# Automatic Definition of Model Transformations at Instance Level

Adrian Rutle[1], Alessandro Rossini[2], Yngve Lamo[1] and Uwe Wolter[2]
[1]Faculty of Engineering, Bergen University College, Norway
[2]Department of Informatics, University of Bergen, Norway

## ABSTRACT

A model transformation is the generation of a target model from a source model. Usually, it is defined for metamodels, i.e. models at the meta-level, and executed by a transformation engine to transform instances of those metamodels. In some cases, it is also desired to transform the instances of the transformed models. In this paper, we use the Diagram Predicate Framework to show how model transformations which are defined at the metamodel level can be used as guidelines to automatically define model transformations at the model level. This requires a special relationship between the metamodel and the instances of its instances in order to inherit all the properties of the transformations from the metamodel level. A formalisation of this relationship is outlined in this paper.

## 1. INTRODUCTION AND MOTIVATION

Software models are abstract representations of software systems. These models are used to tackle the complexity of present-day software systems by enabling developers to reason at a higher level of abstraction. In Model-Driven Engineering (MDE), we refer to four levels of abstraction, which are summarized in the following *modelling hierarchy*:

$$\begin{tikzcd}
M_3 \arrow[loop above, "conformsTo"] \\
M_2 \arrow[u, "conformsTo"'] \\
M_1 \arrow[u, "conformsTo"'] \\
M_0 \arrow[u, "conformsTo"']
\end{tikzcd}$$

Models at the $M_0$-level are called *instances* which represent the running system. These instances must conform to *models* at the $M_1$-level, which are structures specifying what instances should look like. These models, in their turn, must conform to a *metamodel* at the $M_2$-level, against which models can be checked for validity. Metamodels correspond to modelling languages, for example the Unified Modeling Language (UML) and Common Warehouse Model (CWM). The highest level of abstraction (as defined by the Object Management Group [2]) is the $M_3$-level. A model at this level is often called *meta-metamodel*; it conforms to itself and it is used to describe metamodels.

In MDE models are the primary artefacts of the development process. Model transformations play a central role and have many applications in MDE, such as; model integration, model refinement, model evolution, multi-modelling, and code-generation, to mention a few. These transformations are defined at the metamodel level, and executed at the model level. For example, when we want to transform a Java object, we define the transformation for its class. In the same way, when we want to transform a Java class, we define the transformation for the meta-type of Java classes, which is the class `Class`.

An advantage of using the metamodel level is to enable the definition of transformations at a higher level of abstraction. Another advantage is to work with a smaller set of model elements. This is because metamodels are typically smaller and more compact than their instances.

The focus of this paper will be on the automatic definition of transformations at the model level based on the transformation definition at the metamodel level. This kind of automatisation is investigated algebraically using pullbacks [1], however, our approach is more generic and is not bound to a specific algebraic operation. A requirement for the automatisation is the existence of a special relationship between metamodels and the instances of their instances. A short description of this relationship is given in Section 2.

## 2. DIAGRAM PREDICATE FRAMEWORK

Diagram Predicate Framework (DPF)[1] [3, 4, 5] provides a formal approach to modelling based on category theory – the mathematics of graph-based structures. In DPF each modelling language $L$ corresponds to a diagrammatic signature $\Sigma_L$ and a metamodel $MM_L$. $L$-models are represented by $\Sigma_L$-specifications[2] which consist of a graph and a set of constraints. The graph represents the structure of the model, and predicates from $\Sigma_L$ are used to add the constraints to the graph [5]. Signatures, constraints and diagrammatic specifications are defined as follows:

*Definition 1.* A (diagrammatic predicate) signature $\Sigma := (\Pi, \alpha)$ is an abstract structure consisting of a collection of predicate symbols $\Pi$ with a mapping that assigns an arity (graph) $\alpha(p)$ to each predicate symbol $p \in \Pi$.

*Definition 2.* A constraint $(p, \delta)$ in a graph $G(M)$ is given by a predicate symbol $p$ and a graph homomorphism $\delta : \alpha(p) \to G(M)$, where $\alpha(p)$ is the arity of $p$.

*Definition 3.* A $\Sigma$-specification $M := (G(M), M(\Pi))$, is a graph $G(M)$ with a set $M(\Pi)$ of constraints $(p, \delta)$ in G(M) with $p \in \Pi$.

Fig. 1a shows an example of a $\Sigma$-specification $M = (G(M), M(\Pi))$. $G(M)$ in Fig. 1b is the graph of $M$ without any

[1]Formerly named Diagrammatic Predicate Logic (DPL).
[2]For the rest of the paper we use the terms “model” and “diagrammatic specification” interchangeably.

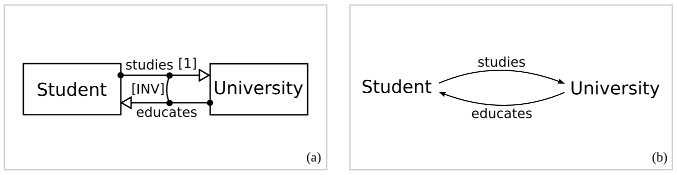

**Figure 1: A Diagrammatic Specification: (a) $M = (G(M), M(\Pi))$, (b) its graph $G(M)$.**

constraints on it. In $M$, every university educates *one or more* students; this is forced by the constraint $([total], \delta_1)$ on the arrow *educates*, which is visualised as a filled circle at the beginning of the arrow. Moreover, every student studies at *exactly one* university; this is forced by the constraint $([single\text{-}valued], \delta_2)$ on the arrow *studies*, which is visualised as the marker `[1]` at the cap of the arrow.

A signature with details of the semantics of its predicates is shown in [5]. Here, only an informal definition of the concept of instances of diagrammatic specifications is given. An instance $(\iota_M, I)$ of a diagrammatic specification $M$ is a graph homomorphism $\iota_M : I \to G(M)$, i.e. the elements of $I$ are typed by $G(M)$, the graph of $M$, such that the constraints in $M$ are satisfied. The set of the instances of a diagrammatic specification $M$ is denoted $Inst(M)$.

As mentioned, in a modelling hierarchy each model at $M_i$-level conforms to a model at $M_{i+1}$-level, for $0 \leq i < 3$. In a modelling hierarchy with transitive conformance, if an instance $I$ at $M_0$-level conforms to a model $M$ at $M_1$-level, and $M$ in its turn conforms to a metamodel $MM$ at $M_2$-level, then $I$ also conforms to $MM$. This property is formalised in the next definition; its importance for the automatic definition of model transformations at the model level based on transformation definition at the metamodel level is explained in Sec. 3.

*Definition 4.* In a modelling hierarchy with transitive conformance, given any diagrammatic specification $MM$, for each $(\iota_{MM}, M) \in Inst(MM)$ if $(\iota_M, I) \in Inst(M)$ then $(\iota_M; \iota_{MM}, I) \in Inst(MM)$.

## 3. MODEL TRANSFORMATIONS

As an example, suppose we want to generate a relational database model from a class model. The class model contains a hierarchy of classes with references between them. The relational database model contains a set of database tables and relationships between them. We can easily develop a model transformation to transform the elements of a specific class model to a database model. Alternatively, we can define a generic transformation which can be used to transform any class model to a database model. Obviously the former procedure is less convenient since it implies that for every class model we have to define a new transformation. To achieve the latter, we define the transformation for the metamodels of class models and database models. Then we use a transformation engine to (automatically) transform any model that conforms to the metamodel of class models, to a model which conforms to the metamodel of relational databases.

Moreover, in some cases, we want also to transform instances of these class models. Thus we have to define a new model transformation between the models themselves. The

$$\begin{tikzcd}
MM \arrow[r, "MT-def."', ""{name=U}] & MM' \\
Inst(MM) \arrow[r, Rightarrow, "MT-exec."', ""{name=D}] & Inst(MM')
\arrow[from=U, to=D, Rightarrow, "MT-engine"]
\end{tikzcd}$$

**Figure 2: Model transformations: overview.**

$$\begin{tikzcd}
 & P \arrow[ld, "\iota_P"'] \arrow[d, "m", ""{name=A}] & K \arrow[l, hook', "in_P"'] \arrow[r, hook, "in_{P'}"] & P' \arrow[d, "m'"', ""{name=B}] \arrow[rd, "\iota_{P'}"] & \\
MM & M \arrow[l, "\iota_{MM}"] \arrow[lu, phantom, "{[=]}"] & & M' \arrow[r, "\iota_{MM'}"'] \arrow[ru, phantom, "{[=]}"] & MM'
\arrow[from=A, to=B, mapsto, "[\![r]\!]"]
\end{tikzcd}$$

**Figure 3: A transformation rule $r : P \to P'$.**

results of this paper enables us to reuse the model transformation which was defined at the metamodel level to define the model transformation at the model level. This is outlined in Sec. 3.1. But first, we have to explain some concepts which are used in model transformations.

Fig. 2 shows the overview of model transformations. Each model transformation is given by a transformation definition *MT-def* which describes how instances of a source model $MM$ can be transformed to instances of a target model $MM'$. The transformation definition is specified in a transformation language, and is executed by a transformation engine. That is, given a model transformation definition, a source model, and a target model, the transformation engine tries to create an instance of the target model for each instance of the source model.

Every transformation definition consists of a set of transformation rules. Each rule defines which elements of the source instances are to be transformed to which elements of the target instances. This is done through an input pattern $P$ and an output pattern $P'$, as shown in Fig. 3. Thus the transformation engine generates a match of the output pattern in the target instance whenever it finds a match of the input pattern in the source instance. Details of how the transformation engine is controlled and how patterns are coordinated (through the coordination set $K$) are given in [4]. Here, patterns and matches of patterns are defined as follows:

*Definition 5.* A pattern $P$ over a diagrammatic specification $MM$ is an instance $(\iota_P, P)$ of $MM$, where the items in $P$ are formal parameters or variables.

*Definition 6.* A match $m : P \to M$ of a pattern $P$ in an instance $(\iota_{MM}, M)$ of a diagrammatic specification $MM$ is a graph homomorphism $m : P \to M$ where variables in $P$ are assigned values from $M$, and $m; \iota_{MM} = \iota_P$.

For a match $m : P \to M$, the analogy is that $P$ is the formal parameter of $m$ and some part of $M$ is the actual parameter. That is, a pattern $P$ can be seen as a scheme and a match $m$ assigns values from $M$ to the variables in $P$. We use $Match(P)$ to denote the set of all matches of $P$ in $Inst(MM)$, i.e. all matches of $P$ in all instances $(\iota_{MM}, M)$ of $MM$. Moreover, we use $Match^M(P)$ for $M \in Inst(MM)$ to denote the set of all matches of the pattern $P$ in $M$.

Recall that each transformation definition consists of a set of transformation rules. These rules and their semantics are defined as follows:

*Definition 7.* A transformation rule $r$ is declared by $r : (\iota_P, MM) \to (\iota_{P'}, MM')$ (abbreviated $r : P \to P'$) where both $P$ and $P'$ are patterns over the models $MM$ and $MM'$, respectively.

*Definition 8.* The semantics of a transformation rule $r : P \to P'$ is a mapping $[\![r]\!]$ which assigns to each match $m \in Match(P)$ a match $m' \in Match(P')$, i.e. $[\![r]\!] : Match(P) \to Match(P')$.

## 3.1 Automatisation of Transformations

An overview of the automatic construction of transformation rules at the model level is shown in Fig. 4. Based on the execution of each transformation rule $r : P \to P'$ at the metamodel level, a (set of) transformation rule(s) $r^*$ at the model level will be created. These transformation rules use $\mathcal{P}^M$, where $M \in Inst(MM)$ and $Match^M(P) \neq \emptyset$, as input patterns; and $\mathcal{P}^{M'}$, where $M' \in Inst(MM')$ and $Match^{M'}(P') \neq \emptyset$, as output patterns. These patterns are defined as follows:

*Definition 9.* $\mathcal{P}^M$ is the set of input patterns such that $\forall m(P) \in Match^M(P) : \exists P^* \in \mathcal{P}^M$, and $\mathcal{P}^{M'}$ is the set of input patterns such that $\forall m'(P') \in Match^{M'}(P') : \exists P^{*'} \in \mathcal{P}^{M'}$.

The input and output patterns in $\mathcal{P}^M$ and $\mathcal{P}^{M'}$ are created by adding a *free variable* to the matches of the patterns of $r$. In Fig. 5, these constructions are abbreviated as $f : M \Rightarrow \mathcal{P}^M$ and $g : M' \Rightarrow \mathcal{P}^{M'}$. For example, if a transformation rule $r$ takes the pattern $P =$ `x:Class` as input and the pattern $P' =$ `x:Table` as output, then for the input matches `Student:Class`[3] and `University:Class` in the model $M$, the construction $f^* : Match^M(P) \to \mathcal{P}^M$ will create `x`$_1$`:Student:Class` and `x`$_2$`:University:Class` as input patterns $P_1^*$ and $P_2^*$ for the rules $r_1^*$ and $r_2^*$, respectively. Similarly, for the output matches `Student:Table` and `University:Table` in the model $M'$, the construction $g^* : Match^{M'}(P') \to \mathcal{P}^{M'}$ will create `x`$_1$`:Student:Table` and `x`$_2$`:University:Table` as output patterns $P_1^{*'}$ and $P_2^{*'}$ for $r_1^*$ and $r_2^*$, respectively.

*Definition 10.* $f^* : Match^M(P) \to \mathcal{P}^M$ is a construction such that, $\forall m(P) \in Match^M(P)$, $f^*(m(P)) = P^*$ where $P^*$ is a pattern over $M$. $g^*$ is defined similarly.

*Theorem 1.* Given a transformation rule $r : P \to P'$ at the metamodel level, it is possible to create a set of transformation rules $\mathcal{R}^* : \mathcal{P}^M \to \mathcal{P}^{M'}$ at the model level.

*Corollary 1.* The semantics of a transformation rule $r_i^* \in \mathcal{R}^* : \mathcal{P}^M \to \mathcal{P}^{M'}$ is a mapping $[\![r_i^*]\!]$ which assigns to each match $m^* : P^* \in \mathcal{P}^M \to I$ a match $m^{*'} : P^{*'} \in \mathcal{P}^{M'} \to I'$, i.e. $[\![r_i^*]\!] : Match(\mathcal{P}^M) \to Match(\mathcal{P}^{M'})$.

In modelling hierarchies with transitive conformance, the application of automatically generated model transformations at the $M_1$-level – which are based on model transformations at the $M_2$-level – will generate models at the $M_0$-level which are instances of the target metamodel. This feature is important in order to inherit the properties of the model transformations which are defined at a higher level of abstraction. For example, if a model transformation at the $M_2$-level is correct, then the automatically constructed model transformation at the $M_1$-level will also be correct.

[3]Notice that (`Student:Class`) is a "user-friendly" notation for the assignment ($\iota_{MM} : Student \mapsto Class$).

$$\begin{tikzcd}
MM \arrow[rr, "MT\text{-}def."', ""{name=A}] & & MM' \\
M \arrow[u, "\iota_{MM}"] \arrow[rr, Rightarrow, "MT\text{-}exec."', ""{name=B}] \arrow[rr, bend right, "MT\text{-}def."', ""{name=C}] & & M' \arrow[u, "\iota_{MM'}"'] \\
I \arrow[u, "\iota_M"] \arrow[rr, Rightarrow, "MT\text{-}exec."', ""{name=D}] & & I' \arrow[u, "\iota_{M'}"']
\arrow[Rightarrow, from=A, to=B, "MT\text{-}engine"]
\arrow[Rightarrow, from=B, to=C, "Auto"]
\arrow[Rightarrow, from=C, to=D, "MT\text{-}engine"]
\end{tikzcd}$$

**Figure 4: Model transformations: automatisation.**

$$\begin{tikzcd}
 & P \arrow[dl, "\iota_P"'] \arrow[d, "m"] \arrow[dl, phantom, "{[=]}"] & K \arrow[l, hook', "in_P"'] \arrow[r, hook, "in_{P'}"] & P' \arrow[d, "m'"'] \arrow[dr, "\iota_{P'}"] \arrow[dr, phantom, "{[=]}"'] & \\
MM & M \arrow[l, "\iota_{MM}"] \arrow[d, Rightarrow, bend left, "f"] \arrow[rr, phantom, "{m \overset{[\![r]\!]}{\longmapsto} m'}"] & & M' \arrow[r, "\iota_{MM'}"'] \arrow[d, Rightarrow, bend right, "g"'] & MM' \\
 & \mathcal{P}^M \arrow[d, bend left, "m^*"] \arrow[rr, phantom, "{m^* \overset{[\![r_i^*]\!]}{\longmapsto} m^{*'}}"] & & \mathcal{P}^{M'} \arrow[d, bend right] & \\
 & I \arrow[uu, bend left, "\iota_M"] \arrow[uul, "\iota_M;\iota_{MM}"] \arrow[uul, phantom, "{[=]}"'] & & I' \arrow[uu, bend right, "\iota_{M'}"'] \arrow[uur, "\iota_{M'};\iota_{MM'}"'] \arrow[uur, phantom, "{[=]}"] &
\end{tikzcd}$$

**Figure 5: Creation of the transformation rules $r^*$ based on the transformation rule $r$.**

## 4. SUMMARY

Transformation rules which are defined between metamodels can be used to automatically derive transformation rules between models. Moreover, if this technique is used for modelling hierarchies with transitive conformance, the properties of the model transformations at the $M_2$-level are preserved in the model transformations at the $M_1$-level.

## 5. REFERENCES

[1] Z. Diskin and J. Dingel. A metamodel Independent Framework for Model Transformation. Technical Report 1/2006, ATEM 2006, Johannes Gutenberg Universität Mainz, Germany, October 2006.

[2] Object Management Group. *Web site.* `http://www.omg.org`.

[3] A. Rutle, U. Wolter, and Y. Lamo. Diagrammatic Software Specifications. In *NWPT 2006*, October 2006.

[4] A. Rutle, U. Wolter, and Y. Lamo. A Diagrammatic Approach to Model Transformations. In *EATIS 2008*, pages 1–8. ACM, 2008.

[5] A. Rutle, U. Wolter, and Y. Lamo. A Formal Approach to Modeling and Model Transformations in Software Engineering. Technical Report 48, Turku Centre for Computer Science, Finland, 2008.