Extending the DPO Approach for Topological Validation of Metamodel-Level Graph Rewriting Rules

TIHAMÉR LEVENDOVSZKY, LÁSZLÓ LENGYEL, HASSAN CHARAF
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann Gy. tér 3, Budapest, H-1111
HUNGARY

Abstract: - Model transformation systems are an important contribution to the field of automated software engineering. This paper summarizes the theoretical background implemented in Visual Modeling and Transformation System (VMTS) to validate the topology of the transformation steps. The mathematical formalism is based on the double pushout approach, which is extended to rewriting steps consisting of metamodel elements. Firstly the instantiation relationship is transformed to a homomorphic mapping, then the theorems from DPO approach is applied in conjunction with the validation-related propositions. The theoretical results are accompanied with practical considerations throughout the paper, and the propositions are turned into algorithms to facilitate their applications.

Key-Words: - Software model transformation, graph rewriting, DPO approach, metamodeling, VMTS

1 Introduction

Due to the standardization of the Unified Modeling Language [1] and the Model-Driven Architecture [2] the demand for model transformation systems has been increased. The Visual Modeling and Transformation System (VMTS) [3][4] was developed to demonstrate the feasibility of graph rewriting-based model transformation systems built on metamodeling techniques. This paper summarizes the theoretical results applied by VMTS to perform topological validation on the transformation steps.

For the sake of notation conventions and the seamless discussion we give a definition for labeled directed graphs (LDG) homomorphism. These follow the conventions applied in [5].

Definition 1. (Labeled directed graphs, LDG):
Let Ω_V and Ω_E be two given alphabet for node and edge labels, respectively. Then the labeled directed graph is a six-tuple:

$$G = \left\langle G_V, G_E, s^G, t^G, lv^G, le^G \right\rangle, \quad (1)$$

where G_V is the set of vertices, G_E is the set of edges; s^G and t^G are the source and target functions ($s^G, t^G : G_E \to G_V$), which map an edge to its source and target vertices, respectively; and finally lv^G: $G_V \to \Omega_V$ and le^G: $G_E \to \Omega_E$, which assign a label to a vertex and an edge from the appropriate alphabet.

We can consider UML models as an LDG, where the node and edge information (type, attribute) is stored in the label. One may store this information as an XML document (a standard approach to this can be found in OMG's XMI specification [6]). Regarding the labels of the vertices and the edges as an XML document can be a suitable implementation strategy for a UML class diagram applied for instance in [4]. To get closer to the category theory framework we define graph homomorphism.

Definition 2. (graph homomorphism for LDGs):
A graph homomorphism f: G→H is a pair of functions: $f = \left\langle f_V : G_V \to H_V, f_E : G_E \to H_E \right\rangle$, which preserves the sources, targets and labels, that is, it satisfies the following conditions:

- $f_V \circ t^G = t^H \circ f_E$
- $f_V \circ s^G = s^H \circ f_E$
- $lv^H \circ f_V = lv^G$
- $le^H \circ f_E = le^G$

One can create a directed graph: where the vertices are LDGs and the edges are homomorphic mapping between the LDG vertices. Thus we have obtained an informal notion of the category of the directed graphs (**Graph**).

2 Related Work

Graph rewriting [5][7] is a powerful tool for graph transformations with strong mathematical background. Originally it was developed as the natural generalization of Chomsky grammars to generate and parse visual languages. Instead of the graph language approach we will use the mechanism of the individual parsing steps, so-called rewriting rules, for graph transformations. The graph transformation is defined as a sequence of rewriting rules, where each such rule is a pair of graphs, called the LHS (left hand side) and RHS (right hand side). The rewriting rule operates as follows: the LHS of a rule is matched against the input graph (or host graph), and the matching portion of that graph is replaced with the RHS of the rule. The replacement process is referred to as firing of the rewriting rule. Model transformation systems mainly use context elements [7] to describe the rewriting rules that change the edges between objects.

Algebraic graph rewriting [5] provides a way to manipulate objects in a graph category, where the objects are labeled directed graphs, and the arrows are graph homomorphisms. There are two main branches of algebraic graph rewriting, namely the double pushout (DPO) and the single pushout (SPO) approaches.

The DPO approach accomplishes the rule firing by two steps: after finding a redex (the part of the host graph matched by the rule), the first step removes the elements (vertices and edges) from the redex which are in the redex, but not in the RHS graph. This modified redex is referred to as interface graph. Then as a second step the elements of the RHS graph not in the interface graph but in the RHS graph are glued to the interface graph. The rewriting rule is characterized by a double pushout. The application of the rules results in a direct derivation of the host graph. The category theory framework provides a more flexible and more general background, so the DPO approach can be applied to many graph-like categories. For labeled and directed graphs the existence of the pushout (which is the condition to fire a rule) can be ensured by forcing the so-called gluing condition. The gluing condition consists of two parts. Firstly, the identification condition, which states that different vertices in the production rule cannot match the same vertex in the host graph. Secondly, the dangling edge condition has to be dealt with as well: if a vertex should be deleted which is connected to an edge that is not inside the redex, the production rule cannot be fired. Unfortunately, this makes impossible to delete a connected vertex without considering its environment.

The single pushout approach uses partial graph homomorphisms to form a single pushout as a rule firing condition, and if a conflict occurs when violating the gluing condition, deletion has priority over preservation.

In the algebraic approaches there is a central topic: the parallel and sequential independence. Two alternative direct derivations are parallel independent, if their redexes overlap in elements (edges or vertices) which are preserved by both derivations. Two consecutive derivations are sequentially independent if the second one is not dependent on the elements added by the first one, and the second one does not delete any items accessed by the first one. In the DPO approach the two independences are equivalent, because the equivalence condition is buried in the gluing condition. To take advantage of this property we have chosen the DPO approach as a formal background for our applications.

3 Contributions

In VMTS there are a few assumptions we used in establishing the formal model. Since VMTS is a metamodeling environment, the metamodel of the input model and the output model is always available. The rewriting rules consist of metamodel elements, and instead of an occurrence an instantiation has to be found in the host graph (input model). These concepts and their practical applications are deeply discussed in [4].

3.1 A Formal Instantiation Relationship

In order to treat the concept of the instantiation within the framework of category theory we need to ensure a homomorphic mapping between the model elements and their metaelements.

First we introduce the notion of type preserving mapping. Let graph *Meta* be a UML class diagram. Let graph *Instance* be a UML object diagram instantiating *Meta*. Type Preserving Mappings (TPM) are mappings from *Meta* to *Instance*, such that every model element in *Instance* is mapped to its type element in *Meta*. This mapping not necessarily utilizes all meta elements (if one considers abstract classes, for instance) in *Meta*. Before resolving this issue we define Type Preserving Mapping (TPM) in a more formal way.

Definition 3. (Type Preserving Mapping):
Let *Meta* be an LDG, with labels conforming to the UML class diagram. Let be another LDG *Instance* given with labels conforming the UML object diagram. Let us further assume that *Instance* instantiates *Meta* according to the instantiation rules enforced by the UML. There are two functions

$$i_V : lv^{Meta}, lv^{Instance} \rightarrow \mathrm{Boolean} \qquad (2)$$

$$i_E : le^{\text{Meta}}, le^{\text{Instance}} \rightarrow \text{Boolean} \qquad (3)$$

which return true if and only if the *Meta* graph element having the label given as the first function argument is the type of the *Instance* element having the label given as the second argument. A mapping between an Instance and a Meta LDG is called Type Preserving Mapping (TPM) if and only if the Instance element is mapped to its type elements and to no other elements.

Practically, if we consider an XMI-like approach outlined above, i_V and i_E can be implemented as simple comparison between the XML tag representing the type of the *Instance* element, and the XML tag holding the name of the *Meta* element. In other tools this relationship may be denoted by a reference or other linking methods peculiar to the applied programming environment. In the following statement we connect graph homomorphism and TPM.

Proposition 1. Let *Meta* be an LDG with labels conforming to the UML class diagram. Let another LDG *Instance* be given with labels conforming the UML object diagram. We further assume that *Instance* instantiates *Meta* according to the instantiation rules enforced by the UML. If *Meta* does not contain inheritance relationship (nor abstract classes, which would be semantically meaningless without inheritance) then the TPM between the *Instance* and *Meta* graph is a graph homomorphism.

Proof. First a TPM is created between *Instance* and *Meta*. Because every vertex and edge has a type, every *Instance* element participates in the mapping. Since there is no inheritance every instance element is mapped to only one *Meta* element. It means that this mapping is a pair of function.
Now it is shown that the adjacent vertices of the *Instance* graph are mapped to adjacent vertices in *Meta*. Suppose we have :A and :B vertices which are connected in *Instance* via link :l, but their types (A and B, respectively) are not connected in *Meta* with L which is the association type for :l (Fig. 1). It implies that *Instance* has violated the rules of instantiation (defined by the UML standard), the *Instance* is not the instance of the *Meta*. That contradicts the condition of the proposition.

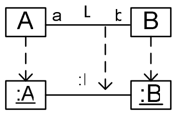

Fig. 1. Instantiation

Proposition 2. Let *Meta* be an LDG with labels conforming to the UML class diagram. Let another LDG *Instance* be given with labels conforming the UML object diagram. In addition we assume that *Instance* instantiates *Meta* according to the instantiation rules enforced by the UML. Let A_n denote the set of the associations connected to the *n*th layer of a class hierarchy. Let L_n represent the set of the links instantiating elements of A_n. *Instance* can be mapped to *Meta* via graph homomorphism if and only if no :O object is attached to link from more than one L_n set, where A_n belongs to the class hierarchy of C, which is the type of :O.

Proof. Firstly, it is shown, that if the conditions in the proposition are satisfied, a TPM exists. The objects of the classes not participating in an inheritance hierarchy are mapped to their types. The objects of the classes participating in an inheritance hierarchy are mapped to the class which has the association instantiated by the link attached to the given object. This mapping is a TPM. Then it is demonstrated that the adjacent vertices of *Instance* are mapped to adjacent vertices of *Meta*. For vertices not participating in the inheritance hierarchy the statement holds based on Proposition 1. For the vertices participating in inheritance hierarchy this property is enforced by the condition. Because of the type compatibility an :X object can be assigned to its exact type X or any ancestors of X, but to only one of them. Thus it is chosen such that the association of the link is attached to it to preserve the adjacency. Secondly, the inverse direction is shown. Let it be assumed that the mapping can be created, but links from more than one inheritance level are instantiated and assigned to an :X object. Hence this should be assigned to X or one of its ancestor. We cannot choose either of them because if we select a class based on an association of a specific level, the other will violate the adjacency in *Meta*, which contradicts our assumption.

The direct consequence of the propositions above is, that, in general, the UML object diagram cannot be mapped to UML class diagram by graph homomorphism. Hence we must construct a metamodel which considers the practical aspects and has the same expressivity as the original, but the instantiation relationship is a homomorphic TPM. First the equivalence of metamodels is defined.

Definition 4 (equivalence of metamodels).
Let *Meta1* and *Meta2* be two LDGs, with labels conforming to the UML class diagram. *Meta1* and *Meta2* are **equivalent** if and only if any LDG provided with labels conforming to UML object

diagram which is instance of *Meta1* is instance of *Meta2* as well, and vice versa. If only one direction is satisfied, namely, all instances of *Meta1* are the instance of *Meta2*, then *Meta2* is **compatible** with *Meta1*. Hence equivalence can be defined as bidirectional compatibility. If at least one instance of *Meta1* are contained by at least one instance of *Meta2*, then *Meta2* is **partially compatible** with *Meta1*.

Contrary to the compatibility property, partial compatibility is symmetric.

Then an algorithm is given which creates an equivalent metamodel where the instantiation relationship is a homomorphic TPM. An example for this algorithm can be found in [3].

Algorithm outline 1 (creating homomorphic meta)

1. Walk through the inheritance hierarchy and copy all inherited data (attributes associations) from the ancestors to the derived class. We take care of the multiplicity values with variable multiplicity.
2. Remove inheritance.
3. Remove all abstract classes.

The algorithm uses a wide-spread technique to eliminate inheritance: it copies all the data and relations to the derived classes, but we must be careful with the association multiplicities, because its value must be distributed among the new nodes: their sum must be the old value. To handle this, we use variable names to denote multiplicity and we refer to them from another association. A detailed discussion of this algorithm can be found in [8].

Proposition 3. The homomorphic metamodel and its generator metamodel is equivalent.

Proof: If an instance LDG instantiates the generator metamodel, it instantiates its homomorphic metamodel as well. The algorithm removes abstract classes only, which are not concerned with instantiation at all, so all the object types (classes) have been preserved with their attribute and other properties. If there was an association connected to a class it still remained after the transformation. If a class had an inherited association after the transformation it has the same association directly. Hence all the links that instantiate the generator metamodel instantiate the homomorphic metamodel as well. If an instance LDG instantiates the homomorphic metamodel, it instantiates its generator metamodel as well. For objects and classes the statement can be shown similarly to the inverse direction. The only changes in the associations are that some of the direct associations have become inherited. This change has no influence on instantiation.

The multiplicity values produced by the algorithm were designed such that it conforms to equivalence.

Proposition 4. Let *Meta1* and *Meta2* be an LDG, with labels conforming to the UML class diagram, and they are homomorphic metamodels. If *Meta1* is compatible with *Meta2*, *Meta1* is a (not always connected) subgraph of *Meta2* not regarding the multiplicity labels. The following formula is always true:

$$M_{\mathrm{Meta1}} \subseteq M_{\mathrm{Meta2}}, \tag{4}$$

where M_{Meta1} and M_{Meta2} are the sets of the allowed multiplicity for *Meta1* and *Meta2* side respectively at the same topological position.

In case of partial compatibility, where the zero multiplicity values are not allowed, it is enough to enforce the following conditions for each corresponding multiplicity pairs:

$$\mathrm{Sup}M_{\mathrm{Meta1}} \leq \mathrm{Sup}M_{\mathrm{Meta2}}, \tag{5}$$

$$M_{\mathrm{Meta1}} \cap M_{\mathrm{Meta2}} \neq \varnothing \tag{6}$$

where Sup is the supremum of the set which contains the allowed multiplicity values.

Proof:

If every instance of *Meta1* also instantiates *Meta2*, it implies the following: if we construct an instance of *Meta1* which instantiates all classes in *Meta1*, then it means that *Meta2* must contain all the classes from *Meta1*.

For partial compatibility the conditions mean that (i) if there is an association in *Meta1* there must be one with at least with the same supremum (that implies the containment) (ii) we ensure that there is an instance of *Meta1* that instantiates every element of *Meta1*, and there is an instance of *Meta2* which contains it. The latter condition is not necessary but rather useful. It is rather obvious to check the situations detailed above for the allowed multiplicity sets.

3.2 Validating the transformation steps

Hence we have a homomorphic metamodel we can establish the category theory framework and the instantiation. To clarify the notations and the basic constructs we define the well-known category of LDGs called **Graph**.

Definition 5. (Category of LDGs, **Graph**): (i) Objects: LDGs, (ii) Arrows: LDG homomorphisms,

(iii) Composite operator: composite of graph homomorphisms.

For these composites we show that they are homomorphisms as well. Suppose $\varphi : G \to H$ and $\psi : H \to K$ mappings are graph homomorphisms, $u \in G_E$, $m, n \in G_V$, and $s^G(u) = m$, $t^G(u) = n$. Based on the definition of the homomorphism:

$$\varphi_E(u) \in H_E$$
$$\varphi_V(m), \varphi_V(n) \in H_V$$
$$s^H(\varphi_E(u)) = \varphi_V(m)$$
$$t^H(\varphi_E(u)) = \varphi_V(m) \tag{7}$$

Applying ψ homomorphism using the definition again:

$$\psi_E(\varphi_E(u)) \in K_E, \psi_V(\varphi_V(m)), \psi_V(\varphi_V(n)) \in K_V,$$
$$s^K(\psi_E(\varphi_E(u))) = \psi_V(\varphi_V(m)),$$
$$t^K(\psi_E(\varphi_E(u))) = \psi(\varphi_V(m)) \tag{8}$$

which means that $\psi \circ \varphi$ is also homomorphism.

Graph homomorphisms can be thought as creating composite functions, which are (i) associative:

$$h \circ (g \circ f) = (h \circ g) \circ f, \tag{9}$$

(ii) and the identity arrows are the identity functions for both vertices and edges. Then it is obvious for an arbitrary *f*: A→B that

$$id_B \circ f = f \text{, and } f \circ id_A = f. \tag{10}$$

Based on the previous definition one can easily validate that both the meta LDGs and instance LDGs are objects in Graph, and partial compatibilities (lkm, lg, gdm, dhm, rhm, krm) inclusions (l, r, l*, r*, m, d, m*) and instantiations (lm, km, rm, gm, dm, hm) are arrows.

Then the transformation step of the DPO approach is extended with the metamodels as it is depicted in Fig. 2.

$$\begin{tikzcd}
LM \arrow[ddd, bend right=60, "lgm"'] & KM \arrow[l, "lkm"'] \arrow[r, "krm"] & RM \arrow[ddd, bend left=60, "rhm"] \\
L \arrow[u, "lm"] \arrow[d, "m"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[u, "km"] \arrow[d, "d"] & R \arrow[u, "rm"] \arrow[d, "m^*"] \\
G \arrow[d, "gm"] & D \arrow[l, "l^*"'] \arrow[r, "r^*"] \arrow[d, "dm"] & H \arrow[d, "hm"] \\
GM & DM \arrow[l, "gdm"'] \arrow[r, "dhm"] & HM
\end{tikzcd}$$

Fig. 2. The diagram of the transformations step with metamodels.

As it is well-known from the DPO approach, the inner double square forms a double pushout. For each participants of this double pushout a metamodel is provided via a TPM homomorphic mapping.

We assume partial compatibility morphisms between the meta-LDGs based on Proposition 4 and for semantic reasons: it is not optimal to have an LHS metamodel which can be instantiated such that it never matches. In case of RHS without ensuring this assumption the rewriting is an error prone operation, because the result may not conform to the destination metamodel. So we summarize this in a definition.

Definition 6. If an LHS and an RHS satisfies the condition of partial compatibility elaborated in Proposition 4 with respect to GM and HM, and they contain nonzero multiplicity values, they are called **proper metamodels**.

Proposition 5. If the morphisms are inclusions in the outer double square, and the inner square is a double pushout, the outer double square is a double pushout.

Proof:

As it is known the category **Graph** has all pushouts, which means that for every pair of morphism there is a pushout in the category. It can be proven [5] that if given two morphism b: A→B, c: A→C, then the pushout ⟨D, g: B→D, f: C→D⟩ can be constructed as $D_V = (B_v + C_V)\approx, D_E = (B_E + C_E)\approx$ i.e. the quotient set of the disjoint union modulo $\approx = \{f(a), g(a) \mid a \in A\}$ smallest equivalence relation that maps each element to its equivalence class. For **Graph** the pushout construction is accomplished componentwise, i.e. for the edge set and the vertex set, respectively. If the morphisms are inclusions, a diagram depicted in Fig. 3 can be established which is a pushout [9]. Based on Proposition 4, it can be seen, that if the match exists, then there is a partial compatibility between corresponding metamodels. The properness conditions (c.f. Definition 6) are assumed to be fulfilled for semantic reasons, so there are inclusion morphisms between the metamodels as it is shown in Fig 2. Hence the outer double square is a pushout.

Assume a rewriting rule which does not violate the dangling edge condition, but we have classes **A** and **B** connected by the association **a** on the metamodel level, which does violate it via removing **B** without removing **a**. Then the rule instantiates **A**,

B, and **a**, and will remove <u>**:B**</u> without removing **l** (which is the instance of **a**), which contradicts to our original assumptions.

$$\begin{tikzcd}
S1 \cap S2 \arrow[r, "\subseteq"] \arrow[d, "\subseteq"'] & S1 \arrow[d, "\subseteq"] \\
S2 \arrow[r, "\subseteq"] & S1 \cup S2
\end{tikzcd}$$

Fig. 3. A Doolittle diagram with inclusions in Set (a category with sets as objects and functions as morphisms)

Based on the proof of Proposition 5 the next proposition is rather straightforward.

Proposition 6. If the meta square is a double pushout and proper metamodels are assumed then the rule cannot violate the dangling edge condition.

Conforming to the DPO approach results in "free" propositions like below:

Proposition 7. If two LHS metamodels do not contain elements in common, then the order of the rules are invariant and can be executed parallel.

Proof: Direct consequence of the DPO Parallelism theorem, because in this case the instance level will not overlap either.

Proposition 7 is especially useful if the transformation environment is expected to offer parallel execution of the rules automatically without collision and side-effects. Proposition 5-6 facilitate to validate the rewriting rule on the metamodel level without knowing what the actual redex is.

Algorithm outline 2. (detecting gluing time RH error)

1. If the partial compatibility does not hold for the LM and GM (it can be checked using Proposition 4), we can signal an error message or automatically adjust it emitting a warning message. In case of RM and HM if the compatibility holds for, no additional error check is necessary.
2. If only the partial compatibility is satisfied then there are two cases: (i)When RM allows a larger instance (with greater multiplicity) then it is considered an error which can be detected and reported. (ii)When RM allows a smaller instance it can be a part of the instance specified by HM. In this case the rewriting might cause an error, but seamless situations are also possible. This type of error can only be detected at gluing time checking the created changes again HM.
3. When HM is not partially compatible with RM, an error message should be reported or an automatic adjustment can be performed (based on the conditions in Proposition 4) with warning.

4 Conclusion

A formal background has been proposed for checking topological validity of model transformation rules. The applied mathematical formalism is category theory and the double pushout approach, as it was used successfully in the field of graph rewriting.

Since it is required by the category theory, the instantiation relationship needs to be transformed to conform to the rules of the homomorphic mapping. To ensure the interchangeability between the transformed and the original metamodel, the definitions of equivalence and compatibility have been introduced. It has been proved, that the original and the transformed metamodels are equivalent on the instance level. Then criteria have been proven to check the topological validity of the rewriting rules and with the provided algorithm several erroneous transformation rule can be detected and prompted back to the user. A proposition condition for parallel execution has been also proven for the metamodel-based rewriting rules.

Future work includes an extension to the SPO approach and performing constraint transformations for homomorphized metamodels.

References:

[1] Object Management Group, Unified Modeling Language Specification, v1.5, www.uml.org

[2] OMG Model Driven Architecture homepage, www.omg.org/mda/

[3] The VMTS Homepage. http://avalon.aut.bme.hu/~tihamer/research/vmts/

[4] T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf, A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, *Electronic Notes in Theoretical Computer Science*, International Workshop on Graph-Based Tools (GraBaTs) Rome, 2004

[5]G. Rozenberg (ed.), *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, Vol.1. World Scientific, Singapore, 1997.

[6] XML Metadata Interchange (XMI), v2.0, www.omg.org

[7]D. Blostein, H. Fahmy, A. Grbavec, Practical Use of Graph Rewriting, *Technical Report No. 95-373*, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, January, 1995.

[8]T. Levendovszky, H. Charaf, Parametrized Multiplicity Constraints in UML Like Metamodels, *microCAD 2004*, Miskolc, pp. 287-291

[9] M. Barr, C. Wells: Category Theory Lecture Notes for ESSLLI, www.folli.uva.nl/CD/1999/library/pdf/barrwells.pdf, 1999.