A Generic Approach Simplifying Model-to-Model Transformation Chains

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, Technical University Munich,
Boltzmannstr. 3, 85748 Garching, Germany
knoll@in.tum.de

Abstract. The model-driven architecture proposes stepwise model refinement. The resulting model-to-model (M2M) transformation chains can consist of many steps. For realizing the transformations two approaches exist: Exogenous transformations, where input and output use different metamodels, and endogenous transformations, that use the same metamodel for input and output. Due to the particularities of embedded systems, using only endogenous transformations is not appropriate. For exogenous transformations, problems arise with respect to creation and maintenance of the subsequent metamodels. Another problem of these M2M transformation chains is that for one transformation step typically large parts of the model data remain unchanged. The resulting M2M transformation does often include many copy operations that distract the developers from the "real" transformations and increase implementation overhead. This paper introduces a generic approach that solves these issues by a (semi-) automatic metamodel construction and copy operation of unchanged model data between subsequent steps.

Keywords: Transformation Chain, Model-to-Model Transformation, Metamodel-to-Metamodel Transformation, Model-driven Software Development, Model-driven Architecture.

1 Introduction

The model-driven architecture (MDA) [1] has been successfully used to cope with large and complex systems. MDA suggests transforming platform independent models (PIMs) by a series of model-to-model (M2M) transformations into platform specific models (PSMs). Especially in the context of model-driven software development (MDSD) [2] of embedded systems, this stepwise refinement is very helpful. Embedded systems are characterized by the importance of extra-functional requirements, timing issues, and the heterogeneity of the involved components and platforms. Therefore, the transformations from PIM to PSM have to take into account several tasks. To enhance readability and maintainability, every M2M transformation step should ideally perform one task.

These tasks could be for example the mapping of software to hardware components or the calculation of an execution schedule.

In the domain of embedded systems the metamodels used for user input and for code generation very often differ significantly [3]. Due to this difference, a big metamodel, whose structure is suited both for user input and code generation, would be inadequate. The ideal transformation process consists of a series of refinements resulting in intermediate models based on different metamodels. In this approach of handling only one task within one M2M transformation step, the changes between steps at the metamodel and model level are rather small and only represent intermediate steps of the transformation towards the final metamodel and model.

One problem of M2M transformation chains is the creation and maintenance of the related metamodels. Successive metamodels typically have large parts in common. As there is currently no tool support available for constructing these metamodels, they are created manually, typically using Copy&Paste. An additional problem arises if later a metamodel in a M2M transformation chain is changed, e.g., by adding a new attribute. Usually the same adaptation has to be applied to subsequent metamodels as well. A manual execution of such changes is error-prone and tedious, hence should be avoided. Very often these problems are avoided by reducing the number of steps in a M2M transformation chain. This paper presents an approach to create and maintain the metamodels based on difference descriptions.

If transformations between models with different metamodels (exogenous transformation) [4] are implemented using an operational M2M transformation language, such as Xtend[1] or QVTOperational [5], the unchanged parts of the system have to be copied manually. As a result, the size of the code for the "real" M2M transformation is very often negligible compared to these manual copy operations. To avoid the additional overhead for implementing these copy operations, different refinement steps are very often combined or even only one large M2M transformation is used. Such M2M transformations contradict state-of-the-art in modern software engineering, which is based on modularity and demands to focus on one task at a time. Therefore, this paper presents a way to deal with the copy of data between successive models.

Section 2 clearly defines the problem statement and the focus of this paper. An overview of our solution is given in Section 3. The approach supports both the creation and maintenance of M2M transformation chains with respect to the two above mentioned problems. Section 4 presents an incremental definition of subsequent metamodels on the metamodel level. A semi-automatic conduction of data copy and type transformation operations for unchanged parts[2] between models based on different metamodels is described in Section 5. The "real" M2M transformation still needs to be specified manually. The implementation and the evaluation of the approach in the context of two MDSD tools for embedded

[1] Xtend/Xpand: `http://wiki.eclipse.org/Xpand`

[2] Underlying metamodel structure has not changed.

systems are contained in Section 6. The paper is concluded by a discussion of related work in Section 7 and a summary in Section 8.

2 Problem Statement

Following the MDSD methodology [2] user-defined models are stepwise combined and refined to a model adequate for code generation. This is especially useful when applying MDSD in the domain of embedded system. Due to the high heterogeneity of platforms and hence of implementations, PIMs abstract from the underlying implementations to simplify the modeling task for developers. In the process of stepwise refinement, a PSM should be calculated that is an optimal representation for code generation of a specific platform. One example is schedule specification. While it is simpler to model the execution through dependencies between tasks, the code generation is simplified if a concrete schedule with start times is calculated during M2M transformations. The same is true for the combination of models. To separate concerns and to reduce complexity, the description of embedded system is very often done using several, aligned models targeting different aspects of the system. Aligned models are models, which were created with respect to each other. They share information and can reference elements of each other without any problems. By working with different models, the developers can concentrate on selected system aspects and their associated data. Examples can be a model describing the used hardware and a model to describe the application. The code generation is simplified when these different "views" are merged.

Ideally the M2M transformation between the input model(s) and the final output model is split up in many small transformations. Each of these transformations then focuses on one task, e.g., schedule calculation or identifier assignment. Hence, a transformation changes only a small part of the model data. The transformations in the chain can be implemented using exogenous or endogenous transformations [4]. Exogenous transformations are transformations between models based on different metamodels. In contrast, endogenous transformations are transformations between models based on the same metamodel. It is possible to perform transformations with big structural changes through endogenous transformations. However, due to the big difference between PIMs and PSMs for embedded systems, the use of endogenous transformations alone is usually not advisable. Hence, this paper focuses on the **support of step-wise model refinement using exogenous transformations**[3].

One problem with exogenous transformations is the necessity to create and maintain further metamodels with large common parts. Very often the similarity between these metamodels leads to a construction using Copy&Paste. During maintenance, problems can arise when metamodels are extended and adapted to new needs. This usually requires applying the same changes in subsequent metamodels. Depending on the length of the M2M transformation chain this can be

[3] The problems with exogenous transformations discussed in the following, do not exist for endogenous transformations.

very time-consuming. Moreover, the refactoring is tedious and error-prone. Figure 1 shows the structure of a M2M transformation chain from PIM to PSM. Metamodel evolution [6] is very similar as it considers the migration of models, after the corresponding metamodel has been changed. The major difference between a M2M transformation chain and metamodel evolution is the life cycles of metamodels. In metamodel evolution, only the latest metamodel is of concern as this metamodel presents the latest version of the tool. The migration of models is only performed once. In transformation chains, all metamodels are required and the full chain of metamodels is processed every time the tooling is invoked for an application. This difference causes some practicability issues that are discussed in the following sections. M2M transformation chains and metamodel evolution are orthogonal to each other as shown in Figure 1. This paper proposes an approach for **creating and maintaining metamodels in exogenous M2M transformation chains based on difference specifications between metamodels**.

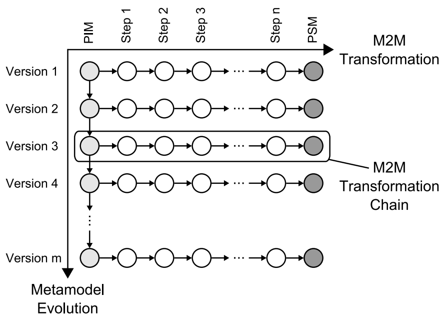

Fig. 1. Relation between Transformation Chains and Metamodel Evolution

Another disadvantage of exogenous transformations is the need to transform all the data of input model(s) into the output model. This transformation also includes copying data, which is not modified by the current M2M transformation, but needs to be transformed into the namespace of the new model. For operational (imperative) M2M transformation languages these copy operations have to be specified by the developers for all model elements. This is a very time-consuming and tedious job. Furthermore, the developers have to ensure that all data are copied from the input model(s) to the output model. The resulting M2M transformation code is very often a mixture of copy and "real" M2M transformation operations. As a consequence, these copy operations hinder the identification of the essential parts and ideas of the M2M transformation itself. To avoid the additional overhead, different M2M transformation steps are often combined or even only one large M2M transformation is used. This paper proposes an approach to **(semi-) automatically copy unchanged parts between models through the use of a function library**.

3 General Approach

The problem statement targets both the metamodel and model level of M2M transformation chains. To simplify discussion, we will deal with each of the problems in a separate section. As model transformations are based on the metamodels created by metamodel transformations, they can only be executed after the metamodel transformations. Hence, we start with the discussion of metamodel-to-metamodel (MM2MM) transformations. Typically, the transformations on models are executed more often than transformations on metamodels. The reason for this is that metamodel transformations belong to a change in the tool, whereas model transformations are part of the tool application to create new applications. Performance is therefore mainly an issue for M2M transformations and can be neglected to a certain extent for MM2MM transformations.

Figure 2 shows the proposed approach. The models / metamodels of the different steps are connected through transformations belonging to a M2M transformation chain. Numbers indicate the designated order of steps. The developers start with defining the input metamodels (1). Afterwards a difference model (2) [7,8,9] is used to create the metamodel of the next step (3). Based on this new metamodel the developer can define the model transformation containing function calls to copy unchanged model data and the "real" transformation (4). Since the "real" transformations represent the intelligence of the tool, they still have to be implemented by the developers without any further support. These steps can be repeated as often as needed (5–8). To create a new application the user needs to define the required models (9) and start the processing (10–12). In the approach only the differences between steps are specified manually. Similarities are handled automatically.

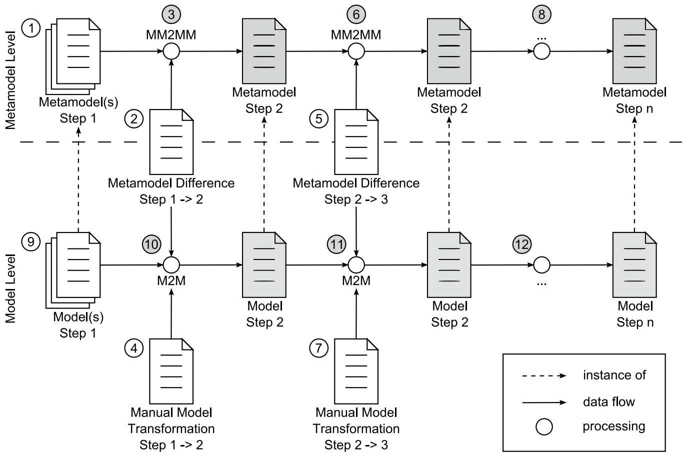

Fig. 2. Schematic Illustration of Model-to-Model Transformation Chain Approach. Gray elements indicate generated artifacts and automatic steps.

4 Metamodel-to-Metamodel Transformations

For supporting M2M transformation chains on the metamodel level, an easy way of creating the metamodels used in the different steps is needed. To keep the overhead for managing the metamodels as small as possible, we suggest specifying only the changes between metamodels (model deltas), e.g., adding, deleting, or modifying of packages, classes, attributes, references, or operations. This is closely related to metamodel evolution. The main difference is that metamodel evolution usually only focuses on calculating a difference model to (semi-) automate the M2M transformation (model migration). The difference model is either calculated by tracking changes of the developers when changing the metamodel or by comparing the old and new metamodel. In contrast, we use the difference model to calculate a successor metamodel out of the given ones. It is important to note that the predecessor metamodels might be affected by changes to the preceding metamodel chain. Hence, the tool must also support the developers by notifying if a difference model becomes partly invalid. In addition, our difference model must be able to specify combinations (used to merge different views) and adaptations of more than one metamodel, whereas metamodel evolution can only relate two metamodels with each other.

Example. Before the approach is presented in detail, a simple example is given. A system consisting of hard- and software components is modeled using separate models (views). These models shall be merged and then further modified. A suitable metamodel is needed to store the newly calculated data. Therefore, the metamodels have to be merged into one metamodel. The merge of the two metamodels raises a conflict as both contain a class *Component*. To resolve the conflict the *Component* classes are renamed into *HWComponent* respectively *SWComponent*. The commonalities of the classes are moved into a new abstract *Component* class. In addition, an *id* attribute is added to give all *Component*s unique identifiers. Figure 3 shows on the left side the preceding metamodels and on the right side the newly generated metamodel. The figure also depicts an intermediate step, which will be described in the discussion of the algorithm.

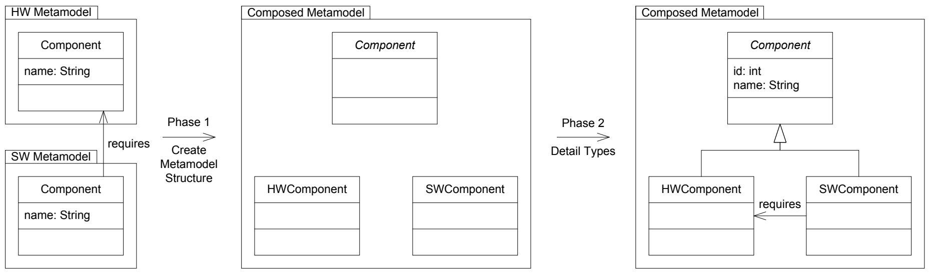

Fig. 3. Simple Example of a Metamodel-to-Metamodel Transformation

To create a subsequent metamodel the developers only need to specify the input metamodels and the changes, which shall be applied. A tool then creates the new metamodel. Figure 4 shows the difference model used to create the new metamodel. Any number of difference models can be specified to create an arbitrary number of subsequent metamodels, where one MM2MM transformation with its difference model builds upon the result of the previous one.

```
Transformation: composed (www.fortiss.org/tooling/m2m/composed)
  Metamodel: hardware (www.fortiss.org/tooling/m2m/hardware)
    Class: Component -> HWComponent [super class = Component] => Modify
      Attribute: name => Delete
  Metamodel: software (www.fortiss.org/tooling/m2m/software)
    Class: Component -> SWComponent [super class = Component] => Modify
      Attribute: name => Delete
  Class: Component [abstract] => Add
    Attribute: id [int] => Add
    Attribute: name [String] => Add
```

Fig. 4. Difference Model Used to Create the new Metamodel of the Simple Example

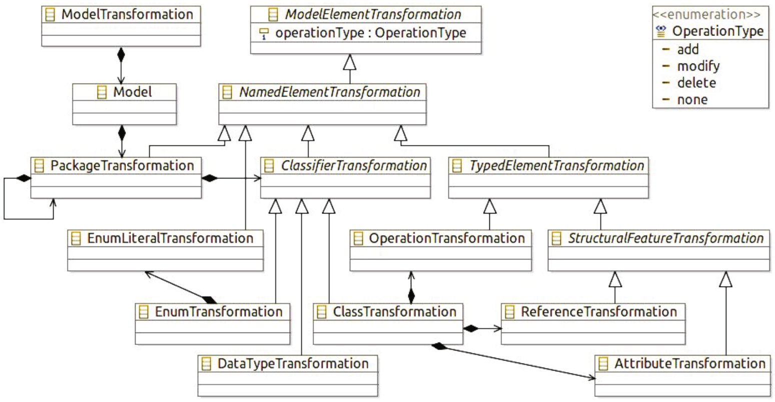

Fig. 5. Simplified Metamodel to Specify Metamodel Transformations. *ModelTransformation* constitute the root element. *OperationType* defines the kind of operation to perform, where *none* is used if only child elements are affected by transformations.

Supported Operations. To specify the adaptations of the metamodels in an unambiguous way, we provide a metamodel for specifying the difference model for MM2MM transformations. Figure 5 shows the basic structure of the metamodel used to specify MM2MM transformations. Based on this metamodel, it is easy to define all changes. The difference model allows to specify adding, deleting, and modifying of metamodel elements. The low level specification of changes gives high flexibility and allows a fine grained transformation of metamodels. Figure 6 shows a table with all supported operations. The column *Add To / Remove From* state where the elements can be added / deleted and *Modify* contains a list of changeable properties.

Element	Add To / Delete From	Modify
Package	package	name
Class	package	name, abstract class, super class
Attribute	class	name, type, multiplicity
Reference	class	name, type, multiplicity, containment
Operation	class	name, return type, parameters
Enumeration	package	name
Enumeration literal	enumeration	name, value
Data type	package	name, instance type name
Annotation	element	key, value

Fig. 6. Supported Metamodel Transformations

Transformation Algorithm. For the MM2MM transformation a difference model is taken as input, which references the metamodels of the previous step and includes a specification of all changes to apply. As result an adapted and potentially combined metamodel is returned. In addition, the specified changes are checked for consistency to cope with potential changes to the preceding metamodels. This prevents the generation of inconsistent metamodels, e.g., metamodel containing classes with same name or usage of not existing data types.

The actual transformation is carried out in two phases. In the first phase, all packages and types (classes, enumerations, and data types) are created. For each package and type the algorithm checks, whether it is not specified to be deleted, before creating them in the new metamodel. This is done for all packages and types of the referenced input metamodels. New packages and types are created in addition. The first phase takes care of creating types incorporating type renamings, without creating the internal structure of classes. By executing the transformation in this way, it can be ensured that all types already exist before they are used by other metamodel elements, e.g., data type of an attribute, super class. After creating all types in the new metamodel, a second phase takes care of the correct construction of the internal structure of classes. This includes the creation of attributes, references, and operations. Assignment of super classes is also part of this second phase.

Due to the fact that only changes between subsequent metamodels are specified through a difference model, changes to metamodels are automatically propagated to all subsequent metamodels along the M2M transformation chain. The implicit propagation of changes along the M2M transformation chain relieves the developers from applying the same adaptation many times and helps to focus on the differences between consecutive metamodels. Furthermore, careless mistakes are avoided by automating the metamodel adaptations.

5 Model-to-Model Transformations

The second aspect of M2M transformation chains targets M2M transformations themselves. The focus of this paper lies on removing the burden of writing data copy operations in operational transformation languages from developers. This is achieved by copying all unchanged (unaffected) model data from the input models to the output model. The developers can then fully concentrate on the "real" transformation. In comparison to metamodel evolution, our focus lies not on specifying the complete M2M transformation, so we split the M2M transformation in a generic part realizing the copy of unchanged data and a manual part. The developers still have to write the code for the "real" transformation. Furthermore, our approach supports the combination of more than one model describing a system from different viewpoints. In addition, we want to reuse the information from the MM2MM transformation to perform a more comprehensive M2M transformation considering exogenous transformations including renaming of metamodel elements.

Example. The approach is illustrated in Figure 7 based on the example of the previous section. The transformation consists of a (semi-) automated and a manual phase. First, generic copy operations based on the difference model are invoked to copy as much data as possible to the successor model. Afterwards, a manual transformation specified by the developers calculates the values for the new id attributes.

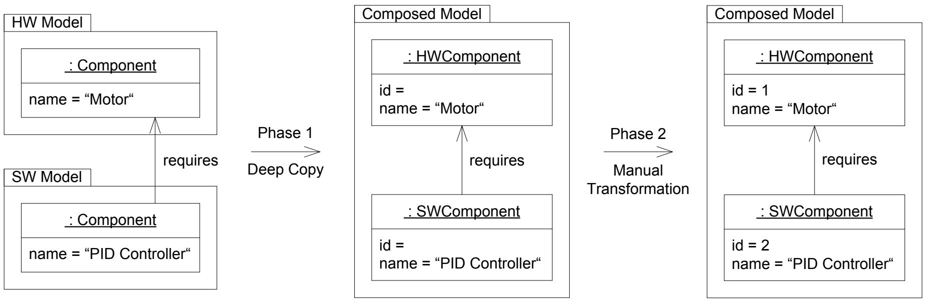

Fig. 7. Simple Example of a Model-to-Model Transformation

To reduce the effort for transformation encoding, all unchanged data between the steps are copied through calls of library functions. The library function *transformObject* takes care of converting the objects between the different namespaces – a deep copy is performed. Through the information contained in the difference model of the MM2MM transformation, the function is also capable of handling renamings of classes, attributes, or references, e.g., class *Component* $\rightarrow$ *HWComponent*. Thus, the developers can concentrate on the "real" transformation (assignment of unique identifiers to components). Figure 8 shows the encoding of the M2M transformation from a *HW* and *SW Model* to a *Composed Model* in the M2M transformation chain. The transformation is encoded using Xtend and contains 4 library functions calls. For usual examples the number of calls shall be lower than 10 and follow a similar structure.

```
//Manual transformation function
Void calculateAndAssignId(Component component):
  component.setId(component.eContainer.components.indexOf(component) + 1);

//Orchestration function of M2M transformation
create ComposedModel this (hw::SWModel hwmodel, sw::SWModel swmodel,
                           ModelTransformation transformation)
  initM2MTransformation(transformation) -> //Initialize M2M transformation

  //Call M2M transformation function copying unchanged data (deep copy)
  this.components.addAll(hwmodel.components.transformObject()) ->
  this.components.addAll(swmodel.components.transformObject()) ->

  //Manual M2M transformation
  this.components.calculateAndAsignId();

  finiM2MTransformation(this); //Finalize M2M transformation
```

Fig. 8. Manual Specification of a Model-to-Model Transformation. For simplification reasons each model contains a root element which stores all the other elements.

Transformation Algorithm. The transformation algorithm is started on an object of an input model. From there it traverses all reachable objects. Every time an object is reached, the algorithm tries to create an equivalent object in the output model and copies as much data as possible between those objects. This includes the transformation between data types of different namespaces. The class of the object, which has to be created in the output model, is determined by using the information contained in the difference model of the metamodels. If elements like classes or attributes do not exist in the next metamodel of the M2M transformation chain, e.g., they are deleted, their data is ignored. The same holds for newly created elements, for which no data exists. To simplify the algorithm all objects are created when they are reached for the first time regardless whether they are reached through a containment or normal association. For keeping track of already created objects a map is used, which relates input objects with their corresponding output objects. Later on, this map can be used during the manual transformation to navigate from input to output objects and vice versa and access their data as needed.

Since the algorithm starts at a specific object, it is possible that only a sub tree of the input model is traversed. The developers have to orchestrate the M2M transformation, so that all required parts are copied. This task is simplified by the fact, that the provided *transformObject* abstracts whether the object has already been transformed. The *finiM2MTransformation* in addition takes care that the resulting model contains no objects without corresponding container (storing) object.

The algorithm is provided as a JAVA library. This library contains functions for initializing a transformation (*initM2MTransformation*), transforming objects (*transformObject*), storing relations between input and output objects (*storeMapping*), getting related objects (*getDestinationObjects* and *getSourceObjects*), and finalizing a transformation (*finiM2MTransformation*). In the example the library has been used from Xtend, but it can be used with any other model transformation language. It is even possible to perform the generic transformation in JAVA and do the manual part with a model transformation language.

The library can also be used directly without specifying the metamodel differences. In this case the output metamodel needs to be specified. Unchanged parts are then copied to the output model by relying only on the information represented in the structure of the underlying metamodels. The type, attribute, and reference names are than used as matching criteria. This requires unique type names over all input metamodels, which are combined.

6 Implementation and Evaluation

In the following, the approach and its implementation are evaluated in the context of two MDSD tools of the embedded systems domain: FTOS and ϵSOA. The tools are built according to state-of-the-art for embedded systems development and rely on M2M transformations to calculate data needed for code generation. In both tools, the approach has been integrated to simplify their M2M transformations. Both tools are based on the Eclipse Modeling Framework (EMF)[4] and use the languages Xtend and Java for M2M transformations.

6.1 Implementation Details

The presented approach has been realized based on EMF, which can be considered as an implementation of the Essential Meta Object Facility (EMOF) [10]. The implementation consists of a metamodel used to specify MM2MM transformations, a script to perform MM2MM transformations based on difference models, and a library to support the (semi-) automatic copy of data in exogenous M2M transformations for operational (imperative) model transformation languages. Figure 9 shows the size of the implementation containing support for both M2M and MM2MM transformations, where the code for MM2MM transformations forms the majority. The implementation and an extended example are available at `http://tooling.fortiss.org/`.

Criteria	JAVA Code	Xtend Code
# Functions	76	14
# Statements	999	28

Fig. 9. Size of Implementation Supporting Model-to-Model Transformation Chains

6.2 Evaluation of FTOS

FTOS [3,11] targets fault-tolerant real-time systems. It generates an application specific run-time system including automated selection and configuration of appropriate fault-tolerance mechanisms. FTOS is based on four input models with their corresponding metamodels. In the hardware model, developers can describe the hardware topology (nodes and networks). A software model is used to specify the application components with a coarse schedule. The set of faults

[4] EMF: `http://www.eclipse.org/modeling/emf/`

that might occur in the system are defined in a fault model. The fault tolerance model is used to select fault detection tests and fault tolerance strategies.

During M2M transformations the four models are merged into one model, appropriate fault-detection mechanisms, and a refined schedule are calculated. A detailed discussion of the different calculations can be found in [3]. To avoid nasty copy operations, a huge and complex M2M transformation was used instead of applying a series of fine-grained M2M transformations. Another problem was the manual creation of the output metamodel, since the input metamodels are frequently extended to support further hardware components or other fault-tolerance mechanisms.

For FTOS, we applied the approach without relying on a difference model for MM2MM transformations. Hence, there is no support for metamodel changes in the M2M transformation chain and for the handling of renamed objects during M2M transformation. Figures 10 and 11 depict the results of the improvement. As can be seen in Figure 10, the code size reduction of the M2M transformation is significant. This results only from the elimination of copy statements. Even in this bad setup by using only one big transformation containing a lot of calculations, the ratio of simple copy instructions contained is high. The increase of the runtime for an equivalent transformation by using the generic library instead of a manual optimized transformation is instead negligible, as stated in Figure 11. Even without using MM2MM transformations the major benefits are the significant reduction of transformation functions and statements. This reduction can be explained by the fact that many elements and their properties are simply copied during the M2M transformation. In addition, the readability of the manual M2M transformation code has been improved, since the remaining code mainly contains code describing the "real" transformation.

Tool	**# Metamodel Elements**	**Criteria**	**JAVA Code**			**Xtend Code**		
			Old Vers.	**M2M Vers.**	**Improvement**	**Old Vers.**	**M2M Vers.**	**Improvement**
FTOS	101	# Functions	124	93	25.0 %	406	286	29.6 %
		# Statements	1285	1045	18.7 %	1881	1146	39.1 %
ϵSOA*	13 (+ 79)**	# Functions	8	1	87.5 %	22	6	72.7 %
		# Statements	59	20	66.1 %	72	22	69.4 %

Fig. 10. Evaluation Results without and with the Presented Approach (* Only code related to M2M transformation and handling of instances of manual created metamodel elements are considered. Other code is ignored, e.g., routing calculation or handling of instances of generated metamodel objects. ** Generated metamodel elements).

6.3 Evaluation of ϵSOA

ϵSOA [12] is used to develop sensor / actuator networks. During M2M transformations communication routes are calculated and the routing tables are prepared. ϵSOA is based on four input models with their corresponding metamodels.

Developers define and configure the nodes engaged in the system and their connection with each other in the nodes and network model. The available services are defined in the service model. In the application model developers can instantiate services on nodes and configure their communication relations.

During M2M transformations an appropriate network routing is calculated for the specified communication. Along with this, unique identifiers are assigned to instantiated services. Most of the calculations and storing of data are done in JAVA. This setting contradicts the main philosophy of MDSD as calculated data is stored outside of models. The major reason why this approach was selected, was to avoid the extension of the underlying metamodel, since the affected part of the metamodel is generated and quite often changed. Otherwise the integration of the manual changes had to be repeated after each regeneration of the corresponding metamodel part. For comfort reasons only one M2M transformation was implemented.

In the context of ϵSOA, both MM2MM and M2M transformations were applied. The advantages of the MM2MM transformation support are obvious, since the metamodels of the input models are extended frequently. By using MM2MM transformations, the changes between metamodels in the M2M transformation chain needed to be specified only once and can now be reapplied whenever an input metamodel changes. The MM2MM transformation consists of the merge of 6 metamodels. In addition, 1 new class is added and 2 are modified (not abstract anymore, renaming due to a name conflict), 1 enumeration is renamed due to a name conflict, 6 references are added and 1 is deleted, and 5 attributes are added and 1 is deleted. Figures 10 and 11 depict the results of the improvement. As can be seen in Figure 10, the code size of the M2M transformation could be dramatically reduced. Even by ignoring the improvements on major parts. The main reason for the huge reduction is that the M2M transformation is mainly a model combination. Since a combination of models consists predominantly of copy operations, it was easy to get rid of these operations by our approach. Figure 11 shows that even a decrease in the runtime for an equivalent transformation by using the generic library has been achieved. The difference between the run times observed in the context of ϵSOA and of FTOS can be motivated by the fact that the manual M2M transformations of FTOS require an additional traversing of the model whereas most of the transformations of ϵSOA are already realized by the library. The speed up is motivated by the execution of compiled JAVA code compared to interpreted Xtend code.

Tool	**# Objects in Application Model**	**Runtime**		
		Old Version	**M2M Version**	**Improvement**
FTOS	121	468.2 ms	478.8 ms	-2.3 %
ϵSOA	112	212.2 ms	199.4 ms	6.0 %

Fig. 11. Runtime without and with the Presented Approach (average over 5 runs)

7 Related Work

The M2M transformation part of our approach is highly related to model transformation languages like Xtend[5], QVTOperational [5], or the imperative part of ATL[6] [13] and constitutes an extension of such languages through a library. This extension is used to relieve the developers from specifying copy operations for unchanged model data by providing support for deep copy. As demonstrated in this paper, the amount of copy operations in a M2M transformation can be rather high. For similar reasons ATL offers a refine mode, which can be used to copy model elements, but works only for endogenous transformations.

The specification of differences between metamodels is closely related to the representation of model differences [7] and delta models used in software product lines (SPLs) [8,9]. Differences in SPLs are called features [14] and are used to integrate functionality into a base configuration. Features are ordered relatively to each other to ensure a consistent integration. When creating a new product, all the required features are selected. The features are ordered and their applications lead to the configured product. In our approach, we support the construction of M2M transformation chains defining a fixed order of the transformations. We do not only take care of transforming the model (product), but also consider the creation of the metamodels required by M2M transformation chains.

Glue Generator Tool (GGT) [15,16] is a framework dedicated to the reuse of PIMs and PSMs of existing applications. Composition rules are specified using GGTs own metamodel. Correspondence rules are used to relate model elements. For composition merge rules are used. Modifications are handled by override rules. In contrast to GGT, our approach is more concerned with the various calculations done in M2M transformation chains and their optimal support. Therefore, only aligned models are considered.

Epsilon Merging Language (EML) [17] is a metamodel based language for expressing model merges. It contains a model comparison and transformation language. Like GGT it is rule based. Match rules specify matching elements, which are then merged through according rules. Not matched elements are handled by transformation rules. EML is concerned with the merge of models based on a specification including copy operations. Our solution is focused on the automation of those copy operations based on type equality. Along with this, our approach offers a way to describe the adaptation of metamodels.

Epsilon Flock [18] is a model migration tool build on top of EML. It contains a rule based transformation language used to define adaptations for metamodel evolutions. This language includes a conservative copy algorithm, which is used to copy unchanged model elements to the new model version. As Epsilon Flock is used to adapt models after a metamodel evolution happened, it does not consider changing metamodels by itself. But as has been shown in this paper, the support of metamodel changes is important for M2M transformation chains. The same holds for other metamodel evolution tools, e.g., COPE [19] or Ecore2Ecore [20].

[5] Xtend/Xpand: `http://wiki.eclipse.org/Xpand`

[6] ATL: `http://www.eclipse.org/atl/`

Atlas Model Weaver (AMW)[7] is a model composition framework that uses a specification for model transformations called weaving model to produce an executable model transformation. The weaving model contains composition operators specifying the relation between the various input models. This weaving model is used by AMW to compose various models. In this sense, it is more a model transformation language. Copy operations can be automated based on the various relations stored in the weaving model. However, the construction of the output metamodel is not in the focus of AMW and not further supported through weaving models.

8 Conclusions

MDA proposes a model refinement in several steps from PIMs to PSMs. However, this requires the management of many similar metamodels and the copy of data between the corresponding models. If large parts of the model remain unchanged, the developers have to specify many copy operations. To avoid this problem, the developers typically use only few steps between PIMs and PSMs.

In this paper an approach was presented that supports on the one hand the (semi-) automatic metamodel construction to specify metamodel chains and to cope with later changes. On the other hand the (semi-) automatic copy of unchanged model data during M2M transformations is supported. The MM2MM transformation support has been applied to one MDSD tool, clearly showing its benefits there. The M2M transformation was applied to two MDSD tools. Both tools show a significant reduction of the code for M2M transformations (up to 70 %). This reduction is only related to avoiding simple copy operations. However, besides lower effort for specifying M2M transformations, the readability is improved drastically.

References

1. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1 (June 2003)
2. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006)
3. Buckl, C.: Model-Based Development of Fault-Tolerant Real-Time Systems. Dissertation, Technische Universität München, München, Germany (2008)
4. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
5. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1 (January 2011)
6. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
7. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A metamodel independent approach to difference representation. Journal of Object Technology 6(9), 165–185 (2007)

[7] AMW: `http://www.eclipse.org/gmt/amw/`

8. Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
9. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE 2010), pp. 13–22 (2010)
10. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification Version 2.0 (January 2006)
11. Buckl, C., Sojer, D., Knoll, A.: FTOS: Model-driven development of fault-tolerant automation systems. In: 15th International Conference on Emerging Technologies and Factory Automation (ETFA 2010), Bilbao, Spain, pp. 1–8 (2010)
12. Buckl, C., Sommer, S., Scholz, A., Knoll, A., Kemper, A., Heuer, J., Schmitt, A.: Services to the field: An approach for resource constrained sensor/actor networks. In: International Conference on Advanced Information Networking and Applications Workshops (WAINA 2009), Bradford, UK, pp. 476–481 (2009)
13. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
14. Batory, D., Azanza, M., Saraiva, J.A.: The Objects and Arrows of Computational Design. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 1–20. Springer, Heidelberg (2008)
15. Bouzitouna, S., Gervais, M.P.: Composition rules for PIM reuse. In: 2nd European Workshop on Model Driven Architecture with Emphasis on Methodologies and Transformations (EDWMDA 2004), pp. 36–43 (2004)
16. Bouzitouna, S., Gervais, M.P., Blanc, X.: Model reuse in MDA. In: International Conference on Software Engineering Research and Practice (SERP 2005), Las Vegas, USA, pp. 354–360 (2005)
17. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book (2010)
18. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model Migration with Epsilon Flock. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 184–198. Springer, Heidelberg (2010)
19. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
20. Paternostro, M., Hussey, K.: Advanced features of the eclipse modeling framework. In: EclipseCON (March 2006)