Towards a Transformation Chain Modeling Language*

Bert Vanhooff, Stefan Van Baelen, Aram Hovsepyan,
Wouter Joosen, and Yolande Berbers

Department of Computer Science, K.U. Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
{bert.vanhooff, Stefan.VanBaelen, Aram.Hovsepyan, Wouter.Joosen, Yolande.Berbers}@cs.kuleuven.be

Abstract. The Model Driven Development (MDD) paradigm stimulates the use of models as the main artifacts for software development. These models can be situated at high levels of abstraction, close to the application's business domain. Many consecutive automatic transformations (a transformation chain) can be applied to these models to add the necessary details in order to generate a concrete implementation. This means that a large part of the total development effort is relocated to the development of transformations and hence we should have the necessary tooling support for designing transformation chains. In this paper we propose a metamodel for a transformation chain modeling language that enables implementation independent composition of transformations. We also propose a concrete syntax for this language that is based on UML activity diagrams.

1 Introduction

Model Driven Development (MDD) is an approach to developing software that proposes using machine-readable models as its main artifacts. These models can be constructed with domain specific modeling languages (DSMLs), which are tailored to a specific type of applications and often have a rich visual syntax that hides implementation-level details. These highly abstract models can then be (semi-)automatically transformed to lower-level models by filling in missing details, which eventually makes its straightforward to generate a concrete implementation.

The Object Management Group (OMG) is one of the major endorsers of MDD. Their specific approach is well-known as the Model Driven Architecture (MDA), which is both a specific vision on MDD as well as a collection of technology specifications that support this vision. These specifications include a metamodeling language (MOF) [1], a generic software modeling language (UML) [2], a (not yet fully standardized) transformation specification language (QVT) [3] and many more.

Because the MDD philosophy relocates much of the development effort to transformations it is important to take up the transformation development task with care. In this paper we argue for the need of multiple transformations to get from the highest level models (possibly DSMLs) to the lowest level models (section 2). This requires configuring many transformations in a certain sequence in order to address the concerns of

* The described work is part of the EUREKA-ITEA MARTES project, and partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders).

a specific type of application. Such a transformation sequence is referred to as a transformation chain. We propose a first step towards a modeling language for specifying transformation chains that is based on UML Activity Diagrams. To accomplish this, we provide a metamodel for this language in section 3 and map this to a UML profile in section 4. We wrap up this paper by drawing conclusions and indicating future work in section 5. Related work will be discussed throughout the paper when appropriate.

2 Multiple Transformations

Many papers concerning MDD use the notions of PIM (Platform Independent Model) and PSM (Platform Specific Model), which were introduced by MDA. A PIM is a model of a system that contains no technical details while a PSM is a representation of the same system containing all technical details that are needed to realize it on a concrete technology platform. The mapping between PIM and PSM is realized using an automatic transformation. Such a single-level transformation process allows us to capitalize on the stable platform independent matters and generate PSMs for a range of different technology platforms (figure 1). The platform specific knowledge is moved to the transformations, effectively separating those concerns from the main application model.

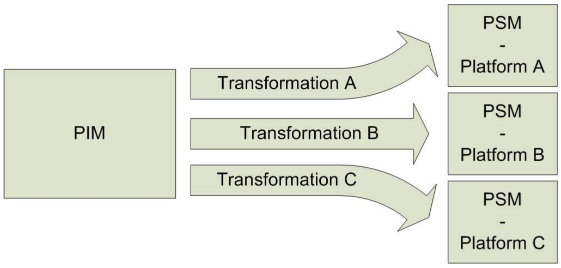

Fig. 1. Single-level transformation (PIM to PSM)

We believe that single-level transformations are not the best way to fully exploit the MDD opportunities. The use of transformations can provide a more elaborate sense of separation of concerns than just pure technical concerns (as in the single PIM/PSM case). Other concerns can be functional, non-functional or just convenience-related such as preventing manual modeling inconsistencies or offering rich domain specific modeling environments. It would be hard and impractical to integrate all these concerns into one big do-it-all transformation. Therefore we argue that it is better to feed a model to a chain of many (small) transformations that each manipulate the model with regard to one specific concern. This would allow us to better modularize the transformations themselves and as a consequence make individual transformations easier to implement and reuse.

Transformation reuse will be most clear in product-line oriented development, as is the case for many embedded applications. Product-lines share a common set of concerns that have to be included or excluded depending on the specific product. If we can encode each of those concerns in a separate transformation, we can more easily leave out unwanted stuff and incorporate new things without having to redo the whole application.

Figure 2 shows an abstract example of composing transformations into a transformation chain. This chain could be the replacement of one of the paths in figure 1.

If multiple transformations are in place, each intermediate model can be seen as being specific to a virtual intermediate platform while being independent of platforms further up the transformation chain. The notion of such a platform is introduced in [4] as abstract platform and is defined as "an acceptable or, to some extent, ideal platform from an application developer's point of view". Abstract platforms not only allow the developer to model an application using appropriate concepts but also allow intermediate transformation developers to create mappings between models using concepts that make sense at their particular level. We could for example treat distribution at one level and timing constraints at another. We consider abstract platforms an integral part of a transformation chain and consequently they are also represented in the model (figure 2).

If we use multiple transformations, the design of their composition (the transformation chain) becomes important next to their implementation. A transformation chain model specifies the composition of many transformations by describing their sequence, input and output model types, dependencies among transformations (such as traceability), platforms etc. Such a model can serve as a construction plan for implementation or as an execution plan after implementation.

Mind that we should very carefully consider how we distribute concerns over transformations. Even though a specific concern can be tackled during one transformation step, it is not always that obvious how all these concerns can be integrated in the overall system. This is especially true for non-functional concerns since they often have subtle effects on one another. A same type of problem arises in the Aspect Oriented Programming (AOP) community, where the application of several aspects on top of each other can produce undesired effects. Our approach to transformation chain modeling does not specifically address these problems.

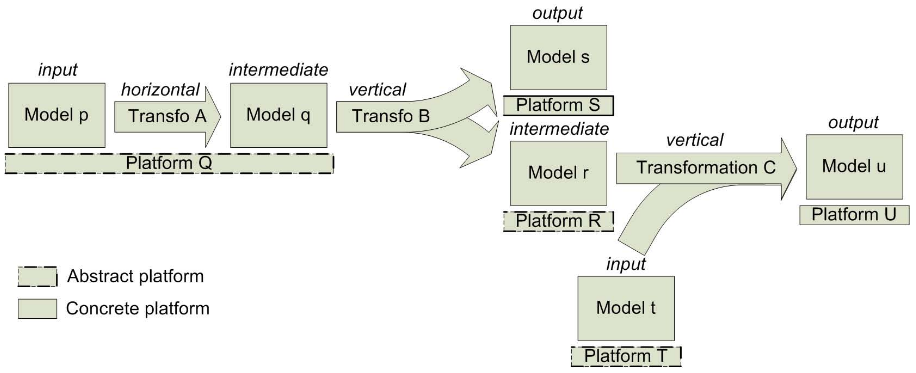

Fig. 2. Multi-level transformation showing intermediate platforms, multi-in/output transformations and vertical/horizontal transformations (respectively between platforms or within a single platform)

In the next section we provide a metamodel that contains the necessary concepts to model transformation chains.

3 Transformation Chain Modeling

In this section we identify the basic requirements of a transformation chain specification language and we present the metamodel that we have conceived to answer to these requirements.

3.1 Requirements

We did not start from scratch in defining a transformation chain specification language. The ORMSC proposal for an MDA Foundation Model [5] gives a good starting point. It consists of a metamodel that defines and relates basic transformation concepts, but does not include concepts specific to composing transformations in transformation chains. We identified the following shortcomings:

1. No specific support for connecting several transformations together.
2. No notion of (abstract) platform; the only typing of models is done through metamodels.
3. No notion of composite transformations, which are reusable transformations that are defined as a chain of lower level transformations themselves.
4. No support for input/output model constraints (pre/post conditions) other than those enforced by the metamodel.
5. No technical considerations – each transformation is assumed to produce models in compatible formats. In real life, even compatible metamodels can be expressed in incompatible formats.

At the same time the MDA Foundation Model proposal contains some concepts that are not that important for defining transformation chains or that are too MDA specific. We consider the listed shortcomings as the additional requirements that our model has to address. In the next subsection we present a metamodel that specifically addresses the shortcomings.

3.2 Transformation Chain Metamodel

In order to adhere to good MDD practice, we define the abstract syntax of the transformation chain by using a metamodel (figure 3).

The model can be divided in two parts: a specification part (*TransformationSpecification*) and an executional part (*TransformationExecution*). The *TransformationSpecification* has two orthogonal specializations: *Atomic* or *Composite* and *Directed* and contains one or more *TransformationFormalParameter*s. Such parameters are typed by an abstract *Platform*, which is in turn characterized by a *Metamodel*, optional *ModelConstraint*s on that metamodel and possibly additional functionality offered by a *ModelLibrary*. A *TransformationFormalParameter* can also have specific *ParameterConstraint*s with which we can enforce additional pre- and postconditions.

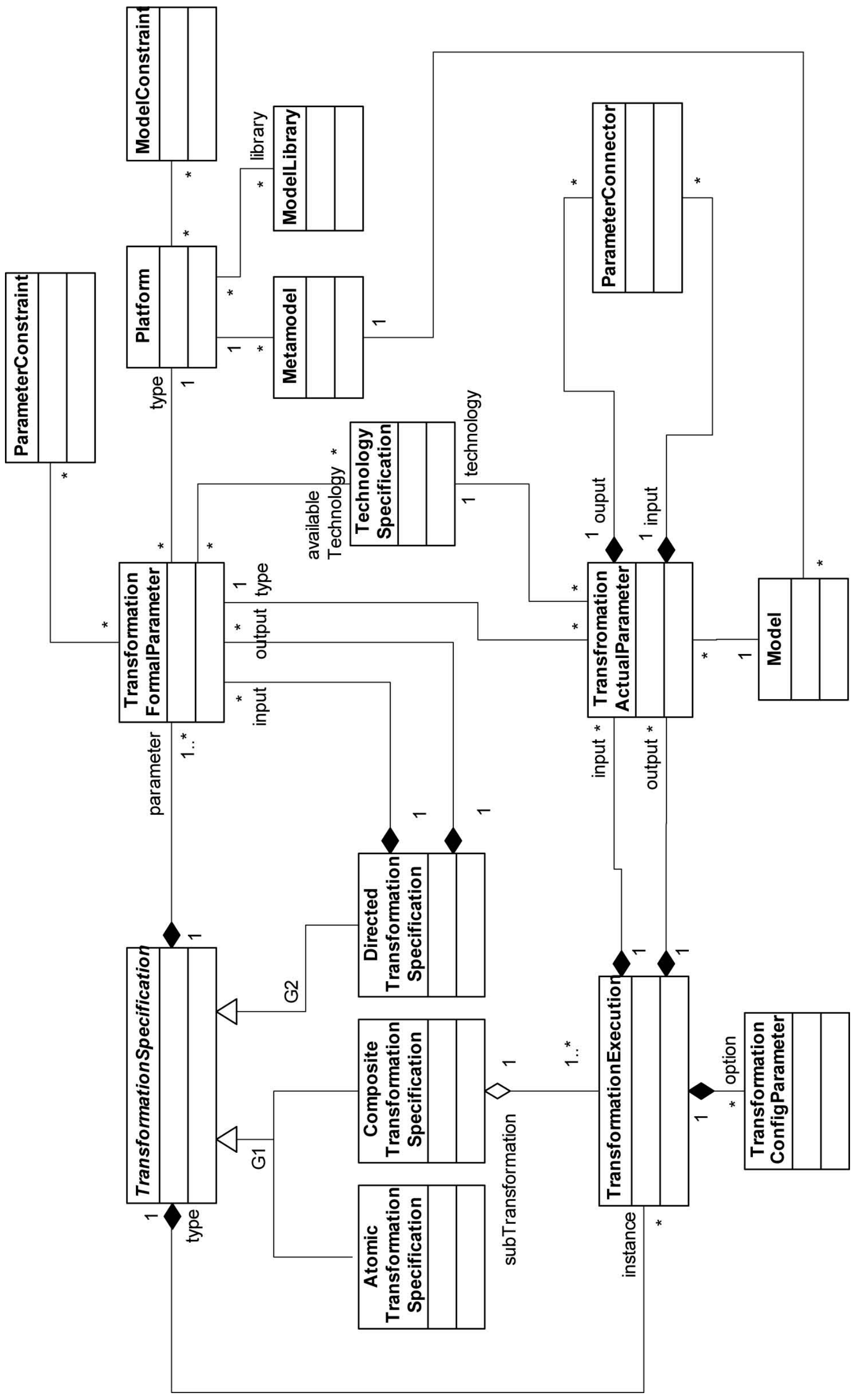

Fig. 3. The metamodel to specify transformation chains

A *TransformationExecution* is always directed (an undirected *TransformationSpecification* becomes directed when it is executed) and has a number of *TransformationActualParameter*s, typed by the *TransformationFormalParameter*s of the related *TransformationSpecification*. The *Model*, referred to by the actual parameter must adhere to platform of the formal parameter. Each *TransformationActualParameter* in the role of *output* can be connected to one or more other *TransformationActualParameter*s in the role of input through a *ParameterConnector*.

The *TechnologySpecification* element is in place to be able to define technical specifications of transformation in- and outputs besides their types of metamodels. This is needed because even if two models adhere to the same metamodel, they can be expressed using a number of different technologies (e.g. XMI v1.x, HUTN – Human Useable Textual Notation, JMI – Java Metadata Interface). Each *TransformationActualParameter* belongs to a concrete implemented transformation so it has to specify a technology for its model. In case of a *CompositeTransformationSpecification* the *TechnologySpecification*s of the containing *TransformationActualParameter*s will be propagated to the *TransformationFormalParameter*s (hence the association between *TransformationFormalParameter* and *TechnologySpecification*).

The issue of specifying type (*Platform* and *ParameterConstraint*) and technology (*Technology*) of transformation parameters is related to interoperability between transformations. This subject is extensively addressed in [6], where a distinction is made between functional (types) and protocol (technology) connectivity.

To make the metamodel complete we need to add some additional constraints, for example to ensure that a *Model* bound to a *TransformationActualParameter* is compliant with the *Metamodel* that can be reached though the associated *TransformationFromalParameter*. We do not show these constraints due to space restrictions.

In the following section we attach a concrete syntax to our conceptual metamodel.

4 Transformation Chain Profile

A metamodel is worth little without an accompanying concrete modeling language to specify its models. There are roughly two options to specify a concrete syntax:

- A heavyweight approach: define a completely new language with its own symbols (DSML) or extend an existing language (e.g. UML) at the meta level. This approach allows the most freedom in tailoring the language to your own taste.
- A lightweight approach: adapt an existing language to your needs. In this case the base language needs to support a kind of extension mechanism. The UML is probably the most well-known language that allows such adaption in the form of UML profiles (stereotypes, tagged values and constraints).

The first approach is conceptually the best but it has some practical drawbacks. Having to precisely define semantics besides abstract and concrete syntax from the ground up together with the need for custom tool support kept us from applying this approach. The UML on the contrary contains the *Activity package* that is used to model actions executed against a flow of objects, which is similar to transformations and models flowing between them. Therefore we chose to define a UML profile that tailors the standard activity diagrams to our specific needs. Also, both the MEDAL [7] and VMT

[8] approaches to MDD make use of activity diagrams to specify transformations but they operate at the transformation implementation level instead of at the transformation chaining level.

In figure 4, we show a mapping of the transformation concepts from the metamodel of figure 3 to stereotypes and tagged values. The figure is only shown as an illustration and does certainly not contain the complete mapping, which would take too much space. We also do not show constraints to prevent the use of unwanted activity elements such as *ControlFlow* and *CentralBufferNode*.

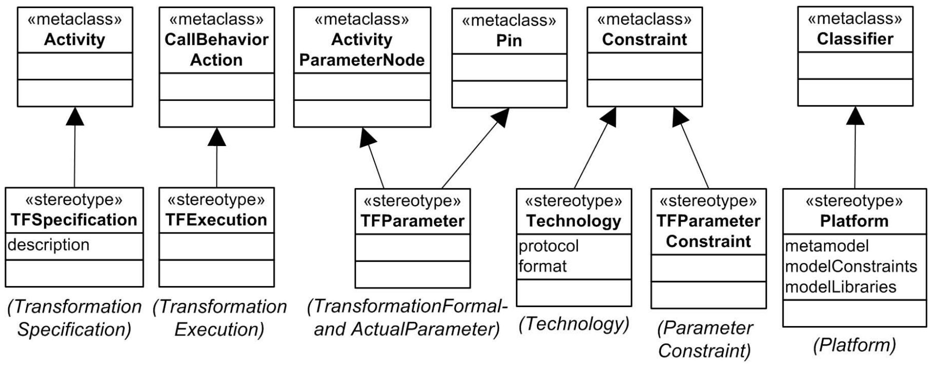

Fig. 4. Partial mapping of the metamodel elements (in italic) to a UML profile; only the most important stereotypes and tagged values are shown, constraints are omitted

The figure shows the UML metaclasses that are specialized using stereotypes and refers to the original metamodel element. The mapping specializes the *Activity* metaclass with the *TFSpecification* stereotype. A *TFSpecification* is *atomic*, if it does not contain any *TFExecution*s (a specialization of *CallBehaviorAction*) or is *composite* when it does. *ActivityParameterNode*s as well as *Pin*s must be stereotyped with *TFParameter*, respectively indicating a formal or an actual parameter. Two types of *Constraint*s are introduced: *Technology* and *TFParameterConstraint*. Finally a *Platform* is a specialization of the *Classifier* element. Besides these, many other UML Activity elements need to be specialized or excluded from the model in order to make the mapping complete.

Because the metamodel-profile mapping still leaves much to the imagination, we give some examples using the concrete activity diagram notation in the next subsection.

4.1 Example of a Transformation Chain Model

Comprehending a modeling language is the easiest when looking at examples of its concrete syntax. We therefore provide two examples.

Our first example is shown in figure 5. It specifies a transformation component that transforms between domain specific models of cellphone applications and UML component models.

We can see that the Phone2UML *TFSpecification* contains two parameters and has a description, which should be more detailed in a real situation. The *TFParameter*s are

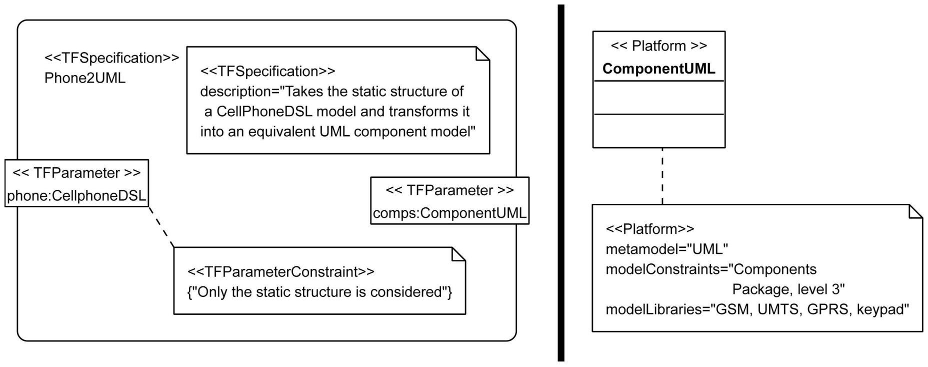
<<TFSpecification>>
Phone2UML
<<TFSpecification>>
description="Takes the static structure of a CellPhoneDSL model and transforms it into an equivalent UML component model"
<< TFParameter >>
phone:CellphoneDSL
<< TFParameter >>
comps:ComponentUML
<<TFParameterConstraint>>
{"Only the static structure is considered"}
<< Platform >>
ComponentUML
<<Platform>>
metamodel="UML"
modelConstraints="Components Package, level 3"
modelLibraries="GSM, UMTS, GPRS, keypad"

Fig. 5. Example of an atomic transformation specification (on the left) and an accompanying platform specification (on the right)

typed by a platform, specified by the *Platform* classifier. The *ComponentUML* platform on the right is given as an example. It is based on the UML *metamodel*, constrained to use component package at compliance level 3 (*modelConstraints*) and includes some cellphone specific *modelLibraries*. The input parameter has an additional *TFParameterConstraint* saying that only the static structure of our cellphone model will be taken into account.

The second example (figure 6) shows a composite transformation that has two inputs and one output and is specified by an internal structure of two *TFExecution*s. The top one (*Phone2UML*) is reused from figure 5. The *TFExecution* at the bottom

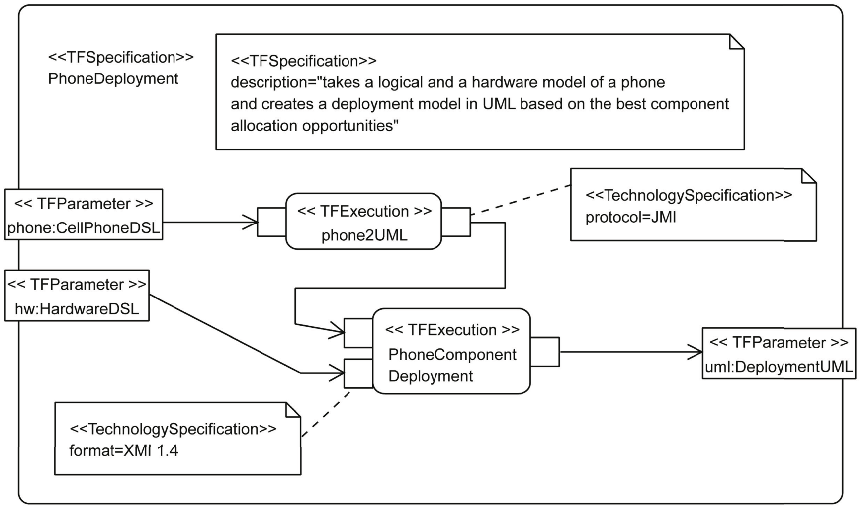
<<TFSpecification>>
PhoneDeployment
<<TFSpecification>>
description="takes a logical and a hardware model of a phone and creates a deployment model in UML based on the best component allocation opportunities"
<< TFParameter >>
phone:CellPhoneDSL
<< TFExecution >>
phone2UML
<<TechnologySpecification>>
protocol=JMI
<< TFParameter >>
hw:HardwareDSL
<< TFExecution >>
PhoneComponent Deployment
<< TFParameter >>
uml:DeploymentUML
<<TechnologySpecification>>
format=XMI 1.4

Fig. 6. Example of a composite TransformationSpecification

(*PhoneComponentDeployment*) takes the UML component model (output from *Phone2UML*) together with a hardware model for a specific type of phone and generates the most optimal UML deployment model. We can further see that two *TechnologySpecification*s are given: the output of *Phone2UML* is accessible through a Java Metadata Interface (JMI), while the second input of the *PhoneComponentDeployment* is required to be given in XMI format.

The given examples just give a flavor of what can be specified in a transformation chain model. Real world models will need to be more detailed in many ways, for example in the specification of metamodel and parameter technology. More formal specification of constraints on input/output parameters can be done using the OCL-based approach of [9].

5 Conclusions and Future Work

An important part of the effort in an MDD-based project lies in the development of an appropriate transformation chain, which in turn eases the construction of the application(s) described in the project. Having multiple levels of transformations facilitates transformation reuse, especially in product-line oriented development, which is often the case for embedded systems.

Being faithful to the MDD philosophy, transformation chains also have to be designed and modeled before implementation. We have proposed a transformation chain modeling language of which we defined the abstract syntax using a metamodel. This metamodel is an elaboration of the MDA Foundation Model proposal. We then mapped the metamodel's elements onto UML's Activity Diagrams, which are well-suited, though not ideal, to model transformation chains.

The design of a transformation chain is only part of a project-specific MDD infrastructure. In order to support the concrete realization of transformation chains we will design and implement a transformation chaining framework that uses the proposed language to allow easy concatenation of transformation components that may be implemented in different transformation languages. The results of experimenting with this infrastructure will be used to refine the proposed transformation language. We are also developing a design process to guide the development of transformation chains [10]. This process should help MDD developers in identifying the correct transformation components, platforms, etc.

References

1. Object Management Group: Meta object facility 2.0 core specification. Misc (2004)
2. Object Management Group: Uml 2.0 superstructure conv. document. Misc (2004)
3. Object Management Group: Qvt-merge group submission for mof 2.0 query/view/transformation. Misc (2005)
4. Almeida, J.P., Dijkman, R.M., van Sinderen, M., Pires, L.F.: On the notion of abstract platform in mda development. In: EDOC. (2004) 253–263
5. Object Management Group ORMSC: A proposal for an mda foundation model, white paper (2005)

6. Blanc, X., Gervais, M.P., Sriplakich, P.: Model bus: Towards the interoperability of modelling tools. In: MDAFA. (2004) 17–32
7. Guelfi, N., Ries, B., Sterges, P.: MEDAL: A CASE Tool Extension for Model-Driven Software Engineering. In: SWSTE '03: Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering, Washington, DC, USA, IEEE Computer Society (2003) 33
8. Sendall, S., Perrouin, G., Guelfi, N., Biberstein, O.: Supporting model-to-model transformations: The vmt approach. Technical report (2003)
9. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: Ocl for the specification of model transformation contracts. In Patrascoiu, O., ed.: OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal, University of Kent (2004) 69–83
10. Vanhooff, B., Ayed, D., Berbers, Y.: Towards a Transformation Chain Design Process. (2006)