

Formalizing Model Transformation Rules for UML/MOF 2

Carsten Amelunxen and Andy Schürr

Darmstadt University of Technology, Real-Time Systems Lab,
Merckstrasse 25, 64283 Darmstadt, Germany
[amelunxen|schuerr]@es.tu-darmstadt.de
<http://www.es.tu-darmstadt.de>

Abstract. Model-Driven software development, as today's state-of-the-art approach to the design of software, can be applied in various domains and thus demands a variety of domain-specific modelling languages. The specification of a domain-specific modelling language's syntax and semantics can in turn be specified based on models which represent the approach of metamodelling as a special form of language engineering. The latest version of UML 2 and its subset MOF 2 provide sufficient support for metamodelling a modelling language's abstract syntax. Furthermore, based on the description of the abstract syntax, a language's static semantics can simply be specified by OCL as UML/MOF's natural constraint language, whereas the description of a MOF compliant language's dynamic semantics is still not covered. In this article we try to close this gap by integrating MOF/OCL with graph transformations for the specification of dynamic aspects of modelling languages and tools. The formalization of such an integration is non-trivial due to the fact that UML/MOF 2 offer a rather unusual and sophisticated association concept (graph model). Although there are many approaches which formalize graph transformations in general and first approaches that offer a precise specification of the semantics of the association concepts of UML/MOF 2, there is still a lack in bringing both together. In this paper we close this gap by formalizing graph transformations that work on a UML/MOF 2 compatible graph model.

1 Introduction

From the very first beginning of computer programming, software languages played an important role as a trade-off between human understandable abstraction of executable structures and directly executable machine code. Over the years software languages evolved to cover higher levels of abstraction until, nowadays, model-driven development became one of the key features of software development. The high-level graphical description of concepts and their relations in the form of Unified Modeling Language (UML) [1] class diagrams is widely known and accepted as a high level description of software structures. But despite UML's claim of being a unified modelling language, there are domains in which special requirements and domain-specific peculiarities demand

the application of a special (modelling) language which regards domain-specific issues. The modelling of (feedback) control algorithms for simulation and code generation in the area of embedded systems demands a different style of modelling language than the modelling of work flows or the modelling of enterprise software systems, for instance. Thus, software language engineering in the form of specifying graphical domain-specific modelling languages is a very important task.

For a tool-based application of a domain-specific modelling language, the definition of the language's (abstract and concrete) syntax, static and dynamic semantics are essential. Since such a definition is quite a complex and expensive task, there is a need of a formal and tool-based framework which simplifies the definition of domain specific modelling languages. The metamodelling framework MOFLON [2], [3] aims to provide such support by offering tool support for a metamodel-based language engineering. MOFLON provides a specification language which is composed of OMG's metamodelling standard Meta Object Facility (MOF) 2.0 [4] together with the Object Constraint Language (OCL) [5] and graph transformations in the form of Story-Driven Modelling (SDM) [6], [7] charts. This composition allows to specify a domain-specific language's abstract syntax¹ together with its static semantics, the first with MOF class diagrams and the latter with OCL. Since the object structures described by MOF are directed and labelled graphs the integration of graph transformation techniques provides additional support for the description of static semantics with graph queries and the possibility to implement any kind of behaviour or manipulation of models (execution, refactoring, ...) based on the language description. From such a specification, MOFLON is able to generate a repository enriched by executable features which can be used as a base for any kind of tool support.

Possible scenarios for the application of such descriptions of domain-specific languages can be found, for instance, in the area of tool integration or in the area of reengineering, just to mention a few. Beside those quite obvious application scenarios, MOFLON can also be used to specify its own (abstract) syntax and semantics. Since, the static semantics of MOFLON's specification language are specified by the adopted standards, there is just a lack of a formal specification of its dynamic semantics. This formal specification is presented in this article. Due to the fact that MOFLON is based on MOF 2.0 which at least represents a subset of UML's class diagrams, the formalization of MOFLON's dynamic semantics can be applied to transformations based on the common core of MOF 2.0 and UML 2. The formalization is based on the principles of graph transformation. Thereby, we concentrate on the application of single transformation rules. Based on the formalization, further aspects like, for instance, the controlling of the application of several rules can be formalized as described in [8], but are out-of-scope of this article.

¹ The definition of concrete syntax is out of MOFLON's scope, since MOFLON is basically used for the adaptation of existing modelling tools and their languages with existing concrete syntax.

First of all, section 2 presents a closer look at MOFLON and the surrounding conditions of the formalization, followed by an overview of related work in section 3. A running example is introduced in section 4. We proceed with basic definitions in section 5 and an example of a transformation in section 6. Afterwards, we formalize transformations on a very simple subset of MOF in section 7 which is successively ² extended throughout section 8. Finally, we end up with a conclusion in section 9.

2 Language Engineering and MOFLON

The metamodeling framework MOFLON provides tool support for metamodel-based language engineering. Its specification language, the MOFLON Specification Language (MOSL) [9], is composed of the existing languages of MOF 2.0, OCL 2.0 and SDM. Hence, the quality of MOSL's formal specification depends on the one hand on the formal specification of its part languages and on the other hand on a smooth and tight integration of its parts. In the centre of MOSL, there is MOF 2.0 for the specification of a domain-specific language's abstract syntax. Since in most specifications the abstract syntax represents the main part of the complete specification, the formal specification of MOF 2.0 is of particular importance. With a set of corrections and adjustments, the MOF 2.0 specification can basically be considered to be sufficient formal for a tool based application.

Beside MOF, MOSL adapts OCL, as MOF's natural extension from the OMG world, for the specification of a language's static semantics. Both standards are innately tightly coupled, therefore, their integration is not of a particular interest. In fact, the integration of MOF 2.0 and the SDM graph transformations as well as the integration of the textual queries of OCL into the graph transformations of SDM attract the most interest. Since the integration of MOF 2.0 and SDM is kind of a precondition, the integration of SDM and OCL will be examined in future work. In the following we investigate the integration of core concepts of graph transformations with MOF 2.0. We neglect syntax and static semantics and concentrate on the dynamic semantics.

On a first sight, the integration of MOF 2.0 and graph transformations might not give the impression of being very promising since the application of class diagrams as graph schema language is well known and has widely been studied. But, compared to its predecessors, MOF 2.0 provides a more sophisticated association concept which has direct influence on the application of graph transformation. Since the new 2.x version of MOF and UML share a common core [10], the mentioned issues are also relevant for UML as well. As the new features of MOF 2.0 are neither by the OMG's own transformation approach called Query/View/Transformation (QVT) [11] nor any other transformation approach taken into account, the presented formalization provides a transferable solution for most UML/MOF based transformation languages. In the following, we

² The final set of formulae can be found in a compact summary at http://www.es.tu-darmstadt.de/download/publications/amelunxen/MOFLON_Dynamic_Semantics.pdf

present a set theoretic formalization of the application of graph transformations to the sophisticated associations of UML/MOF. We chose a set theoretic formalization rather than MOSL itself, since for the purpose of a written presentation a MOSL specification would be too close to its implementation.

The presented formalization is successively extended throughout this article. We start with a very simple notion of a metamodel and introduce relevant concepts by and by. Furthermore, we assume all metamodels to be compliant to the static semantics of MOF 2.0 and, therefore, omit any kind of metamodel constraints. Model constraints are introduced when necessary. But we do not claim to provide a complete set of model constraints. There are undoubtedly more constraints than those essential constraints presented here. We also concentrate basically on MOF 2.0 associations as the most relevant constructs and omit all further remaining MOF 2.0 elements.

3 Related Work

The success story of UML is based on its efforts to provide an easy and quite intuitive language for multiple domains (which is influenced by the experiences gained by several modelling approaches) rather than on its precise definition. A language should always provide formal semantics. Thus, from the early beginnings of UML, this lack of formalism has been criticized [12] and there were many attempts to fill this gap by providing mappings onto formal languages like Z or Object-Z ([13], [14]). There are also actual attempts to provide formal semantics for the new version of UML [15], since even for the new version of UML, precise formal semantics have not been taken into consideration.

Before, beside as well as inside UML, the modelling of relations between concepts in an object oriented modelling approach has always gained special interests ([16], [17] just to mention a few). In the meantime there are also approaches dealing with the new association concepts [18] and some first attempts to formalize the new association features [19].

Considering the area of transformations and especially model transformations, there are a lot of approaches available [20]. Many of those approaches lack a formal description as well. Whereas, there is a long tradition of formalization in the area of graph transformations. In general, there are the three different techniques of category theoretical, logical and set theoretical approaches for the formalization of graph transformations [21], [22]. We chose the set theoretical approach, since we regard this approach to be most intuitive for people outside the graph transformation community.

There are also projects which combine UML and graph transformation by applying UML class diagrams as graph schema language, for instance GReAT [23], AGG [24], PROGRES [25] and Fujaba [6]. All tools use the basic concepts of UML class diagrams but not the sophisticated association concepts of MOF/UML 2. The presented approach is quite close to the Fujaba approach and can be interpreted as an addition concerning the formal base and its application.

4 Running Example

In the following, we introduce a simplified but MOF 2 compliant metamodel of C and C++ projects. Such a metamodel might be used for reengineering purposes based on project structures and header file includes. It is especially designed to introduce the association features which are addressed by the formalization. We use instances of this metamodel to demonstrate and illustrate several aspects of transformation. Instances of classes are denoted as objects, instances of associations are denoted as links. References to elements of the metamodel are typed sans-serif.

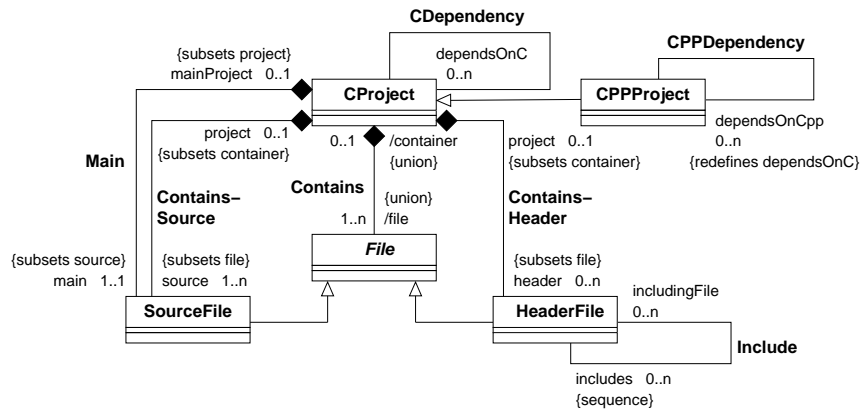


Fig. 1. A simple MOF 2.0 model for C and C++ projects

The metamodel depicted in Fig. 1 represents C Projects which, in general, consist of Files and, more detailed, especially of SourceFiles and HeaderFiles. Header files are characterized by their primary task of acting as modularization interface by grouping declarations and type definitions. As such, header files can be included by other header files through simple include directives³. This feature is represented in the metamodel by the association **Include**. For the purpose of refactoring, for instance, it is important that, on the one hand a header file can be included several times in the same file and on the other hand, that the include directives are ordered in the including file.

Thus, the association end `includes` is defined as non-unique which allows two objects of `HeaderFile` to be linked by `Include` several times. The additional definition of association end `includes` as ordered demands information on each link concerning the index within the total set of links. Both, uniqueness and ordering are together denoted by the constraint `{sequence}`.

³ The fact that header files can also be included by source files is neglected to keep the example as simple as possible

Beside header files, a C project also consists of source files which contain the source code. Additionally, source files as well as header files can be grouped by a project that acts as superior entity. The special relation between files and projects is covered by defining CProject as composite of Files. The special dependency between a composite (CProject) and its part(s) (SourceFile, HeaderFile), causes links of composite relations to connect two objects more tightly than links of a common (non-composite) association would do. This composite relation represents the fact, that the deletion of a complete project causes the deletion of its comprised header and source files, whereas the deletion of files does not affect the project at all.

The execution of a C program begins at a special method called *main()*. Since a project represents the superior entity which integrates several source files to a program, a project has to know the file which contains the *main*-method to be directly executable. This is modelled by the special association Main. A project has to include exactly one *main*-method and hence exactly one source file containing the *main*-method. Thus, the association end *main* is constrained by a lower and an upper bound of 1 to include exactly one SourceFile object. As a consequence, a refactoring of moving the *main*-method from one source file *A* to another source file *B*, would cause *A* to be detached from the project in the sense of being a *main*-method containing file and *B* to be declared as such.

The special characteristics of a *main*-method containing file does not prevent the file from being a source file. Thus, the source file should additionally appear in the set of the project's source files. This is modelled by declaring the association Main (or rather its association ends) as subset of ContainsSource. The subset relation causes links of Main to be also accessible through ContainsSource. Hence, the accessible link set of ContainsSource is composed by the direct links of ContainsSource combined with the links of Main.

Another similar relation is used between the associations Contains and ContainsSource and ContainsHeader, respectively. In this case, the superset association is marked as union of its subsets which states that the set of links of the superset association is exclusively determined by the combination of all its subsets. This reflects the fact, that a project's file set is composed of header and source files and nothing else. The declaration of an association as union of its subsets is a kind of derivation rule. Derived associations cannot be set directly since all instances have to comply with the derivation rule. In general, there could be arbitrary derivation rules (specified with OCL), but such a case is not covered by the example.

The C project metamodel is expanded by the class CPPProject to cover also C++ projects as a special form of C projects. Since, a C project can depend on other C projects, the dependency is modelled by the association CDependency. A C++ project as type compliant instance of CProject can be linked by CDependency with objects of CProject which, indeed, is not intended by the modeller. C++ projects are intended just to depend on C++ projects. This annoying side effect can be prevented by introducing a more special association CPPDependency for C++ projects as redefinition of CDependency. The redefinition ensures

that redefined and redefining association end represent exactly the same set of objects. Therefore, it is implicitly inhibited that a C project can be linked with a C++ project through the redefined association CDependency. In the following, we will introduce basic definitions to start with the formalization.

5 Basic Formalization of UML/MOF

The formalization of model transformations for UML/MOF assumes a formal model of UML/MOF itself. This formal model is presented by a set theoretic approach. We use the convention that sets are denoted by capital letters. Furthermore, we use the following basic definitions.

Basic Definitions

\perp denotes the undefined state

\mathbb{N}^+ denotes the set of positive natural numbers including ∞ and excluding 0.

\mathbb{N}_0^+ denotes the set of positive natural numbers including ∞ and 0.

$\mathbb{N}^\perp := \mathbb{N}_0^+ \cup \perp$ denotes the set of positive natural numbers including 0, ∞ and an undefined state.

$\mathbb{B} := \{true, false\}$ denotes the set of logical values.

$\mathbb{D} := \{first, second\}$ denotes a set of values for the distinction between the first and the second end of an association.

$proj_n((x_1, \dots, x_n, \dots, x_m)) = x_n$ denotes the n-th element in a given tuple.

$\mathcal{P}(X)$ denotes the powerset of a given set X

$\mathcal{B}(X)$ denotes set of all multi-sets over a given set X

$\#(X)$ denotes the cardinality of a given set X

$\#_y(X)$ denotes the cardinality of element y in a given multi-set X

r^* denotes the reflexive transitive closure of a binary relation r and a set-valued function (which is another representation of a binary relation)

r^+ denotes the transitive closure of a binary relation r

Given the basic definitions, we start with the formalization of a simple metamodel. In the following, this and further simple definitions will be replaced by extended definitions.

Definition 1: Metamodel

Let the alphabets

C denote a finite set of class identifiers and

A denote a finite set of binary association identifiers.

Then a metamodel is defined as follows

$$MM := (C, A, firstEnd, secondEnd) \quad (1)$$

where

$firstEnd : A \rightarrow C$ returns the first class of a given association and
 $secondEnd : A \rightarrow C$ returns the second class of a given association.

A metamodel simply consists of classes and binary associations. This concept of a metamodel reduces the features of UML/MOF to the absolute minimum. In fact, to this level of abstraction, the definition could also be considered to describe any other modelling language based on entities and relations. But we will converge to UML/MOF 2 again by successively expanding this definition and introducing several association features, since these are the main aspects for the application of graph transformation. Firstly, we proceed with the definition of a model.

Definition 2: Model

Let MM be a metamodel and let the alphabets

O_{MM} denote the infinite set of possible object IDs for all classes $c \in C$,
 where $C := proj_1(MM)$ and

L_{MM} denote the infinite set of possible link IDs for all associations $a \in A$,
 where $A := proj_2(MM)$.

Then a model is defined as follows

$$M = (O, L, class, association, firstObject, secondObject) \quad (2)$$

where

$O \subset O_{MM}$ denotes the finite set of the model's object IDs

$L \subset L_{MM}$ denotes the finite set of the model's link IDs

$class : O \rightarrow C$ returns the class $c \in C$ of a given object $o \in O$

$association : L \rightarrow A$ returns the association $a \in A$ of a given link $l \in L$

$firstObject : L \rightarrow O$ returns the first object $o \in O$ for a given link $l \in L$

$secondObject : L \rightarrow O$ returns the second object $o \in O$ for a given link $l \in L$

Furthermore, $Models_{MM}$ denotes the set of all (consistent) models for a given metamodel MM as defined above. A model $M \in Models_{MM}$ is denoted as consistent iff the following consistency constraint holds.

A model consists of objects and links. Each link connects either exactly two objects or one object with itself. Thus a model cannot contain dangling links. The remaining model represents a directed and labelled ⁴ graph and thus represents the basis for the application of known graph transformation techniques. In order to follow the wording of our application domain, we speak of model transformation instead of graph transformation, although, in our case, both terms represent the same. As a consequence, we also deviate from the common wording by using the terms object instead of node and link instead of edge, since we believe that this kind of vocabulary is more comfortable for people outside the graph transformation community. For the sake of simplicity, we use an abbreviation style which is introduced in the following.

⁴ The labelling is represented by the functions *class* for the labelling of objects (nodes) and *association* for the labelling of links (edges).

Abbreviation

Let MM be a metamodel and let $M_i \in Models_{MM}$. Then we use the following shortcuts:

$$\begin{aligned} O_i &:= proj_1(M_i) \\ L_i &:= proj_2(M_i) \\ class_i &:= proj_3(M_i) \\ association_i &:= proj_4(M_i) \\ firstObject_i &:= proj_5(M_i) \\ secondObject_i &:= proj_6(M_i) \end{aligned}$$

In contrast to our definition of a metamodel, we cannot revert to static semantics which are already specified in the specification of MOF/UML for our specification of a model. Thus, in the following we will provide an initial set of constraints for the definition of a model, which is also expanded successively in the remaining article. We do not claim to present a widespread and complete set of constraints. We rather concentrate on a minimal set of constraints that is essential for the application of transformations.

Constraint 1: Consistency

Let MM be a metamodel.

$$\begin{aligned} \forall M_i \in Models_{MM} \mid (\forall l \in L_i \mid \\ (class(firstObject(l)) = firstEnd(association(l))) \\ \wedge (class(secondObject(l)) = secondEnd(association(l)))) \end{aligned}$$

The consistency constraint ensures that only objects can be connected by a link whose classes correspond to the classes connected by the link's association. In fact, in such a way, that the first object's class corresponds to the first class of the link's association and that the second object's class corresponds to the second class of the link's association. This constraint is essential for all models. We proceed with the definition of a submodel which is necessary for the definition of pattern matching and transformation operations.

Definition 3: Submodel

Let MM be a metamodel and let $M, M_s \in Models_{MM}$. The model M_s is a submodel of M ($M_s \subseteq M$) iff:

1. $O_s \subseteq O$
2. $\forall o \in O_s \mid class_s(o) = class(o)$
3. $L_s \subseteq L$
4. $\forall l \in L_s \mid (association_s(l) = association(l)) \\ \wedge (firstObject_s(l) = firstObject(l)) \wedge (secondObject_s(l) = secondObject(l))$

Generally speaking, a submodel is nothing more than just a cut-out of that model, of which it is denoted to be a submodel of. The object sets as well as the link sets of model and submodel are in a subset relation with each other. Furthermore, the model's mapping into the metamodel remains unchanged for

the submodel. Since the submodel is also an element of $Models_{MM}$, it is free of dangling links as well. The three definitions mentioned until now, provide a formal base for the application of transformations. In the following, we introduce the concept of transformation by means of an example to proceed afterwards with the formalization.

6 Example of Transformations

In general, a model transformation describes the rewriting from one model to another one. The description of such a conversion can be designed in two different styles. The first and most obvious approach is an imperative approach which describes several actions like destroying/creating objects, navigating links, etc., whose applications represent the transformation. The disadvantage of such an imperative description is the fact, that the result of the transformation can only be determined after the application of all transformation steps, since the result is not explicitly declared.

The second approach is the declarative description of a transformation which describes a transformation by the specification of the source and the result of a transformation. This approach describes what is done by the application of a transformation rather than how the transformation is executed. For the purpose of graph transformations, the declarative description promises to be most beneficial, since the execution can be derived automatically.



Fig. 2. Description of a transformation

Thus, we describe a transformation by a pair of model patterns (left and right model pattern⁵) as depicted in Fig. 2. The left model pattern represents the model before the application of the transformation, whereas the right model pattern represents the model after the application of the transformation. The example depicted in part (a) of Fig. 2 shows the deletion of a link and an object since the object y and its link are not part of the right model pattern and thus deleted. In part (b) the opposite case is shown, which depicts the creation of a link and an object. Since the object y and its link are not part of the left

⁵ In the graph transformation community these models would be denoted as left- and right-hand side of the transformation.

model pattern but part of the right model pattern, they are created by the transformation.

We use the term model pattern rather than model since both model patterns are only descriptions of the transformation. The transformation is applied to a model which we call source model. Analogously, we call the result of the transformation target model. Thus, there has to be a mapping between the left model pattern and a source model and also between the right model pattern and a target model. Fig. 3 shows such a mapping which represents simultaneously the application of the transformation on the given source model. Identical objects and links on left and right side are denoted by using the same identifiers on both sides.

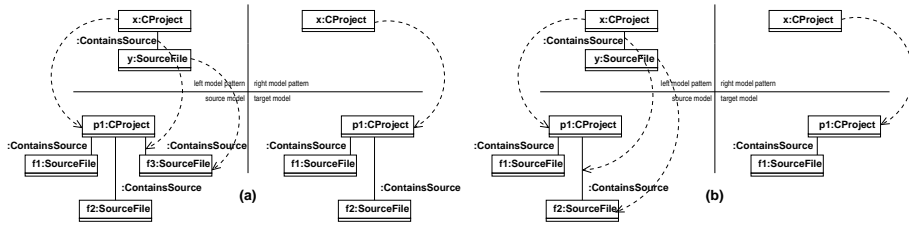


Fig. 3. Example of applications of transformation

In part (a) of Fig. 3, the left model pattern is mapped by the application of several mapping rules, which are introduced later on, onto an appropriate part of the source model. If there is more than one possible mapping, the application is non deterministic. The part of the source model onto which the left model pattern is mapped, or matched as we also call it, is modified by the transformation. It is modified in such a way, that after the application of the transformation, there is a mapping of the right model pattern onto the combined model consisting of the non matched part of the source model and the modified matched part of the source model. This combination is called target model. In the case of Fig. 3, the transformation effects the deletion of the object $f3$ and its link. After the application, there are still submodels in the source model onto which the left model pattern can be matched again. Thus, part (b) of Fig. 3 shows a further application of the transformation. In the following, we formalize those mentioned aspects of a transformation.

7 Formalization of Transformations

Since, the basic character of graph transformations is the specification of the transformation by a description of the graph patterns before and after the transformation rather than by a sequence of imperative statements, the fundamental approach is to map a graph (model) pattern onto a graph (model). Thus, it is

necessary to define a morphism for the mapping of a model (pattern) onto a model.

Definition 4: Model morphism

Let MM be a metamodel and let $M_1, M_2 \in Models_{MM}$.

Then a model morphism $h : M_1 \rightarrow M_2$ is a pair of functions $h = (h_O, h_L)$ that maps M_1 onto M_2 , where

$$h_O : O_1 \rightarrow O_2 \tag{3}$$

$$h_L : L_1 \rightarrow L_2 \tag{4}$$

$$\forall o \in O_1 \mid class_2(h_O(o)) = class_1(o) \tag{5}$$

$$\forall l \in L_1 \mid association_2(h_L(l)) = association_1(l) \tag{6}$$

$$\forall l \in L_1 \mid h_O(firstObject_1(l)) = firstObject_2(h_L(l)) \tag{7}$$

$$\forall l \in L_1 \mid h_O(secondObject_1(l)) = secondObject_2(h_L(l)) \tag{8}$$

Furthermore, $Morphisms_{MM}(M_1, M_2)$ denotes the set of all possible model morphisms between two given models $M_1, M_2 \in Models_{MM}$ for a given metamodel MM .

In contrast to the definition of a submodel which defines the relation between different parts of one model, a morphism describes the relation between (parts of) different models. A model can be considered being homomorphic to a second model if all objects and all links of the first model can be mapped onto objects and links of the second model, in such a way, that the classes and associations of the mapped objects and links remain unchanged in the second model. Furthermore, a link can only be mapped, if its first and second objects can be mapped onto the first and second objects of the link's equivalence as well. Thus, the basic idea of a model morphism is to define a grade of similarity between two models. This kind of similarity can be used to define a model transformation.

Definition 5: Transformation

Let MM be a metamodel and let $M_l, M_r \in Models_{MM}$. Then a transformation T is a pair of model(s) (patterns) $T := (M_l, M_r)$. Furthermore, $Transformations_{MM} := (Models_{MM} \times Models_{MM})$ denotes the set of all transformations for a given metamodel MM . Additionally, we define

$$leftModelPattern_T := proj_1(T) \text{ with } T \in Transformations_{MM}$$

$$rightModelPattern_T := proj_2(T) \text{ with } T \in Transformations_{MM}$$

A transformation is defined as the tuple of models. In fact, the first model acts as left model pattern of the transformation and consequently the second model acts as right model pattern of the transformation. Together with the definition of morphism, the definition of a transformation can be used to define the application of a transformation.

Definition 6: Application of a Transformation

Let MM be a metamodel and let $M_1, M_2 \in Models_{MM}$, $T \in Transformations_{MM}$ and $M_l = leftModelPattern(T)$, $M_r = rightModelPattern(T)$ whereas M_2 is one possible result of the application of T to M_1 . Then a transformation T defines a binary relation on models $\rightsquigarrow_T \subseteq Models_{MM} \times Models_{MM}$.

$(M_1, M_2) \in \rightsquigarrow_T$ holds iff

$$\begin{aligned} \exists h^l &= (h^l_O, h^l_L) \in Morphisms_{MM}(M_l, M_1), \\ \exists h^r &= (h^r_O, h^r_L) \in Morphisms_{MM}(M_r, M_2) \mid \\ & (h^l_O(O_l \cap O_r) = h^r_O(O_l \cap O_r)) \\ & \wedge (h^l_L(L_l \cap L_r) = h^r_L(L_l \cap L_r)) \end{aligned} \tag{9}$$

$$\begin{aligned} \exists M_i \in Models_{MM} \mid \\ & (O_i = O_1 \setminus DelObj_T) \wedge (L_i = L_1 \setminus DelLink_T) \\ & \wedge (O_i = O_2 \setminus AddObj_T) \wedge (L_i = L_2 \setminus AddLink_T) \\ & \wedge (M_i \subseteq M_1) \wedge (M_i \subseteq M_2) \end{aligned} \tag{10}$$

$$\begin{aligned} \exists M_c \in Models_{MM} \mid \\ & (O_c = O_l \cap O_r) \wedge (L_c = L_l \cap L_r) \\ & \wedge (M_c \subseteq M_l) \wedge (M_c \subseteq M_r) \end{aligned} \tag{11}$$

where

1. $DelObj_T = h^l_O(O_l) \setminus h^r_O(O_r)$ are the objects which are deleted by the application of T
2. $DelLink_T = h^l_L(L_l) \setminus h^r_L(L_r)$ are the links which are deleted by the application of T
3. $CoreObj_T = h^l_O(O_l) \cap h^r_O(O_r)$ are the objects⁶ which are not affected by the application of T
4. $AddObj_T = h^r_O(O_r) \setminus h^l_O(O_l)$ are the objects which are created by the application of T
5. $AddLink_T = h^r_L(L_r) \setminus h^l_L(L_l)$ are the links which are created by the application of T

In the following, we denote M_1 as *source model* and M_2 as *target model*. A target model M_2 can be the result of a transformation T which is applied to a source model M_1 , if three constraints hold. First of all, as formulated in formula (9), the matched but unaffected objects and links of the transformation, which are determined by the intersection of the left and right model pattern's object

⁶ Although this set is firstly used in following definitions, it is already introduced since it is perfectly in line with the remaining sets.

and link sets, have to be mapped on the same objects and links in the source and target model as well. Considering the example in Fig. 3, the unaffected part of the transformation is the object x . It is essential, that x in the right model pattern is mapped onto the same object ($p1$) onto which it is mapped from the left model pattern. The same restriction must hold for links as well, although not mentioned in the example.

Secondly, as formulated in formula (10), there has to be an intermediate model, which is equally constructed, on the one hand, from the source model's objects and links without the deleted objects/links and on the other hand, from the target model's objects/links without the newly created objects/links. Furthermore, that intermediate model has to be a submodel of the source model and a submodel of the target model, as well. By demanding the existence of such an intermediate model, it is ensured that there are no dangling links left. Dangling links arise when only one of a link's objects is deleted and the remaining object and link are not matched. A transformation which would leave dangling links cannot be applied⁷. Thus, avoiding dangling links is a matter of the transformation rather than of its application. The further demand of the intermediate model being a submodel of the source and target models, ensures that the objects and links retain their classes and associations.

Thirdly, the left and right model pattern must have a common core which we call core model pattern. It represents all those elements which are matched by the transformation and preserved. This core model pattern, as described by formula (11) is simply composed by the intersection of the left and right model pattern's elements. Similar to the intermediate model, which is a kind of analogon to the core model pattern, the core model pattern also has to be a submodel of the left and right model pattern.

7.1 Model transformations with multiple inheritance

Since, due to missing features of abstraction, the previous definition of a metamodel is far away from being useful for practical concerns, we expand the definition of a metamodel by the feature of (multiple) inheritance.

Definition 1a: Metamodel with inheritance

Definition 1 remains unchanged except for formula (1) which is replaced by formula (12).

$$MM := (C, A, firstEnd, secondEnd, subClasses) \quad (12)$$

$subClasses : C \rightarrow \mathcal{P}(C)$ returns the set of classes which directly inherit from a given class

The expanded definition of a metamodel opens new aspects for the definition of a model morphism as depicted in Fig. 4. Since, the inheritance of a special

⁷ This restriction will be relaxed during the remaining article.

class represents the specialization of the concept which is modelled by the general class, a transformation that is formulated in the context of the general concept should also be applicable in the specialized context. That is, considering the example in Fig. 4, a transformation which is applicable in the context of the class `File` should also be applicable in the context of class `HeaderFile` as depicted in part (a) and also in the context of class `SourceFile` as depicted in part (b). Thus, we expand the definition of a morphism to take this aspect into account.

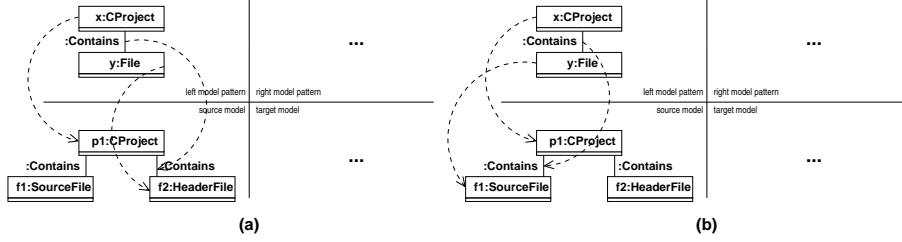


Fig. 4. Example of inheritance and model morphism

Definition 4a: Model morphism with inheritance

Definition 4 remains unchanged except for formula (5) which is replaced by formula (13).

$$\forall o \in O_1 \mid class_2(h_O(o)) \in subClasses^*(class_1(o)) \tag{13}$$

The former definition of morphism allowed the mapping of an object onto a target object only in that special case in which the target object has exactly the same class as the object that has to be mapped. This restriction is relaxed in such a way, that an object can also be mapped onto a target object whose class is a direct or an indirect subclass of the mapped object’s class. Consequently, the consistency constraint has to be relaxed as well. A model is denoted to be consistent if the classes of linked objects are equal to or subclasses of the corresponding classes of the link’s association. Thus, constraint 1 is replaced by constraint 1a.

Constraint 1a: Consistency

Let MM be a metamodel.

$$\begin{aligned} \forall M_i \in Models_{MM} \mid (\forall l \in L_i \mid \\ & (class(firstObject(l)) \in subClasses^*(firstEnd(association(l)))) \\ & \wedge (class(secondObject(l)) \in subClasses^*(secondEnd(association(l)))))) \end{aligned}$$

8 Formalizing the impact of UML/MOF associations

Until now, we have not considered any of the association features provided by UML/MOF. Thus, in the following, we successively introduce each feature starting with simple ordered associations and finally close with subsetted and redefined associations.

8.1 Ordering

The feature of ordering is a feature of association ends rather than of the association itself. An association end represents the set of objects which are linked to a given object. In case of an ordered association end, this set of objects is rather a sequence than a set. Thus, there is additional ordering information that has to be considered for the matching of links. The following definition introduces the necessary metamodel extension.

Definition 1b: Metamodel with ordering and inheritance

Definition 1a remains unchanged except for formula (12) which is replaced by formula (14).

$$MM := (C, A, firstEnd, secondEnd, subClasses, isOrdered) \quad (14)$$

$isOrdered : A \times \mathbb{D} \rightarrow \mathbb{B}$ returns the information whether the first or second end of a given association is ordered or not.

Since, the ordering information has to be stored in the model, the definition of a model has to be extended as well.

Definition 2b: Model with ordering and inheritance

Definition 2a remains unchanged except for formula (2) which is replaced by formula (15).

$$M = (O, L, class, association, firstObject, secondObject, index) \quad (15)$$

$index : L \times \mathbb{D} \rightarrow \mathbb{N}^\perp$ returns the ordering information of a given link for the first or second object.

A link carries ordering information for its first and second object. The ordering information represents the index of the position at which the object is stored in the opposite object's sequence. Ordering information is only available if the appropriate association end is specified as ordered. Thus, a further constraint is needed.

Constraint 2: Ordering

Let MM be a metamodel.

$$\begin{aligned} \forall M_i \in Models_{MM} \mid (\forall l \in L_i \mid \\ (index(l, first) \neq \perp \leftrightarrow isOrdered(association(l), first)) \\ \wedge (index(l, second) \neq \perp \leftrightarrow isOrdered(association(l), second))) \end{aligned}$$

Beside the constraint, the extension of the morphism is required as well. If there is ordering information on a link, this link can only be mapped onto a link which contains the same ordering information.

Definition 4b: Model morphism with ordering and inheritance

Definition 4a is extended by the following formulae but besides remains unchanged.

$$\forall l \in L_1 \mid \text{index}_1(l, \text{first}) \neq \perp \Rightarrow (\text{index}_1(l, \text{first}) = \text{index}_2(h_L(l), \text{first}))$$

$$\forall l \in L_1 \mid \text{index}_1(l, \text{second}) \neq \perp \Rightarrow (\text{index}_1(l, \text{second}) = \text{index}_2(h_L(l), \text{second}))$$

8.2 Composite

In contrast to the feature of ordering, composition has quite a wide influence on the application of transformations. Since, the purpose of a composition is to model a life-cycle dependency, the deletion of a composite object should always effect the deletion of its parts as well. The consequences for the application of a transformation are depicted in Fig. 5.

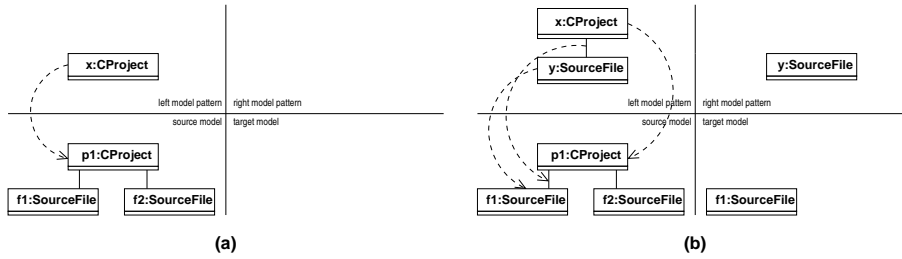


Fig. 5. Example of composites and transformation

Part (a) of Fig. 5 depicts a transformation which deletes a composite object `x`. Due to the composite association from `SourceFile` to `CProject`, all objects of `SourceFile` have to be deleted together with their `CProject` object. There is one exception to this rule, namely in the case, if one of the child objects is explicitly preserved. Such a case is depicted in part (b) of Fig. 5. The left model pattern matches a composite object (`x`) and one of its linked child objects (`f1` as depicted in the figure). Since, in the right model pattern the child object is still existent, `f1` must not be deleted. Nevertheless, the rule mentioned before is still valid and therefore all other child objects (`f2`) have to be deleted as well. Note that the dangling link between `f1` and `p1` which is left after the deletion of `p1` has to be deleted as well. In the following, we introduce these aspects into the definitions, starting with the extension of the metamodel.

Definition 1c: Metamodel with composition, ordering, ...

Definition 1b remains unchanged except for formula (14) which is replaced by formula (16).

$$MM := (C, A, firstEnd, secondEnd, subClasses, isOrdered, isComposite) \quad (16)$$

isComposite : $A \times \mathbb{D} \rightarrow \mathbb{B}$ returns the information whether an end of a given association is a composite end or not.

The function *isComposite* returns *true* for one association end x if the association's other end y is specified as composite and as such association end x is decorated with a black diamond. In fact, MOF/UML demands the composite feature to be specified at the opposite end (see [10], p. 111). Since it is our intention to draw our formalization as close to MOF/UML as possible, we stick to this definition. In contrast to the extension of the metamodel, it is not mandatory to extend the definition of a model. Whereas, the deletion of a composite's child objects has to be integrated in the definition of the application of a transformation. We introduce two functions to simplify the definition. Both introduced functions are meant to identify those objects and links which have to be deleted as a consequence of a composite deletion. Note that the query of the composite feature at the opposite link side, during the calculation of *compositeLinks*, is a result of the inverted definition of the function *isComposite*. Furthermore, we introduce two sets, which contain all the objects (*DelCompObj_T*) and links (*DelCompLink_T*) which have to be explicitly deleted by the transformation T and implicitly due to a deleted composite.

Definition 6c: Application of a transformation with composition, ...

compositeLinks : $O_1 \rightarrow \mathcal{P}(L_1)$ returns the links in which a given object acts as composite object.

childObjects : $O_1 \rightarrow \mathcal{P}(O_1)$ returns all child objects for a given composite object.

$$\begin{aligned} compositeLinks(o) = \{ & l \in L_1 \mid \\ & ((firstObject(l) = o) \wedge (isComposite(association(l), second))) \\ & \vee ((secondObject(l) = o) \wedge (isComposite(association(l), first))) \} \end{aligned}$$

$$\begin{aligned} childObjects(o) = \{ & o_c \in O_1 \mid (o_c \neq o) \wedge (\exists l \in compositeLinks(o) \mid \\ & ((o_c = firstObject(l)) \vee (o_c = secondObject(l)))) \} \end{aligned}$$

$$DelCompObj_T = \left(\bigcup_{o \in DelObj_T} childObjects^*(o) \right) \setminus CoreObj_T \supseteq DelObj_T$$

$$\begin{aligned} DelCompLink_T = \{ & l \in L_1 \mid (firstObject(l) \in DelCompObj_T) \\ & \vee (secondObject(l) \in DelCompObj_T) \} \cup DelLink_T \end{aligned}$$

Definition 6a remains unchanged except for formula (10) which is replaced by formula (17).

$$\begin{aligned}
& \exists M_i \in Models_{MM} \mid & (17) \\
& (O_i = O_1 \setminus DelCompObj_T) \wedge (L_i = L_1 \setminus DelCompLink_T) \\
& \wedge (O_i = O_2 \setminus AddObj_T) \wedge (L_i = L_2 \setminus AddLink_T) \\
& \wedge (M_i \subseteq M_1) \wedge (M_i \subseteq M_2)
\end{aligned}$$

The new definition differs from the former version only in the way in which the intermediate model is composed. Since, the intermediate model's object set is composed by the source model's object set without the objects that are explicitly deleted by the transformation, the implicitly deleted objects, as determined by the set $DelCompObj_T$, have to be removed from the source model's object set as well. Due to the fact that $DelCompObj_T \supseteq DelObj_T$, the subtraction of $DelCompObj_T$ deletes the explicitly deleted object as well. The same applies to the set of links. Beside the implicitly deleted links, the set $DelCompLink_T$ includes the explicitly deleted links as well as all dangling links, too. The core model pattern of the transformation description is not affected at all. But finally, for the combined deletion it is essential, that there are no cycles of composite deletion. Due to the importance of this constraint, it is explicitly defined in the following.

Constraint 3: No composite cycles

Let MM be a metamodel.

$$\forall M_i \in MM \mid (\forall o \in O_i \mid o \notin childObjects^+(o))$$

8.3 Multiplicity

The most significant association feature is the feature of multiplicity. Multiplicity simply limits the number of objects which can be linked with a given object. Again, this feature applies to association ends. First of all we extend the metamodel with all necessary functions.

Definition 1d: Metamodel with multiplicity, ...

Definition 1c remains unchanged except for formula (16) which is replaced by formula (18).

$$\begin{aligned}
MM := (C, A, firstEnd, secondEnd, subClasses, isOrdered, & (18) \\
isComposite, lower, upper)
\end{aligned}$$

$lower : A \times \mathbb{D} \rightarrow \mathbb{N}_0^+ \setminus \infty$ returns the lower bound of one of a given association's ends.

$upper : A \times \mathbb{D} \rightarrow \mathbb{N}^+$ returns the upper bound of one of a given association's ends.

The multiplicity of an association end is determined by a lower and an upper bound. A lower bound must neither be less than zero nor infinite. An upper bound must be greater than zero and can also be infinite in case of an unlimited multiplicity.

Definition 2d: Model with multiplicity, ...

Definition 2c is extended by the following formulae but besides remains unchanged.

$links_M : A \rightarrow \mathcal{P}(L)$ returns all links of a given association

$query_M : O \times A \times \mathbb{D} \rightarrow \mathcal{B}(O)$ returns the multi-set of all objects which are linked to a given object by one of a given association's ends

$$links_M(a) = \{l \in L \mid association(l) = a\}$$

$$query_M(o, a, second) = \{o_l \in O \mid (\exists l \in links(a) \mid (firstObject(l) = o) \wedge (secondObject(l) = o_l))\}$$

$$query_M(o, a, first) = \{o_l \in O \mid (\exists l \in links(a) \mid (firstObject(l) = o_l) \wedge (secondObject(l) = o))\}$$

The most obvious effect of multiplicity is, that the application of a transformation can violate the multiplicity constraints by adding or removing too many objects. Thus, it is necessary to constrain a model to be compliant to the multiplicity constraints. With such a constraint, a transformation which would create a model that is not valid concerning its multiplicity, is prevented from being applied, since a resulting model M is part of $Models_{MM}$ and all $M \in Models_{MM}$ respect the specified model constraints. But the demand, that all $M \in Models_{MM}$ respect all the model constraints causes problems with the definition of the intermediate model. The example depicted in Fig. 6 illustrates the problem.

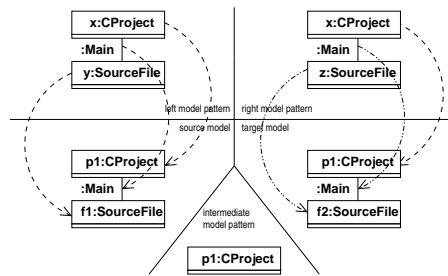


Fig. 6. Example of multiplicity and transformation

The transformation matches a CProject object and exchanges the SourceFile object which is linked by an instance of the association Main. Such a transformation can be applied without any problems, since, the resulting target model respects all model constraints. However, the intermediate model, which is exceptionally depicted as well, does not respect the multiplicity demanded by the association end main's lower bound of one. This would prevent the transformation from being applied although it would create a valid target model. Thus, if there is a multiplicity constraint, $M_i \in Models_{MM}$ cannot be demanded. In the following, we will, therefore, introduce an additional set of *valid* models and the before mentioned constraint. We denote a model to be valid, if it respects all multiplicities.

Constraint 4: Validity

The set $ValidModels_{MM} \subset Models_{MM}$ contains all models which respect to the following constraint. Let MM be a metamodel.

$$\begin{aligned} \forall M \in ValidModels_{MM} \mid (\forall l \in L \mid & \\ & (\#query_M(firstObject(l), association(l), second) \\ & \geq lower(association(l), second)) \\ & \wedge (\#query_M(firstObject(l), association(l), second) \\ & \leq upper(association(l), second)) \\ & \wedge (\#query_M(secondObject(l), association(l), first) \\ & \geq lower(association(l), first)) \\ & \wedge (\#query_M(secondObject(l), association(l), first) \\ & \leq upper(association(l), first))) \end{aligned}$$

Since, we do not demand that a model $M \in Models_{MM}$ respects the multiplicity constraint, we have to adapt the definition of the application of a transformation in such a way, that only models are produced which respect the multiplicity constraint.

Definition 6d: Application of a transformation with multiplicity

The definition of the binary relation \rightsquigarrow_T is changed from $\rightsquigarrow_T \subseteq Models_{MM} \times Models_{MM}$ to $\rightsquigarrow_T \subseteq ValidModels_{MM} \times ValidModels_{MM}$.

The rest of the specification remains unchanged except for $M_1 \in ValidModels_{MM}$ and $M_2 \in ValidModels_{MM}$

8.4 Uniqueness

Compared to former versions of MOF/UML, the first new feature of MOF/UML 2 is the possibility to declare an association (end) as non-unique. Although the impact of this feature on transformations is quite low, we take it into account to provide a formalization which covers as much association features as possible. Beside that, there are, of course, useful applications. For the purpose of reengineering of C/C++ source code, for instance, it is important to detect multiple

includes of the same header file. With unique associations, it would only be possible to state whether a header file is included or not. The also existing case that a header file can be included several times cannot be covered without non-unique associations. The effect of such a non-unique association is depicted in Fig. 7. If there is more than one link of the association `Include` between two objects of the class `HeaderFile`, a transformation which deletes the include relation between two header files can be applied more than once.

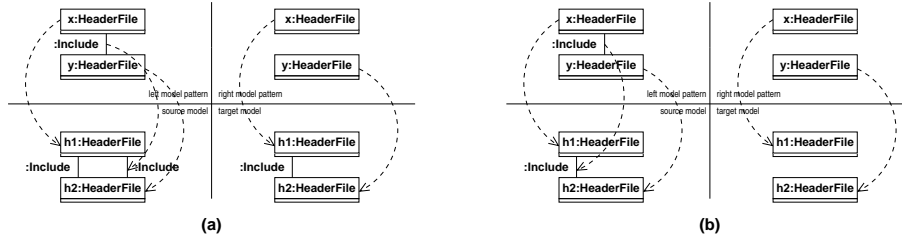


Fig. 7. Example of non-uniqueness and transformation

In fact, there is no difference between the application of a transformation on a unique and on a non-unique association. If there are multiple morphisms between left model pattern and source model, a transformation can be applied in several ways. Nevertheless, we extend the metamodel by the feature of (non-)uniqueness.

Definition 1e: Metamodel with uniqueness, ...

Definition 1d remains unchanged except for formula (18) which is replaced by formula (19).

$$MM := (C, A, firstEnd, secondEnd, subClasses, isOrdered, isComposite, lower, upper, isUnique) \quad (19)$$

$isUnique : A \times \mathbb{D} \rightarrow \mathbb{B}$ returns the information whether one of a given association's ends is declared as unique.

The only additional definition which is required for the purpose of (non-)uniqueness is a constraint which limits the number of links between two given objects to one in case of a unique association. Since this constraint cannot be violated by the composition of the intermediate model, it must hold for each $M \in Models_{MM}$.

Constraint 5: Uniqueness

The following constraint holds for each $M \in Models_{MM}$.

Let MM be a metamodel.

$$\begin{aligned} & \forall M \in Models_{MM} \mid (\forall l \in L \mid \\ & \quad (isUnique(association(l), first) \wedge isUnique(association(l), second)) \\ & \quad \Rightarrow (\forall o \in O \mid \#_o(query_M(firstObject(l), association(l), second)) \leq 1) \\ & \quad \wedge (\forall o \in O \mid \#_o(query_M(secondObject(l), association(l), first)) \leq 1)) \end{aligned}$$

8.5 Subsetting and Inheritance of Associations

One of the major improvements of MOF/UML 2 compared to its predecessors is the subsetting of association ends. In fact, MOF/UML defines the feature of subsetting to be applicable on association ends rather than on an association as a whole. In case of subsetting association ends, “given a set of specific instances for the other ends of both associations, the collection denoted by the subsetting end is fully included in the collection denoted by the subsetted end” ([10], p. 111). But since, for each of those objects in the collection there is a link to the given object, the opposite association ends also maintain a subset relation. A subset relation on one association end implies also a subset relation on the opposite association end.

Although subsetting obviously affects the whole association and not just single ends, the application to association ends is quite reasonable for practical purposes. The application to associations would demand a continuous naming of associations. Since, a name inside a namespace, in the case of associations inside a package, has to be unique, the demand of a continuous and reasonable naming complicates the already difficult task of naming associations additionally. Thus, the alternative to use the usually easy to name association ends instead, suggests itself. Nevertheless, there are no semantic differences between declaring the first, the second or even both association ends as subsets of another association’s ends.

Therefore, we treat subsetting as association feature. The subsetting of complete associations instead of association ends raise the question how subsetting relates to the also available feature of association inheritance. “The existence of a link of a specializing association implies the existence of a link relating the same set of instances in a specialized association” ([10], p.111). Hence, inheritance arise the same effects as subsetting. Thus, we treat subsetting in its three variants⁸ and inheritance just as syntactical variations, which can be used depending on local diagram aspects, and provide the same semantics for all variants. The metamodel is extended by a single function which returns all the associations that apply to the previous mentioned semantics.

Definition 1f: Metamodel with subsetting, ...

Definition 1e remains unchanged except for formula (19) which is replaced by formula (20).

⁸ 1) The first end is a subset. 2) The second end is a subset. 3) Both ends are subsets.

$$MM := (C, A, firstEnd, secondEnd, subClasses, isOrdered, \quad (20) \\ isComposite, lower, upper, isUnique, subAssociations)$$

$subAssociations : A \rightarrow \mathcal{P}(A)$ returns all associations which subset a given association.

The definition of a model mostly remains unchanged except for the definition of the function $query$. If there are subsets of the association end which is queried, the links of the subsetting association have to be considered as well. Since, a subsetting association can also be subsetted by a further association the subsetting associations have to be determined transitively.

Definition 2f: Model with subsetting, ...

Definition 2d remains unchanged except for the definition of $query$ which is replaced by the following.

$$query_M(o, a, second) = \{o_l \in O \mid (\exists l \in \bigcup_{a' \in subAssociations^*(a)} links_M(a') \mid \\ (firstObject(l) = o) \wedge (secondObject(l) = o_l))\}$$

$$query_M(o, a, first) = \{o_l \in O \mid (\exists l \in \bigcup_{a' \in subAssociations^*(a)} links_M(a') \mid \\ (firstObject(l) = o_l) \wedge (secondObject(l) = o))\}$$

The extension of the function $query$ influences the definitions of multiplicity and uniqueness which use the function as well. The validity constraint uses the function $query$ to limit the number of objects given by an association end. The application of the new definition of $query$ changes the constraint in such a way, that if there is a subset relation, the set of objects given by $query$ is expanded by the set of objects which is given by a query on the subassociation. Thus, the upper and lower bound of an association end have to be checked against all subassociations. As a consequence, the lower bound of a subsetted association must not be less than the sum of all its subassociations' lower bounds and the upper bound has to be greater or equal than the sum of its subassociations' upper bounds.

Furthermore, there are consequences for the uniqueness constraint as well. The uniqueness constraint states that in case of a unique association, there must not be more than one link between two objects. With the new definition of $query$, the constraint states that there must not be more than one link between two objects in the association itself and all its subassociations. Thus, all subassociations of a unique association must be unique as well.

Between those rather less considerable effects, there is a major effect on the definition of a model morphism. Analogously to the inheritance of classes, a

link of a subsetted (or inherited) association can be mapped onto a link of a subsetting (or inheriting) association. There are some examples depicted in Fig. 8.

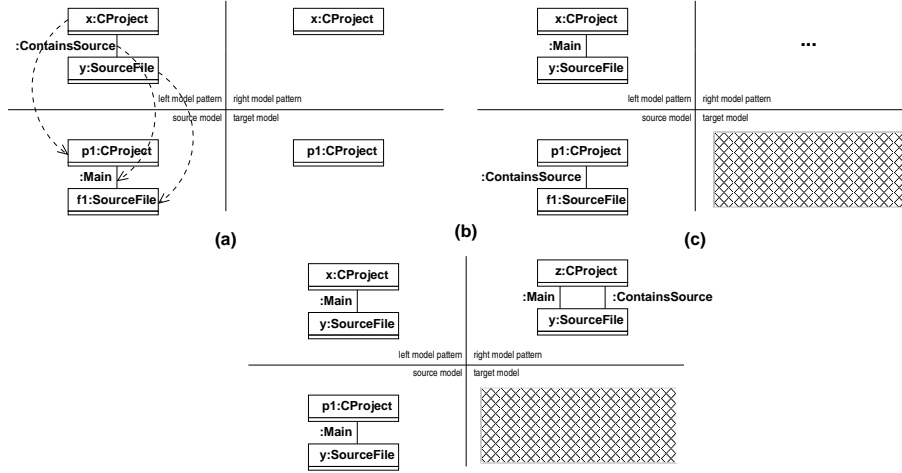


Fig. 8. Example of subsetting and transformation

Part (a) of Fig. 8 depicts a case as mentioned before. The transformation matches one of a matched project's source files through a link of the association `ContainsSource`. Since the association `Main` subsets the association `ContainsSource` the depicted morphism can be applied. Part (b) and (c) of Fig. 8 depict cases in which there are no morphisms between left model pattern and source model and thus both transformations cannot be applied. Part (c) shows the inverse case as depicted in Part (a). A link of the specializing associations cannot be mapped onto a link of the specialized association. Finally, Part (b) shows a case in which an additional link of the subsetted association would violate the multiplicity constraint and thus prevents the association from being applied. Compared to the important consequences, the effective changes to the definition of a model morphism are quite simple. With this extension the rule for the mapping of links becomes similar to the rule for the mapping of objects again. Now, instances in general can be mapped onto instances of a more special classifier.

Definition 4f: Morphism with subsetting

Definition 4b remains unchanged except for formula (6) which is replaced by formula (21).

$$\forall l \in L_1 \mid association_2(h_L(l)) \in subAssociations^*(association_1(l)) \quad (21)$$

In general, the specification of an association end as subset of another (sub-setted) association end implicitly specifies the subsetted association end as a superset. A further feature concerning subsetting is the specification of such an implicit superset as an exclusive union of all its subsets. This feature is also applied in the example of Fig. 1. Beside the association `Contains` between the classes `File` and `CProject`, there are the associations `ContainsSource` and `ContainsHeader` which connect specialized classes of `File` with `CProject`. Consequently, both associations are a subset of the general association `Contains` and thereby provide the possibility to query the set of files for a given C project. Since, a C project consists either of source files or header files, the set of files can be specified as an exclusive union of its subsets (the set of source files and the set of header files). Thus, both association ends of the general association `Contains` are specified as union of their subsets and as such can be regarded as derived. Hence, an association end which is declared as union is often denoted as derived union.

In practice, the specification of supersets as union is quite a popular solution to reflect the abstraction of class hierarchies on associations. Therefore, we will point out the impact of unions on the application of transformations although the overall impact is quite low. Actually, there is no difference between the matching behaviour of implicit supersets and derived unions. The derivation only effects the rewriting. The deletion of a union association's link does not cause any problems as well. Such a link can only be matched on a (deletable) link of one of the subsetting associations since there are no direct links of a union association. On the other hand, exactly due to this reason, the creation of a union association's link cannot be executed at all. But, since such an illegal modification can be detected by a static analysis over the left and right model patterns, there is no need for a check at runtime. Transformations which try to create a link of an association declared as derived union are illegal and as such not executed.

8.6 Redefinition

Beside the subsetting of associations, the redefinition of associations is another major improvement of UML/MOF 2 compared to its predecessors. Again, this feature is applied to association ends. But this time the application to association ends instead to the complete association is essential, as we will demonstrate in the following.

In general there are two ways in which the redefinition of an association end can be interpreted. On the one hand, redefinition can be interpreted as a feature which "prevents inheritance of a redefined element into the redefinition context thereby making the name of the redefined element available for reuse, either for the redefining element, or for some other" (see [10], p.127). Since most object oriented languages do not support inheritance and cancellation, such a interpretation would complicate the generation of source code.

On the other hand, there is the set theoretic interpretation which provides a more realistic approach. In the case of a redefined association end "`redefined`", given a

set of specific instances for the other ends of both associations, the collections denoted by the redefining and redefined ends are the same” ([10], p. 111). Such an interpretation is very well representable in source code; therefore, we stick to this definition. Since the definition demands the redefining and the redefined association end to denote the same set of objects, it implicitly limits the redefined association end to those objects which can also be part of the redefining association end.

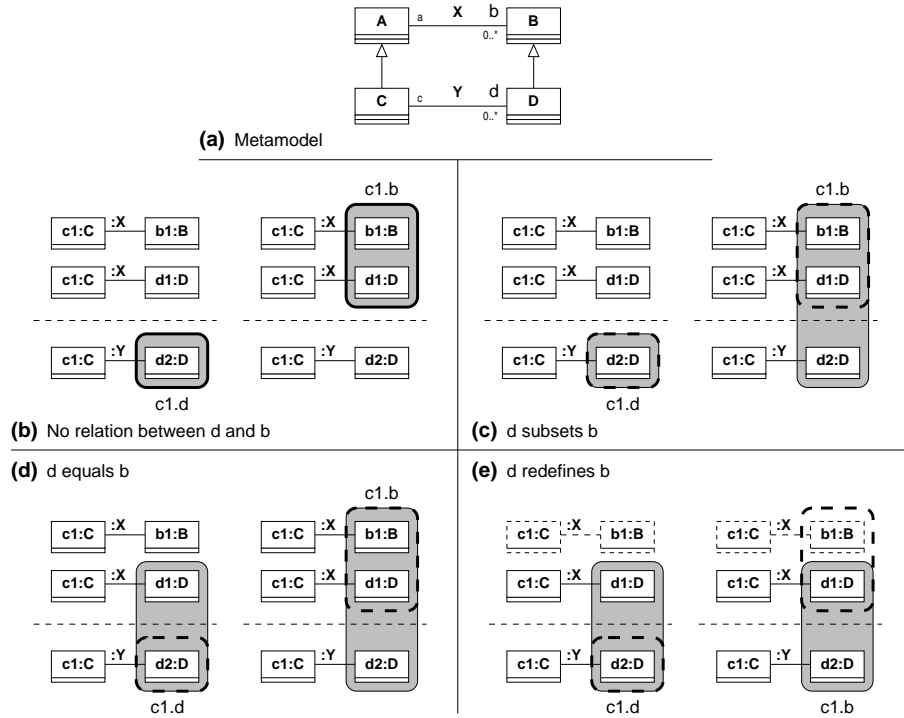


Fig. 9. Example of relations between associations

Fig. 9 demonstrates this aspect in correlation with all other possible relations between associations. Part (a) of Fig. 9 depicts a simple metamodel consisting of four classes and two associations. Throughout the remaining parts (b) - (e) of the figure, a set of linked objects of this metamodel is repeatedly used to demonstrate several query results. This set consists of two links of the association X above the dashed line and of one link of the association Y below the dashed line. Each link's first object is the object *c1* which is an instance of class C. This set of objects is used to demonstrate the effects of the several relations between associations on the result of an association query. On the left hand side of the figure's part (b), the result of the query of association end *d* based on the object

c1 is shown (highlighted by the grey background). Whereas, on the right hand side, the result of the query of association end *b* based on the same object is shown. Since there is no relation between both involved association ends, the two results are completely independent. In the following we will refer to these results as original queries.

In part (c) of Fig. 9 the results of both queries are shown in the case that the association end *d* is a subset of association end *b*. As explained before, the original query of *b* is expanded by the query of *d*. Before we address the issue of redefinition, we extend the idea of subsetting and introduce the feature of equality. Equal associations share links in both directions. Similar to subsetting, the links of the special association are also available as links of the general association. Additionally, also those links of the general association, whose objects are compliant to the classes of the special association, are available as links of the special association. This effect is depicted in part (d) of Fig. 9. The query of association end *b* is the same as in the case of subsetting. The difference compared to subsetting is, that the original query of *d* is expanded by those objects of the original query of *b* which are compliant to the class *D*. This feature of equality applies to the association as a whole due to the same reasons as subsetting applies to the association as a whole. We suggest that such a relation should be denoted by the constraint {equals}. An equals relation on the first association end implies an equals relation on the second end.

Based on the feature of equal association ends, the effects of a redefinition as depicted in part (e) of Fig. 9 get more clear. Since per definition, the query of *d* has to be the same as the query of *b*, those objects have to be excluded from the query of *b* whose class is not compliant to the class of the redefining association end *d*. In case of the example, the link between *c1* and *b1* cannot be queried due to the redefinition anymore and thus, must not be created at all. If those links which cannot be queried can neither be created, the effects of a redefinition are congruent to the effects of equality (cp. parts (d) and (e)). Thus, the effects of redefinition on the application of transformations can be covered by describing the effects of equal association ends together with an additional constraint which prevents the creation of links violating the redefinition.

Due to the equal relation, a transformation on a redefined or redefining association can be applied as depicted in Fig. 10. The parts (a) and (c) of Fig. 10 show the two possible variations. In the first case a general link is mapped onto a more special link whereas in the latter case the inverted case of mapping a special link onto a more general link is depicted. Furthermore, the impact of the constraint is shown in part (b) of Fig. 10. Due to the redefinition of `dependsOnC`, an instance of `CPPProject` can only be linked through `CDependency` with an instance of `CPPProject`. Therefore, the transformation cannot be applied. The metamodel has to be extended by a function which returns all the associations that redefine a given association. This is necessary for the mentioned constraint, which is introduced later on. And additionally, we introduce a function that returns all the associations which maintain an equal relation to a given association.

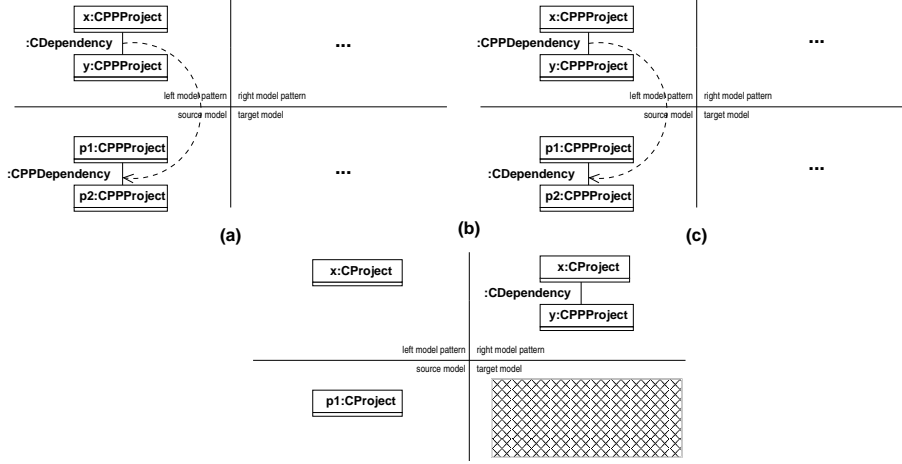


Fig. 10. Examples of redefinition and transformation

Definition 1g: Metamodel with redefinition, ...

Definition 1f remains unchanged except for formula (20) which is replaced by formula (22).

$$MM := (C, A, firstEnd, secondEnd, subClasses, isOrdered, isComposite, lower, upper, isUnique, subAssociations, redefinitions, equalAssociations) \quad (22)$$

redefinitions : $A \times \mathbb{D} \rightarrow \mathcal{P}((A, first)) \cup \mathcal{P}((A, second))$ returns the associations which are redefined by one of a given association's ends.

equalAssociations : $A \rightarrow \mathcal{P}(A)$ returns all associations which are equal to a given association.

Similar to subsetting, the definition of a model remains unchanged except for the adaptation of the function *query*. The set of links has to be expanded by the links of all equal associations. But, since equal associations can be more general than the association at which the query is applied, an additional check has to ensure that all resulting objects are type compliant.

Definition 2g: Model with redefinition, ...

Definition 2f remains unchanged except for the definition of *query* which is replaced by the following.

$$query_M(o, a, second) = \{o_l \in O \mid (\exists l \in$$

$$\bigcup_{a' \in \text{subAssociations}^*(a)} \text{links}_M(a') \cup \bigcup_{a'' \in \text{equalAssociations}^+(a)} \text{links}_M(a'' \mid \\ (\text{firstObject}(l) = o) \wedge (\text{secondObject}(l) = o_l) \\ \wedge (\text{class}(o_l) \in \text{subClasses}^*(\text{secondEnd}(a))))\}$$

$$\text{query}_M(o, a, \text{first}) = \{o_l \in O \mid (\exists l \in \\ \bigcup_{a' \in \text{subAssociations}^*(a)} \text{links}_M(a') \cup \bigcup_{a'' \in \text{equalAssociations}^+(a)} \text{links}_M(a'' \mid \\ (\text{firstObject}(l) = o_l) \wedge (\text{secondObject}(l) = o) \\ \wedge (\text{class}(o_l) \in \text{subClasses}^*(\text{firstEnd}(a))))\}$$

The modification of the function *query*, again, affects the constraints for multiplicity and uniqueness. Additional to the side effects of subsetting, there are side effects considering the lower and upper bounds of equal association ends. These side effects only affect objects of the more special association's classes regardless by which of both equal associations they are linked. Objects of the more general association's classes are not affected at all. In fact, for the first case, the smaller one of the upper bounds of two equal association ends defines the effective upper bound of both ends. Accordingly, the effective lower bound of both ends is defined by the greater one of both ends' lower bounds. Considering uniqueness, the definition of an association end as unique implies all its equal ends to be unique as well.

In the following, we introduce the constraint as mentioned before. In general, the classes of a link's objects can either be exactly the same classes as connected by its association or subclasses of those classes. However, this is only true, if there is no redefinition involved. A redefinition of an association end implies, that there is no link in the redefined association, which connects an object of the redefinition's opposite class with an object of the redefined end's class.

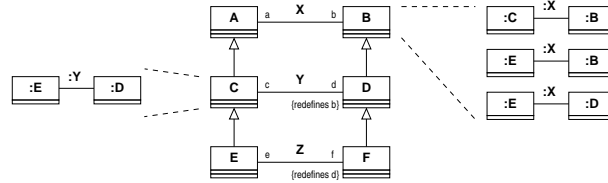


Fig. 11. Examples of forbidden links due to redefinition

Fig. 11 gives a simple example to illustrate this relation. The figure shows a small, artificial metamodel together with forbidden links for the association *Y*, due to the redefinition of *d*, on the left side and for the association *X*, due to the redefinition of *b* and to the transitive redefinition of *d*, on the right side. Due

to the redefinition of d , an object of E can only be linked with an object of F , which is automatically true for Z but not for Y . Thus, there has to be a constraint that prevents such a linkage. Furthermore, the creation of the redefined association could be restricted even more, if the redefining association end is derived. A derivation rule of the redefining association end must not be violated by links of the redefined association end as well. But, since the consideration of arbitrary restrictions would demand a compliant formalization of the used constraint language, this is out-of-scope of this article

Constraint 6: Redefinition

Let MM be a metamodel.

$$\begin{aligned}
& \forall M \in Models_{MM} \mid (\forall l \in L \mid \\
& \quad \neg((class(firstObject(l)) = firstEnd(a)) \\
& \quad \quad \wedge(class(secondObject(l)) = secondEnd(a))) \\
& \Rightarrow (\neg \exists(a', second) \in redefinitions^+(association(l), second) \mid \\
& \quad \quad firstEnd(a') = class(firstObject(l))) \\
& \quad \wedge(\neg \exists(a', first) \in redefinitions^+(association(l), second) \mid \\
& \quad \quad secondEnd(a') = class(firstObject(l))) \\
& \quad \wedge(\neg \exists(a', first) \in redefinitions^+(association(l), first) \mid \\
& \quad \quad secondEnd(a') = class(secondObject(l))) \\
& \quad \wedge(\neg \exists(a', second) \in redefinitions^+(association(l), first) \mid \\
& \quad \quad firstEnd(a') = class(secondObject(l))))
\end{aligned}$$

Beside the additional constraint, the definition of a morphism has to be expanded. Due to the concept of equal associations, a link can also be mapped onto a link of an equal and more special association which is similar to subsetting and additionally onto a link of an equal and more general association as well. Since, a more general but equal association can contain links that are not compliant to the more special association one might expect a kind of type checking. But such checks are not necessary, since the compliance to the association's types is already demanded by (7), (8) and (13).

Definition 4g: Morphism with redefinition and subsetting

Definition 4f remains unchanged except for formula (21) which is replaced by formula (23).

$$\begin{aligned}
& \forall l \in L_1 \mid \tag{23} \\
& \quad association_2(h_L(l)) \in (subAssociations^*(association_1(l)) \\
& \quad \cup equalAssociations^*(association_1(l)))
\end{aligned}$$

9 Conclusion

We presented a set theoretic formalization of graph transformations based on the sophisticated association concept of UML/MOF 2. This formalization describes the effects of new association features like subsetting and redefinition on the application of transformations. Those new features add new aspects to the application of graph transformations which result in a different matching behaviour compared to the application of former versions of UML/MOF as graph schema language. Inside an integrated approach of MOF 2.0, OCL 2.0 and graph transformations the combination of graph transformations and MOF 2.0 provides a very promising approach for the purpose of language engineering. The sophisticated association concept of MOF 2.0 provides the base for the modularization of huge language specifications. By adapting these concepts to graph transformations, the base for modularization is also provided for the specification of a language's dynamic semantics.

The implementation of the combined approach by the MOFLON framework provides powerful support for the purpose of language engineering. Based on the description of a domain-specific language's abstract syntax, static and dynamic semantics, MOFLON generates a repository implementation for the manipulation, evaluation and transformation of domain-specific languages. The presented formalization closes a gap in the formal specification of MOFLON's specification language MOSL. In future work we will extend this formalization to cover also the omitted features like attributes, etc. Furthermore, the formalization of transformation and derivation as well as the integration of transformations based on OCL queries are challenging tasks.

References

1. Object Management Group. Unified Modeling Language: Superstructure; 2007. Formal/2007-02-05.
2. Amelunxen C, Königs A, Röttschke T, Schürr A. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink A, Warmer J, editors. Model Driven Architecture - Foundations and Applications: Second European Conference. vol. 4066 of Lecture Notes in Computer Science (LNCS). Heidelberg: Springer Verlag; 2006. p. 361–375.
3. MOFLON Homepage; 2007. <http://www.mofflon.org>.
4. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification; 2006. Formal/06-01-01.
5. Object Management Group. Object Constraint Language; 2006. Formal/06-05-01.
6. Zündorf A. Rigorous Object Oriented Software Development. University of Paderborn; 2001. Habilitation Thesis.
7. FUJABA Homepage; 2007. <http://www.fujaba.de>.
8. Schürr A. Programmed Graph Replacement Systems. In: Rozenberg G, editor. Handbook of graph grammars and computing by graph transformation: vol. 1: foundations. vol. 1. Singapur: World Scientific; 1997. p. 479–546.
9. Amelunxen C, Königs A, Röttschke T, Schürr A. MOSL: Composing a Visual Language for a Metamodeling Framework. In: Proc. of the IEEE Symposium on Visual Languages and Human Centered Computing; 2006. .

10. Object Management Group. Unified Modeling Language: Infrastructure; 2007. Formal/07-02-06.
11. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification; 2007. Ptc/07-07-07.
12. France R, Evans A, Lano K. The UML as a Formal Modeling Notation. In: Kilov H, Rumpe B, Simmonds I, editors. Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics. Technische Universität München, TUM-I9737; 1997. p. 75–81. Available from: citeseer.ist.psu.edu/france97uml.html.
13. Shroff M, France R. Towards a Formalization of UML Class Structures in Z. In: Proc. 21st International Computer Software and Applications Conference (COMP-SAC 1997); 1997. p. 646. Available from: citeseer.ist.psu.edu/shroff97towards.html.
14. Kim SK, Carrington D. Formalizing the UML Class Diagram Using Object-Z. In: Proc. 2nd International Conference on UML: UML'99. Springer Verlag; 1999. p. 83–98.
15. Broy M, Crane ML, Dingel J, Hartman A, Rumpe B, Selic B. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In: Kühne T, editor. MoDELS 2006 Workshops. vol. 4364 of LNCS. Springer; 2006. p. 318–323.
16. Rumbaugh J. Relations as Semantic Constructs in an Object-Oriented Language. In: Proc. of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87). Orlando, Florida, USA: ACM Press; 1987. p. 466–481.
17. Stevens P. On Associations in the Unified Modeling Language. In: Gogolla M, Kobryn C, editors. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference. Toronto, Canada: Springer Verlag; 2001. p. 361–375.
18. Diskin Z, Dingel J. Mappings, maps and tables: a formal semantics for UML2 associations. In: Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006). Springer Verlag; 2006. p. 230–244.
19. Alanen M, Porres I. Basic Operations over Models Containing Subset and Union Properties. In: Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006). Springer Verlag; 2006. p. 469–483.
20. Czarnecki, Helsen. Classification Of Model Transformation Approaches. In: Proc. Generative Techniques in the context of Model Driven Architecture; 2003. <http://www.softmetaware.com/oopsla2003/czarnecki.pdf>.
21. Rozenberg G, editor. Handbook of graph grammars and computing by graph transformation: vol. 1: foundations. vol. 1. Singapur: World Scientific; 1997.
22. Ehrig H, Ehrig K, Prange U, Taentzer G. Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer; 2006. Available from: <http://www.springer.com/3-540-31187-4>.
23. Agrawal A, Karsai G, Shi F. Graph Transformations on Domain-Specific Models. Vanderbilt University; 2003. ISIS-03-403.
24. Taentzer G, Ermel C, Rudolf M. The AGG-Approach: Language and Tool Environment. In: Ehrig H, Engels G, Kreowski HJ, Rozenberg G, editors. Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools. World Scientific; 1999. p. 551–603.
25. Schürr A, Winter A, Zündorf A. PROGRES: Language and Environment. In: Ehrig H, Engels G, Kreowski H, Rozenberg G, editors. Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools. vol. 2. Singapur: World Scientific; 1999. p. 487–550.