Maintenance of Multi-Level Models – An Analysis of Elementary Change Operations

Daniel Töpel[1] and Björn Benner[2]

[1] Department of Computing, Sheffield Hallam University, UK
[2] Information Systems and Enterprise Modeling, University of Duisburg-Essen, DE

Abstract. The application of multi-level-modelling inevitably leads to the necessity to constantly evolve a model from one iteration to the next. During each of the iterations potentially all levels are subject to change and even more levels are affected by those changes. A set of well defined change operations is required to enable the modeler to improve his models. In this paper we will derive all possible change operations for a given meta-model and give one possible definition for their behaviour.

Keywords: Co-Evolution, Maintenance, Multi-Level Modeling, FMML^X

1 Introduction

Multi-level modeling is an emerging field and provides advanced modeling capabilities over traditional approaches of conceptual modeling [1]. Hence, there are several publications which investigate multi-level modeling [1,2,3,4], however, to the best of our knowledge there are no guidelines which support the creation of multi-level models.

Due to this lack of guidelines, it is not trivial to create a convincing multi-level model. Therefore, a multi-level model will presumably not be created by monotonically adding elements, but it will probably require several iterations with reoccurring tasks of adding, modifying and deleting model elements. Due to this omnipresence of change, it is relevant to choose an appropriate means for creating such a model.

The use of pencil, paper and eraser will probably provide the most freedom in terms of creating a multi-level model. This means does not include any mechanism for checking the validity, thus, multi-level models can be changed freely. However, due to the lack of validation, it is not possible to ensure the model's consistency automatically. Furthermore, the developed models are created as a sketch on a paper, thus, these models cannot be interpreted or executed by a computer.

In contrast, modeling tools do allow a automatic consistency check as well as a intepretation and execution by a computer. However, prevalent modeling tools do not allow conducting non-monotonic changes as freely as by using pencil, paper and eraser. Modeling tools lack freedom because they restrict the model creation and modification either explicitly or implicitly. Tools like MetaEdit+ [5]

or EMF/GMF [6] limit the model creation explicitly by enforcing a user to follow a clearly defined order of tasks. Other Tools like the XModeler [7,8] restrict the model creation implicitly, as they allow to perform the model creation in any order, but some orders will create such an inconsistent state, that the model has to be re-created from scratch.

An adequate modeling tool for multi-level modeling should restrict the modeler as little as possible while ensuring the model's consistency. In order to develop such a modeling tool, it is necessary to identify all possible operations of change which can be performed on a multi-level model.

Therefore, this paper aims at identifying possible change operations on a multi-level model. Due to the complexity of multi-level modeling, identifying all possible changes is a challenging endeavor by itself. Therefore, this work presents a starting point by identifying all possible changes in a reduced metamodel for multi-level modeling, denoted as minimal model. Thus, the outcome of this research does not represent a full-fledged approach for handling change in multi-level models.

The remaining parts of the paper are structured as follows: Section 2 discusses multi-level modeling and the relevance of change therein. Afterwards, the handling of change in traditional conceptual modeling is explored in Section 3, followed by the presentation of the developed approach in Section 4. Following, the results are discussed in Section 5. Eventually, the paper closes with a conclusion and an outlook for future work in Section 6.

2 Multilevel Modeling and Change

2.1 Multi-level Modeling

Frank characterizes multilevel modelling by pointing out three major differences to traditional conceptual modeling: *Flexible Number of Classification Levels*, *Relaxing the Rigid Instantiation/Specialization Dichotomy*, *No Strict Separation of Language Levels* [1].

As multilevel modeling allows for a *Flexible Number of Classification Levels*, those models are not limited to a fixed language architecture, but allow for facilitating as many classification levels as required.

Furthermore, the *Rigid Instantiation/Specialization Dichotomy* is relaxed. In traditional conceptual modeling, there is a strict distinction between *Instantiation* and *Specialization*, i. e., these are mutually exclusive: Two elements might connected either by an instance-of-relation or by a specialization-relation, but never by both at the same time. In contrast, this dichotomy is relaxed in multi-level modeling, as it accounts for elements, which are both instance and specialization of another element at the same time.

Moreover, in multi-level modeling there is no *Strict Separation of Language Levels*. Therefore, in multi-level modeling it is possible to create associations that cross classification levels. Thus, multi-level modeling corresponds to natural technical languages, which might require connecting concepts on different levels of abstraction. Hence, multi-level modeling allows for overcoming the artificial borders of classifications levels created by traditional conceptual modeling.

One multi-level modeling language is the FMML^X [1]. The acronym stands for *Flexible Meta-Modeling and Execution Language*; "the 'x' is intended both to express the flexible classification level of the metamodel and to indicate that the metamodel, as well as all models instantiated from it, are executable" [1].

The FMML^X allows an arbitrary number of classification levels, whereby each level is denoted by an unique identifier [1]. Objects are on the level M0; Classes are on M1 and are illustrated with a white background color. For each further level, the index is increased by 1. Metaclasses are on M2 and have a black background. Furthermore, elements on M3 have blue background, on M4 a red and on M5 a green.

Furthermore, FMML^X provides *intrinsic features* which are visualized by a white number on a black background in front of an attribute [1]. The number implies the instantiation level of the attribute, e.g., "1" implies that an attribute is instantiated on M1. *Intrinsic features* can also be applied on associations and attributes.

2.2 XModeler

XModeler is a meta-modelling tool, that supports multi-level modeling as well as executable modeling [1,7,8]. Models can be edited graphically or using the command line interface. XModeler uses a specifically designed language called XOCL and based on a kernel model called XCore, on which all other features of XModeler are built on. Furthermore, XCore has been enhanced in order to support multi-level modelling with FMML^X .

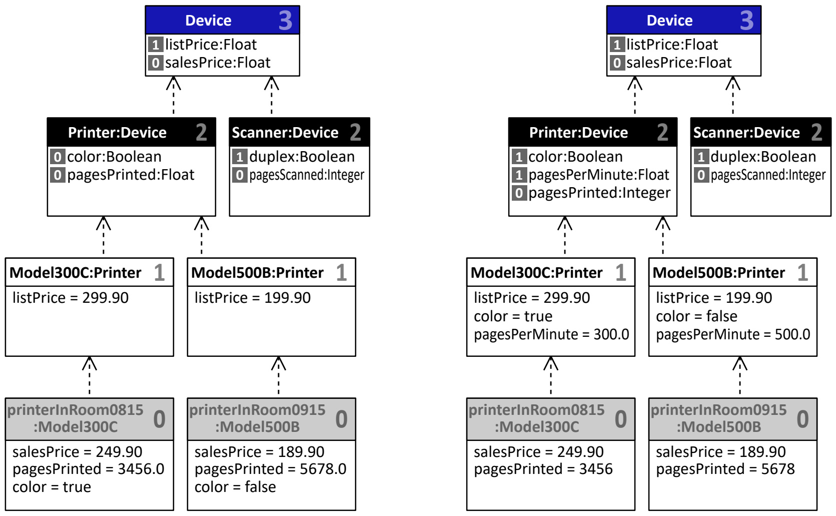
The architecture of XModeler keeps code and model within the same framework, so that any changes in the model go into effect immediately without generating a new version of code. That means for example, that when a new class is added to the model, its constructor becomes available and can be invoked to create instances. These instances themselves can then be referenced such as in the body of operations. Operations in turn can be compiled and invoked at runtime.

While the tool basically supports multi-level modeling, only a specific order of modeling is currently supported. The results from this and subsequent papers are intended to be used as specifications for implementing a multi-level modeling editor which is not limited to top-down-modeling

2.3 Application for Change in Multi-level Modeling

Following, challenges and potential solutions are illustrated, which have to be faced by a modelling tool as XModeler. Fig. 1a shows an example in which the concept of peripheral device is refined over the concept of printer down to the specification of specific printer types and instances of those types (cf. [1]).

If we review the model on the left, we discover that some mistakes were made by the designer. A minor mistake is, that *Printer::pagesPrinted* is of type



(a) before (b) after

Fig. 1: Example

Float, where *Integer* were appropriate. When the type is changed, there is the problem, that the value does not fit anymore. That is a situation which is not much different to a two-level system, but the change must be adapted to a multi-level environment, as the slot is two classification levels away here. To avoid inconsistencies the values have been replaced by the null value. They have to be reset manually. Ideally a conversion function could be supplied here to convert from *Float* to *Integer*, so that the values would not need to be adapted manually.

Also the attribute *color* in *Printer* could be regarded as wrong. This attribute, which tells the reader whether this printer is a color-printer, is wrongly stated as to be instantiated on level 0. Therefore, the actual printers on level 0 were all created with this slot. The attribute should better be a property applying to level 1. When the instantiation level of the attribute is changed, the slots on level 0 must be removed, and new ones have to be created on level 1. In this case, as all instances (for each class respectively) had the same slot value, that value can be reused for their classes. The final state after all these changes is shown in Fig. 1b

Considering the discussed example, changes in a multi-level modeling might be more complexe than in traditional conceptual modeling: In traditional modeling, it is possible to perform several changes on a metamodel in the first step and adapting the instantiated models afterwards. However, multi-level modeling requires to adapt both aspects at the same time. If only the metamodel is adapted, the overall multi-level model becomes inconsistent, because some instances do not correspond to their classes anymore. Thus, adaption constitute one inseparable unit in the realm of multi-level modeling.

Therefore, changes in multi-level models do differ from changes in traditional conceptual models. Thus, it is necessary to investigate possible changes in the context of multi-level modeling in detail with the intention to develop potential guidelines for the adaption of multi-level models.

3 Traditional Conceptual Modeling and Change

3.1 Model Evolution and Co-Evolution

The role of changes in traditional conceptual modelling has been discussed widely (e.g., [9,10,11,12,13,14,15,16]). In that context, the process of adapting a model has been coined as "Model Evolution" or "Model Adaption" [9]. Exemplary areas of application for model evolution are database management systems [17] or Model-driven Engineering [16,18,19].

Model evolution is also relevant for metamodeling. For example, if there are changes in a specific domain of discourse, it might imply changes in a corresponding domain specific modeling language (DSML). Thus, it is necessary to change the corresponding metamodel [12]. The process of adapting a metamodel is called *metamodel evolution*.

Metamodel evolution is a special case of model evolution, because it implies a *model co-evolution* of the instantiated models. Model co-evolution describes the evolution of models with the intention to comply to its evolved metamodel. It is mandatory to conduct a model co-evolution after a metamodel evolution in order to maintain a consistent language architecture [15]. Or to put it differently: If only the metamodel is changed without adapting the instantiated models, the models do not comply to the metamodel anymore. Thus, the language architecture would get inconsistent.

3.2 Existing approaches

In the context of object oriented modeling, there are several approaches which aim at dealing with model evolution and/or model co-evolution.

Gruschko et al. classify possible changes of a metamodel evolution into *Not Breaking Changes*, *Breaking and Resolvable Changes* and *Breaking and Unresolvable Changes* [20]. *Not Breaking Changes* occur in the metamodel, but do not make the instantiated models inconsistent. *Breaking and Resolvable Changes* do make instantiated models inconsistent, however, the consistency can be regained automatically. Lastly, *Breaking and Unresolvable Changes* do also break the consistency, but the consistency cannot be regained automatically.

Wachsmuth develops change operations for models which allow a stepwise meta-model evolution [12]. Thereby, Wachsmuth distinguishes between three adaptation groups: refactoring, construction and destruction. Furthermore, the model co-evolution is also considered in this approach by corresponding adaption mechanisms.

Vermolen et al. investigate the model evolution from an ex-post perspective, i.e., they compare two versions of one metamodel in order to identify the differences [13]. Based on these differences, the required change operations are derived. The required change operations are a composition of previously identified basics change operations, which are derived from model refactoring mechanisms.

Herrmannsdoerfer et al. develop an approach which aims at the coupled conduction of metamodel evolution and model co-evolution [11]. Therefore, they developed a library of coupled operations, which describes a set of operations for metamodel evolution and their implied operations for model co-evolution [10].

Narayanan et al. developed a language for dealing with metamodel evolution and model co-evolution [21]. With this language, the conducted metamodel evolution is explicated by defining mapping-relation between elements in the pre-evolutionary metamodel and the post-evolutionary metamodel. Based on this, an automated model co-evolution can be performed.

Similarly, Cicchetti et al. present an approach, in which the differences due to evolution are explicated with a *difference model* [18,19]. A difference model contains all differences between to stages of a metamodel evolution. Similar to the approach of Narayanan et al., a difference model allows for an automatic model co-evolution.

Jahn presents an approach, where he derives a set of change operations for a multi-level system [22]. He claims to have a complete set of change operations, induced by having a change operation for each property of each element of his meta-model and a deleting operation for each element of his meta-model. Operations, which are adding elements, are intentionally omitted as not causing inconsistencies. Additionally there are some moving operations which are necessary for completeness, but not covered by the argument above. Also, we think the adding operations should not be treated as trivial, as for instance the operation adding an attribute would leave some existing instances in an inconsistent state.

Jahn's approach uses class migration as a default method to resolve some of the inconsistencies arising. The *delete Concept* operation for example would leave the instances of the deleted concept orphaned. To resolve that issue, these instances now become top-level concepts where the *instanceOf* link is not set. Instead, we would like to avoid class migration as a default method. We might add alternative change operations in the future, which offer an advantage by using class migration, but for this paper we will focus on change operations free of class migration to avoid side effects.

Those existing approaches are used as inspiration for analysing the potential change operations in a multi-level modelling.

4 Evolution in Multilevel Models

4.1 Minimal Model

Following, a minimal meta-model or multi-level modelling is described step-by-step. The complete model is shown in Figure 2.

To begin with, the meta model will consist of *Entities*. These entities can serve as classes and instances at the same time as explained in section 2.1. These entities have a name, a classification level and a class they were instantiated from. An entity can be instantiated from the meta-model, which requires a name and a classification level to be chosen. When instantiated, the instance requires a name as well, but the classification level will be in any case one less than its class's classification level. If an instance has classification level zero, it cannot be instantiated further.

The meta model also includes attributes. An instantiable entity may have a named attribute, which causes its instance to have a slot which can have a value.

In a two-level system there is no choice when to create the slot in an object from an attribute in a class. In multi-level models, there is the option to add the attribute not only in the entity directly above the level where the slot is to be created, but also in any level above. An attribute which cause slot creation only after more than one instantiation is called an intrinsic attribute. In addition to the name, an attribute now needs one more parameter, which describes when the slot will be available.

The attribute also has a type, which can be any instantiable entity. In a two-level architecture the slot must contain a direct instance of the type, as we can instantiate it only once. In a multi-level architecture we can instantiate multiple times. The exact classification level of the slot value has not been specified yet. The corresponding slot must hold a value which is an intrinsic^{1+}-instance of the given type. The null-value satisfies all types.

The meta model also allows inheritance. In this context inheritance means, that an instantiable entity has a link to one or more `parents`. The entity and its parents must have the same level. Attribute names must be unique within a class and all its parents recursively. Also, loops are not allowed.

Multiple inheritance is allowed, as it does not cause additional issues here. As long as the concept of overriding – either attributes or operations – is not introduced, duplicate attributes are not an issue, as those duplicate attributes are in fact identical attributes received through different means, which materialise in only one slot for each attribute added to an entity.

The following terms will be used to refer from one entity's perspective to another entity. An entity may have a **Class** which it was instantiated of. The class of its class is called a **Meta-Class**. We expand this concept by adding an exponent to "Meta", where the exponent indicates the number of instantiation steps from the referred entity to the referring entity, which have been made beyond the classical single instantiation. That means: When a class has been instantiated once, there have been no additional instantiation steps made, the full term would be **Meta^0-Class**, which is equivalent to **Class**. Also, a **Meta-Class** would be

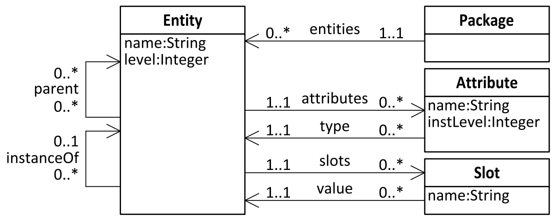


Fig. 2: Minimal Model

Meta^{1}-Class in the full notation. Finally we add a + to the exponent when referring to all entities where are at least that number of additional instantiation steps were made. The same type of notation is used for the instances of an entity, where the term **Intrinsic^{n}-Instance** indicates the number of instantiation steps done beyond the classical one. The terms **Instance** and **Intrinsic-Instance** are used to indicate zero or one step respectively. The modifier + can also be used here.

4.2 Finding the Evolutionary Steps

To make sure we will get a complete list of change operations, we derive them from the properties and links in the diagram in Figure 2. When describing the state of a model, we describe the state of an instance of the class *Package*. When we start modelling we have such an object which represents the empty model. This is the root element, of which exactly one exists.

We can now add and remove the other objects i. e., *Entity*, *Attribute* or *Slot*. These objects cannot exist on their own as they all are on one end of a link which requires them to be linked to something else. E.g., an *Attribute* must be connected to exactly one *Entity*. These links give us the first group of change operations. An object of the zero-to-many end of the link can be created and added to one object of the one-to-one side of the link. It can also be removed and discarded. And there is a third change operations which for a given object from the zero-to-many end replaces the object on the one-to-one end.

Four of these possible change operations are not used. There are no operations concerning *Slot*, as constraints prohibit independent changes on slots. As we only have one root element, an *Entity* cannot be transferred to a different root. Additionally, because the *instanceOf* and the *level* of *Entity* are immutable, two different operations are necessary for creating entities: One for creating a top-level-class from the meta-model and a second one for creating an instance from an existing entity.

There are two links which are many-to-one. These are *Attribute* to *Entity*, named *type* and *Slot* to *Entity*, named *Value*. That means that every attribute refers to exactly one entity, which in turn may be referred to by any number of attributes. Therefore we only need a *replace*-operation which replaces the type entity in the attribute. The same holds true for the value entity in slot.

The link *Entity* to *Entity*, named *instanceOf* will not be considered. These links are only added or removed when an entity is added or removed.

There is one more link: *Entity* to *Entity*, named *parent*. This is a many-to-many link, so we only need an *add*- and a *remove*-operation. An operation for moving a parent link to another target may appear necessary at a first glance. Nevertheless the many-to-many link does not demand such an operation, as adding the new link first and removing the old one does the required tasks without data loss. Fortunately the constraints do not prevent us from having the old and new link simultaneously.

There are five properties, all with a multiplicity of one. That means, they only need a *change*-operation. The property *level* in *Entity* is immutable and the property *name* in *slot* may not be altered on its own. For the other three, an operation is available.

4.3 Change Operations

addEntity(level, name): This operation adds a new top-level-entity to the model. This new entity does not have a class it is instantiated from. It requires a name and a level. Implicitly the *instanceOf*-link is set to the null-reference. The class name is required to be unique within the model. Also it must follow a given rule set, which is not yet specified, e. g., it must not contain white spaces. The level cannot be negative, as negative levels are not defined. While a level of zero might be thinkable, it is rather useless, as such an object does neither have slots or any other features. While a level of one does not make use of any multi-level features, it still serves at least one interesting purpose: This allows any two level-architecture to be a special case of a multi-level architecture. Therefore it is decided, that the level must be at least one. No slots will be created as this entity is not instantiated from anything which could have applicable attributes.

addEntity(of, name): This operation adds a new entity to the model. This new entity is instantiated from another entity, known as its class. The name property and the instanceOf-link is set from the argument given. The level property is implicitly set one less that the level of the class. As the new level cannot be less than zero, the level of the class must not be less than one. The name property follows the same rules as above. Where the meta^{0+}-classes have attributes with an instantiation level equal to this entity's level, the entity has to have slots to be in a consistent state.

removeEntity(name): This operation removes an entity from the model if possible. It checks if the entity is not referenced in any way, and if that is the case, the entity is removed. If it is not possible, the entity is not removed. It is not specified yet how to notify the invoker, but this issue is not specific to this operation anyway, as for other operations the pre-conditions might not be met as well. Aside from that, this operation will not attempt to remove usage of this entity on its own. It will nevertheless be useful and possible to have an aggregated operation doing so in the future.

changeEntityName: This operation changes the *name* property. As for *addEntity*, the new name must follow certain rules. The entity is referred to only by links and not its name, so no other data needs to be adjusted.

addAttribute: This operation creates an *Attribute* and links it to an *Entity*. As arguments the entity, a name, a type and an instantiation level are required. The instantiation level is a number which indicates the level where a corresponding slot is created and is non-negative but less than the level of the entity. The type can by any entity which can be instantiated, i. e., has a level of one or higher.

The name used here will be used for the slots as well. As no two slots in one entity may have the same name, this has implications on the attributes as well. An entity gets a slot from every attribute it finds in an entity which it is linked to through instanceOf and parent links, going only in the direction from source to target, and where the attribute's instantiation level equals its own level. This hypothetical instance where a name clash could occur does not necessarily exist yet. We need to check for this issue here, otherwise we will have an entity which should be instantiable by merit of their level, but cannot be instantiated without violating consistency. We will use the phrase *related entity* to refer to two entities which have the potential to cause a name clash in entities later on, without exactly specifying their relationship here.

If those instances are already existent, then a slot has to be created for each of them. Slot creation is explained in 4.3: addEntity(of, name).

removeAttribute: This operation removes an attribute from an entity. As a consequence, in all intrinsic^{0+}-instances of the entity which are on the level specified by the attribute, the slot with the same name as the attribute is removed.

changeAttributeName: This operation changes the *name* property of an *Attribute*. The new name must follow the same rules as in 4.3: addAttribute. All slots derived from this attribute must have their *name*-property adapted.

changeAttributeLevel: This operation changes the *level* property of an *Attribute*. This actually means that existing slots have to be removed in some entities, whereas in other entities slots have to be created. If we used *removeAttribute* and *addAttribute* instead, the slot values are lost. When the level is decreased, the entity where the new slot is created has its slot linked with the value of its direct meta^{n}-class's slot. This only works on a decrease, as an entity has only one direct meta^{n}-class's on a given level but has possibly any number of intrinsic^{n}-instances on any given level. While not doing it here, we still have the option to add some rules on how to determine a new slot value from a set of former slot values. For the moment the new slot value for an increasing level will be the null-value.

changeAttributeType: This operation changes the *type* link of an *Attribute*. This is likely to violate the constraint, that a slot's value must be an intrinsic^{0+}-instance of the type of the attribute it was created from. Some slots may still have a conforming value, so they can remain. Other slots may have a non-conforming value,which have to be dealt with. The easiest solution is to replace them with a null-value. A future option would be an instruction how to initialize the slot values, i. e., a conversion function.

moveAttribute: This operation is aimed at moving attributes between related entities. A classic example would be moving an attribute form a class to its superclass. If we move the attribute up along inheritance or classification, we will have objects which gain a slot, if we move down, some objects will lose a slot. It is not necessary to move attributes up and down in one step, as there is no loss doing this in separate steps. For simplification it is sensible to limit this change to directly related classes, and use it repeatedly for longer distances.

changeSlotValue: This operation changes the target of the *value*-link of *Slot*. There is no danger of inconsistencies as long as the new value is an intrinsic^{0+}-instance the type of the attribute the slot was created from. Additionally a null-value is always considered a legal value.

addParent: This operation adds a new *parent*-link between two *Entities*. Inconsistencies may arise from the fact, that the scope of attributes is changed. This is not limited to the attributes of the two entities involved. With a possibly larger scope, some attributes may no longer have a unique name therein. Also, this new link must not result in any loops. That means, there must be no closed loop paths through any number of *parent* or *instanceOf* links following their direction specified. The entity on the source end is referred to as subclass, the entity on the target will is to as superclass. As inheritance affects the slot creation from the attributes, both entities must be able to have attributes, i. e., they have a level of at least one. If the link can be added legally, the subclass's intrinsic^{0+}-instances may gain a slot.

removeParent: This operation removes an existing *parent*-link between two *Entities*. This may affect slots in existing intrinsic^{0+}-instances of the class at the source end. Also, those very instances may now be illegal values in other slots, as they not necessarily intrinsic^{0+}-instances of the required type any more.

5 Discussion

In the following, the contribution of this paper in terms of an approach for multilevel model evolution is discussed in detail.

To our best knowledge, only one [22] approach for multilevel model evolution has been conducted so far. That approach is a good inspiration, but not sufficient for our demand. Therefore, a corresponding approach is proposed in this paper. For supporting the development of a well-founded approach, the work from adjacent fields – namely object oriented modeling and metamodeling – has been utilized. Due to the wider subject of multilevel modeling, the approaches for evolution in object oriented modeling and metamodeling are not suitable for being used in multilevel modeling. However, due to similarities (e.g., instantiation relations between entities, the characterization of entities with attributes, etc.), the approaches can be used as a starting point respectively an inspiration for developing a suitable approach.

In the fields of object oriented modeling and metamodeling, the terms model evolution and model co-evolution has been coined. While model evolution concerns the adaption of models respectively metamodels, model co-evolution describes the corresponding adaption of the dependent model instances respectively metamodel instances (i.e., models). This terminology is only partly applicable for multilevel modeling, because a multilevel model comprises several classification levels, e.g., metamodel, model and model instance. Therefore, the separation between model evolution and model co-evolution is not reasonable for multilevel modeling. If a classification layer is evolved, it is mandatory to adapt the depending classification layers in order to maintain the multilevel model's consistency.

The development of a full-fledged conception for multilevel model evolution is an challenging endeavor due to the number of modeling constructs in multilevel modeling. In order to reduce this number and therefore to reduce the possible number of change operations, a minimal model for multilevel modeling has been defined.

For identifying all possible change operations, the basic change operations "add","modify" and "delete" have been applied on the elements of this minimal model. As not all possible change operations are meaningful in terms of the multilevel modeling, the list is reduced by all pointless change operations.

By removing the meaningless operations, all operations are identified which are possible and meaningful in terms of the minimal model. Thus, this set is a complete set of change operations in terms of the minimal model.

Although the set of change operations is complete for the minimal model, it is not a full-fledged approach for multilevel model evolution. In order develop a holistic multilevel model evolution approach, it is necessary to enrich the minimal model as well as the set of change operations step wise by further modeling constructs respectively further change operations.

Furthermore, the current conception of multilevel model evolution is only on a theoretic level. It is further necessary to provide a prove of concept by developing a multilevel modeling tool which supports the multilevel model evolution.

Additionally, the presented approach comprises only atomic change operations, i.e., those operations represent the minimal possible changes to a multilevel model which maintain the model's consistency and do not lead to an information loss. It would be desirable to incorporate mechanisms for bundling atomic operations to more complex operation. For instance, the presented changes include only an operation for removing an entity; however, it might be desirable to delete an entire entity hierarchy at once.

Furthermore, the transfer of model evolution and model co-evolution to multilevel modeling is just one approach for dealing with change in multilevel models. Another imaginable approach would be to omit the co-evolution, but to consider the evolved multilevel model as a new version of it. In this new version, there are only those entities present which are consistent with the new version. For example, if an entity is enhanced by an attribute, its instances are not taken over to the new version. In order to make those instances part of the new version, each instance has to be migrated individually to an consistent instance of the entity's new version. However, this approach would require to deal with class migration in multilevel models.

6 Conclusion / Future Work

In this paper, a conceptualization of change in multilevel models is developed. Due to the complexity of multilevel modeling, change has been investigated in terms of a minimal metamodel of multilevel modeling (i.e., reduced metamodel). By applying the operations "add", "remove" and "modify" on all elements of the minimal model, all possible change operations have been identified. Due to the conceptualization of multilevel modeling, not each of those operation is meaningful. Thus, this set of operation has been reduced by meaningless operations in order to derive a complete list of all relevant change operations on multilevel models.

As those change operations are founded on a minimal model, this set is complete in terms of this minimal model but does not represent a full-fledged approach for multilevel modeling. Therefore it is necessary to enhance both the minimal model and the list of change operations stepwise by further language constructs respectively by further operations. Furthermore, this conceptualization has only been created on an theoretical level, i.e., it is necessary to develop a corresponding modeling tool, which supports the application of those changes on a multilevel model.

Another issue arises when the meta model includes operations that contain a body. These operation bodies will refer to named properties like attributes or other operations. These references must be kept consistent with the metamodel. While refactoring operation bodies in a fully typed, classical two-level environment has been widely addressed[23], a multi-level system requires those approaches to be adapted.

Thus, future work aims at developing a full-fledged conceptualization of change in multilevel modeling. Furthermore, this conceptualization should be underpinned by a proof of concept. This should be as a multilevel modeling tool which supports the application of all relevant changes on models without compromising the model's integrity.

References

1. Frank, U.: Multilevel modeling: Toward a new paradigm of conceptual modeling and information systems design. Business and Information Systems Engineering (2014) 319–337
2. Clark, T., Gonzalez-Perez, C., Henderson-Sellers, B.: A foundation for multi-level modelling. In: MULTI@ MoDELS. (2014) 43–52
3. Atkinson, C., Kühne, T.: In defence of deep modelling. Inf. Softw. Technol. **64**(C) (August 2015) 36–51
4. Carvalho, V.A., Almeida, J.P.A.: Toward a well-founded theory for multi-level conceptual modeling. Software & Systems Modeling (2016) 1–27

5. MetaCase: MetaEdit+. `http://www.metacase.com/products.html` (2017)
6. The Eclipse Foundation: Graphical Modeling Framework. `https://www.eclipse.org/modeling/gmp/` (2017)
7. Clark, T., Sammut, P., Willans, J.: Superlanguages: Developing languages and applications with XMF. (2008)
8. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling A Foundation For Language Driven Development: Second edition. (2008)
9. Favre, J.M.: Meta-model and model co-evolution within the 3D software space. In: ELISA: Workshop on Evolution of Large-Scale Industrial Software Applications. (2003) 98–109
10. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope - automating coupled evolution of metamodels and models. In: Proceedings of the 23rd European Conference on ECOOP 2009. (2009) 52–76
11. Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In: Software Language Engineering. (2010) 163–182
12. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: European Conference on Object-Oriented Programming. (2007) 600–624
13. Vermolen, S.D., Wachsmuth, G., Visser, E.: Reconstructing complex metamodel evolution. In: International Conference on Software Language Engineering. (2011) 201–221
14. Khelladi, D.E., Hebig, R., Bendraou, R., Robin, J., Gervais, M.P.: Detecting Complex Changes During Metamodel Evolution. In: Advanced Information Systems Engineering, Springer, Cham (2015) 263–278
15. Demuth, A., Riedl-Ehrenleitner, M., Lopez-Herrejon, R.E., Egyed, A.: Co-evolution of metamodels and models through consistent change propagation. Journal of Systems and Software (2016) 281–297
16. Iovino, L., Pierantonio, A., Malavolta, I.: On the Impact Significance of Metamodel Evolution in MDE. Journal of Object Technology (2012)
17. Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., Madec, J.: Schema and database evolution in the o2 object database system. In: Proceedings of the 21th International Conference on Very Large Data Bases. VLDB '95 (1995) 170–181
18. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: Enterprise Distributed Object Computing Conference. (2008) 222–231
19. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Meta-model differences for supporting model co-evolution. In: Proceedings of the 2nd Workshop on Model-Driven Software Evolution-MODSE. (2008) 1–10
20. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: Proceedings of the International Workshop on Model-Driven Software Evolution, IEEE (2007)
21. Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G.: Automatic Domain Model Migration to Manage Metamodel Evolution. In: Model Driven Engineering Languages and Systems. (2009) 706–711
22. Jahn, M.: Evolution von Meta-Modellen mit sprachbasierten Mustern. PhD thesis, Bayreuth (2014)
23. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering (2012) 5–18