A UML Class Diagram-Based Pattern Language for Model Transformation Systems

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 methods are vital in several applications such as OMG's Model-Driven Architecture and Visual Model Processors. This paper contributes a metamodel-based rewriting rule representation similar to the UML class diagram and the supporting algorithms to determine valid instances of the patterns. The proposed algorithms are illustrated by specific examples throughout the paper. The results turn out to be useful not only for UML class diagram-based rewriting rule formulation patterns, but it provides a method for checking valid instantiation of UML class diagrams in modeling environments.

Key-Words: instantiation, metamodel-based model transformation, graph rewriting, UML class diagram, association multiplicity

1 Introduction

Model transformation systems are a highly researched field of applied computer science. For instance the Object Management Group's (OMG) Model-Driven Architecture (MDA) standard [1] strongly builds on model compilers, which automatically create a platform specific model (PSM) from the platform independent models (PIM) specified by the development team. Model transformation systems can serve as a basis for model compilers. Another applications are Visual Model Processors (VMPs)[2], which can provide a more maintainable solution for processing system models than their traversing counterparts.

To create a representation for the transformation steps it is reasonable to use a well-known visual formalism like the UML class diagram [3][4].

This paper is organized as follows: Section 2 enumerates the achievements contributed to the results included this paper, Section 3 supplies detailed information about a UML class diagram-based rewriting rule formalism. Section 4 provides the theoretical and algorithmic background to determine the allowed instantiations, and in Section 5 conclusions are drawn.

2 Backgrounds and Related Work

According to the Unified Modeling Language (UML) standard [3], the binary association and the multiplicity sets assigned to the association ends are interpreted as follows:

The integer set *s1* (Fig. 1) contains the valid number of links (association instances on the object diagram) originated from *A* type objects to a *B* type object.

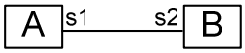


Fig. 1. Association with multiplicity sets

The integer set *s2* is conceived symmetrically. To rephrase this in terms of graph theory, one can say that the allowed valencies (degrees) of an object vertex of type *A* concerning the links connected to object vertices of type *B* are denoted on the *B* side of the association that resides between classes *A* and *B*.

In the previous statement we assumed that only one association exist between *A* and *B*. If there are more than one associations between *A* and *B* (this structure requires association classes or role names to make distinction between the associations), the same rule holds for each association and their instantiating links, respectively.

According to the UML standard there is another property of the association end which affects the instantiation of an association, namely, the Boolean *navigable* property. If an association end is navigable (e.g. the *B* side of the diagram in Fig. 1), the instances of the class on the other end (in our example the instances of *A*) can access the instances of the class where the association end is navigable (instances of class *B*).

Since the UML class diagram is mainly used to generate programming language code, navigability means that the given side of the association is simply ignored by the generator, that is, no code will

be generated into the class residing on the opposite side for the association end (which is usually a variable named after role name, and with the type of an array or with a single type, depending on the multiplicity values). The UML class diagram can be regarded as the metamodel of the object diagram, thus we use the two expressions interchangeably in the context of this paper.

Graph rewriting is a sequence of rule applications. A rule consists of two graphs: left hand side (LHS) and right hand side (RHS) graphs. The goal is to find an occurrence of LHS (a subgraph isomorphic to LHS) in the input graph, which is replaced with the RHS of the rule. As far as the rule firing is concerned the theory of double pushout approach (DPO) in algebraic graph rewriting [5] is applied: firstly only the intersection of LHS and RHS is kept in the input graph, other parts are removed, and then the rest of the RHS (RHS-(RHS∩LHS)) is glued to the input graph. In our metamodel-based case introduced in [6] the input graph is the model to be transformed, and the rules are metamodels: instead of finding an occurrence (isomorphic subgraph) a subgraph is found which instantiates LHS.

As far as pattern languages for graph rewriting are concerned PROGRES [7] provides a constructs similar to multiplicities (cardinality assertions).

The GReAT framework [8] is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts. It uses a proprietary notation and interpretation instead of instantiation between the rules expressed with metaelements and the match.

Previous work has introduced a software package called Visual Modeling and Transformation System (VMTS) [2][9]. In this paper a technique is worked out, with the help of which we can apply UML class diagram elements (classes, associations, multiplicities) to create a pattern language for the LHS and the RHS part of the graph rewriting rules used in model transformation engine of VMTS.

3 Motivation

In case of a model transformation system the description language of the transformation steps is crucial from the users' point of view. If the achievements elaborated in the previous sections can be turned into a representation which the developers are familiar with, it facilitates a powerful, easy-to-learn transformation tool. UML class diagram is a popular and standard way of modeling the static structure of software systems under design. Our goal is to provide a rule specification feature in VMTS which is really similar in both concepts and notations to the mechanisms applied in UML class diagram.

The main differences are as follows: (i) In rewriting rules we have to force a modified version of the identification condition [5]. The original identification condition states that different vertices in the production rule must not match the same vertex in the host graph. In case of metamodels, however, one metamodel element can have more objects instantiating it. The identification condition can be generalized such that the sets containing the instances of different pattern elements shall be disjoint. It implies that a metamodel element can appear more than one times as opposed to the UML class diagram. An intuitive example can be seen in Fig. 2.

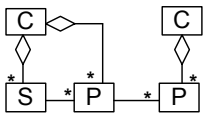


Fig. 2. Example for the modified identity condition

In Fig. 2 a pattern from the hierarchical data flow diagram flattening problem is depicted, which is elaborated in [6]. A Compound element (*C*) can contain Simple elements (*S*) to facilitate the hierarchical structure. Both *C* and *S* elements are connected to Ports (*P*). In order to flatten the diagram one should find the ports connected to a *P* element and the *C* element containing the *P* element, and a port of another *C* element to short circuit them. After this step *C* elements can be removed. The rule needs two different *C* elements, with two totally different topological orientations: one contains an S element and a *P* element connected to an *S* element, the other one contains a *P*, and not necessarily an *S* component. Moreover our experience has shown that this is the intuitive interpretation of this construction as well. In addition it is easy to see that this interpretation is exactly what the modified identification condition ensures. Processing LHS containing elements benefiting from this modified identification condition can be processed easily: the classes appearing more then once are treated as different classes during the matching, and the disjoint property is always checked as the matching process proceeds.

(ii) The other distinction is that the *navigable* property is not taken into account at all during the matching process, because the reachability has no semantic meaning in the context of matching. Consequently, each multiplicity value has to be

valid, because it is always taken into consideration by the rewriting engine, and the instantiation are evaluated as if the *navigable* properties were set to true.
To implement and process these class diagram-based rules we have to determine what the possible instantiations of an UML class diagram are.
This is not always simple, since there are valid UML patterns for which no valid instantiation exists (Fig. 3)

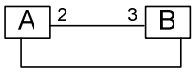


Fig 3. UML class diagram with no valid instances

Although this situation rarely occurs in case of UML modeling, where the multiplicities are mainly one of the values 1, 0..1,1..*, 0..*, considering a pattern language the other cases also have to be dealt with. The rest of this paper is devoted to this issue and the related algorithms.

4 Contributions

4.1 Properties of the instantiation

As it was illustrated in the previous section not every class diagram can be instantiated. In this section we examine the valid instantiations related to the multiplicity values with respect to the case where multiplicity values consist of a nonzero integer element.

Fig. 4. A general bidirectional association with nonzero integer multiplicities

Proposition 1. The class diagram depicted in Fig. 4 can be instantiated by *na'* objects of type A and *nb'* objects of type B, where *n* is an arbitrary positive integer and

$$\begin{aligned} a' &= \frac{a}{GCD(a,b)} \\ b' &= \frac{b}{GCD(a,b)} \end{aligned} \tag{1}$$

where GCD denotes the greatest common divisor.

Proof: It follows from the UML class diagram semantics that each object of type *A* must have exactly *b* number of *B* type objects. Assume we have *y* number of *A* type object. Then $by = z$ edges exist on the object level. Similarly, if *x* number of *B* type objects exist, $ax = z$ edges are set up (LCM denotes the least common multiple).

$$\begin{aligned} ax &= by = nLCM(a,b) \\ ax &= nLCM(a,b) \\ x &= \frac{nLCM(a,b)}{a} \end{aligned} \tag{2}$$

Because of the symmetry of the model (Fig. 4):

$$y = \frac{nLCM(a,b)}{b} \tag{3}$$

Using a well-known formula we can express the solution dependently on the other variable:

$$\begin{aligned} GCD(a,b)LCM(a,b) &= ab \\ LCM(a,b) &= \frac{ab}{GCD(a,b)} \\ x &= \frac{nb}{GCD(a,b)} \\ y &= \frac{na}{GCD(a,b)} \end{aligned} \tag{4}$$

Based on Proposition 1 an example can be created (Fig. 5):

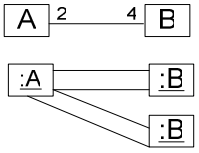


Fig. 5. Parallel links

Although this is a construction that conforms to the UML standard [3], multiple edges, however, are not used in practical applications. If multiple edges are not allowed, as it is the case regarding the LHS rules, another proposition can be used.

Proposition 2. If no multiple edges allowed for associations in the object diagram, the class diagram in Fig. 4 can be instantiated by *na* objects of type A and *nb* objects of type B, where *n* is an arbitrary positive integer.

Proof. We use the *x,y* variables from the previous proof. As a first step an *A* node is taken and tied to *b* number of **different** *B* type node to avoid multiple

edges. It means that this step requires *b* number of *B* type node to be available. We do not restrict the connection made during a particular step, but it always holds that in worst case the (*ak+1)*th step requires the availability of *(k+1)b B* type object (k=1,2,…,N), because the worst case does not use new *B* type object until it is necessary i.e. when all the already tied *B* type objects are exhausted. Consequently:

$$a(k-1)+1 \le s \le ak \Rightarrow kb \le x,$$

where *s* denotes the ordinal number of the current step. Each step uses and exhaust the connection facilities of exactly one *A* type object, hence $s = y$.

$$a(k-1)+1 \le y \le ak \Rightarrow kb \le x \qquad (5)$$

Recall that because of the edge numbers

$$ax = by \qquad (6)$$

$$a(k-1)+1 \le y \le ak \Rightarrow kb \le \frac{by}{a}$$

$$a(k-1)+1 \le y \le ak \Rightarrow ak \le y,$$

Then the only solution:

$$ak = y,$$

And because of the symmetry:

$$bk = x.$$

We do not consider *k=0* as a solution, thus the proposition directly follows from the formulae above. Because this is a special case of the multiple edge version, if we do not simplify with the greatest common divisor, we obtain (1) given in Proposition 1.

4.2 Instantiating the model structure

It is worth examining that which *n* values should be checked for a specific value *a.* It would be advantageous if we could formulate an upper bound for *n* to be examined, and we could decide whether it can be a part of the match. Unfortunately, in general such an upper bound cannot be given in all cases, further parts of the object graph has to be examined.

Assume that *ma* number of *A* type object and *mb* number of *B* type are available. Examining *na* and *nb* objects it cannot be decided whether these objects form a valid instantiation if $n < m$.

We create an instance construct with a parameter *n*, as it is depicted in Fig 6. The structure contains *m* number of blocks, and each block contains *a* number of *A* type objects and *b* number of *B* type objects. As it follows from Proposition 2, a block can form a satisfying instantiation of the given class diagram.

However, if an edge is removed from blocks, which are instances in themselves, each block will contain an *A* and a *B* type object able to accept an edge.

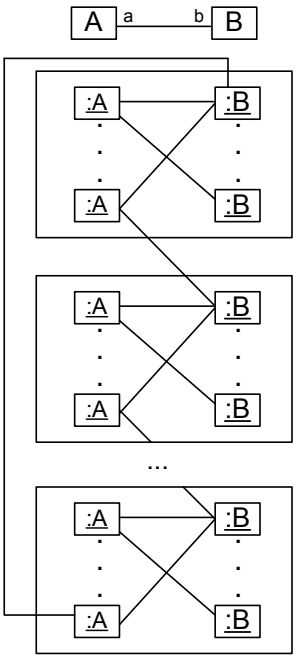


Fig. 6. An infinite construction for a class diagram which cannot be analyzed by blocks

This edge is drawn from a free *A* type object of the *nth* block to the free B type object of the *n+1th* block, and in case of the *mth* block the first block is considered. Consequently, the threshold for examined *n* values can be at most

$$\min\left[\frac{\#A}{a}, \frac{\#B}{b}\right]. \qquad (7)$$

For computing the allowed numbers of the participating objects in whole instantiation model graph we consider a specific representation derived from the metamodel, which is referred to as Incidence Matrix with Multiplicity (IMM) in the sequel.

Algorithm 1. CREATEIMM( )

1 Traverse the metamodel graph

2 Take each edge e_j.

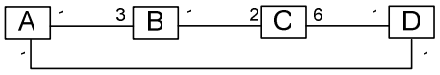
3 If the v_k and v_l are the vertices incident upon e_j, then set IMM_{kj} to the v_k side multiplicity and IMM_{lj} to the v_l side multiplicity of e_j.

IMM can be considered as a short representation of the equations established for each node based on

Proposition 2. In our example it can be written as follows:

$$\begin{aligned} A &: x_0 = x_3 \\ B &: 3x_0 = x_1 \\ C &: 2x_1 = 6x_2 \\ D &: x_2 = x_3 \end{aligned} \tag{8}$$

Obviously, in the formula above all x_i variables are nonzero integers.



$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Fig. 7. An example for creating IMM.

To solve this sort of equations an elimination algorithm is proposed. The source code of the algorithm can be downloaded from [9].

Algorithm 2. ELIMINATION(IMM IMM)

```
 1 for each j column index
 2   r0=index of row with first nonzero element
 3   r1=index of row with second nonzero element
 4   if there is no r0 and r1 then break
 5   lcm=LCM(imm[r0,j],imm[r1,j])
 6   MultiplyRow(r0, lcm/imm[r0,j])
 7   MultiplyRow(r1, lcm/imm[r1,j])
 8   for each j2 column index
 9     if imm[r0,j2]= = imm[r1,j2] then
10       imm[r1,j2]=0
11     else if imm[r0,j2]!=0 and imm[r1,j2]!=0 then
12       Inconsistent parallel paths. No instantiation.
13     else if imm[r1,j2]!=0
14       imm[r0,j2]=imm[r1,j2];
15       imm[r1,j2]=0;
16     end if
17   end for
18 end for
19 rowlcm=LCM of the 0th row of imm
20 for each j column index
21   imm[0,j]=rowlcm/ imm[0,j]
22 end for
```

After the elimination the 0th row of the IMM contains a factor f_i for each edge e_i. Each m_1, m_2 multiplicity on an edge e_i must be multiplied by factor f_i. The number of the m_1 side node can be $m_1 f_i k$, where k is an arbitrary nonzero integer and the cardinality of the other node can be computed similarly. An example for the elimination algorithm:

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 0 & 0 & 3 \\ 3 & 1 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 3 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 6 & 2 & 0 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 6 & 2 & 6 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \Rightarrow$$

$$\Rightarrow \begin{bmatrix} 6 & 2 & 6 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 6 & 2 & 6 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \text{(LCM=6)},$$

$$\begin{bmatrix} 1 & 3 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{matrix} x_0 = 1 \\ x_1 = 3 \\ x_2 = 1 \\ x_3 = 1 \end{matrix} \Rightarrow \begin{matrix} A : 1n \\ B : 3n \\ C : 6n \\ D : 1n \end{matrix}$$

As a second example we consider our introductory counterexample depicted in Fig. 3.

$$\begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 6 & 3 \\ 6 & 2 \end{bmatrix} \Rightarrow \text{Inconsistent equation.}$$

Taking another unsuccessful example it can be seen that a tool can recommend a correct, consistent multiplicity in special cases:

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 7 \\ 2 & 5 & 1 & 0 & 0 \\ 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \Rightarrow \ldots \Rightarrow$$

$$\begin{bmatrix} 24 & 60 & 12 & 15 & 56 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 15 & 3 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \text{Inconsistent equation.}$$

Although this equation cannot be solved, one can offer a solution for the multiplicities in the last column:

$$8m_1 = 3m_2 = lLCM(8,3) = 24, l = 1,2,...$$

$$m_1 = 3$$

$$m_2 = 8$$

Substituting the result into the original matrix:

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 3 \\ 2 & 5 & 1 & 0 & 0 \\ 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 & 8 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \Rightarrow \ldots \Rightarrow$$

$$\begin{bmatrix} 24 & 60 & 12 & 15 & 24 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

5 Conclusion

This paper has proposed a metamodel-based formalism for rewriting rules used as model transformation steps in VMTS. An algorithm for determining the valid instantiations for nonzero integer multiplicity values has also been contributed. The provided algorithms can be easily extended to set multiplicities as well. (i) The zero multiplicity case should be treated differently, because it implies no links connected to the object of the given type. (ii) For set multiplicities (e. g. intervals), all pairs from the Cartesian product of the two multiplicity sets have to be examined for set values with the IMM algorithm.

However this method has been developed for metamodel-based rewriting rules, it can be used by modeling environments to check whether a valid instantiation exists for a specified class diagram, when the *navigable* properties are turned on.

Future work includes the examination of the case when *navigable* properties do not hold, and reducing the search space with ruling out the invalid instantiations in matching part of the rule firing algorithm.

References:

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

[2] 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

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

[4] T. Pender, T. Pender, *UML Bible*, Wiley, 2003

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

[6]T. Levendovszky, G. Karsai, M. Maroti, A. Ledeczi, H. Charaf, Model reuse with metamodel-based transformations, *Lecture Notes in Computer Science* - ICSR7, Austin, TX, April 18, 2002, pp. 166-178

[7] A. Zündorf, Graph Pattern Matching in PROGRES, *Graph Grammars and Their Applications in Computer Science,* LNCS 1073, J. Cuny et al. (eds), Springer-Verlag, 1996, pp. 454-468.

[8] G. Karsai, A. Agrawal, F. Shi, On the Use of Graph Transformation in the Formal Specification of Model Interpreters, *Journal of Universal Computer Science*, Special issue on Formal Specification of CBS, 2003. pp. 1296-1321

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