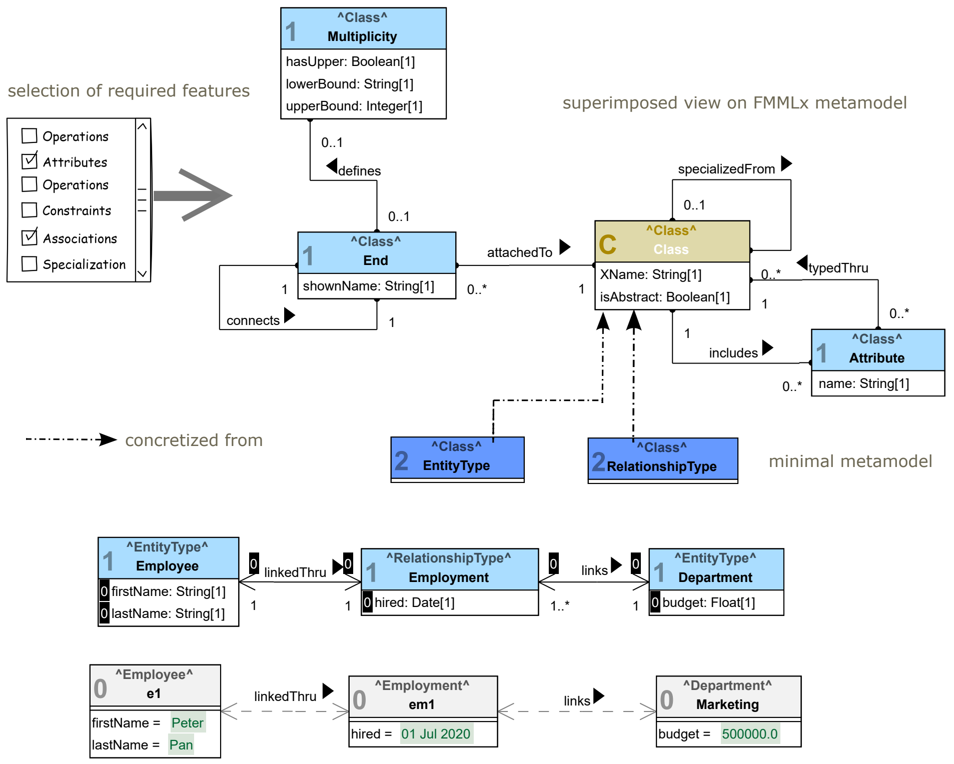


Figure 5: Reduced metamodel of ERM

REFERENCES

[1] Colin Atkinson and Ralph Gerbig. 2016. Flexible Deep Modeling with Melanee. In *GI-Edition / Proceedings*, Vol. 255. Köllen, Bonn, 117–121.

[2] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. 2012. On-the-Fly Emendation of Multi-Level Models. In *Modelling Foundations and Applications (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 194–209. https://doi.org/10.1007/978-3-642-31491-9_16

[3] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. 2009. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering* 35, 6 (Nov. 2009), 742–755. https://doi.org/10.1109/TSE.2009.31

[4] Colin Atkinson and Thomas Kühne. 2003. Model-Driven Development: A Metamodeling Foundation. *IEEE Software* 20, 5 (Sept. 2003), 36–41. https://doi.org/10.1109/MS.2003.1231149

[5] Colin Atkinson and Thomas Kühne. 2008. Reducing Accidental Complexity in Domain Models. *Software & Systems Modeling* 7, 3 (July 2008), 345–359. https://doi.org/10.1007/s10270-007-0061-0

[6] Tony Clark, Paul Sammut, and James Willans. 2008. *Applied Metamodelling: A Foundation for Language Driven Development* (2 ed.). Ceteva. http://www.eis.mdx.ac.uk/staffpages/tonyclark/Papers/Applied%20Metamodelling%20%28Second%20Edition%29.pdf

[7] Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. 2014. Extending Deep Meta-Modelling for Practical Model-Driven Engineering. *Comput. J.* 57, 1 (Jan. 2014), 36–58. https://doi.org/10.1093/comjnl/bxs144

[8] Ulrich Frank. 2014. Multilevel Modeling: Toward a New Paradigm of Conceptual Modeling and Information Systems Design. *Business & Information Systems Engineering* 6, 6 (Dec. 2014), 319–337. https://doi.org/10.1007/s12599-014-0350-4

[9] 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.

[10] Ulrich Frank. 2021. 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.

[11] Ulrich Frank. 2022. Multi-level modeling: cornerstones of a rationale. *Software and Systems Modeling* Online First (2022). https://doi.org/10.1007/s10270-021-00955-1

[12] Ulrich Frank, Luca L. Mattei, Tony Clark, and Daniel Töpel. 2022. Beyond Low Code Platforms: The XModeler^{ML} - an Integrated Multi-Level Modeling and Execution Environment. In *Proceedings of the Modellierung 2022 Satellite Events*, Judith Michael, Jérôme Pfeiffer, and Andreas Wortmann (Eds.). GI, Bonn, 235–244. https://doi.org/10.18420/MODELLIERUNG2022WS-032

[13] Ulrich Frank and Daniel Töpel. 2020. Contingent Level Classes: Motivation, Conceptualization, Modeling Guidelines, and Implications for Model Management. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, Esther Guerra and Ludovico Iovino (Eds.). New York, NY, USA, 622–631.

[14] Santiago Jacome and Juan De Lara. 2018. Controlling Meta-Model Extensibility in Model-Driven Engineering. *IEEE Access* 6 (2018), 19923–19939. https://doi.org/10.1109/ACCESS.2018.2821111

[15] Sybren De Kinderen, Monika Kaczmarek-Hes, and Kristina Rosenthal. 2021. Towards an Empirical Perspective on Multi-Level Modeling and a Comparison with Conventional Meta Modeling. In *24th International Conference on Model-Driven Engineering Languages and Systems*. IEEE, Piscataway, NJ, 531–535. https://doi.org/10.1109/MODELS-C53483.2021.00082

[16] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and How to Use Multilevel Modelling. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 12:1–12:46. https://doi.org/10.1145/2685615

[17] Bernd Neumayr, Katharina Grün, and Michael Schrefl. 2009. Multi-Level Domain Modeling with M-Objects and M-Relationships. In *Proceedings of the 6th Asia-Pacific Conference on Conceptual Modeling (APCCM)*, Sebastian Link and Markus Kirchberg (Eds.). Australian Computer Society, Wellington, 107–116.