Remodeling to Powertype Pattern

Matthias Jahn, Bastian Roth, Stefan Jablonski
Chair for Databases & Information Systems
University of Bayreuth
Bayreuth, Germany
{matthias.jahn, bastian.roth, stefan.jablonski} @ uni-bayreuth.de

***Abstract*— Nowadays, models often stand as first class objects in the field of software development. That's why clarity and understandability are important markers of high quality models. Therefore, several patterns exists that can help to improve model quality. However, developing a domain specific language is affected by understanding the domain of interest which often evolves during the development of the software system. This evolution again causes the language to change either. As a consequence of that, meta-modeling patterns are oftentimes inserted in an existing meta-model which results in various adaptions to migrate the system into a valid state. Since the current research has not discovered any techniques to cope with a remodeling to such a pattern these adaptions have to be done manually. Focusing on this challenge, we present in this article an evolution operator that creates a powertype within an existing model and furthermore adapts the other related models simultaneously.**

Keywords-powertype, extended powertype, remodeling to patterns, meta-model evolution, meta-model, deep instantiation

I. MOTIVATION

Today, developers often tend to define a separate modeling language for special parts of the domain of interest. That is especially the case if standard modeling languages do not cope with special application settings. This trend is referred to as domain specific modeling (DSM) and the resulting language is hence called domain specific language (DSL).

A modeling language in general consists of three parts: a definition of an abstract syntax, a definition of a concrete syntax, and a rule set (constraints) [1]. Thereby, meta-models are oftentimes used to express both the abstract and the concrete syntax. Hence, the quality of the resulting language is highly-coupled to the quality of the meta-models describing it. Consequently, these meta-model have to be concise and human-readable.

Therefore, current research has discovered several patterns (in the following called language patterns to distinguish them from design patterns) that enrich meta-models in different aspects, e.g., helping persons of different perspectives in the software development process (e.g., the software developer or the method engineer) to understand the meta-model easier [2] or improving their conciseness [3].

One of these language patterns with the above mentioned benefits is the powertype pattern [4], [5]. However, introducing a powertype pattern into an existing meta-model often results in several manual adaptions in other meta-levels for migrating models to the new meta-model. Hence, such a remodeling to powertype patterns can be a time-consuming and error-prone task [6]. Focusing on this problem, we present below an operator that introduces a powertype pattern into an existing meta-model. Simultaneously, the operator adapts corresponding models into a valid state.

Therefore, in the following section we are going to show the state of the art. Subsequently, we explain the powertype (pattern). After that, we will present an extension for this pattern: the extended powertype. In section V we present the Create-Powertype-For operator which introduces an (extended) powertype pattern into a meta-model. In the subsequent section we provide an example model on which we apply the operator. Finally, we give a conclusion and an outlook to our future work.

II. RELATED WORK

The presented work belongs to the research field of meta-model evolution. The Create-Powertype-For operator changes the (meta-) meta-model and migrates other (meta-) models to become valid to the new meta-model.

In the current research such an approach is called coupled evolution [7]. Since most of the work in this field considers merely two meta-levels the coupled evolution definition is limited to a model and a meta-model. As we do support more than two meta-levels in our modeling environment we extend this definition to arbitrary levels.

Meta-model evolution, in general, faces two main challenges. First, adaptations and changes performed on a meta-model need to be captured [8]. Second, evolving a meta-model might render models as instances of a meta-model invalid, e.g., when attributes are removed or a type within a meta-model is defined to be abstract within an evolution step. Hence, these invalid models have to be migrated which is called co-evolution [9].

According to the work of Herrmannsdorfer et al. [8], approaches for capturing meta model evolution can be categorized into three kinds: state based, change based and operation based approaches. State based approaches store two versions of a model and derive differences between those two versions after changes were actually performed (which is an implementation of the Model Management operator DIFF [10]). Contrariwise, change based approaches record differences at the moment they occur. Operation based approaches are a subclass of the change based approaches since changes on meta models are defined by means of transformation operators before they are actually

 ISBN: 978-1-61208-276-9

performed. In today's systems often state based recording is chosen although it is not as powerful as the operation based approach [8]. Our presented approach belongs to the operation based approaches.

Practical application scenarios of the varied approaches can be found in the work of Gruschko et. al. [11] (state based), Aboulsamh et. al. [12] (change based) and Herrmannsdorfer et. al. [8], [13] (operation based).

Similar to the above presented work, Wachsmuth [14] and Herrmannsdorfer et. al. [13] provide an operation set that is used to evolve the meta-model explicitly, i.e., by means of well-defined transformations the user evolves the meta-model stepwise. In consequence, co-evolution can be performed without the need to handle ambiguity which is a challenge for state-based approaches [15].

Up to our knowledge, the current research in meta-model evolution mostly considers common meta-modeling concepts like classes, attributes and relations. Only some approaches (e.g., [13], [14], [16]) also analyze inheritance hierarchies for evolution and explain solution for handling co-evolution. However, there is no approach that considers other language pattern like the powertype pattern, deep instantiation or materialization [17].

Besides handling the evolution itself, handling co-evolution is another important topic in this field of research. To face this challenge, various approaches can be observed: matching of two meta models (see model management [18]), operation based co-evolution and manually specification of migration [15]. An Example for an operation based co-evolution can be found at the work of Wachsmuth [14], within the COPE System [19] and also within this paper.

III. THE POWERTYPE PATTERN

The powertype pattern is a language pattern used to describe that a concept A extends another concept Part (this is called the partitioned type) and at the same time this concept A is an instance of concept Pow (which is then called powertype).

A. Example

Below, there is an example of the powertype pattern that shows a simple meta-model (named M2) with two concepts: Tree and TreeKind. The concept Tree stands of course for a tree and TreeKind is a representation for a kind of a tree. Furthermore, a model (M1) is shown with only one concept Maple which stands for a correspondent real world object.

If one wants to model trees there are at least two different views of seeing a maple. On the one hand, this maple is a specialization of the class tree. On the other hand, maple partitions the set of trees because it is a kind of a tree. Hence, maple can be seen as a specialization of tree. To combine these two views, one can introduce the powertype pattern (Figure 1). Then, Tree is partitioned with TreeKind (the powertype) and Maple is an instance of TreeKind and together with that a specialization of Tree.

As a consequence, Maple has two different facets. The first one is the *type* facet that extends Tree and the second one is the *instance* facet, an instance of TreeKind.

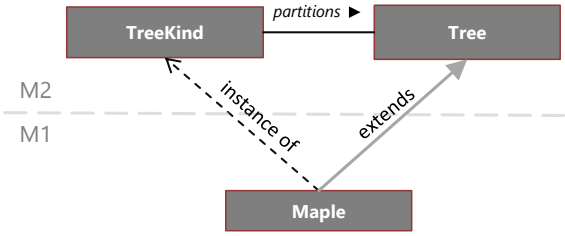

Figure 1. Example of a powertype pattern

Such a mixture of a class and an object is called clabject [20] or concept [21]. The specialization relationship is often not visualized within meta-model diagrams.

IV. THE EXTENDED POWERTYPE PATTERN

One rule of practice in modeling is that all attributes being common in all subclasses are added to the superclass [22]. Other attributes that do not belong to each of the subclasses are not declared in the superclass, in general. Instead, often new subclasses are created that stand between the super- and the subclasses in the inheritance hierarchy. As a consequence, a deep inheritance hierarchy could result which is often seen as bad design [23]. Furthermore, this approach leads to multiple inheritance which sometimes causes problems [24], [25] like the diamond of death.

To avoid this complex inheritance hierarchy, one can use the extended powertype pattern [26], [27]. This pattern enhances the powertype pattern with so called feature attributes.

These boolean attributes are declared at the powertype with a link to an attribute of the partitioned type (the enabled attribute). Afterwards, one can decide for each instance of the powertype if an attribute of the partitioned type is inherited or not. If a feature attribute at an instance of the powertype is set to *true* the corresponding enabled attribute of the partitioned type is inherited. Needless to say that if a feature attribute has the value false no attribute is inherited.

Hence, all attributes of the sub-concepts can be collected in the partitioned type and for each sub-concept one can decide the set of attributes that are inherited.

A. Example

In Figure 2 a simple graph-based process modeling language with an extended powertype pattern is shown.

To visualize the complete meta-model stack we use a tree editor with syntax similar to object-oriented programming languages. The root of the tree is the whole meta-model stack. The children of that are the different meta-levels. The next higher meta-level which is instantiated by the current level is shown after the colon. Each level again contains at least one or more packages structuring the level. In a package lie concepts (clabjects) and these concepts can have attributes and/or assignments. Again, after the colon all instantiated concepts are listed. Other relations like extends or partitions are also shown together with the corresponding other concept.

 ISBN: 978-1-61208-276-9

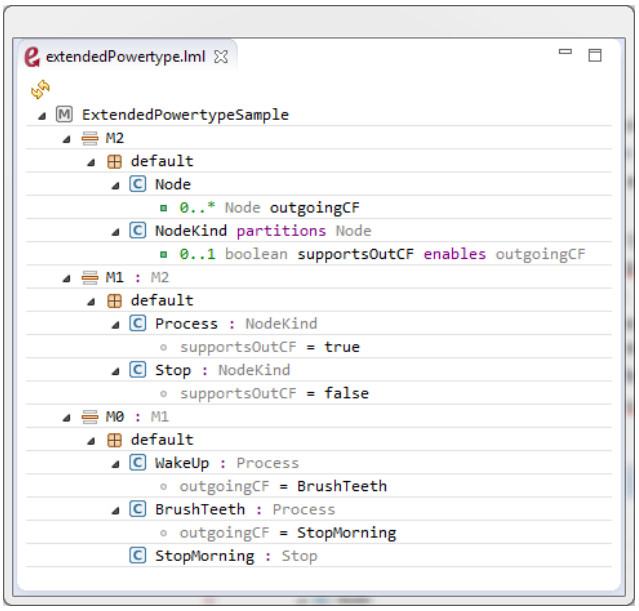

Figure 2. Morning Process Example

Furthermore, the deep instantiation counter (also called deep instantiation potency) is displayed, if the value is greater than 1 (see section VI.A). Attributes have a cardinality (0..1, 0..*, 1..*) and an attribute type. Assignments consist of a corresponding attribute and a value for it.

In the meta-meta-model (M2), Nodes are connected with each other using outgoing control flows. This is done with the corresponding outgoingCF attribute at Node. Besides, an extended powertype (NodeKind) is modeled due to the fact that NodeKind has a partitions relation to Node. NodeKind again has a boolean feature attribute supportsOutCF enabling or disabling the outgoingCF attribute of Node.

At level M1, Process and Stop are instances of the powertype. Since a stop interface does not have any outgoing control flows the supportOutCF attribute is set to false whereas the Process attribute is set to true.

Level M0 contains a little model that describes a (spare) morning process. After waking up, the concerning person brushes his/her teeth and then stops the morning process. Since the feature attribute of Stop was set to false setting the value of outgoingCF in StopMorning would cause a validation error.

V. THE CREATE-POWERTYPE-FOR OPERATOR

In the following, we present an Evolution operator that introduces a powertype into an existing (meta-) model and simultaneously adapts the meta-model hierarchy to be valid again.

A. Operator Process

In Figure 3 the process of the Create-Powertype-For operator is shown. Therein all steps that need an input from the user are highlighted with black boxes. “The Move concept to upper level” and “the Add instantiation to powertype” steps are also highlighted as they are other complex evolution operators that will be presented below.

Initially, the operator is invoked with a source concept (e.g., chosen by the user). In the following, this concept is called Part as it will be the partitioned typed after the operator has finished. In the next step the operator collects all concepts that specialize Part. This set of concepts (in the following called SCs) is important because all members could potentially be an instance of the newly created powertype.

After that, the user decides which member of SCs will become an instance of the powertype and hence creates a subset of SCs (SubSCs). Then, for each member of SubSCs the specialization relation to concept Part is deleted. Afterwards, each concept of SubSCs is checked whether it is instantiated or not. If one concept is instantiated, concept

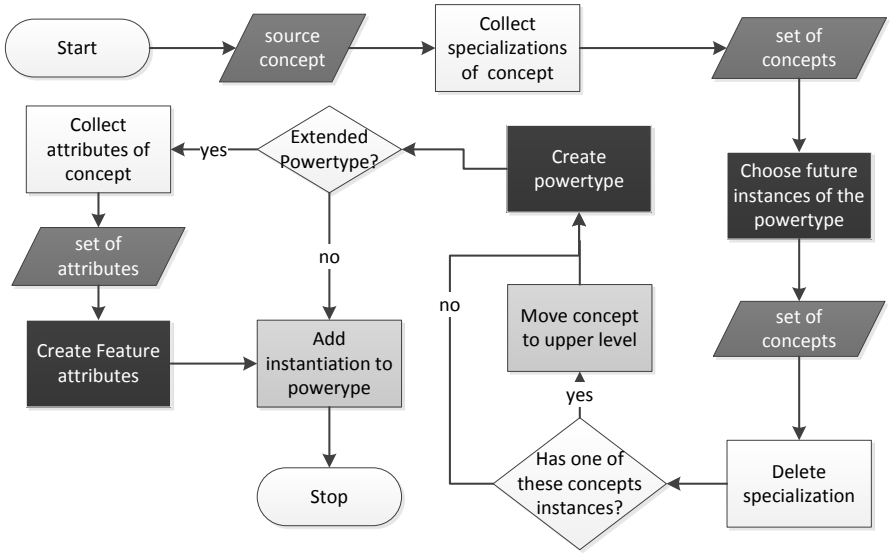

Figure 3. Create-Powertype-For operator process

Part and all related concepts (see section V.B) have to be moved to the upper level. Otherwise, it would not be possible to instantiate the instances of the powertype again. Of course, in case a modeling environment does not support several levels this step cannot be done and hence the instantiation has to be deleted.

Then, a new concept (the powertype) is created by the operator. The user specifies the properties of the concept like the name, whether the concept is abstract or final, its visibility, its instantiated concepts (optional), its extended concepts (optional) and its concretely used concepts (for instance specialization see [21]), also optional). The user also must specify whether the concept is an extended powertype or not. The concept Part will then be added to the set of partitioned concepts whereby the *partitions* relation of the powertype is created.

If the created powertype is an extended one the operator collects the attributes of the initially given concept Part. Then, the user chooses the attributes that will get a corresponding feature attribute which will be created in the powertype. Finally, each of the previously chosen concepts (SubSCs) will become an instance of the new powertype using the corresponding operator (see section V.C).

B. THE MOVE-CONCEPT-TO-UPPER-LEVEL Operator

The Move-Concept-To-Upper-Level operator moves, as the name indicates, a concept from a given level upon the next upper level. The process of the operator is shown in Figure 4.

1) Operator Process

The operator gets as input a concept that will be moved one meta-level up.

In the first step the operator tries to get the upper level and checks whether the level exists or not. If not a new level is created and the name of it has to be set. Then the operator changes the level of the given concept to the upper level.

Afterwards, the operator increments the deep instantiation counter of the given concept if the concept is instantiated.

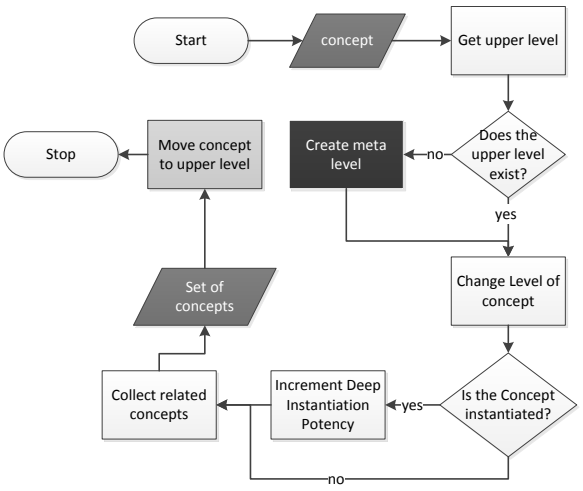

Figure 4. Move-Concept-To-Upper-Level operator process

Changing the value of the deep instantiation counter [28] causes that instances of the concept can instantiate the concept again although they are more than one (exactly two) meta-level lower. If deep instantiation is not supported other techniques like nested meta levels [29] may be used at this point.

For correct migration of the meta-model the operator has to invoke itself recursively on all related concepts. Thus, these concepts are collected in the next step. Related concepts are those concepts that stand in a relationship with the given concept (includes relationships like *extends* (for specialization), *partitions* (for powertype relation) or *concreteUseOf* (for instance specialization) [21]. Thereby, the operator has to detect cycles to avoid an endless loop.

C. The Add-Instantiation-To-Powertype operator

This operator adds an *instanceOf* relation from a given concept to a given powertype. In Figure 5 the process of the operator is presented.

1) Operator Process

Initially, the operator is invoked with a concept (the future instance) and a powertype. If the powertype is not an extended one merely the *instanceOf* relation between the concept and the powertype is created. Thereby, a constraint has to be considered. In case the instance of the powertype is already an instance of another concept this would end in multiple instantiation which breaks, e.g., strict meta modeling [30]. Thus, for such environments the operator has to delete one instantiation.

If the powertype is an extended powertype the operator has to provide a possibility to move the attributes from the given concept to the partitioned type. Therefore, the user has to choose all attributes of the concept that should be moved.

For each reference attributes (the attribute type is a concept) the operator has to check whether the attribute type is a specialization of the partitioned type.

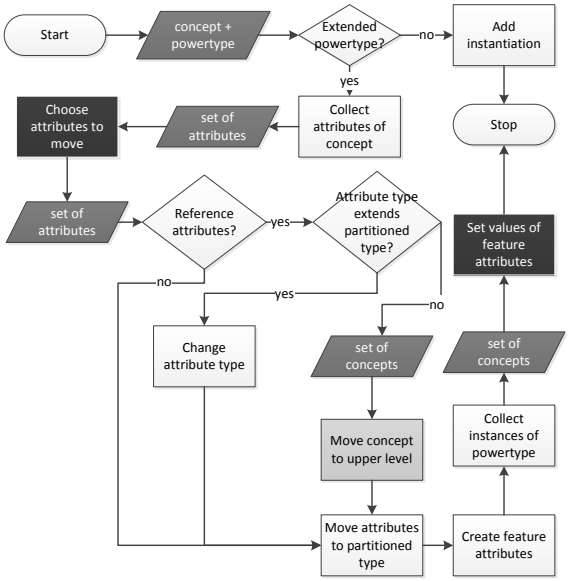

Figure 5. Add-Instantiation-To-Powertype operator process

If so, the attribute type has to be changed to the partitioned type. Otherwise, the referenced concept has to be put one level up because relationships cannot cross levels. Hence the Move-Concept-To-Upper-Level operator is called.

Now the operator can move all selected attributes to the partitioned type.

Afterwards, the operator creates corresponding feature attributes in the powertype for the moved attributes. Then, the operator collects all instances of the powertype and the user chooses those ones which should inherit the moved attributes.

Finally, the operator sets according to the user selection before the feature attribute values of all powertype instances. Of course, the value of the feature attributes for the given concept is set to true (since this concept declared the attributes before).

VI. EXAMPLE

In this section we give an example for the application of the Create-Powertype-For operator. The example shows a simple feature model of a car product line inspired by [3]. This simple feature model gives the opportunity to model Features and link them with the help of Associations together.

Figure 6 shows the complete meta-model stack. Therein M1 is the meta-model for M0. On M1 there are two concepts: Feature and Association. Each Association element connects one Feature element as source and zero or more Feature elements as target. On the other side, Features can refer to zero or one Association. Thus, this relationship is bidirectional.

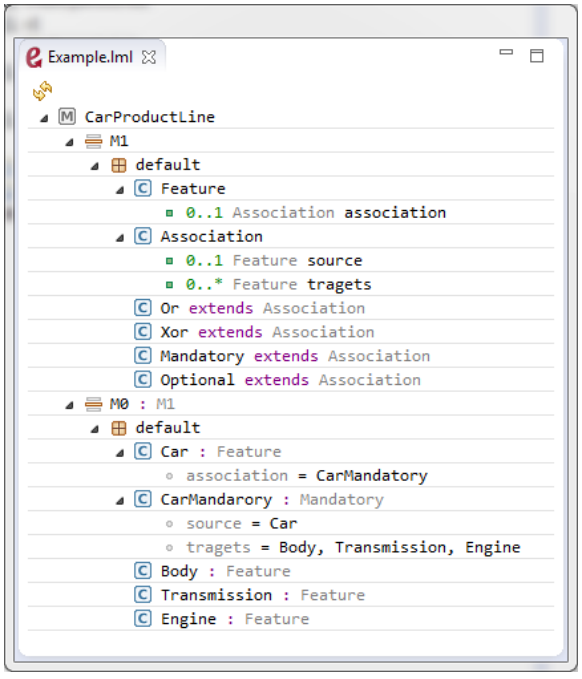

Figure 6. Car product line model

Furthermore, the concept Association is specialized in form of the concepts Or, Xor, Mandatory and Optional. Xor and Or can be used to express that at least one of several target features have to be selected. Instances of Optional can set a target whereas instances of Mandatory have to select a target.

Based on M1, there is a model M0 that declares four features (Car, Body, Transmission and Engine) and one association (CarMandatory). These features are linked together with the association so that following constraint is expressed: A car must have a body, an engine and a transmission.

A. *Application of the operator*

Now, we apply the Create-Powertype-For operator to the above introduced model. The result is shown in Figure 7.

First, we select the concept Association and invoke the operator on it. The operator uses the given concept and collects all its specializations since these concepts are candidates for instances of the future powertype. The outcome of this step is a set of four concepts: Or, Xor, Mandatory and Optional.

Afterwards, we have to review this set and tell the operator which concepts will become instances of the future powertype. In our example, we choose all of them. Then, the operator checks all selected concepts if they were instantiated before. This is true for Mandatory. Thus, the operator has to move the future powertype to the upper level and invokes the corresponding operator.

Hence, the concept Association is delivered to the Move-Concept-To-Upper-Level operator.

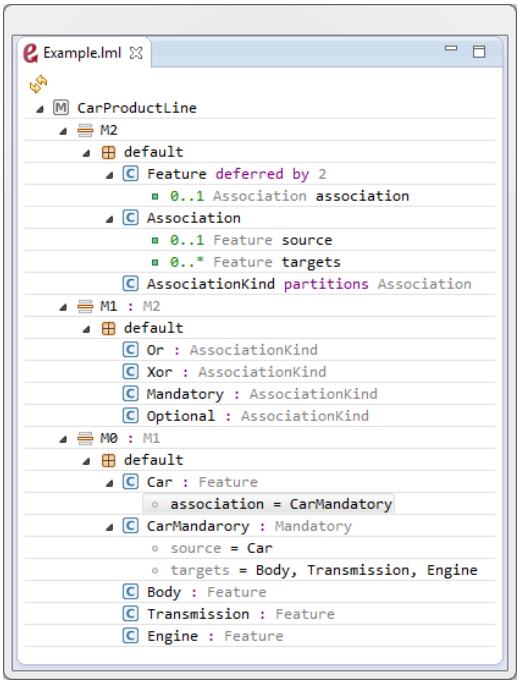

Figure 7. The resulting car product line model after application of the operator

 ISBN: 978-1-61208-276-9

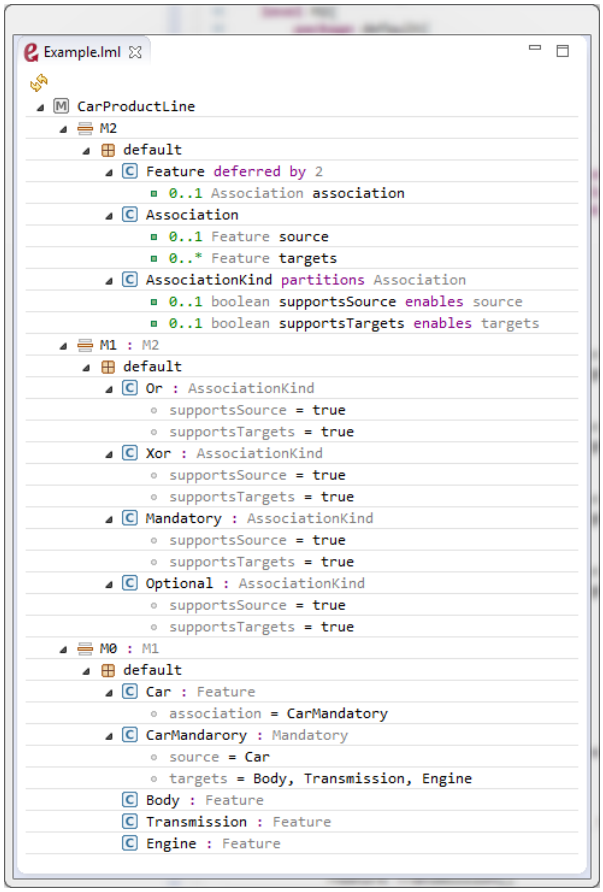

Figure 8. Result of the operator with extended powertype

After that, the operator checks if an upper level exists which is false. That's why it creates a new level that we name M2. Then the operator changes the level of Association to M2. After that, the specialization relation of Or, Xor, Mandatory and Optional is deleted. As Association is not instantiated, no deep instantiation counter has to be changed.

In the next step all related concepts are collected by the operator, which is only Feature (relationship to and from Association). Thus, the operator Move-Concept-To-Upper-Level is again called for Feature.

Because M2 already exists no meta-level has to be created. Subsequently, the meta-level of Feature is changed to M2 and the deep instantiation counter is incremented as it is instantiated in form of Car, Body, Transmission and Engine. Hence, the deep instantiation counter of Feature is now *2* (shown after the keyword *deferred by* in Figure 7). Since Feature has no related concepts because Association is already visited, the Move-Concept-To-Upper-Level operator terminates.

Afterwards, the Create-Concept-For-Powertype operator starts again with creating a new concept that we name AssociationKind and setting the partitions relation to Association.

If we decide to create a "simple" powertype Or, Xor, Mandatory and Optional just become instances of AssociationKind.

Otherwise, the operator collects for each concept (Or, Xor, Mandatory and Optional) all declared attributes. Since none of the concepts have attributes no user selection is needed and no reference attribute is part of the selection.

The operator continues with the creation of the feature attributes for targets and source (supportSource, supportTargets). Since Or, Xor, Mandatory and Optional were specializations of Association the feature attribute values for all concepts are set to true.

The result of creating an extended powertype is shown in Figure 8.

VII. CONCLUSION

Nowadays, meta-modeling is an often used approach for developing a domain specific language. Since these languages evolve during modeling of the domain of interest it is important to support this evolution to avoid manual migration of models.

Current research has discovered several patterns helping to improve the quality of (meta-) models [3]. Unfortunately, a remodeling of a meta-model to such a pattern is not supported today.

Facing this challenge, we presented in this article an operator that allows introducing a powertype pattern into an existing meta-model hierarchy considering migration of invalid models.

Currently, we have developed an Eclipse-based editor that supports several basic evolution operators like creating levels, packages, concepts and attributes. Furthermore some complex operators like the presented Create-Powertype-For, the Move-Concept-To-Upper-Level and the Add-Instantiation-To-Powertype operator are implemented as well.

In future work we will present complex evolution operators that support other language patterns like deep instantiation [28], materialization [17] or instance specialization [21]. Furthermore, we envision providing a preview of evolution operators similar to refactoring previews in modern IDEs. With the help of these previews, users can compare possible evolution steps.

REFERENCES

[1] H. Cho, "A demonstration-based approach for designing domain-specific modeling languages," *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 51–54, 2011.

[2] B. Henderson-Sellers and C. Gonzalez-Perez, "The rationale of powertype-based metamodelling to underpin software development methodologies," *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*, vol. 43, pp. 7–16, 2005.

[3] B. Neumayr, M. Schrefl, and B. Thalheim, "Modeling techniques for multi-level abstraction," *The evolution of conceptual modeling*, pp. 68–92, 2011.

[4] C. Gonzalez-Perez and B. Henderson-Sellers, "A powertype-based metamodelling framework," *Software and Systems Modeling*, vol. 5, pp. 72–90, 2006.

 ISBN: 978-1-61208-276-9

[5] J. Odell, “Power types,” *Journal of Object-Oriented Programming*, vol. 7(2), pp. 8–12, 1994.

[6] A. Demuth, “Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking,” *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 452–455, 2011.

[7] M. Herrmannsdoerfer, S. Benz, and E. Juergens, “COPE-automating coupled evolution of metamodels and models,” *European Conference on Object-Oriented Programming*, pp. 52–76, 2009.

[8] M. Herrmannsdoerfer and M. Koegel, “Towards a generic operation recorder for model evolution,” *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pp. 76–81, 2010.

[9] D. Di Ruscio, “What is needed for managing co-evolution in MDE?,” *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, pp. 30–38, 2011.

[10] [P. A. Bernstein and S. Melnik, “Model Management 2.0: Manipulating Richer Mappings,” *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 1 – 12, 2007.

[11] B. Gruschko, D. S. Kolovos, and R. F. Paige, “Towards Synchronizing Models with Evolving Metamodels,” *Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.

[12] M. A. Aboulsamh and J. Davies, “A Metamodel-Based Approach to Information Systems Evolution and Data Migration,” *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances*, pp. 155–161, 2010.

[13] M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth, “An extensive catalog of operators for the coupled evolution of metamodels and models,” *Software Language Engineering*, pp. 163–182, 2011.

[14] G. Wachsmuth, “Metamodel adaptation and model co-adaptation,” *European Conference on Object-Oriented Programming*, pp. 600–624, 2007.

[15] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, “An analysis of approaches to model migration,” *In Proceedings of Models and Evolution (MoDSE-MCCM) Workshop*, pp. 6–15, 2009.

[16] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth, “Language evolution in practice: The history of GMF,” *Software Language Engineering*, pp. 3–22, 2010.

[17] M. Dahchour, A. Pirotte, and E. Zimányi, “Materialization and its metaclass implementation,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 5, pp. 1078–1094, 2002.

[18] P. A. Bernstein, “Applying Model Management to Classical Meta Data Problems,” *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[19] M. Herrmannsdoerfer, S. Benz, and E. Juergens, “COPE: a language for the coupled evolution of metamodels and models,” *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management.*, 2008.

[20] C. Atkinson and T. Kühne, “Meta-level independent modelling,” *International Workshop on Model Engineering at the 14th European Conference on Object-Oriented Programming*, vol. 12, p. 16, 2000.

[21] B. Volz, “Werkzeugunterstützung für methodenneutrale Metamodellierung,” Dissertation, Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth, Bayreuth, 2011.

[22] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. Prentice Hall International; Auflage: 6th edition. Global Edition., 2010, p. 1155.

[23] Microsoft, “When to Use Inheritance,” 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/27db6csx(v=vs.90).aspx. [Accessed: 26-Sep-2012].

[24] G. B. Singh, “Single versus multiple inheritance in object oriented programming,” *ACM SIGPLAN OOPS Messenger*, vol. 6, no. 1, pp. 30–39, Jan. 1995.

[25] B. Zengler, J. Hahn, C. Rupp, M. Jeckle, and S. Queins, *UML 2 glasklar: Praxiswissen für die UML-Modellierung und -Zertifizierung*. München - Wien: Hanser Fachbuchverlag, 2007, p. 559.

[26] S. Jablonski, B. Volz, and S. Dornstauder, “On the Implementation of Tools for Domain Specific Process Modelling,” *International Conference on the Evaluation of Novel Approaches to Software Engineering*, vol. 4, pp. 109–120, 2009.

[27] B. Volz and S. Dornstauder, “Implementing Domain Specific Process Modelling,” *Communications in Computer and Information Science*, vol. 69, pp. 120–132, 2010.

[28] T. Kühne and F. Steimann, “Tiefe charakterisierung,” *Modellierung 2004 : Proceedings zur Tagung*, pp. 109–120, 2004.

[29] C. Atkinson and T. Kühne, “The essence of multilevel metamodeling,” *«UML» 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pp. 19–33, 2001.

[30] C. Atkinson, “Supporting and applying the UML conceptual framework,” *Lecture Notes in Computer Science, UML’98*, pp. 21–36, 1999.

 ISBN: 978-1-61208-276-9