Automated Model-to-Metamodel Transformations Based on the Concepts of Deep Instantiation

Gerd Kainz[1], Christian Buckl[1], and Alois Knoll[2]

[1] fortiss, Cyber-Physical Systems
Guerickestr. 25, 80805 Munich, Germany
{kainz,buckl}@fortiss.org
[2] Faculty of Informatics, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
knoll@in.tum.de

Abstract. Numerous systems, especially component-based systems, are based on a multi-phase development process where an ontological hierarchy is established. Solutions based on modeling / metamodeling can be used for such systems, but all of them are afflicted with different drawbacks. The main problem is that elements representing both CLAsses and oBJECTs (clabjects), which are needed to specify an ontological hierarchy, are not supported by standard metamodeling frameworks. This paper presents the combination of two approaches, namely deep instantiation and model-to-metamodel transformations. The resulting approach combines the clean and compact specification of deep instantiation with the easy applicability of model-to-metamodel transformations in an automated way. Along with this a set of generic operators to specify these transformations is identified.

Keywords: Model-to-Metamodel (M2MM), Model-to-Model (M2M), Model Transformation, Deep Instantiation, Transformation Operator, Clabject, Model-Driven Software Development (MDSD).

1 Introduction

Nowadays model-driven software development (MDSD) is widely used for the development of applications. Relevant tools are in general based on the modeling hierarchy as defined by the Object Management Group (OMG)[1]. The modeling hierarchy is shown in figure 1. It consists of four layers: M3 represents the meta-metamodel layer and describes the concepts used to define application specific metamodels. M3 contains very basic concepts such as classes and attributes. Hence, M3 is generic enough to describe itself and to terminate the modeling hierarchy. M2 defines application specific metamodels defining the application concepts and their related data. Based on the metamodels of M2, the application

[1] OMG: http://www.omg.org/

J. Whittle, T. Clark, and T. Kühne (Eds.): MODELS 2011, LNCS 6981, pp. 17–31, 2011.

developer can define models in M1, which conform to metamodels of M2 and are used to specify the application. M0 can be interpreted as the real world, which is represented by the models of M1.

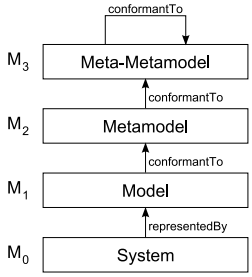

Fig. 1. Model Hierarchy [1] as Specified by OMG

The model hierarchy is well suited if an application can be described by using only the two modeling levels M2 and M1. However, for many systems, this assumption is not true. Examples are the UML specification [2] and the description of hardware components [3]. Here, some system elements have a dual role: in their first role they represent instances of a metamodel element; in their second role they constitute metamodel elements for other system objects. Elements with this dual role have been named clabjects (CLAsses and oBJECTs) by Atkinson [4].

Clabjects have been investigated intensively under various circumstances. The research resulted in many different solutions. Most of them try to find an adequate mapping of the problem to the existing modeling hierarchy. A totally different approach has been proposed by Atkinson and Kühne. In [2] they suggested to change the current instantiation model from shallow instantiation to deep instantiation. Deep instantiation allows that elements of a modeling level have an object and a class facet at the same time. Such a realization requires a fundamental change of the underlying modeling theory, leading to a clean way of describing this and other problems. Another approach based on model-to-metamodel (M2MM) transformations was presented in our previous work [3]. M2MM transformations are based on an iterative development process, where models of one phase (object facet) are transformed into metamodels (class facet) describing the models of the next phase. In our previous work, these M2MM transformations had to be implemented manually without any further support. The implementation complexity and therefore the effort increases drastically with each additional M2MM transformation phase.

This paper presents a combination of deep instantiation and M2MM transformations. The resulting approach combines the clean and compact description of deep instantiation with the easy applicability of M2MM transformations without having to change the underlying metamodeling framework. The automation of the M2MM transformations approach helps in applying this approach to similar

problems and reduces the time and effort for implementation. In addition, a set of generic M2MM transformation operators has been identified.

The remainder is structured as follows: Section 2 gives an overview of the problem. Related work is discussed in section 3. Section 4 contains a small motivating example. A detailed description of the suggested approach and the set of generic operators is the content of section 5. Details about the implementation and an evaluated based on a real-world use case is given in section 6. The content of the paper is summarized in section 7.

2 Problem Statement

Modeling languages are described in two orthogonal dimensions: a linguistic and an ontological dimension. The linguistic dimension specifies how a language is constructed and is typically represented by the different modeling levels (metaclass ← class ← object). The ontological dimension represents elements and their instance-of-relationship of a certain domain (e.g. *Component* ← *C : Component* ← *CI : C*), the so-called ontological hierarchy [5]. Since state-of-the-art modeling frameworks are based on the one-dimensional modeling hierarchy, these two dimensions cannot be represented adequately. This problem becomes obvious as soon as the ontological hierarchy spans more than two levels or can be changed / extended by the user [2]. Examples, where this problem arises, are component based systems where the user is able to define components and store them in a generic library. By doing so the user specifies a new component type which can be instantiated / copied later for use. Ptolemy II [6] and MATLAB/Simulink[2] include for example a library mechanism as described. Since they have no direct support for an ontological hierarchy integrated into their underlying programming model, it requires an enormous effort to emulate ontological support on top of their underlying programming model. Many other examples exist. A simple instance of that kind of problem is presented in figure 2.

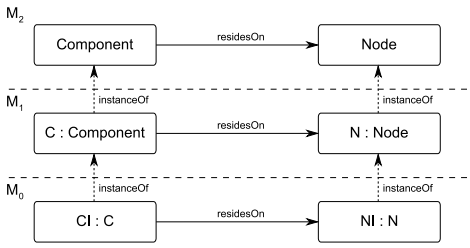

Fig. 2. Ontological Hierarchy Example Based on Components and Nodes [2]

The fundamental problem of expressing ontological hierarchies with current modeling systems is based on the duality of model elements, which is not supported by most metamodeling frameworks. Duality means that model elements

[2] MATLAB/Simulink: `http://www.mathworks.com/`

represent objects and classes at the same time. For example: in figure 2 *C* is an object of type *Component*, furthermore it represents the type of *CI*. This duality of model elements has been named clabject by Atkinson.

As model elements can represent types of other model elements, they construct their own ontological hierarchy introducing additional levels to the model hierarchy. Since these additional levels are not supported by standard metamodeling frameworks, the ontological hierarchy has to be folded into one level of the linguistic hierarchy. This leads to the problems of ambiguous classification and replication of concepts [2].

- Ambiguous classification: Model elements can be seen as both instances of their linguistic and ontological type, e.g. *C* has the linguistic type *Class* and the ontological type *Component*.
- Replication of concepts: As it is not possible to propagate attributes and associations over instantiation relations of the ontological hierarchy, the workaround is to replicate concepts, e.g. define class *Component* to represent *C*, *ComponentInstance* to represent *CI* and for both of them a separate *residesOn* association.

The goal of this paper is to propose an approach for a clean and compact description, which can easily be applied using state-of-the-art metamodeling frameworks. We start with a survey of existing solutions for the above mentioned problems and discuss of their strengths and weaknesses.

3 Related Work

The most promising approach in the context of the problem statement is deep instantiation introduced by Atkinson and Kühne [2, 7–9]. As the name says, the approach is based on a deep instead of a shallow instantiation mechanism, which is used in classical metamodeling frameworks. This enables the specification of model elements (classes, attributes, associations ...), which cannot only affect the direct underlying model level, but also other model levels underneath. To control the behavior of the deep instantiation mechanism, the concepts of *level* and *potency* are added to every model element. *Level* defines for each element at which model level in the hierarchy it resides. *Potency* on the other hand determines the number of times a model element can be instantiated. These extensions allow a compact specification of multi-level metamodeling. A first implementation of the deep instantiation mechanism called DeepJava[3] is available in the context of Java programming [7]. One major drawback of DeepJava is the missing support by integrated development environments (IDEs) supporting it. Furthermore, the definition of new ontological types requires that Java code has to be written by the developer. Therefore, this approach is only suited for software developers and cannot be directly applied by application users themselves. An advantage is that the whole ontological hierarchy is available and can be accessed at any time in the runtime system.

[3] DeepJava: `http://homepages.mcs.vuw.ac.nz/~tk/dj/`

To avoid the problem of missing IDE and metamodeling framework support, we previously suggested an approach based on M2MM transformations [3]. M2MM transformations are based on multi-phase metamodeling, where models of one phase are transformed into the metamodels of the next phase. This allows users to define new types, e.g. *ControlMotor : Component*, in a model. The subsequent M2MM transformation takes care that the corresponding type is created in the metamodel of the next phase. Each phase constitutes a modeling tool on its own. This means that by generating the metamodel of the next phase also the modeling tool of the next phase is altered to reflect the change of the underlying metamodel. As M2MM transformations are used to regenerate parts of the metamodels of the system based on the input data of the previous model, they do not need additional levels in the metamodeling hierarchy. This is both a strength and a weakness: on the one hand the metamodel only contains the information required in the specific phase, but on the other hand it is hard to determine the relationship between classes / objects at the different levels. Another drawback is the manual specification of M2MM transformations. The transformations must be encoded by the developer, who has to take care that all needed data is transformed according to the requirements of the succeeding phases. This can also imply that data has to be copied to guarantee that it is available in the following phases. If more than one M2MM transformations are executed in a row, it is very hard for the developers to implement those. With each additional M2MM transformation step, it gets harder to deal with the arising complexity of the transformations. The reason for the increasing complexity originates from the additional variability introduced with each new M2MM transformation. While the first M2MM transformation is based completely on a static metamodel, the dynamic part of the subsequent metamodels increases. The increasing complexity and the time consuming implementation of M2MM transformations make this approach very hard to apply. Furthermore, the transformation descriptions are encoded using a program language. This makes it hard to identify how the input model is transformed into the succeeding metamodel.

The power types concept of Odell [10, 11] constitutes another solution to integrate ontological hierarchies into the modeling hierarchy. A power type is defined to be a type whose instances are subtypes of another type. The relation between the power type and its instances is defined by a normal association. When working with power types this fact has to be considered. Furthermore, power types merely describe how to model an ontological hierarchy but offer no additional support for their handling.

Like power types, the prototypical concept pattern presented by Atkinson and Kühne [8] tries to solve the problem of ontological hierarchies within the modeling hierarchy by combining inheritance and instantiation. Compared to power types the prototypical concept pattern uses no normal association to connect the power type with the other type. Instead of the association, the instantiation mechanism of the modeling hierarchy is used. By doing so the number of levels in the modeling hierarchy is extended, which results in the already

mentioned problems regarding implementation using current metamodeling frameworks. Moreover the prototypical concept pattern offers no support for the handling of the introduced ontological hierarchy. Hence, it possesses no advantage compared to the deep instantiation approach and can be neglected.

Bragança and Machado [12] describe a similar approach to M2MM transformations supporting multi-phase modeling. In their work they use the term model promotion instead of M2MM transformation. Compared to the M2MM transformations approach where flexibility is provided in each transformation step, they can only specialize their initial metamodel by annotating models with information utilized for M2MM transformations. This restricts the power of their M2MM transformations to the predefined set of transformations offered through annotations. It also limits the usable domain concepts to the concepts introduced in their first metamodel and requires the specification of metamodel information in the model. An advantage is that the number of possible M2MM transformations is unbounded.

4 Motivating Example

As outlined in the previous section each of the presented solutions has different strengths and weaknesses. Therefore, we propose an approach combining the clean and compact description of deep instantiation with the easy applicability of M2MM transformations.

In this section we give a small motivating example based on figure 2. This shall help to better understand the automated M2MM transformations approach. The example is concerned with the definition of components residing on nodes containing various devices. Figure 3 shows the metamodel of the example at M2[4]. The superscript of the model elements presents the value of potency. The level is depicted at the model elements as subscript. The metamodel defines three classes *Component*, *Node* and *Device* and two associations between them. Additionally three instantiation operators (notes attached to classes) and one split field operator (note attached to association) are specified.

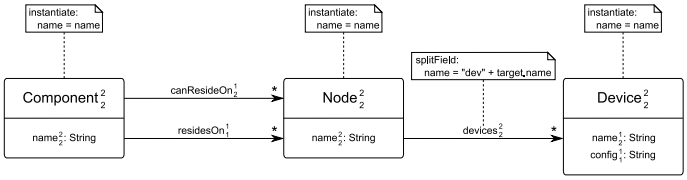

Fig. 3. Model Level 2 of Components-Nodes Example

Based on the metamodel shown in figure 3 the user is able to define the model presented on the upper part of figure 4. Applying an automated M2MM

[4] When we talk about metamodel, we mean the class facet of a clabject. We refer to the object facet by talking about the model.

transformation to the model results in the metamodel that is shown at the bottom of figure 4. As is visible in the metamodel the *ReadSensor*, *PC*, *CPU* and *Sensor* objects are transformed into classes. Furthermore, the *devices* association has been refined into a *devCPU* and *devSensor* association for the *PC* class.

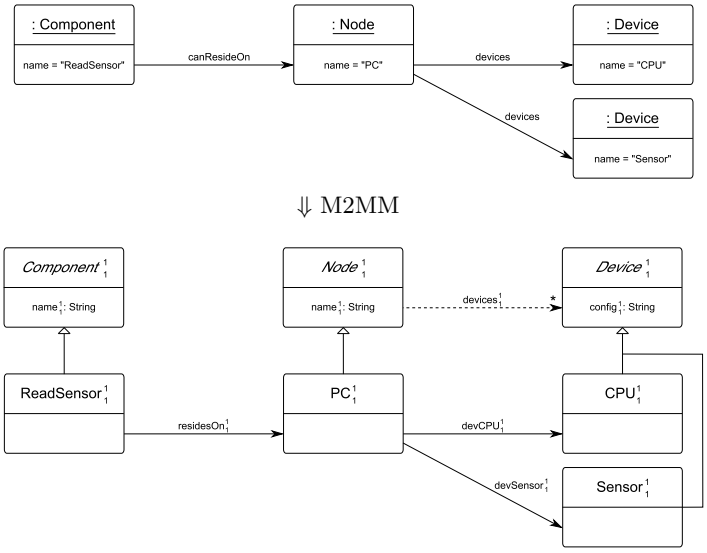

Fig. 4. Model Level 1 of Components-Nodes Example

The user can afterwards use the generated metamodel of figure 4 to define a model at modeling level 0. Such a model is displayed in figure 5, defining the component instance *ReadSensor1* and the node instance *Node1* with its *CPU* and *Sensor* devices.

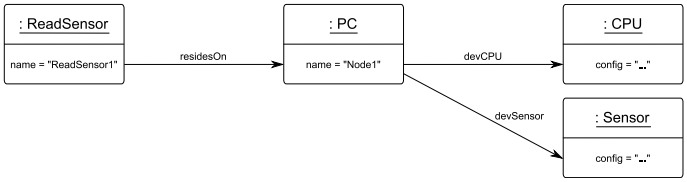

Fig. 5. Model Level 0 of Components-Nodes Example

5 Approach

The idea of this work is to integrate the concepts of deep instantiation in current metamodeling frameworks. To avoid a reimplementation of the metamodeling frameworks for a full support of deep instantiation automated M2MM transformations are used.

For simplification reasons, we will focus on *classes*, *attributes*, *references* (representing associations) and *operations* of the Essential Meta Object Facility (EMOF) [13]. As these are the main concepts, this presents no serious restriction.

Deep instantiation uses level and potency to define at which level a model element exists and how many times it can be instantiated. As this is a very clean and compact description to establish an ontological hierarchy, we adopted these concepts and extended the linguistic metamodel elements *classes*, *attributes*, *references* and *operations* with these attributes. This allows the definition of the basic properties to semi-automatically establish an ontological hierarchy. Additional to level and potency, operations are specified, which are applied during transformation. This information is used by the automated M2MM transformation to generate a metamodel out of an input model. During a M2MM transformation the model data is converted into a new metamodel predominantly by transforming objects into classes. Additional operations allow to steer the automated M2MM transformation and provide the missing information, e.g. the name of a new created class. All the available modification operators are defined later in this section.

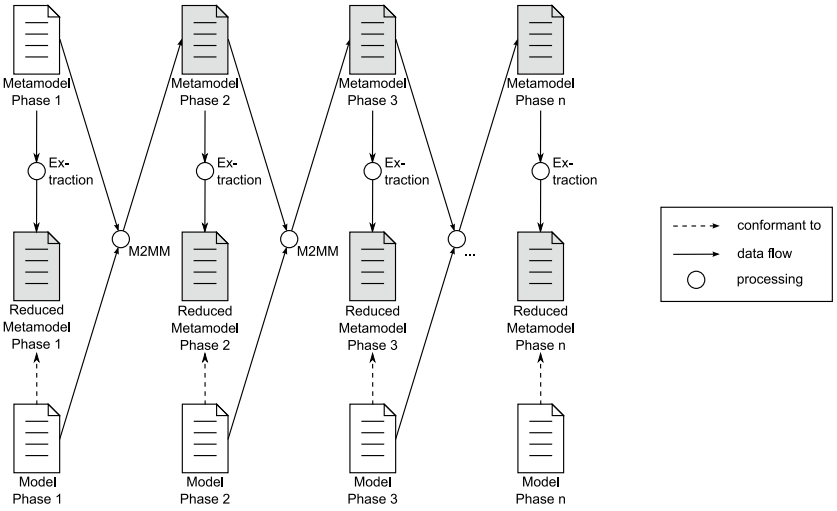

Fig. 6. Principle of the Automated M2MM Transformations Approach (All the Gray Metamodels are Generated)

Figure 6 shows how automated M2MM transformations work. The developer specifies the metamodel of the first phase including all the information needed for the automatic application of all following M2MM transformations. Afterwards a fully automated extraction step is conducted. During the extraction all model elements, which have no effect on the current model level, are eliminated. In short, these are all model elements with a level different to the current model level. This step is only included to ensure that existing metamodeling tools with

no support of level, potency and the additional specified operations are still able to handle the new kind of metamodels. If eventually all used metamodeling tools are able to cope with the additional information this step can be skipped. The models can then be defined based on a reduced metamodel of the current phase. Based on the specified model and the complete metamodel (not the reduced metamodel) the M2MM transformation is executed resulting in the complete metamodel of the next phase. A M2MM transformation affects mainly the model elements of the current model level with a potency value greater than 1. These model elements are converted into their corresponding model elements with level and potency reduced by 1[5].

Before going into details on how the algorithm of the automated M2MM transformation works, we will explain all available operators, which can be applied to model elements during a M2MM transformation.

5.1 M2MM Transformation Operators

To reuse common functionality between M2MM transformations we identified a set of various operators by analyzing the use case described in section 6. Since this use case is rather complex, we are quite confident that additional operators are not required. The set of M2MM transformation operators currently supported is: *instantiation*, *change property*, *split field*, *generate enumeration* and *execute*. Typically the operators work on model elements of the current model level with a potency greater than 1. These model elements are converted according to the operator specification into their corresponding model elements with level and potency reduced by 1.

To incorporate the user input during the application of the operators, the operators have access to the model data. By providing the operators with descriptions of how to process the model data to extract needed information, each operator can be adapted to concrete use cases. This makes the operators more flexible and generic. The descriptions are needed to automate the execution of the M2MM transformations.

In the following a comprehensive description of all operators is given. To facilitate the understanding of the M2MM transformation operators, examples for their application are given according to the motivating example of section 4.

Instantiation. The instantiation operator constitutes the main M2MM transformation operator. It is responsible for the transformation of objects into their own classes. Hence it implements the connection between the two facets of clabjects. Since M2MM transformations are based on an iterative definition process of the ontological hierarchy, which means that in each phase only one specific level of the hierarchy can be defined / manipulated, during the application of the instantiation operator the old class is transformed into the new metamodel

[5] Decrementing of level and potency logically happens when instantiating new model elements. As level and potency are not directly present during modeling this is done in the M2MM transformation afterwards.

representing the super class of its newly created sub classes. To prevent further manipulation of the super class it is automatically converted into an abstract class. For the fully automatic application of the instantiation operator a description of how to construct the names of the new sub classes out of the object data is needed. For example the component object *ReadSensor* is transformed into a component type *ReadSensor*, which is sub class of an abstract *Component* class.

Additionally the instantiation operator takes care of the transformation of all attributes, references and operations of the class. In cases where sub classes define different values for properties of a contained model element, the model element is moved into the sub classes. To prevent unnecessary type casts to access these elements when working directly with the object model an additional access operation is added to the super class.

Following is the operator definition. It takes as input a class specification, all instances of that class and a description for the calculation of the new sub class names and returns the transformed class and all new created sub classes.

```
instantiation (in class: Class, in instances: Set<Object>,
               in name: Description): Set<Class>
```

Change Property. The change property operator allows the adaption of model element properties. For example a new default value for the attribute *name* of class *PC* can be specified with *"PC" + Counter.getNextID()*, where the function returns the number of a running counter. Even the refinement of the data type of an attribute is possible. This operator is very generic and allows to adapt the next metamodel in a flexible way. As already mentioned at the instantiation operator, special attention has to be taken when properties of elements in sub classes are set to different values. In such cases the elements have to be dragged from the super class into all sub classes. To further support access to those elements based on the super type, access operations must be installed. Sometimes it makes sense to apply this operator on model elements which are not transformed by the M2MM transformation but are coming into life for the first time, e.g. if the value of an attribute can be changed depends on previous model data.

As can be seen from the definition below, the operator takes an identification of the property which shall be changed and a calculation description for the new value as input. To consider the model data for the new property value the corresponding object is given to the operator. The result of the operator is the adaption of the given model element according to the specification.

```
changeProperty (in property: PropertyKind, in value: Description,
                in instance: Object, inout element: ModelElement)
```

Split Field. A very interesting operator is the split field operator. Its task is to allow the refinement of associations between super class and sub classes. Imagine the following example: after specifying that nodes of type *PC* can have the devices *CPU* and *Sensor*, it should only be allowed to link nodes of type *PC* with devices of type *CPU* and *Sensor* but nothing else. As can be seen

from the example, this operator can establish very strong constraints on sub types. The additional constrains help preventing a lot of careless mistakes during model handling. Access to referenced object via the previous relation can still be ensured trough the definition of an access operation instead of the relation in the base type. The M2MM transformation can additionally take care of providing an appropriate realization for the access operation for each sub type.

The split field operator is realized in two separate parts. The first part is responsible for transforming the original reference of the super class into an appropriate operation. The second part takes as input the reference, a description of how to define the names of the new references, and a list of all the objects referenced by the object, which is going to be transformed in a sub class. A list containing all new references and the access operation including an appropriate implementation is returned.

```
splitFieldSuperClass (in reference: Reference): Operation

splitFieldSubClass (in reference: Reference, in name: Description,
                    in referencedObjects: List<Object>)
                   : List<ModelElement>
```

Backtrack. Since M2MM transformations introduce a cut between two succeeding phases, a backtrack operator is offered to get full access to the model data of previous phases. This operator is able to return the object belonging to a class, so it can be used to traverse the M2MM transformations in reverse order. It is not only available during M2MM transformations but can also be used when working directly with the object model of a phase. In the context of our example the backtrack operator applied to the type *PC* of *Node1* at M0 would result in the object defining *PC* at M1.

The definition of the backtrack operator takes a class as input and returns the related object in the model of the previous phase. The operator is only defined in the context of classes representing the class facet of a clabject with both object and class facet. The behavior for clabjects without any object facet or any other object is undefined.

```
backtrack (in class: Class): Object
```

Generate Enumeration. Generate enumeration is used to create new enumerations. It has been shown during the application of automated M2MM transformations that sometimes the user defines a list of allowed values for a type in one phase and wants to use the generated enumeration for an attribute in the next phase. This helps to assure that only valid values are assigned to the attribute. For example at M1 it could be possible to specify the valid operating systems for the node type *PC* in an additional field *os*. This list is then transformed into a new enumeration. The operating system running on node instance *Node1* can then be only selected among those values.

To create a new enumeration the operator takes a description of the enumeration name and all literals as input and returns the generated enumeration. The literals of the enumeration consist of a name value pair.

```
generateEnumeration (in name: Description,
                     in literals: Description): Enumeration
```

Execute. There will always be special cases, which are not foreseen. To support such situations an execute operator is available in automated M2MM transformations. This operator offers the highest flexibility to transform data according to special needs. In general all presented operators can be emulated using the execute operator. Through its high flexibility this operator can be used to implement highly specialized transformations in a M2MM transformation.

To offer its high degree of flexibility the execute operator gets as input the current class, the complete model and a description of the transformation to execute.

```
execute (in class: Class, in model: Object,
         in modification: Description,
         inout metamodel: List<Object>)
```

Operator Application Specification. The MOF has been designed with expandability in mind. For extensions annotations exists. They can be attached to all model elements. We make extensive use of annotations to specify all the operators with their corresponding data. The operator specifications are attached to the model elements, on which they shall be applied. As it is important to apply the operator during the right M2MM transformation, all the annotations specify the transformation to which they belong.

5.2 Automated M2MM Transformations Algorithm

After the introduction of the different transformation operators the M2MM transformation algorithm is explained in detail. The algorithm is parameterized with the metamodel containing all operator specifications and the model of a phase and returns the metamodel of the next phase. To simplify the transformation the algorithm consists of two parts.

During the first part all types are created. Therefore the *instantiation*, *generate enumeration* and *execute* operators are executed for model elements, whose level is equal to the model level of the next phase plus 1 and have a potency greater than 1. Additionally, all types belonging to a model level lower than or equals to the model level of the next phase are copied. In this process the effect of *change property* operators are incorporated. This part is only responsible for defining all types, but does not take care of their internal structure. By doing so problems of referencing not yet created types is effectively prevented.

The second part is responsible for the completion of the created and copied types. This includes the transformation and copy of all attributes, references and

operations. While transforming those model elements special cares have to be taken if for a model element different values are assigned to properties of sub classes created by the *instantiation* operator. In those cases the model elements are moved into the sub classes and an additional access operation is added to the super class. Additionally, the *execute* operators are processed again to finish their tasks.

After the completion of both steps the metamodel of the next phase is completely constructed. It contains a complete definition of the structure of the current and all following model levels. Model elements and specifications belonging to the previous model levels are completely removed.

5.3 Differences between Automated M2MM Transformations and the Two Original Approaches

Beside the transformation of models into metamodels, the specification of additional operations to provide the missing information for the automated M2MM transformations is a big difference compared to deep instantiation. In contrast to the deep instantiation approach, automated M2MM transformations rely on fully automatic creation of new types. Therefore the model data is taken and all needed information is extracted through specified operations. Thus the user does not need to know how to define new types in the metamodel or programming language. The only knowledge needed is how to insert correct model data. The creation of new types is then automatically conducted during M2MM transformations. This relieves the user from knowing how to modify a metamodel or program and helps him to concentrate on the ontology specification via modeling.

Compared to deep instantiation, potency has a slightly different meaning in the context of automated M2MM transformations. In the context of deep instantiation, potency specifies how many times a model element can be instantiated. This fact can be utilized to define abstract elements at the metamodel level with a potency value of 0, which makes the abstract flag obsolete. For automated M2MM transformations this additional utilization is not allowed, because M2MM transformations rely on potency for defining how many times a model element can be instantiated or copied, if it is abstract. Copying model elements is necessary as in contrast to deep instantiation not the complete ontological hierarchy is available for direct access at a specific model level. This requires that parts of the ontological hierarchy are copied to succeeding model levels as needed.

Deep instantiation also defines the concept of simple and dual fields. Field is the generalized term unifying attributes and references on the metamodel level and slots on the model level. A simple field is defined to be a field, which takes only a value when its potency is 0. In contrast a dual field can have a value for each model level. In the context of M2MM transformations it has been shown that the explicit distinction between simple and dual field makes no sense. The distinction is implicitly achieved through the specification of level and potency. Level defines the model level in which the field exits. In cases where the level

number is lower than the number of the current model level, the field can be treated as nonexistent. Potency on the other hand specifies how many times the field shall get a value. Through the assumption that an existing field can get a value, the distinction between simple and dual fields is no longer needed. In cases where the assignment of values shall be delayed to a later model level the level can be set accordingly.

As automated M2MM transformations can be seen as an improvement of the M2MM transformations approach the only difference between those two approaches lies in the automation of the transformations. Through the definition of transformation operators the developer is relieved from programming the whole transformation. By using automated M2MM transformations large parts of the transformation can be executed automatically based on the specification of level, potency and the operators to apply.

6 Implementation and Evaluation

A first implementation of the presented approach is available based on the Eclipse Modeling Framework (EMF) [14]. This implementation has been used to demonstrate the usefulness of the approach on the example presented in our previous work [3][6]. The application of the automated M2MM transformations approach on this example resulted in a much simpler and more compact system description. Furthermore, the original three phase approach could be enhanced with an additional fourth phase, to define the different capability types. It also turned out that the new approach simplified the M2MM transformations. Most of the M2MM transformations are described using 43 standard operators (*instantiation* 9, *change property* 29, *split field* 2, *backtrack* 1 and *generate enumeration* 2). Only a special transformation had to be implemented with an *execute* operator.

7 Conclusion

In this paper we presented a combination of the deep instantiation and the M2MM transformations approach. The resulting approach uses the clean and compact description of the deep instantiation to automate the M2MM transformations approach. By combining these approaches main drawbacks of the original approaches are eliminated. The automated M2MM transformations approach does not require a fundamental change of the underlying metamodeling framework. All known and used modeling tools are further utilizable. In addition, the time consuming manual implementation of M2MM transformations is replaced by a clean and compact specification of transformation operators. The M2MM transformation operators support the developer in all transformation cases. For unsupported transformations a generic execute operator exists.

[6] Due to space limitations it is not possible to go into details about the example. Interested readers can refer to [3] for more information.

Furthermore, we introduced and presented a set of generic M2MM transformation operators. The operators are used to guide the M2MM transformations and provide the transformation with all needed information to ensure an automatic execution.

Finally a prototype of the automated M2MM transformations approach has been implemented for EMF and its usefulness has been demonstrated in the context of a real world example.

References

1. Bézivin, J.: In search of a basic principle for model driven engineering. UPGRADE-The European Journal for the Informatics Professional 5(2), 21–24 (2004)
2. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 19–33. Springer, Heidelberg (2001)
3. Kainz, G., Buckl, C., Sommer, S., Knoll, A.: Model-to-metamodel transformation for the development of component-based systems. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6395, pp. 391–405. Springer, Heidelberg (2010)
4. Atkinson, C.: Meta-modeling for distributed object environments. In: Proceedings of the 1st International Conference on Enterprise Distributed Object Computing, EDOC 1997, Washington, USA, pp. 90–101 (1997)
5. Atkinson, C., Kühne, T.: Model-driven development: A metamodeling foundation. IEEE Software 20(5), 36–41 (2003)
6. Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y.: Taming heterogeneity - the ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (2003)
7. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style: multi-level programming with deepjava. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications. OOPSLA 2007, Montreal, Canada, pp. 229–244 (2007)
8. Atkinson, C., Kühne, T.: Processes and products in a multi-level metamodeling architecture. International Journal of Software Engineering and Knowledge Engineering 11(6), 761–783 (2001)
9. Gutheil, M., Kennel, B., Atkinson, C.: A systematic approach to connectors in a multi-level modeling environment. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 843–857. Springer, Heidelberg (2008)
10. Odell, J.: Power types. Journal of Object-Oriented Programming 7(2), 8–12 (1994)
11. Martin, J., Odell, J.J.: Object-oriented methods: a foundation, UMLed, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)
12. Bragança, A., Machado, R.J.: Transformation patterns for multi-staged model driven software development. In: Proceedings of the 12th International Software Product Line Conference, SPLC 2008, Washington, USA, pp. 329–338 (2008)
13. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification Version 2.0 (January 2006)
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley, Reading (2009)