# Towards Flexible Creation of Multi-Level Models: Bottom-Up Change Support in the Modeling and Programming Environment XModeler

Daniel Töpel
University of Duisburg-Essen
Essen, Germany
daniel.toepel@uni-due.de

Monika Kaczmarek-Heß
University of Duisburg-Essen
Essen, Germany
monika.kaczmarek-hess@uni-due.de

## ABSTRACT

A process of a multi-level model creation follows typically the top-down approach, i.e., it requires first defining concepts and relations on the highest classification levels, which only then can be used to create concepts on the lower ones. Empirical insights into the process of multi-level model creation suggest however, that this strategy may be counter-intuitive and challenging, especially for non-experts. This paper addresses this problem by focusing on the idea of flexible multi-level model creation, understood as an intertwined application of top-down and bottom-up strategies. As a first step towards realizing this vision for multi-level models in general, and those created with the XModeler and Flexible Meta-Modeling and Execution Language (FMML$^x$) in particular, in this paper, we select a set of relevant multi-level refactoring patterns, adapt them to our approach, and implement them in the supporting tool. We illustrate the flexible creation process using an exemplary scenario.

## CCS CONCEPTS

• **Computing methodologies → Modeling methodologies**; • **Software and its engineering → Domain specific languages**.

## KEYWORDS

multi-level modeling, flexible modeling process, bottom-up modeling, multi-level refactoring patterns, Flexible Meta-Modeling and Execution Language FMML$^x$, XModeler

**ACM Reference Format:**
Daniel Töpel and Monika Kaczmarek-Heß. 2022. Towards Flexible Creation of Multi-Level Models: Bottom-Up Change Support in the Modeling and Programming Environment XModeler. 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/10.1145/3550356.3561553

## 1 INTRODUCTION

The term multi-level modeling covers any modeling approach that aims to provide systematic support for representing multiple classification levels within a single body of model content [4]. Application of multi-level modeling comes with numerous potential benefits [5, 7, 12, 16], which have inspired and given rise to a few multi-level modeling approaches, for instance, potency-based multi-level modeling [5], multi-level objects and relations [28], MultEcore [26, 27], DeepTelos [22], MetaDepth [10], and the Flexible Meta-Modeling and Execution Language (FMML$^x$) [14]. Although each approach has (on purpose) a different focus, and thus, differences exist when it comes to, among others, modeling discipline; a typical process of a multi-level model construction follows a typical process of meta-model construction [9, 16]. It means that the construction of a multi-level model requires first defining concepts and relations on the highest classification level, which only then can be used to create concepts on lower classification levels. This strategy may be however counter-intuitive and/or difficult for non multi-level modeling experts or domain-experts, who may prefer to define first concepts on lower-classification levels, and only later abstract those into meta concepts.

This observation has been initially tested from an empirical angle in [9], where the creation of multi-level models using the FMML$^x$ and its supporting modeling and programming environment, the XModeler [17], has been investigated. Within the study, data on subjects' modeling processes has been collected to identify and analyze modeling difficulties these subjects face, by using, among others, the concept of a cognitive breakdown [30]. One of the insights gained from this study is that although the application of the tool, in that case the XModeler, forced the subjects to design a multi-level model starting from the highest classification level, the participants indicated the need to allow for bottom-up design of a multi-level model as well. This view on the creation process of a multi-level model is indeed in line with the observation that objects, or instances, are something familiar to domain experts, in opposition to classes [25].

Taking the above into account, we postulate an idea of flexible multi-level model creation, encompassing not only supporting the top-down strategy, but also supporting a bottom-up creation process[1]. Please note that supporting the bottom-up process might require applying refactoring of multi-level models [11], and as such

[1]Please note that we use the term 'flexible modeling' in different way than [27]. There the term flexible multi-level modeling refers to the underlying framework, which is to, among others, support flexible hierarchies allowing to skip levels or support flexible model transformations. In our case, we focus on the process of model creation, and the possibility to follow different modeling strategies at will.

may come with numerous challenges, such as, among others, dealing with 'class migration', merging properties of already defined concepts, or changing the 'owner' of a property definition. In addition, each change operation may trigger a need to propagate the changes through the entire multi-level model, both horizontally and vertically, and must not result in a multi-level model becoming inconsistent.

The main aim of this paper is to provide a basic support for bottom-up modeling using FMML$^X$ and the XModeler, understood as a first step towards supporting a fully-fledged flexible creation of multi-level models. To this aim, after sketching the targeted process of model creation and reaching awareness of the required and missing change support in selected multi-level modeling tools, we adapt and extend the selected refactoring patterns, as proposed in [11], to specifics of FMML$^X$, and implement them in the XModeler tool.

We follow the design science research approach, particularly the steps of the engineering cycle as detailed in [34]. Therefore, after reaching the problem awareness, we identify requirements towards a tool supporting the bottom-up modeling strategy. Based on the requirements, we design and implement a set of required change operations in the selected tool. Then, we illustrate the implemented change operations using an exemplary scenario. Thus, in this paper we report on the coverage of one full engineering cycle [34].

The paper is structured as follows. After discussing the process of multi-level models creation, as well as selected strategies and approaches to change support (Section 2), we present the idea of flexible modeling (Section 3). There we also point to the extent to which bottom-up modeling is supported by selected multi-level modeling tools, and identify a set of change operations that should be offered to support the bottom-up creation process. Next, in Section 4, we describe how relevant refactoring patterns can be adapted for the multi-level FMML$^X$ models. Finally, we illustrate the implemented support for bottom-up modeling using a driving scenario in Section 5. The paper concludes with final remarks and an outlook on future work.

## 2 BACKGROUND

As mentioned in the introduction, different multi-level modeling approaches have been proposed, sharing a set of common ideas, cf. [2, 14, 16], such as: (1) support for arbitrary-depth classification hierarchies, (2) relaxing the type/instance dichotomy, and (3) offering deferred instantiation. However, each approach has (on purpose) a different focus. As a result, the approaches differ when it comes to, among others, (1) the level of modeling discipline, see, e.g., [2], (2) the way in which deferred instantiation is designed and implemented, and (3) additional mechanisms and software tool support, see, e.g., [19, 20, 24, 27]. The existing multi-level modeling approaches also differ regarding how the multi-level models are created, and to what extent performing subsequent changes to the already created models is supported by the corresponding modeling tools.

Atkinson et al. [3] elaborate on two possible modes in which models are developed, namely "constructive" and "exploratory". In constructive modeling, sharing similarities with the top-down modeling, the role of classes (or elements on higher classification levels) is to "serve as templates from which populations of instances can be generated at a future point in time" [3, p. 2], and thus, types come before instances. In turn, in exploratory modeling, sharing similarities with the bottom-up modeling, the role of classes (or elements on higher classification levels) is to "capture classification information wrapped up in an already existing population on instances" [3, p. 2]. So it also follows that "the types are usually identified from, and therefore after, the instances they classify" [3, p. 4].

As already signalized in the introduction, a typical process of a multi-level model construction follows the "constructive" approach, as also done in a typical process of meta-model construction, cf. [25]. Although first guidelines and heuristics regarding the process of multi-level creation may be found, e.g., [16], this strategy may be however counter-intuitive and/or difficult for non multi-level modeling experts or domain-experts [9], who may prefer to first define concepts on lower-classification levels, and only later abstract those into meta concepts, i.e., follow the exploratory approach. Following such an approach however, may require dealing with introducing (and propagating) changes within multi-level models.

A few initiatives exist already, focusing on managing changes in the multi-level modeling environment. The existing approaches, on the one hand, acknowledge the existing extensive efforts focusing on refactoring for programming languages, i.e., restructuring a piece of code while keeping its external behaviour, e.g., [13, 29], model refactorings in meta modeling (in the context of two-level modeling), e.g., [21], as well as existing initiatives to support a bottom-up meta model construction, e.g., meta-model construction by example as proposed by [25]. On the other hand, they stress additional challenges connected with dealing with changes in the context of multi-level modeling, namely since the number of classification levels is unlimited, a change to a model element in one classification level may have an impact on a large number of elements over an unlimited number of lower and higher classification levels [1]. In addition, it is important to stress that whereas the change propagation in some cases may be performed automatically [1, 11, 32], quite often there is a need to obtain further input from modelers about the intended effects of changes. Indeed, the introduction of changes/refactoring of multi-level models in most cases needs to be guided by the modeler [11].

And so, dealing with changes in the multi-level modeling environment has been, e.g., studied by [1], where the authors explore the use of emendation[2] services in Melanee to repair a multi-level model upon changing some of its elements, e.g., modifying the potency of a clabject or adding an attribute. To the best of our knowledge this is the first work dedicated to ensuring that whenever changes are made in a multi-level model, in this case within ontological levels [1], the collection of data across all levels remains consistent.

Next, in the context of multi-level modeling in general, and FMML$^X$ in particular, Töpel and Benner [32] identify a set of elementary change operations that should be supported in the context of maintenance of multi-level models. The focus is assigned to a

[2]Please note that Atkinson et al. on purpose use the term emendation and not refactoring pointing to the semantic difference between the terms [1, p. 195].

minimal model and a set of elementary changes only, and thus, the topic of complex changes (or refactoring) is not tackled.

To the best of our knowledge the most comprehensive study of change support in the multi-level environment, is the work of de Lara and Guerra [11], where the authors focus on refactoring of multi-level models that may involve a sequence of modifications, and consider their purpose of use, pre-conditions and variants. In their work the authors identify seventeen refactoring patterns that apply in case of multi-level modeling and implement selected patterns in the MetaDepth environment. Those patterns are as follows: (1) collapse clabject with instances/clabject with type/inheritance hierarchy, (2) set/unset/create/extract clabject type, (3) set/unset/create feature type, (4) pull up/push down/split clabject, (5) pull up/push down feature, (6) replace reference by instantiation, (7) stratify potency. Please note that those patterns are to help designers rearrange elements both across different levels, as well as within affected levels; and account for the possible side effects. Although proposed in the context of potency-based languages, the refactoring patterns, on the conceptual level, should be also applicable to other tools/approaches supporting multi-level modeling. Our work addresses this problem by adapting and extending the selected refactoring patterns, among others, the composite pattern 'extract clabject type', to our approach.

## 3 TOWARDS FLEXIBLE MODELING: RATIONALE AND REQUIRED CHANGE SUPPORT

As discussed in the the previous section, two main strategies of (multi-level) models construction exist: top-down (constructive) and bottom-up (exploratory) modeling. Using top-down modeling, the modeler starts for each hierarchy of concepts from the highest level of abstraction, defining concepts on the highest classification levels first, and then concretizing these concepts down towards lower classification levels. Bottom-up modeling by contrast means that the modeler starts on a relatively low level of abstraction, for instance on level 1. Investigating commonalities between the concepts on lower levels, they will systematically abstract broader concepts on higher levels. Likewise, where properties are shared by concepts which have been abstracted over, the modeler will pull up the definition of those properties to higher levels.

We argue that these two approaches in no way exclude each other. In our on-going work, see [9], we have been studying the creation process of multi-level models. The analysis of think-aloud protocols indicates clearly that constructive and exploratory modes intertwine in the initial phase of the multi-level model creation, and that designers are unlikely to follow a strict order, where they would finish one level of classification before continuing with the next one. Indeed, we have observed that most of the users start with the exploratory approach, which gives them some ideas regarding the potential higher classification levels. Thus, a modeler who creates a concept on a high level will already have done some bottom-up abstraction, either in their mind or during a previous iteration of modeling. Then, they switch to the constructive mode. When some doubts regarding, e.g., the level of classification at which some information should be accounted for occur, they switch again to the exploratory mode and investigate the objects they are more familiar with, in order to find an answer.

In addition to that, both the conducted experiments as well as our own practical experiences with designing multi-level models, e.g., [7, 8, 23], clearly indicate that a multi-level model will usually not be right the first time, and a modeler will likely change their mind because of various reasons. For instance, a model may change as the designers gain experience, or when additional scenarios and/or additional perspectives have been identified that should be supported by the model. Therefore, there is a clear need to support introducing changes in multi-level models with the aim to enable not only a top-down but also a bottom-up modeling process.

Based on the analysis of relevant literature as well as mentioned experiments, we argue that to enable the bottom-up approach, the following three requirements constitute a minimal set of operations that should be supported: (1) Providing a support for changing a classifier of a selected class/set of classes; (2) Providing a support for changing an owner of a property (moving the definition of a property to a higher classification level, pull-up); and (3) Providing support for applying the change operations 1 and 2 in tandem, and propagating changes vertically and horizontally, to ensure the model consistency. Thus, in the light of those requirements, there is a need to support at least the following refactoring patterns [11]: the composite refactoring extract clabject type (involving create clabject type, create feature type, set feature type and set clabject type), as well as pull up feature, see [11, p. 10].

Taking the above requirements into account, the question appears to what extent the bottom-up creation process is supported by multi-level modeling tools. Although a few multi-level modeling tools exist, let us focus on two selected ones supporting two different families of approaches, i.e., Melanee and XModeler.

Melanee is a multi-level graphical modeling environment supporting the potency-based multi-level modeling [5]. In deep modeling the so-called ontological classification relationships are governed by potency, and the modeling levels are defined according to the principles of strict modeling. In Melanee, deep models are represented by two languages: (1) the Level-agnostic Modeling Language (LML), which contains constructs such as Entities, Connections and Generalizations, and (2) a multi-level variant of OCL [18]. As deep instantiation is governed by strict meta modeling [2], thus, by default modelers are asked to follow the top-down modeling strategy. Nevertheless, thanks to the emendation service already mentioned [1], some basic change operations within and across levels are possible. However, although Melanee offers support in updating and propagating changes throughout the created model, an explicit support for bottom-up modeling is missing, e.g., there is no functionality supporting users in abstracting a set of concepts into a new supertype or ontological type.

XModeler is a meta-modeling tool supporting integrated modeling and programming [6, 14]. Models can be edited graphically or using the command line interface. XModeler uses a dedicated language called XOCL, based on a kernel model called XCore, on which all other features of the XModeler are built on. As the architecture of XModeler keeps code and model within the same framework, any changes in the model go into effect immediately without generating a new version of code. That means for example, that when a new class is added to the model, its constructor becomes available and can be invoked to create instances. These instances themselves can then be referenced, e.g., in the body of operations. Operations in turn can be compiled and invoked at runtime. While the tool supports multi-level modeling with FMML$^X$, only a specific order of modeling is currently supported, i.e., top-down, with a limited set of change operations that may be performed, such as, e.g., changing the already assigned level of intrinsicness to properties, raising the level of classification of a meta class, see the elementary change operations defined in [32]. Nevertheless, such operations as changing the meta class classifier, abstracting a set of classes into one common concept, or moving a definition of a property to a higher classification level, are not offered to users.

Similarly to Melanee and XModeler, also other existing approaches and their supporting tools, see e.g., [20], support mostly the top-down modeling strategy, and offer a set of rather elementary change operations. Indeed, the automated support for bottom-up modeling, especially abstracting concepts, is mostly missing.

## 4 BOTTOM-UP MODELING WITH FMML$^X$ AND XMODELER

In the following we focus on the implementation of the selected change operations supporting the bottom-up modeling process in the context of FMML$^X$ and the XModeler[3].

### 4.1 Main Design Decisions and Specific Characteristics of FMML$^x$ and XModeler

As already mentioned, given that the existing multi-level modeling approaches quite significantly differ in their details, the refactorings in [11] proposed in the context of potency-based languages, cannot be used in the FMML$^X$ approach without being adapted. The main concept in the FMML$^X$, please see [15] for details, is a `MetaClass`, which contains the additional features for multi-level modelling, such as `level`, as well as the altered machinery, i.e., the 'Meta-Object-Protocol'. Each top-level concept of a model is an instance of `MetaClass`. Additionally, it also inherits from `MetaClass`. This allows these concepts to be in turn instantiable and can be repeated down to the desired level. Furthermore, properties in FMML$^X$ always come in pairs of a property definition and a property value, i.e., attributes and slots or associations and links. A property definition has an intrinsicness level assigned pointing to the exact level of classification (so called intrinsicness), where the property value will be assigned/known. Also FMML$^X$ includes operations, which can be defined and can be invoked. Additionally, operations can be overridden, a feature not shared with attributes and associations, and therefore not taken into account in [11]. Multiple inheritance is allowed in FMML$^X$ to a certain extent, requiring additional consistency checks.

As already mentioned in the previous section, the XModeler is a tool supporting integrated modeling and programming, and the code and model are kept within the same framework. Thus, any changes in the model go into effect immediately without generating a new version of code. In this context please note that supporting a bottom-up modeling, especially, abstracting the already defined concepts into higher-level ones, creates a problem which has been widely deemed violating a main principle, namely that an instance would exist before their class has been created. One would either need to adapt the existing instance to its new class, which might even be impossible in a strictly typed language, or a new instance has to be created, its properties transferred, then all references to old instance be diverted to the new one, before the old one finally needs to be disposed of, each step in its own way prone to mistakes from oversight. While it is clear that class migration cannot be avoided, it is nevertheless necessary that its application should be restricted to a certain extent, basically through additional pre-conditions, in order to avoid complexity, which is not needed in the scope of this work, see also 'C: Restrictions on Class Migration'.

### 4.2 Added Change Operations and Their Implementation

In the following, the required change operations realizing the relevant refactoring patterns, required to support the bottom-up modeling in the XModeler are described: (1) Operations to add a new meta class and to add a new superclass to an element. (2) An operation pulling up a property. (3) An operation used to classify a number of classes and pulling up properties in one go. (4) Additionally, as classifying has a precondition regarding the levels involved, an operation manipulating the levels beforehand.

Each change operation is described considering the pre-condition, post-condition, as well as the operation itself. Please note that when performing user-initiated changes, the core assumption is that the multi-level model has been valid before the change has been initiated, and it follows, that the multi-level model shall also be valid after the change has been executed (propagated vertically and horizontally). The validity of a multi-level model in case of FMML$^X$ and the XModeler means that it has to adhere to the XCore/FMML$^X$ meta model [15], that declares the admissible model elements, properties and relations, as well as a rich set of XMF constraints that models should fulfill. Note here that the requirements towards a FMML$^X$ multi-level model being valid, among others due to not adhering to strict meta modeling rules [14], differ from the ones for potency-based modeling, as defined in [1, p. 199] or [11].

The description of the implemented operations follows.

*4.2.1 Change Operation: classify.* This change operation replaces the default meta class of a given element with a given meta class. It corresponds to 'set clabject type' refactoring pattern, cf. [11]. It is further similar to 'create clabject type' with the only difference that the latter creates that classifying class within a single refactoring. In this approach, if that class is not yet existing, the operation 'create meta-class', which is not within the scope of this paper, cf. [32], must be preformed first. Also it must be taken into account, that it is possible that since that operation was performed, the meta class may have some properties added already.

*Pre:* The given element must not have a non-default meta class yet. Furthermore the level of the new meta class must be exactly one level above the level of the element to be classified. Where the new meta class already has properties, they must not collide with the properties of the given element. Otherwise the user input dialog for pulling up properties, cf. 'D. Options for pulling up properties', must be presented. As inheritance in FMML$^X$ is only allowed between

[3]The XModeler tool is available at https://www.wi-inf.uni-duisburg-essen.de/LE4MM/xmodelerml/#download.

elements which have the same meta class, they must be migrated at the same time.

*Post:* All elements supplied are a direct instance of the supplied meta-class.

*Change Operation Scheme:*

- The elements supplied are checked whether they are on the correct level and are a direct instance of the default meta class.
- The elements supplied are checked whether they are in a subclassing relationship and whether the corresponding super/sub-classes are supplied as well.
- The properties of the supplied classes are checked whether there is any kind of collision.
- A user dialog may be presented if meaningful to resolve those collisions.
- The properties are renamed or removed, in line with the decisions made by the user.
- The property values of the properties to be merged are stored.
- The properties to be merged are removed.
- The class of the supplied elements is changed to the supplied meta class.
- The stored property values are recovered.

*4.2.2 Change Operation: addParent.* This change operation adds a superclass to another class. There seems to be no corresponding refactoring pattern in [11]. However it is similar to 'classify' in the sense that an element gains an additional ancestor with possible conflicting properties. Therefore no schema is given here.

*Pre:* The classes to be connected by inheritance must be direct instances of the same meta class. They must further be on the same level, which is guaranteed unless they are direct instances of the default meta class. Also they must not form an inheritance cycle, i.e., the new super-class must not be a descendant of the new subclass. Where the properties of both elements collide, the user must be prompted to resolve those.

*Post:* The supplied classes form an inheritance relationship. There is no cyclic inheritance.

*4.2.3 Change Operation: pullUpProperty.* This change operation takes one property from a given class and pulls it up to its meta class, superclass, or any other ancestor. It corresponds to 'pull up feature' refactoring pattern, cf. [11]. However, this operation does not require the property to be present in all descendants of the new owner of the property. On the contrary, in such a case the — until now — independently from each other existing properties must be explicitly flagged as sharing an identity.

Once the property is pulled up, the change will apply to all descendants of the new owner of the property. Additionally one property from each of the affected classes can be selected to be merged into this property at the same time. This not only caters for properties with the same name, but also for others, where the name may differ, but they are meant to represent the same subject. If, on the other hand, the names collide, but do not represent the same subject, there is the option to rename that property to avoid a collision. Depending on the type of property and its ability to be overridable, there must be no property in the affected classes with the same name as the given property, unless is has been explicitly selected to be merged and has a compatible type. From each affected class only one property may be merged. Also, renaming properties must adhere to the same rules as for an atomic name change. The property will then be pulled up, causing all the descendants of the new owner, on the respective level, to gain property values where applicable. The property values of the properties to be merged will now be referred through the new property, cf. the scenario in Sect. 5.

*Change Operation Scheme:*

- The selected property is checked, whether it causes a name collision in the target class.
- Where necessary, the user is prompted to keep, merge, rename or drop any property in the descendants of the target class. By default the choice for unaffected properties defaults to 'keep'.
- The properties are renamed or removed when decided to do so by the user.
- The property values of the properties to be merged are stored.
- The properties to be merged are removed.
- The stored property values are recovered.

*4.2.4 Change Operation: classifyAll.* This change classifies a set of elements, which will become direct instances of a newly created meta class. It furthermore pulls up a set of properties, for each of which a set of properties will be merged into it.

Basically this change operation can be combined from *classify* and *pullUpProperty*, as well as the very basic change operation for adding a new element, which is not covered in this work. It corresponds to 'extract clabject type' refactoring pattern, cf. [11] applied together with 'pull-Up features'.

*Pre:* The name for the new meta class must be unique. The elements in the set must not have a non-default meta class yet and must be on the same level. A property to be pulled up must satisfy the (slightly adapted) preconditions from *pullUpProperty*, as respectively do the properties to be merged into those.

*Post:* The new meta class is situated one level higher that the elements in the set, and contains the properties which have been selected to be pulled up. The existing property values in the instances remain unchanged, albeit under a new name, if the property has been merged. All instances which did not have that property before gain a new default property value.

*Change Operation Scheme:*

- A new class is created on a level one higher than the level of the classes to be classified.
- The classes to be classified are migrated to the newly created class.
- The user is prompted to pull up, keep, merge, rename or drop any property in the descendants of the newly created class. By default the choice for unaffected properties is set to 'keep'. However, when a property is chosen to be pulled up, the resolution for conflicting properties defaults to 'unresolved'. Only when no property remains unresolved, the process continues.
- The properties are renamed or removed, in line with the decisions made by the user.

- The property values of the properties to be merged are stored.
- The properties to be merged are removed.
- The properties to be pulled up are pulled up, with the property values remaining in place or if not yet existing are set to the default value.
- The stored property values are recovered, replacing the default value set in the previous step.

*4.2.5 Change Operation: liftUpHierarchy.* This change operation lifts up a subset of a model, namely all classes, which are related to each other through a path of ancestry. The number of levels to be lifted up is arbitrary. There seems to be no corresponding refactoring pattern in [11], which could be explained by the fact that cross-level relationships, which may be the consequence from performing this operation, are prohibited in their and some other approaches.

*Pre:* The set of supplied model elements contains only such elements that all ancestors and descendants of it are in the set as well.

*Post:* The level of all elements is increased by the given number. The instantiation level of properties is increased accordingly, so that the property values keep their owner.

*Change Operation Scheme:*

- The level of all elements is increased by the given number.
- The instantiation level of all properties is increased by the given number.

## 4.3 Restrictions on Class Migration

The already described operations require handling class migration. The strongest restriction would be to allow only elements which are direct instances of `MetaClass` to be migrated. The classes they will be migrated to inherit from `MetaClass` by the rules of FMML$^{X}$. As such any existing property values are those from `MetaClass` or its ancestors and will therefore be equally present in the migrated class. That means all properties required in the meta model, or for the functioning of the tool, remain unchanged.

The element will only gain the user defined property values from the class it is migrated to. The change operation will therefore have to deal with similar problems as those which add properties to classes with existing instances. This type of problem has been already widely discussed [11, 33]. For the scope of this work, we will assume that the property is initialized with a null-value or a default value, leaving it to the modeler to later add the values where necessary.

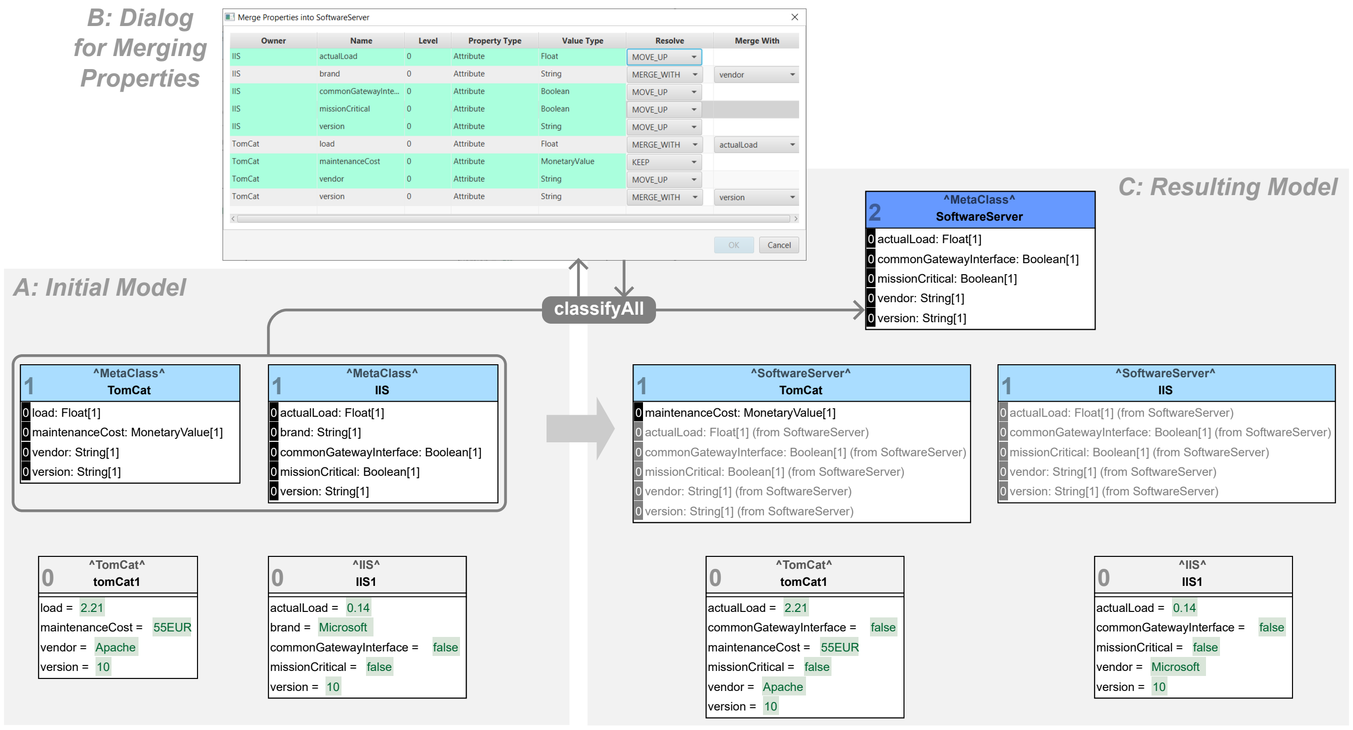
## 4.4 Options for Pulling Up Properties

The support for flexible modeling in general, and for bottom-up model creation in particular, can be only semi-automated, as only a user himself/herself can indicate the intended way the changes should be propagated, cf. also [1]. One of such operations is deciding what should happen with properties of classes being abstracted. Here, a designer is presented with possible options and needs to, for each property within each class, decide what should happen.

*4.4.1 Properties of the Target Class.* For properties already existing in the target class, the only option is to keep the property. The reasoning behind this is that any other change, like, e.g., renaming, can be done separately beforehand without complicating this change operation. The properties are nevertheless shown in the dialog for the convenience of the user, as they may cause conflicts with properties of the same name in the source classes, which are easier to detect as such. In any case, the properties of the target class have priority, and the conflict has to be dealt with in the affected source classes.

*4.4.2 Properties of the Source Classes.* The designer, for each property of each source class, needs to select one of the following options.

- **Keep:** When the property is not to be pulled up, this is the default choice. This option requires the property to be not in conflict with any property in the target class, either already existing or being pulled up during this operation. This option does not change anything.
- **Keep as:** This option is intended to be used for properties not to be pulled up, but being in conflict with a property in the target class. This option changes the property name and then leaves it in place.
- **Pull up:** This option moves the property into the target class. That requires the target class to not have any conflicting properties. When selected, this may render the selected option for other properties invalid. This option takes the same priority as "Keep" for the target class properties. When performed, this option changes the owner of the property, causing some descendants of the target class to gain a property value. The question remains how those new values shall be initialized. However, that issue is the same one, as caused by the change operation for adding a new property to a class, which already has instances, cf. [32].
- **Merge into:** Where a property is to be pulled up, but in conflict with another property, which is to be pulled up from a different source class, there is an option of merging those properties into one. It is not possible to merge a property of a source class with a pre-existing property from the target class, as it would cause ambiguities, namely the question which values to keep. If performable, the merging option copies the values from this property into the newly created value slots of the property to be merged into.
- **Drop:** This option allows a property to be dropped. Basically the separate change operation for removing the property is invoked beforehand.
- **Other:** It is possible to offer further options for convenience. For instance "Pull up" could also have a version were the property is renamed. But the options shown above, together with using other change operations in advance should cover a vast majority of the use-cases.
- **Unresolved:** The dialog also shows this option, which is the default selection initially, intended to push the modeler into actively selecting a meaningful solution. As long as any selection is still unresolved, or any other selection is in conflict with another, the dialog box does not allow the user to proceed.


B: Dialog for Merging Properties
A: Initial Model
C: Resulting Model
classifyAll
^MetaClass^
1 TomCat
0 load: Float[1]
0 maintenanceCost: MonetaryValue[1]
0 vendor: String[1]
0 version: String[1]
^MetaClass^
1 IIS
0 actualLoad: Float[1]
0 brand: String[1]
0 commonGatewayInterface: Boolean[1]
0 missionCritical: Boolean[1]
0 version: String[1]
^TomCat^
0 tomCat1
load = 2.21
maintenanceCost = 55EUR
vendor = Apache
version = 10
^IIS^
0 IIS1
actualLoad = 0.14
brand = Microsoft
commonGatewayInterface = false
missionCritical = false
version = 10
^MetaClass^
2 SoftwareServer
0 actualLoad: Float[1]
0 commonGatewayInterface: Boolean[1]
0 missionCritical: Boolean[1]
0 vendor: String[1]
0 version: String[1]
^SoftwareServer^
1 TomCat
0 maintenanceCost: MonetaryValue[1]
0 actualLoad: Float[1] (from SoftwareServer)
0 commonGatewayInterface: Boolean[1] (from SoftwareServer)
0 missionCritical: Boolean[1] (from SoftwareServer)
0 vendor: String[1] (from SoftwareServer)
0 version: String[1] (from SoftwareServer)
^SoftwareServer^
1 IIS
0 actualLoad: Float[1] (from SoftwareServer)
0 commonGatewayInterface: Boolean[1] (from SoftwareServer)
0 missionCritical: Boolean[1] (from SoftwareServer)
0 vendor: String[1] (from SoftwareServer)
0 version: String[1] (from SoftwareServer)
^TomCat^
0 tomCat1
actualLoad = 2.21
commonGatewayInterface = false
maintenanceCost = 55EUR
missionCritical = false
vendor = Apache
version = 10
^IIS^
0 IIS1
actualLoad = 0.14
commonGatewayInterface = false
missionCritical = false
vendor = Microsoft
version = 10
OK
Cancel

Merge Properties into SoftwareServer

| Owner | Name | Level | Property Type | Value Type | Resolve | Merge With |
|---|---|---|---|---|---|---|
| IIS | actualLoad | 0 | Attribute | Float | MOVE_UP | |
| IIS | brand | 0 | Attribute | String | MERGE_WITH | vendor |
| IIS | commonGatewayInte... | 0 | Attribute | Boolean | MOVE_UP | |
| IIS | missionCritical | 0 | Attribute | Boolean | MOVE_UP | |
| IIS | version | 0 | Attribute | String | MOVE_UP | |
| TomCat | load | 0 | Attribute | Float | MERGE_WITH | actualLoad |
| TomCat | maintenanceCost | 0 | Attribute | MonetaryValue | KEEP | |
| TomCat | vendor | 0 | Attribute | String | MOVE_UP | |
| TomCat | version | 0 | Attribute | String | MERGE_WITH | version |

**Figure 1: Application of 'classifyAll' on two concepts on L1**

*4.4.3 Dealing with Different Kinds of Properties.* The first kind of property, which is generally investigated are attributes and their counterpart, the slots. This is the case here as well, and the treatment of properties presented above therefore apply to those as described. However, there are also other properties, namely operations, constraints and associations, which deviate from that description to a certain extent.

Operations do not have a value counterpart, which needs to be preserved. As such, merging is not applicable. Another difference is the fact, that operations can be overridden, and as such have a different behaviour regarding conflicts. In most cases an element may have an operation matching the signature of an operation in any of its ancestors. One issue however is multiple inheritance. It must be ensured that no element has any ambiguous operations.

Constraints can be treated similar to operations, as their value counterpart — the constraint report — is not persistent. However, in contrast to those, they cannot be overridden.

A further issue affecting operations and constraints is that they contain code, which may refer to other classes and features. When those are altered, the code may no longer be executable. The code is stored in XModeler as an expression tree which may be a starting point for adapting them in a future work. However, as XMF is not typed language, this complicates the matter even further.

Associations do have a counterpart, namely links. Those need to be migrated to other associations, where they are merged. Also the question of conflicts is still subject to discussions, involving the number of identifiers an association has, and in which name-spaces they exist.

## 5 EXEMPLARY SCENARIO

As already mentioned, the change support for bottom-up modeling is guided by the designer, who needs to decide where and how the changes should be performed. It follows that after the user selects a change operation on some element(s) (e.g., classifyAll), first the pre-conditions are being checked, the change operation executed and propagated horizontally and vertically, if necessary, to repair all invalid elements.

Let us consider the following scenario to illustrate the way that the concept of bottom-up modeling, and thus the discussed change operations, are implemented in the XModeler. This scenario illustrates a typical process of a multi-level model creation that emerges based on our empirical investigations.

Let us assume that a model designer wants to create a multi-level model to support analyses and activities in the field of IT infrastructure management. As such, the multi-level model should provide relevant, in the light of the defined goals, characteristics of IT artefacts being part of an enterprise information systems, e.g., servers, hardware, network configuration. The model designer working on a daily-basis with the IT components is at the same time a domain-expert. Following the top-down approach, as initially imposed by the XModeler, the model designer would need to identify the most generic concept, and the highest classification level relevant, in order to start the creation process. Whereas one could be assuming that a suitable candidate for a most generic concept could be, e.g., an 'IT artifact', determining the level of classification at this point of time is problematic.

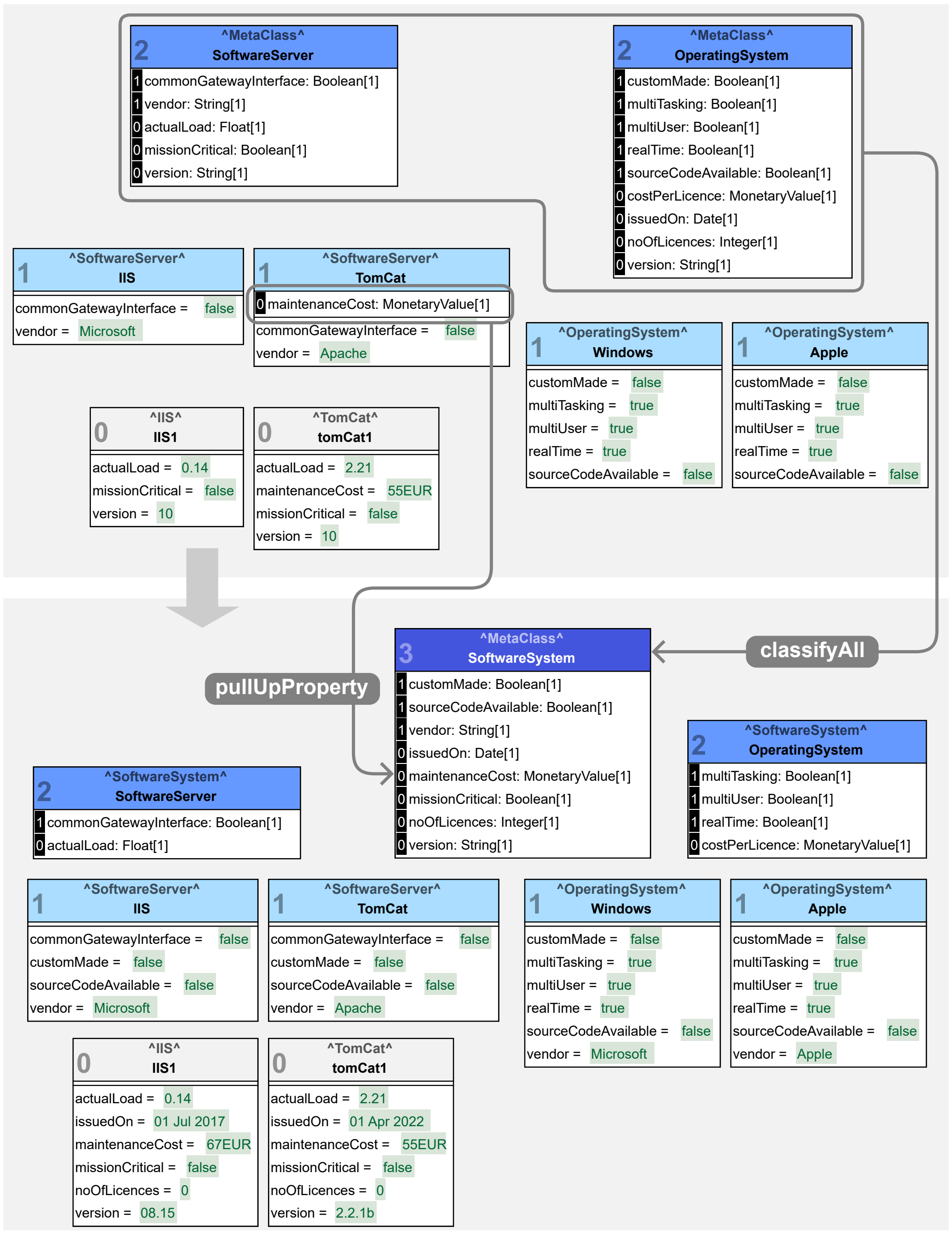


**Figure 2: Application of 'classifyAll' and 'pullUpProperty' in the further modeling phase: to abstract two concepts on L2, and to pull up a property from a concept on L1 to the concept on L3, respectively**

Therefore, the model designer starts exploring the domain, and decides to start modeling by focusing on the actual elements of the infrastructure, e.g., server Apache TomCat version 10 or Microsoft IIS version 10. Considering characteristics of selected types of servers, the designer defines the classes for those concepts on the L1 level, Fig. 1.A. For each class, TomCat and IIS, respectively, the multi-level model designer defines the properties (attributes and operations), and associations, as the user deems relevant.

Now, after instantiating the respective classes on L0 level, Fig. 1.A, the model designer focuses on the fact that actually: (1) both TomCat and IIS are software servers, and thus, share some elements of the description, and that (2) some slot values on L0 (e.g., that TomCat is offered by Apache Organization), actually are known already earlier, and should be thus assigned to the corresponding elements on the L1 level. Therefore, as the multi-level model designer recognizes the potential to abstract the selected concepts into one common meta class (classifier), the designer decides to introduce this change to the model first, before dealing with other issues. Thus, the user marks the classes of interest and selects the *classifyAll* functionality. As the selected classes fulfill the precondition, i.e., they have been defined at the same classification level and are of default meta class type, the change operation is triggered.

In the process, the user is prompted to confirm and adjust, if needed, the selection of concepts, and to provide a name for the to-be created meta class. The new meta class will be situated one level higher than the concepts being abstracted, so in our case on level L2. At this point in time, in the background, the class migration takes place in line with 'classifyAll', see the previous section for details.

In the process, a dialog box is presented to a user, Fig. 1.B, allowing him/her to decide what should happen with properties already defined, i.e., for each property a decision to keep, move_up or merge_with needs to be made, Fig. 1.B. In case some conflict is identified, the property causing the conflict is marked, and the status is set to unresolved. Let us assume that the designer decided, e.g., that whereas attributes defining the average load or kernel space, should stay with their respective owner (option: keep), attributes vendor and brand should be merged into one attribute (under the name vendor), and its definition should be moved to the newly created meta class. The same should also happen with the version (merging and moving up). The attribute mission critical (defined only in IIS), should be moved up. In effect, those properties will be moved to a newly created meta class and the changes introduced are propagated along the hierarchy, in that case, to already existing concepts on L1 and L0. This is done automatically without user's intervention. In that way the multi-level model becomes again consistent, see Fig. 1.C.

Please note that at any point of time the designer may also use the already available operations offered by the XModeler, like, e.g., changing the intrinsicness level, editing the properties, or associations, see also [32], to adjust the model further.

The model designer continues the process and models other types of IT artifact, among others, Operating System (OS) with such types as Windows OS and its versions, Fig. 2 (the upper model). Again, the multi-level model designer notices that both Software Server and Operating System have some commonalities, and decides to abstract them into a common (meta) class Software System. Also in this step, similarly to the already described scenario, a user uses the 'classifyAll' operation in order to obtain support in creating a new meta class one level higher and changing the classifier of the selected concepts. After the designer decided what should happen with the properties from the classes to be abstracted, the corresponding changes are carried out, see Fig. 2, and again propagated vertically (this time from L3 to L0), and horizontally, to all relevant concepts.

At any point, if the designer notices that the decision regarding the properties has been incorrect, or that the level at which a given property has been defined needs to be adjusted, there is a possibility to move the attribute to another classification level. In our case the designer decides to move the attribute 'maintenance cost' defined within the TomCat two levels higher, i.e., to the 'SoftwareSystem', by initiating the 'pull up property' operation, Fig. 2. The relevant change, i.e., changing the owner of the attribute, is propagated down to all descendants of the respective meta class Software System. It follows that also instances of IIS, Windows and Apple have the corresponding slot value, initially empty, and the already provided slot-value, defined for the instance of the TomCat, is kept. The designer may also finally adjust the intrinsicness level of the already mentioned property vendor and change it to '1' (by using the built-in 'Change property level' operation).

## 6 CONCLUSIONS

In this paper we have discussed the need to support flexible multi-level modeling understood as intertwined application of top-down and bottom-up creation process. Whereas the top-down approach, as well as introducing changes to the already created multi-level model, e.g., in terms of changing the name of the class or adjusting the properties, are supported by existing tools, the support for bottom-up modeling is mostly missing. Therefore, we adapted a few selected multi-level model refactoring patterns and implemented them in the XModeler, to support the bottom-up creation process.

When it comes to our future work, five main avenues may be identified. Firstly, in this work we assumed that the task of deriving (meta) classes from their instances is performed by a model designer. Nevertheless, some further automation of this process is envisioned. Similarly, like in the field of ontology engineering a kind of "reasoning" services aiming at identifying (meta)classes or inheritance relations are to be investigated, cf, also [11].

Secondly, additional functionalities are needed in order to support the creative process of multi-level model creation. One of the missing aspects, is that for now, a model designer needs to manually adjust the existing associations and links. Providing an automated support for that is part of our on-going work. Furthermore, in the very early stage of the model creation, it would be useful to be able to state, e.g., that some property may be part of the concept description, however, without providing other details like the data type or the intrinsicness level. Although one could argue that this information might be expressed, e.g., in the form of a comment attached to some concept, a more structured way to capture this information would be desired.

Thirdly, an alternative design of the approach to propagate changes and ensure the validity of multi-level models is to be investigated. We find especially the idea of adopting the constraint-based refactoring, as described in [31], to multi-level setting, appealing.

Fourthly, as the change operations discussed alter the model semantics, it is crucial that a user comprehends the effects of the change operation, before the change is performed. Therefore, we envision designing relevant mechanisms making all of the consequences explicit: we want to improve the understandability of the side effect by providing more comprehensive description and case-specific feedback regarding the changes to be performed.

Finally, all the implemented mechanisms supporting the bottom-up modeling process, as well as our driving assumptions, need to be further evaluated by users. Therefore, we want to continue our empirical studies of the act of multi-level model creation. To this aim, a next round of experiments with the new version of XModeler is planned.

## REFERENCES

[1] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. 2012. On-the-Fly Emendation of Multi-level Models. In *Modelling Foundations and Applications*, Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–209.

[2] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. 2014. Comparing multi-level modeling approaches, In MULTI 2014. *CEUR Workshop Proceedings* 1286, 53–61.

[3] Colin Atkinson, Bastian Kennel, and Björn Goß. 2011. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *Procs. 7th int. workshop on semantic web enabled software engineering.* 1–15.

[4] Colin Atkinson and Thomas Kühne. 2001. The Essence of Multilevel Metamodeling. In *UML 2001*. Springer, London, 19–33.

[5] Colin Atkinson and Thomas Kühne. 2008. Reducing accidental complexity in domain models. *SoSyM* 7, 3 (2008), 345–359.

[6] Tony Clark, Paul Sammut, and James Willans. 2008. *Applied metamodelling: a foundation for language driven development.* Ceteva, Sheffield.

[7] Sybren de Kinderen and Monika Kaczmarek-Heß. 2021. Making a Case for Multi-level Reference Modeling – A Comparison of Conventional and Multi-level Language Architectures for Reference Modeling Challenges. In *Innovation Through Information Systems*, Frederik Ahlemann, Reinhard Schütte, and Stefan Stieglitz (Eds.). Springer International Publishing, Cham, 342–358.

[8] Sybren de Kinderen, Monika Kaczmarek-Heß, and Simon Hacks. 2022. Towards Cybersecurity by Design: A multi-level reference model for requirements-driven smart grid cybersecurity. In *30th European Conference on Information Systems - New Horizons in Digitally United Societies, ECIS 2022, Timisoara, Romania, June 18-24, 2022*, Roman Beck, Dana Petcu, Marin Fotache, Sabine Matook, Remko Helms, Martin Wiener, Lazar Rusu, and Tuure Tuunanen (Eds.). https://aisel.aisnet.org/ecis2022_rp/89

[9] Sybren de Kinderen, Monika Kaczmarek-Heß, and Kristina Rosenthal. 2021. Towards an Empirical Perspective on Multi-Level Modeling and a Comparison with Conventional Meta Modeling. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion*. IEEE, 531–535.

[10] Juan de Lara and Esther Guerra. 2010. Deep Meta-modelling with MetaDepth. In *Objects, Models, Components, Patterns*, Jan Vitek (Ed.). Springer Berlin Heidelberg, 1–20.

[11] Juan de Lara and Esther Guerra. 2018. Refactoring Multi-Level Models. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (Nov. 2018), 17:1–17:56. https://doi.org/10.1145/3280985

[12] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and How to Use Multilevel Modelling. *ACM TOSEM* 24, 2, Article 12 (Dec. 2014), 46 pages.

[13] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

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

[15] Ulrich Frank. 2018. *The Flexible Multi-Level Modelling and Execution Language (FMMLx). Version 2.0: Analysis of requirements and technical terminology.* Technical Report. ICB-Research Report.

[16] Ulrich Frank. 2022. Multi-level modeling: cornerstones of a rationale. *Softw. Syst. Model.* 21, 2 (2022), 451–480.

[17] Ulrich Frank, Luca L. Mattei, Tony Clark, and Daniel Töpel. 2022. Beyond Low Code Platforms: The XModelerML – an Integrated Multi-Level Modeling and Execution Environment. In *Modellierung 2022 Satellite Events, Digital Library, Gesellschaft für Informatik*, MJ. Michael, J. Pfeiffer, and A. Wortmann (Eds.). 235–244.

[18] Ralph Gerbig. 2017. Deep, seamless, multi-format, multi-notation definition and use of domain-specific languages.

[19] Muzaffar Igamberdiev, Georg Grossmann, and Markus Stumptner. 2016. A Feature-based Categorization of Multi-Level Modeling Approaches and Tools. In *MULTI 2016 (CEUR Workshop Proceedings, Vol. 1722)*, Colin Atkinson, Georg Grossmann, and Tony Clark (Eds.). CEUR-WS.org, 45–55.

[20] Santiago P. Jacome-Guerrero and Juan de Lara. 2020. TOTEM: Reconciling multi-level modelling with standard two-level modelling. *Computer Standards & Interfaces* 69 (2020), 103390. https://doi.org/10.1016/j.csi.2019.103390

[21] Matthias Jahn. 2014. *Evolution von Meta-Modellen mit sprachbasierten Mustern.* Dissertation. Universität Bayreuth, Bayreuth.

[22] Manfred A. Jeusfeld and Bernd Neumayr. 2016. DeepTelos: Multi-level Modeling with Most General Instances. In *Conceptual Modeling*, Isabelle Comyn-Wattiau, Katsumi Tanaka, Il-Yeol Song, Shuichiro Yamamoto, and Motoshi Saeki (Eds.). Springer International Publishing, Cham, 198–211.

[23] Monika Kaczmarek-Heß and Sybren de Kinderen. 2017. A Multilevel Model of IT Platforms for the Needs of Enterprise IT Landscape Analyses. *Bus. Inf. Syst. Eng.* 59, 5 (2017), 315–329. https://doi.org/10.1007/s12599-017-0482-4

[24] Monika Kaczmarek-Heß, Mario Nolte, Andreas Fritsch, and Stefanie Betz. 2018. Practical experiences with multi-level modeling using FMMLx: a hierarchy of domain-specific modeling languages in support of life-cycle assessment. In *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018 (CEUR Workshop Proceedings, Vol. 2245)*, Regina Hebig and Thorsten Berger (Eds.). CEUR-WS.org, 698–707.

[25] Jesus Lopez-Fernandez, Jesus Sanchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Example-driven meta-model development. *Software & System Modeling* 14, 4 (2015), 1323–1347.

[26] Fernando Macías, Adrian Rutle, and Volker Stolz. 2016. MultEcore: Combining the Best of Fixed-Level and Multilevel Metamodelling. In *MULTI 2016 (CEUR Workshop Proceedings, Vol. 1722)*, Colin Atkinson, Georg Grossmann, and Tony Clark (Eds.). CEUR-WS.org, 66–75.

[27] Fernando Macías, Adrian Rutle, Volker Stolz, Roberto Rodríguez-Echeverría, and Uwe Wolter. 2018. An Approach to Flexible Multilevel Modelling. *EMISAJ* 13 (2018), 10:1–10:35.

[28] Bernd Neumayr, Katharina Grün, and Michael Schrefl. 2009. Multi-level domain modeling with m-objects and m-relationships. In *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling-Volume 96*. Australian Computer Society, Inc., 107–116.

[29] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign, USA.

[30] Kristina Rosenthal and Stefan Strecker. 2019. Toward a Taxonomy of Modeling Difficulties: A Multi-Modal Study on Individual Modeling Processes. In *ICIS 2019*, Helmut Krcmar, Jane Fedorowicz, Wai Fong Boh, Jan Marco Leimeister, and Sunil Wattal (Eds.). AIS.

[31] Friedrich Steimann. 2015. From well-formedness to meaning preservation: model refactoring for almost free. *Softw. Syst. Model.* 14, 1 (2015), 307–320.

[32] Daniel Töpel and Björn Benner. 2017. Maintenance of Multi-level Models - An Analysis of Elementary Change Operations. In *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017 (CEUR Workshop Proceedings, Vol. 2019)*, Loli Burgueño, Jonathan Corley, Nelly Bencomo, Peter J. Clarke, Philippe Collet, Michalis Famelis, Sudipto Ghosh, Martin Gogolla, Joel Greenyer, Esther Guerra, Sahar Kokaly, Alfonso Pierantonio, Julia Rubin, and Davide Di Ruscio (Eds.). CEUR-WS.org, 243–250. http://ceur-ws.org/Vol-2019/multi_5.pdf

[33] Guido Wachsmuth. 2007. Metamodel Adaptation and Model Co-Adaptation. In *ECOOP 2007 – Object-Oriented Programming (LNCS)*. Springer, Berlin, Heidelberg, 600–624. https://doi.org/10.1007/978-3-540-73589-2_28

[34] Roel J Wieringa. 2014. *Design science methodology for information systems and software engineering.* Springer.