Combining Independent Model Transformations

Anne Etien, Alexis Muller, Thomas Legrand, Xavier Blanc
INRIA Lille-Nord Europe, LIFL CNRS UMR 8022
Université Lille 1
France
firstname.lastname@inria.fr

ABSTRACT

Model transformation is one of the key principles of Model Driven Engineering. Many approaches have been proposed to design and realize them. However, for all the approaches, model transformations are considered as single entities that can only be chained if their input and output metamodels are compatible. This approach has the major drawback to focus on the satisfaction of the compliance property when building a transformation chain.

In this paper we propose a mechanism for combining independent model transformations which jointly work towards a common objective but do not initially handle compatible metamodels. Our proposal is independent of any model transformation approach. It has been validated using Gaspard, an environment dedicated to code generation for embedded systems.

1. INTRODUCTION

Since Model Driven Engineering (MDE) advocates to support the well known principle of `separation of concerns` through the extensive use of models in all the steps of the software development cycle [1, 2], model transformations are the key technology to achieve the `integration of concerns` [3, 4, 5].

Classically, a model transformation is an operation that has a model as input to either modify it or output a new one. The structure of the new model is directly related to the structure of the input. Even if there are different styles to design and perform model transformations [6], all the approaches proposed in the literature such as [7, 8, 9, 10] consider that model transformations are independent entities. Therefore, the technics to reuse existing model transformations are restricted to their improvement (by adding new transformation rules or by modifying ones for instance).

The main flaw of those approaches comes from the need to combine a set of existing independent model transformations. In this paper, "independent transformations" stands for transformations using different metamodels but aiming at the same objective: generate some code, build a target model... For example, the MDA approach, based on a three-level separation of concerns (CIM, PIM, PSM) [1], imposes to combine independent transformations for an automatic code generation from requirement models (e.g CIM to PIM, PIM to PSM and PSM to Code). An example related to the system co-design also illustrates that need: existing model transformations jointly work to build a single output model [11].

The common way for combining independent transformations is to chain them. A coarse-grain model transformation chaining the inputs and the outputs of the existing fine-grain model transformation is hence created. But the designer must face the following problem: make the input and output metamodels of the fine-grain model transformations compatible. Indeed, two transformations can only be chained if the output metamodel of the first one is included into the input metamodel of the second one. This compliance issue forces to design tailored fine-grain model transformations for a dedicated chain. As a consequence, the fine-grain model transformations are highly coupled to a particular chain and are not reusable.

In this paper, we present a convenient approach to design highly flexible chains from existing independent model transformations. We propose to artificially change the input and output types of fine-grain model transformations. The goal is to overcome the type compliance issue and ease the integration of the transformations into several chains. This paper is organized as follows: Section 2 presents existing works related to model transformation chaining and composition. Section 3 describes how we make up a coarse grain model transformation from localized and smaller ones. Section 4 validates our approach within the Gaspard environment [11]. Finally, section 5 draws a conclusion and introduces future works.

2. RELATED WORK SECTION

In [12], Vanhoof and al. have clearly shown the too simplistic characteristic of the recommended "PIM to PSM" MDA's model transformation vision. They highlight the need to chain transformations and explain: a *transformation chain specifies how a number of transformations work together, each elaborating on the source model to come to a target model.* Then, they propose a process for building transformation chains but do not face the point raised in this paper: to combine transformations working towards a single output whereas their input and output metamodels are not obviously compliant.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

In [13], Oldevik provides a framework to make existing transformations more generic and then reusable by composition. His framework considers the transformations as functions to compose if their domains are compliant. Our approach has the same target but does not impose the compliance between the input and output metamodels.

In [14], Olsen and al. define *a reusable transformation [as] a transformation that can be used in several contexts to produce a required asset.* They also clearly identify transformation chaining as one of the techniques allowing the reuse of transformations. Nevertheless, they emit two restrictions: first, existing transformations should be as small as possible and second, their input and output metamodels have to be compliant. We agree that small transformations are more reusable than bigger ones. Thanks to our approach, the type compliance constraint can be overcome.

In [15], Sànchez and Garcia argue that model transformation facilities are currently too focused on rules and patterns and should be tackled at a coarser-grained level. To make model transformation reusable as a whole, authors propose the *factorization* and *composition* techniques. The *factorization* technique aims at extracting a common part of two existing transformations in order to define a new transformation. The *composition* composes two transformations to create a new one. Obviously, those techniques have the same drawback. They require a connection between the input and the output metamodels of the transformations (their intersections must not be empty). Our approach would overcome this drawback because it virtually increases the input and the output metamodels of existing transformations. Our work allows the *factorization* and the *composition* to become more generic.

3. FORMALIZATION OF INDEPENDENT TRANSFORMATION COMBINING

Our investigations tend to make up a complex model transformation from smaller transformations jointly working in order to build a single output model. These transformations involve different parts of the same input metamodel; their application field is localized. To build the complex model transformation, the localized transformations have to be chained despite the input and output metamodels incompatibility (i.e. the output metamodel of a first one is not included in the input metamodel of a second one). In the rest of this paper, independent transformations hold two properties:

1. they jointly work to build a single output model
2. their input and output metamodels are not compatible

The first property provides an obvious reason to chain them. The second property specifies that the transformations to chain are structurally independent.

We propose the `Extend` operator. This operator modifies model transformations by extending their input and output metamodels to make them compatible. Two transformations that were independent can henceforth be chained. The `Extend` operator preserves the behavior of the initial transformation and uses the identity to handle the elements not involved in the initial transformation. In this section, we present the foundations of our approach. The subsection 3.1 briefly recalls the well-known model, metamodel and transformation definitions our approach is based on. The subsection 3.2 then presents the `Extend` operator and explains how it can be used to chain two existing model transformations. Finally, the subsection 3.3 outlines the properties of the chain that can be inferred.

3.1 Definitions

As our purpose is not to propose yet another definition of well-known MDE concepts, we adopt the metamodel, model and conformance definitions established by Alanen *et al.* [16]. A metamodel is a set of classes and a set of properties owned by the classes. A model is a set of elements and a set of slots. Each element is typed as a class in a metamodel. Each slot is owned by an element and corresponds to a property in a metamodel.

From these definitions, the union, the intersection and the difference are defined on both metamodels and models respectively as the union, the intersection or the difference of each set defining the metamodels or the models. Besides, the difference operator does not produce dangling links. For example, if a class does not belong to the difference of two metamodels, there can not be a property referencing it.

We also adopt the traditional definition of unidirectional binary model transformation: a transformation has one input metamodel and one output metamodel. A transformation is a function consuming a model instance of its input metamodel and producing a model instance of its output metamodel.

3.2 `Extend` Operator Semantics

For a given transformation t from a metamodel X to a metamodel Y, our objective is to define an extended transformation T. Its input metamodel would be an extended metamodel encompassing X. But it preserves as much as possible the behavior of the transformation t. This implies that:

1. the behavior of both transformations (t and T), on an element typed by a class of the metamodel X, must be the same.
2. an element typed by a class not belonging to the metamodel X must be unchanged.

An instinctive approach would be to extend the input metamodel X and the output metamodel Y with the same ordinary metamodel A. $X \cup A$ and $Y \cup A$ would thus be respectively the input and output metamodel of the extended transformation. However, we argue that $Y \cup A$ can not be the output metamodel of the extended transformation. If X is partially included in A ($X \cap A \neq \varnothing$), some concepts are inevitably common to X and A. The elements typed by those concepts are transformed into elements typed by concepts of the metamodel Y according to the first hypothesis. Therefore, those concepts from A do not belong to the output metamodel of the extended transformation.

We propose the following definition for the `Extend` operator:

Definition 1 *Let t be a transformation from the metamodel X to the metamodel Y (t : X → Y) and A an ordinary metamodel.*

$\texttt{Extend}_A(t)$ is a transformation T from the $X \cup A$ metamodel to the metamodel B (T : $X \cup A \rightarrow B$), written such as:

- $B = Y \cup (A \setminus X)$
- $T(m) = t(m)$ *if m is a model instance of the metamodel X*
- $T(m) = m$ *if m is a model instance of the metamodel* $A \setminus X$
- $T(m) = T(n) \cup (m \setminus n)$ *with n the part of the m model typed by X*

According to the difference operator of the models, $T(n) \cup (m \setminus n)$ can not produce dangling links in the resulting model.

3.3 Chain Properties

Considering the metamodels X, Y, Z, W and the two transformations $t_1 : X \rightarrow Y$ and $t_2 : Z \rightarrow W$ that cannot be chained (*i.e.* Y is not included into Z, neither W into X). Let's suppose that we plan to combine them to achieve a common objective. t_1 and t_2 are viewed as two transformations involving two different parts of the same metamodel having a non-empty intersection. They jointly build a common output model. The transformation T resulting from the combination aims to be applied on a model instance of a metamodel encompassing X (or Z).

Ideally, we would also expect the behavior of T, on a model only instance of X (respectively Z), to be strictly the same as the behavior of t_1 (respectively t_2).

In order to be composed, t_1 and t_2 have to be extended with the `Extend` operator. According to the way the operator is applied on these transformations and taking into account the relationships between X, Y, Z and W, four cases may occur:

1. t_1 and t_2 can only be combined in one order (t_1,t_2 for example).
2. t_1 and t_2 can be combined in both orders but the resulting models won't be the same.
3. t_1 and t_2 can be combined in both orders and the resulting models will be the same.
4. t_1 and t_2 can not be combined at all.

In the next section, we expose properties relative to each case.

The first property specifies the requirement for two transformations to be combined by imposing the order. Intuitively, two transformations can be combined if the concepts used by the second transformation are not consumed by the first one. It can be formulated as follows:

Property 1 *t_1 can be chained with t_2 if and only if:* $Z \cap (X \setminus Y) = \varnothing$

In that case, $T_1 = \texttt{Extend}_A(t_1)$ *and* $T_2 = \texttt{Extend}_B(t_2)$. *A is the input metamodel of T_1 and B (computed as specified in Definition 1) is the output one.*

If T_1 can not produce the elements needed/required by t_2, T_2 always corresponds to the identity transformation. This is formally expressed by:

Property 2 *Let's suppose that t_1 can be chained with t_2. If* $Z \cap B = \varnothing$ *then T_2 is the identity transformation.*

The order of t_1 and t_2 can be inverted if both t_1 can be combined with t_2 and vice-versa. The Property 3 is built by applying twice the Property 1, once in each order. The output metamodels of each resulting transformation are the same. But such an affirmation can not be ensured for the models.

Property 3 *If* $(X \cap Z) \setminus (Y \cap W) = \varnothing$ *then* $\exists T_1, T_2, T_1', T_2'$

$$as \begin{cases} T_1 : A \cup X \rightarrow B \\ T_2 : B \cup Z \rightarrow C \\ T_2' : A \cup Z \rightarrow B' \\ T_1' : B' \cup X \rightarrow C \end{cases}$$

with $T_1 = \texttt{Extend}_A(t_1)$, $T_2 = \texttt{Extend}_B(t_2)$, $T_1' = \texttt{Extend}_{B'}(t_1)$, $T_2' = \texttt{Extend}_A(t_2)$

If the input metamodels of two transformations have no common elements and if the concepts required by one transformation are not produced by the other, they can be combined and the resulting model does not depend on their execution order. We aware that this property is relatively restrictive. It could be refined by distinguishing the read concepts from those that are either created or modified.

Property 4 *If* $(X \cap Z) = \varnothing$ *and* $(Y \cap Z) = \varnothing$ *and* $(W \cap X) = \varnothing$ *then* $\forall$ *model m,* $T_1(T_2(m)) = T_2(T_1(m))$

The following property specifies a sufficient condition ensuring that the combination of two transformations is impossible. Intuitively, it occurs when each transformation consumes the concepts useful for the execution of the other.

Property 5 *Let t_1 and t_2 be two transformations.* $\forall$ *metamodel* $A, \nexists T_1, T_2$ *as* $T_1 = \texttt{Extend}_A(t_1)$, $T_2 = \texttt{Extend}_B(t_2)$ *or* $T_2 = \texttt{Extend}_A(t_2)$, $T_1 = \texttt{Extend}_B(t_1)$

$$\textit{if and only if:} \begin{cases} Z \cap (X \setminus Y) \neq \varnothing \ \ \textit{and} \\ X \cap (Z \setminus W) \neq \varnothing \end{cases}$$

If the extension operator can be applied on almost all the transformations, it reaches a high usefulness when they are very localized, *i.e.* restricted to the set of involved elements. Each transformation may be dedicated to a single purpose and deals with a part of an initial encompassing metamodel. Thus, the extension operator consists in handling this encompassing metamodel. The transformations defined on the small parts are combined and chained to jointly reach the target model. Their reuse is eased.

In the next section, we describe an example of such transformations through Gaspard, an MDE-based framework dedicated to the co-design of embedded systems.

4. CASE STUDY

This section targets the validation of our approach with a real case study. For this purpose, we demonstrate that several chains can be specified from three fine grain transformations (*i.e.* whose input and output metamodels only

contain concepts implied in the transformation). The `Extend` operator is applied with different metamodels while the execution of the chains leads to a single final objective. These three transformations all exist withing the Gaspard environment. They aim to smoothly replace the MARTE concepts [17] by others closer to SystemC.

Subsection 4.1 presents the context by briefly introducing Gaspard and its business concepts. The subsection 4.2 specifies many transformations leading to the SystemC code generation. They are practically combined to make a chain in accordance with the properties defined in subsection 3. Finally, subsection 4.3 details the concrete implementation within the Eclipse framework.

4.1 Context: Gaspard a Co-Design Environment

Gaspard is a co-design environment dedicated to high performance embedded systems based on massively regular parallelism. In this environment, the user separately designs the hardware architecture and the application at a high abstraction level. To do so, she uses UML enriched with the MARTE standard profile for modeling and analysing real time embedded systems. From these high level specifications, code for high performance computing, hardware-software co-simulation, functional verification or hardware synthesis is automatically produced by model transformations. Gaspard respectively enables the generation of OpenMP C, SystemC, Lustre and VHDL programs.

The generation of such code from UML is complex and requires intermediary steps. Some transformation chains shared characteristics such as the explicit mapping of application tasks onto processing units, the mapping of the data onto memories... The successive transformations are specified by focusing on the involved concepts. They are extended using the `Extend` operator in order to be applied on a model instance of an intermediary metamodel.

The input of each chain is a UML model, defined by the end-user and compliant with the MARTE profile. This model is transformed to a MARTE model: the transformation from the profiled UML metamodel to the MARTE metamodel is a classical transformation involving all the concepts. It has been specified in QVTO and is out of the scope of this paper. It corresponds to a bridge between the specifications given by the user and the transformation developed by the designer. In the rest of the section, we consider the MARTE metamodel to be the input of the chains.

In the specifications of any application, a component corresponds to a task whereas in the architecture part, it refers to a hardware resource, *e.g.* a processor or a memory. Hundred tasks relating to exactly the same action realized with different data are designed by a unique task. It owns a repetition property equals to 100. The way the data are consumed by each task is specified by the *LinkTopology* concept (*Tiler* or *Reshape*). Similarly, 64 identical processing units are represented by only one with a repetition-value set to 64.

Commonly with the component-based approach, a component may be complex and composed of several other ones. A hierarchy is also often proposed. In MARTE, the composed component is a *StructuredComponent* and the contained ones are instance of an other component externally defined. The *AssemblyPart* concept is related to the instances. As a consequence, the hierarchy is defined two levels by two levels.

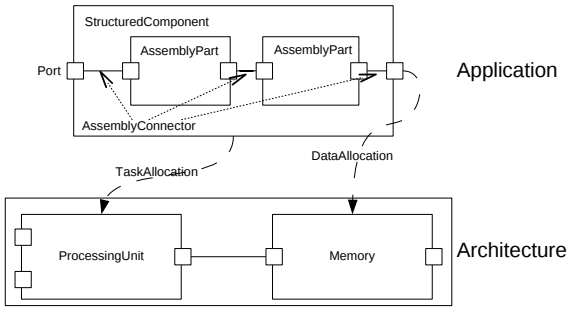

Figure 1: Sketch of a MARTE model

Task can be mapped onto processing units through the *TaskAllocation* link. Data are analogically mapped onto memories with the *DataAllocation* linking the port to the memories.

Figure 1 sketches very roughly the concepts of MARTE used in Gaspard. It simply aims to present the concepts useful to understand the case study. Further details on Gaspard can be found in [11].

4.2 SystemC Code Generation

This section is not addressed to a SystemC specialist. It shortly describes the transformations necessary to generate SystemC code from a MARTE model assuming the parallelism is compactly expressed. These transformations have been defined targeting their combination and their application field is restricted. Input and output metamodels contain few concepts: only the ones involved in the transformation. Three of the ten transformations are exposed to illustrate our approach.

4.2.1 Description of the transformations

Port Instance Introduction (t_1).

Similarly to UML, the MARTE metamodel does not contain the port instance concept. A port instance corresponds to the port of a component instance. Since it is really easier to manipulate the port instance than the couple (port, component instance), t_1 consists of introducing the port instance concept (*i.e. PortPart*) in the MARTE metamodel.

The connectors (*i.e. AssemblyConnector*) contained in a structured component are replaced by *PortConnector*. The new extremities are the same than those of the *AssemblyConnector* except that the couples (*FlowPort*, *AssemblyPart*) are replaced by the corresponding *PortPart*.

In other words, a *PortConnectorEnd* is either a *PartConnectorEnd* referring to a *PortPart* or a *DelegationConnectorEnd* referring to a port of a *StructuredComponent* called *Flowport*. At the end, the topology associated to the *AssemblyConnector* henceforth refers to the *PortConnector*. Figure 2 and Figure 3 respectively correspond to the input and output metamodel of t_1.

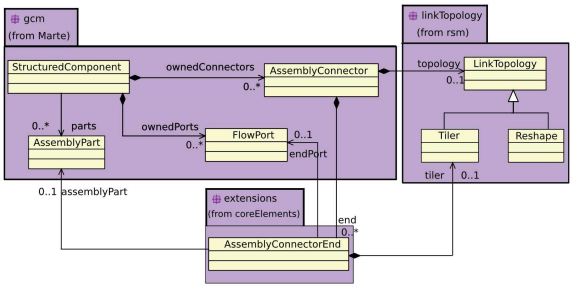

Figure 2: Input metamodel of t_1 (extract from the MARTE metamodel)

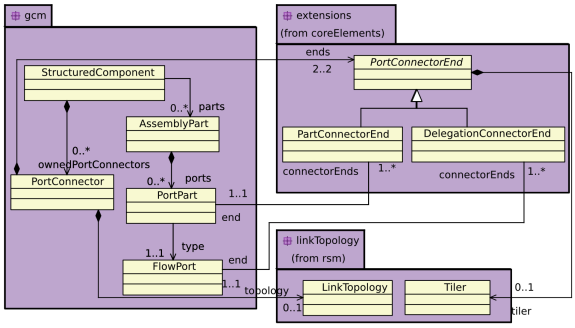

Figure 3: Output metamodel of t_1

Explicit Mapping (t_2).

In order to ease the design, the mapping specification of each application component is not compulsory. An application component which is not explicitly mapped through an *Allocation* link onto a processing unit is implicitly mapped onto the same processing unit than the *StructuredComponent* which contains it. However, the SystemC code generation for a component requires the mapping information to be explicit. t_2 aims to explicitly provide a mapping for each application component. It is a refactoring, the input and output metamodels are the same. Figure 4 shows the concepts involved in the explicit mapping.

Link Topology Management (t_3).

As briefly explained before, the *LinkTopologies* are associated to connectors and express the way the data are consumed or produced by repeated tasks. These connectors are complex. They hide some tasks that must be mapped onto processing units, as any other task. t_3 aims to replace the connectors associated to the *LinkTopologies* (*Tiler* and *Reshape*) by tasks called *LinkTopologyTask*. It also preserves the connections between the various existing tasks. Figure 5 presents the output metamodel of t_3.

Other transformations exist and are based on the same approach: restrict the input metamodels to the set of concepts involved in the transformation. These transformations

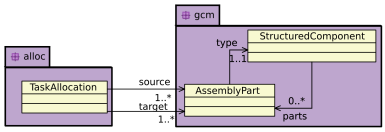

Figure 4: Input and output metamodel of t_2 (extract from the MARTE metamodel)

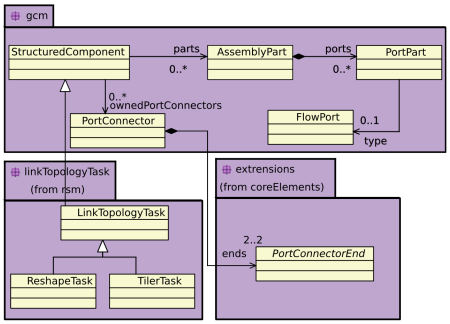

Figure 5: Output metamodel of t_3

are intended for being applied on model instances of a larger metamodel. For this objective, the `Extend` operator allows us to focus on the purpose of the transformation without defining the parts corresponding to the identity.

4.2.2 *Construction of a Transformation Chain*

The SystemC code generation from the MARTE metamodel consists in the combination of the previously defined transformations. Practically, the extended transformations have to be computed.

The input metamodels of t_1 and t_2 are both sub-parts of the MARTE metamodel, that is the entry point of the building chain. t_2 adds some information in the models but still refers to the same metamodel. It does not involve metamodel concepts introduced (*e.g. PortPart* or *PortConnector*) or consumed (*AssemblyConnector*) by t_1. Therefore, t_1 and t_2 verify Property 3: the order they are chained doesn't matter. t_2 can either be performed on a model conform to the MARTE metamodel or a model conform to the output metamodel of $\texttt{Extend}_{MARTE}(t_1)$. Property 4 is not satisfied by the metamodels of t_1 and t_2 as some concepts exists in several metamodels. However, these concepts are only read by the transformations; they provide the information useful to the modification or the creation of other ones. They are themselves neither consumed nor modified and the models resulting from the application of t_1 then t_2 or t_2 then t_1 are exactly the same.

On another hand, an order must be chosen. We suggest to remain as long as possible in the MARTE metamodel and consequently to begin the chain with $T_2 = \texttt{Extend}_{MARTE}(t_2)$ and to continue with $T_1 = \texttt{Extend}_{MARTE}(t_1)$. The output metamodel is constructed according to T_1 definition.

Besides, according to Property 1, t_1 and t_3 can only be chained in this order. t_3 consumes concepts (*e.g. LinkTopology*) that are required by t_1 whereas the contrary does not raise this issue. We extend the t_3 transformation with a metamodel in such a way that the input of T_3 is the output of T_1. T_3 is introduced in the chain after T_1.

It can be noticed that relatively to property 3, the order of t_2 and t_3 can be interchanged. Indeed, the mapping of the tasks created by t_3 on the processing units is not performed by t_2. It requires another specific transformation (not explained in this paper). Other chains could have been defined, for instance t_1, t_3 and then t_2.

From a business point of view, the implicit mappings are deduced from those specified by the end-user to map each application component onto one or more processing unit. Each port of the part is transformed into a port instance. Fi-

nally, the *LinkTopologies* associated to connectors, expressing how data are consumed by the various tasks, are themselves considered as tasks. Once mapped, the transformations corresponding to the compilation (strictly speaking) can be introduced in the chain.

The three transformations described above jointly work to achieve the same target. They cannot a priori be chained. We have shown that the `Extend` operator helps to overcome this issue. Several chains based on these transformations have been built. Some transformations can even be interchanged without any incidence on the models they are applied on. Furthermore, the `Extend` operator allows to break the received idea considering a refactoring dependent on a single metamodel. If the refactoring is very localized, *i.e.* involves few concepts, it can be defined only with these concepts. The `Extend` operator allows to specify this refactoring on any metamodel including the involved concepts.

The next subsection outlines our implementation of this approach.

4.3 Implementation within Eclipse

4.3.1 `Extend` *Operator Implementation*

Practically, applying the `Extend` operator consists in extending both the metamodels and the fine grain transformation.

The extended metamodels are built using the merge operator defined in the UML specification. The merge operator has been implemented thanks to a QVT transformation under the Eclipse/QVTO engine. The advantage is clear: the metamodels involved in the fine grain transformation can be easily extended with any metamodel.

The extended transformation is made of two parts: the fine grain transformation and the identity. For the latter, two alternatives are available: defining transformation rules to copy each concept or shifting the metamodel. The first alternative is a tedious work. The second one reduces the identity operator to a simple metamodel shift but it implies the introduction of an intermediary metamodel containing all the concepts of the input and the output metamodels. It targets the use of the identity transformation on the complete model. We definitely choose the second option. The intermediary metamodel is built as the others using the merge operator.

The extended transformation is divided into four steps:

1. *shift the metamodel.* This step is preliminary. It takes into account the real context where the extended transformation is applied and shifts the metamodel to apply the transformation itself. It modifies the conformance link between the input metamodel and the input model. In practice, the input model is not initially defined as an instance of the metamodel $X \cup A$ (the input metamodel of the extended transformation) but instance of A explaining why the `Extend` operator has been applied with this metamodel. Indeed, if the extended transformation aims to be the first in the chain, a model instance of A may exist and it might have been designed by the user. If this transformation is in the middle of a chain, the input model of the extended transformation is the result of the execution of the previous one. Since the transformations are independent, it is much more likely that the input model is an instance of A and not of $X \cup A$.

2. *shift the metamodel.* As in the previous step, it changes the conformance link between the input metamodel and the input model. This model henceforth conforms the intermediary metamodel. Practically, this step consists in updating the xml namespace from the initial metamodel URI to the intermediary metamodel URI. This stage can only occur if the output metamodel of the coarse grain transformation includes its input metamodel. For practical reasons, the steps 1 and 2 are performed together as a single metamodel shift.

3. *execute the fine grain transformation.* The fine grain transformation is realized on the model using an in-out mode. It involves only the elements instance of the concepts belonging to the metamodel X. The other elements remain unchanged.

4. *shift the metamodel.* A new metamodel shift is mandatory to conform the model to the metamodel B. B is included in the intermediary metamodel and not conversely. But the metamodel shift is possible, because, by construction, the model does not contain any element instance of a concept coming from X. The shift is technologically performed as in the second step.

Figure 6 illustrates this process.

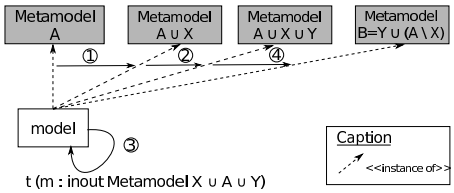

Figure 6: Extending a fine grain transformation

We have achieved these steps in the Eclipse/EMF world, using Java/DOM technology for the first and the third steps and QVTO for the second step.

4.3.2 *The Implementation Chain Process*

Figure 7 shows the valid chain built in the previous subsection from three fine grain transformations.

The `Extend` operator is applied to build the three transformations and their associated metamodels from the fine

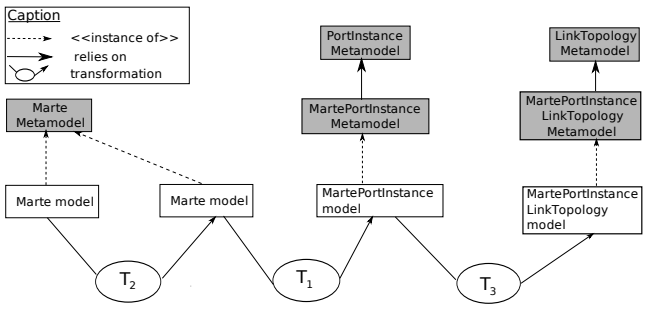

Figure 7: An implementation chain

grain transformations. The implementation of the chain is then classical. The extended *Explicit mapping* transformation (T_2) is first applied on a MARTE model. This transformation is a refactoring. The resulting model is also an instance of the MARTE metamodel. Then the extended *Port Instance Introduction* transformation (T_1) produces a model instance of the MartePortInstance metamodel. This metamodel relies on the PortInstance metamodel input of the fine grain transformation t_1. Finally, the extended *Link Topology Management* transformation (T_3) gets the MartePortInstance model and produces a MartePortInstanceLinkTopology model.

5. CONCLUSION

In this paper, we proposed a mechanism to reuse transformations in chains. Our mechanism aims at chaining two model transformations having incompatible input and output metamodels and jointly working to build a single output. This is important, since most of the industrial model transformations are directly concerned.

We defined the `Extend` operator that virtually extends the input and output metamodel of any existing transformation. From the formal definition, we have expressed the conditions to chain two model transformations. In particular, from two ordinary transformations jointly working to achieve a common target, we have clearly explained when they can not be chained and when they can be chained in both execution orders getting the same result. Some properties rely on strong constraints expressed at the metamodel level. We forese to relax them by differentiating the read and modified concepts.

Our approach has been validated using the complex transformation chains of the Gaspard framework. This validation step has highlighted the efficiency of our `Extend` operator when it relies on small and independent model transformations. This mechanism is currently the core of Gaspard. It ensures a high flexibility for the application developments, reduces the design costs and allows an optimized maintenance.

As a conclusion, our investigations have been validated using QVT but do not depend on any model transformation technique. The model transformations are handled as functions having one input and one output metamodel. This approach facilitates the extension of any other mechanism targeting the reuse of model transformations.

6. REFERENCES

[1] Miller, J., Mukerji, J.: Mda guide version 1.0.1. Technical report, Object Management Group (OMG) (2003)

[2] Selic, B.: The pragmatics of model-driven development. IEEE Software **20**(5) (2003) 19–25

[3] Ossher, H., Harrison, W., Tarr, P.: Software engineering tools and environments: a roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, New York, NY, USA, ACM (2000) 261–277

[4] Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2) (2006) 25–31

[5] Bézivin, J., Gerbé, O.: Towards a precise definition of the omg/mda framework. In: 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA. (2001) 273–280

[6] Czarnecki, K., Helsen, S.: Classification of model transformation approaches. Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture (2003)

[7] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: Atl: a qvt-like transformation language. In: Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA. (2006) 719–720

[8] OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2007)

[9] Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: Rubytl: A practical, extensible transformation language. In Rensink, A., Warmer, J., eds.: ECMDA-FA. Volume 4066 of Lecture Notes in Computer Science., Springer (2006) 158–172

[10] Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: Sitra: Simple transformations in java. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 351–364

[11] Gamatié, A., Le Beux, S., Piel, É., Etien, A., Ben Atitallah, R., Marquet, P., Dekeyser, J.: A model driven design framework for high performance embedded systems. Research Report RR-6614, INRIA (2008)

[12] Vanhooff, B., Ayed, D., Berbers, Y.: A framework for transformation chain development processes. In: Proceedings of the ECMDA Composition of Model Transformations Workshop. (2006) pp. 3–8

[13] Oldevik, J.: Transformation composition modelling framework. Volume 3543 of Lecture Notes in Computer Science., Springer (2005) 108–114

[14] Olsen, G., Aagedal, J., Oldevik, J.: Aspects of reusable model transformations. In: Proceedings of the ECMDA Composition of Model Transformations Workshop. (2006) pp. 21–26

[15] Sanchez Cuadrado, J., Garcia Molina: Approaches for model transformation reuse: Factorization and composition. In: Proceedings of the International Conference on Model Transformation. Volume 5063 of LNCS., Springer-Verlag (2008) pp. 168–182

[16] Alanen, M., Porres, I.: A metamodeling language supporting subset and union properties. Software and System Modeling **7**(1) (2008) 103–124

[17] ProMarte partners: UML Profile for MARTE, Beta 2 (June 2008)