Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 109 (2004) 31–42

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Realizing UML Metamodel Transformations with AGG

Fabian Büttner[1] Martin Gogolla[2]

Dept. of Computer Science
University of Bremen
Germany

Abstract

In this paper, we work out equivalence transformations on the UML metamodel as concrete graph transformations implemented in the AGG tool. We consider two examples for manipulating the static structure of a UML model, namely the transformation of an association class into a ternary association and the transformation of a ternary association into three binary associations. We discuss technical details and pros and cons of the presented approach and shortly put out work into the context of the MDA.

Keywords: AGG, Graph Transformations, Model Driven Architecture, Platform Independent Model, Platform Specific Model, UML.

1 Introduction

The Unified Modeling Language [16] has become widely accepted as a standard for modeling and documenting software systems. UML comprises a number of diagram forms used to describe particular aspects of software artifacts. The diagram forms can be divided depending on whether they are intended to describe structural or behavioral aspects. From a fundamental point of view, a central ingredient of UML is the Object Constraint Language (OCL) (see [20,3,18]), which is close to first order predicate calculus, but which claims to be easy applicable by the 'average business or system modeler'.

[1] Email: green@tzi.de
[2] Email: gogolla@tzi.de

doi:10.1016/j.entcs.2004.02.054

Depending on the target platform, not all constructs available in UML may be directly implementable. The Model Driven Architecture (MDA, see [15,2]) addresses this problem by introducing explicit 'platform independent models' (PIMs) and 'platform specific models' (PSMs). Ideally, PSMs should be automatically derivable from PIMs. Such PIM-to-PSM transformations could be expressed and implemented in terms of (attributed) graph transformations like introduced in [12,5].

In this paper we elaborate equivalence transformations from [GR01] on the UML metamodel as graph transformations using the AGG (Attributed Graph Grammars) tool [11,5,19,1]. The transformations modify the static structure of a model: One transformation replaces association classes by ternary associations, the other replaces ternary associations by binary associations. Because neither association classes nor ternary associations are present in, for example, imperative programming languages like Java or C++, we believe that static structure transformations like ours may be a common part of PIM-to-PSM transformations. The AGG tool is freely available Java software and offers a graphical user interface, as well as a library interface and could possibly assist developers in these kind of transformations. [9] present a meta-programmable transformation tool (GReAT) which possibly could be used as well to realize our transformations.

This paper is structured as follows: Sect. 2 shows how we handle the transformation of association classes into ternary associations. Section 3 shows how we treat the transformation of ternary associations into binary associations. We conclude with some general remarks and observations in Sect. 4.

2 Replacing Association Classes

Figure 1 shows the general idea of transforming an association class to a ternary association: The association class A on the left-hand side is split up into a class A and a ternary association RA on the right-hand side. An example of applying this transformation is shown in Fig. 2. Because of the special semantics of multiplicities in UML (see [16,6]), the multiplicity constraints on the left-hand side cannot be carried over to the right-hand side. Instead, they must be reformulated as OCL class invariants ($multiplicity_1$ and $multiplicity_2$). We further need to ensure that an instance of A is not shared among distinct pairs of instances of C1 and C2 (invariant *unshared*) and that each pair (c1, c2) is connected with at most one A instance (expressed by the multiplicity 0..1 on the *a* role).

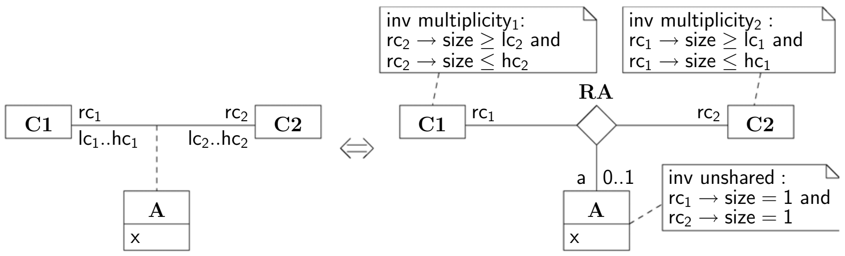

Fig. 1. Replacing an association class by a ternary association

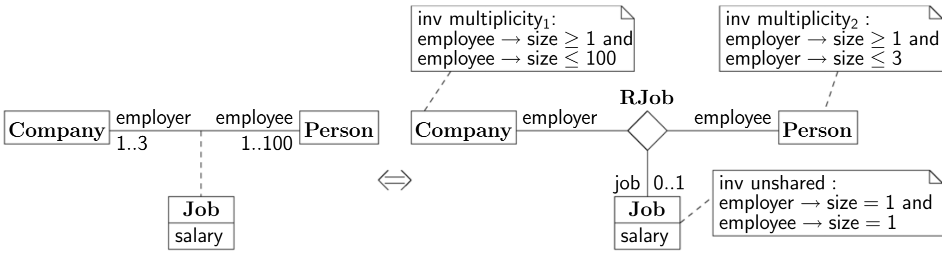

Fig. 2. Example for a transformation

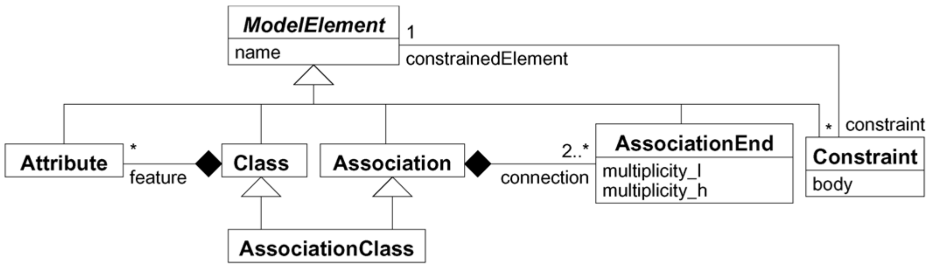

Fig. 3. Excerpt of the UML metamodel

2.1 Implementation as an AGG graph transformation rule

In order to implement this transformation as a graph transformation, we need to formulate it on the meta level. For the sake of simplicity, we use only an excerpt of the UML 1.5 metamodel, as shown in Fig. 3. The considerations presented carry over to UML 2.0 in an analogous way. Essentially, an association is built up from at least two association ends, and a class is built up from attributes. All model elements have a name. Association ends have a lower and an upper multiplicity bound. Multiplicity bounds are represented by integer valued attributes in our simplified metamodel. We encode an unlimited upper bound (‘*’) as -1 . Constraints can be attached to all model elements. The body attribute of a constraint element is a boolean OCL expression which must yield *true* in all valid states.

AGG graph transformation rules consist of a left-hand and a right-hand

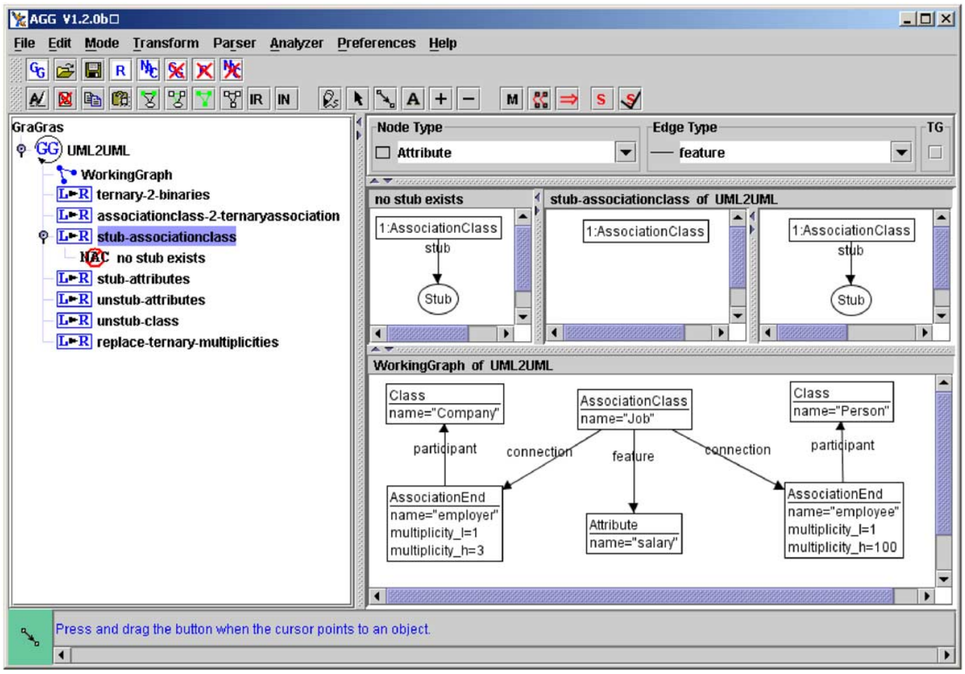

Fig. 4. AGG system

side graph, a mapping morphism between nodes (and edges) on both sides, and a set of ‘negative application conditions’ (NACs). A screenshot of the AGG system is shown in Fig. 4: On the left-hand side, the tree view of the project shows the working graph and the rules which are present. In the upper right, the selected rule can be found, and in the lower right, the actual working graph is shown. Rules having a NAC are displayed by three graphs (NAC, left-hand side, right-hand side), rules without a NAC are displayed by two graphs. Numbers in front of node labels represent the morphism of the rule . The working graph in Fig. 4 corresponds to the left-hand side of the example in Fig. 2.

The main transformation rule named ‘associationclass-2-ternaryassociation’ is shown in Fig. 5. Nodes representing metaobjects are depicted as boxes. ‘Stubs’ (to be explained below), which are helper nodes that do not represent metaobjects, are depicted as ovals. To apply the rule, the AssociationClass node from left-hand side must be matched in the working graph. This can be done through AGGs ‘map’ mode, available through the context menu. The rule is actually executed by clicking the ‘⇒’ button in the toolbar.

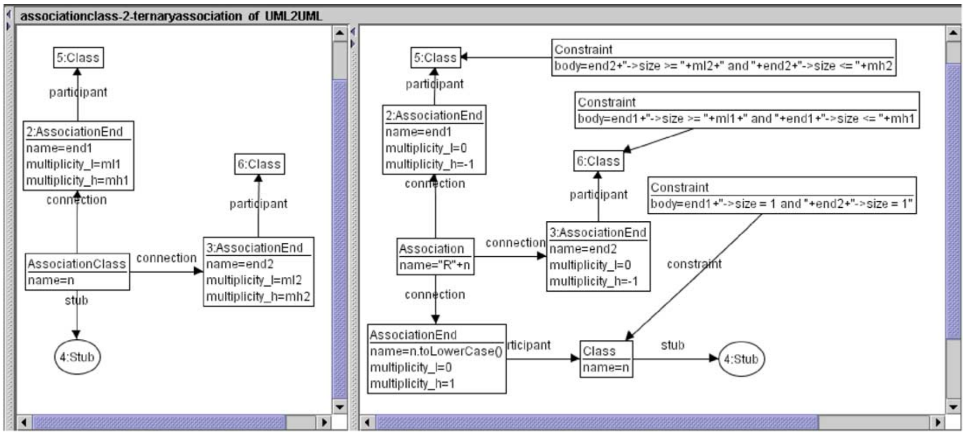

Fig. 5. Main graph transformation rule

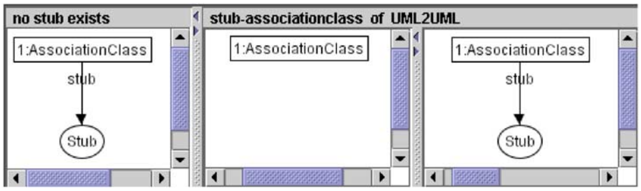

Fig. 6. Create a stub for the class part

2.2 *Stubbing, helper nodes, and unstubbing*

Where do the 'stub' nodes come from? As the metaclass AssociationClass is a subclass of both, Association and Class, it inherits a set of features (Attributes) from Class and a set of connections (AssociationEnds) from Association. The transformation in Fig. 1 replaces an association class by a class and an association. Thus, speaking in terms of graph transformations, one node (the AssociationClass node) must be deleted and two nodes (the Association node and the Class node) must be created. During this transformation, the Attributes connected to the AssociationClass node must be disconnected and attached to the newly created Class node. Because this cannot be directly expressed by an AGG graph transformation rule, we have to perform some preparations before applying the rule. First, we create a Stub node and attach it to the AssociationClass. This rule 'stub-associationclass' is depicted in Fig. 6. The negative application condition 'no stub exists' ensures that no AssociationClass node has more than one stub. After attaching the stub, all Attributes can be moved from the AssociationClass to the stub. The corresponding rule 'stub-attributes' is shown in Fig. 7.

Having moved all attributes to the stub by repeatedly applying 'stub-attributes', we can apply the main transformation rule 'associationclass-2-

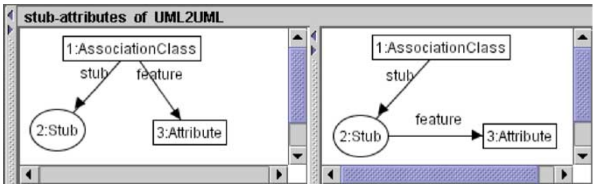

Fig. 7. Move one attribute from an AssociationClass to the stub

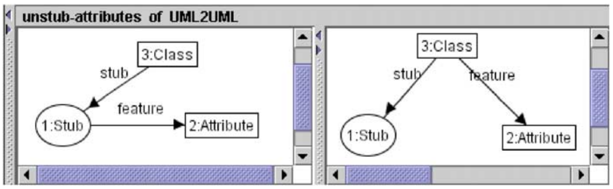

Fig. 8. Move an attribute from the stub to the class

ternaryassociation' (Fig. 5). The rule attaches both AssociationEnd nodes from the left-hand side to the newly created Association node and attaches the Stub node to the newly created Class node. Finally, the Class node is connected to the Association by a new AssociationEnd node - of course, since we transform a binary association to a ternary association. The three OCL constraints from Fig. 1 are inserted as Constraint nodes where neccessary. The constraint texts are constructed by string concatenation and assigned to the 'body' attributes of the Constraint nodes.

Finally, after applying the rule, the attributes must be 'unstubbed' by applying repeatedly the rules 'unstub-attributes' (Fig. 8) and at last 'unstub-class' (Fig. 9).

Summarized, we perform the whole transformation from Fig. 1 by applying the rules in the following sequence:

stub-associationclass, stub-attributes$*$,
associationclass-2-ternaryassociation,
unstub-attributes$*$, unstub-class

It is worth to mention that 'associationclass-2-ternaryassociation' cannot be applied if there are still Attributes connected to the AssociationClass node to be replaced. This is due to the fact that AGG does not allow dangling edges per default. Deleting an AssociationClass node which still has Attributes would leave dangling 'feature' links and is therefore not permitted.

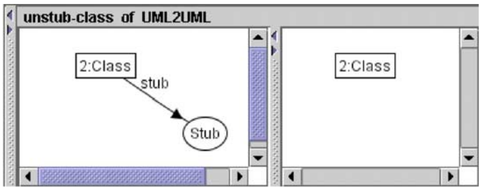

Fig. 9. Remove the stub from the class

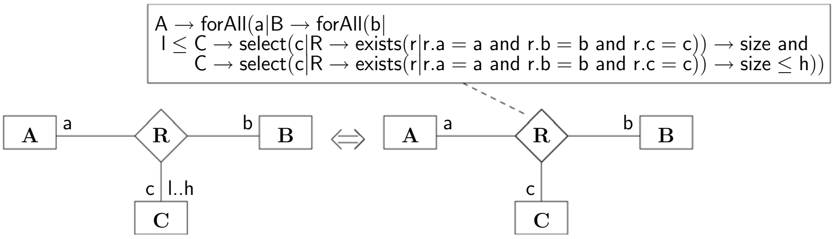
A → forAll(a|B → forAll(b|
l ≤ C → select(c|R → exists(r|r.a = a and r.b = b and r.c = c)) → size and
C → select(c|R → exists(r|r.a = a and r.b = b and r.c = c)) → size ≤ h))
A
a
R
b
B
c
l..h
C
⟺
A
a
R
b
B
c
C

Fig. 10. Replacing multiplicities in ternary associations by OCL constraints

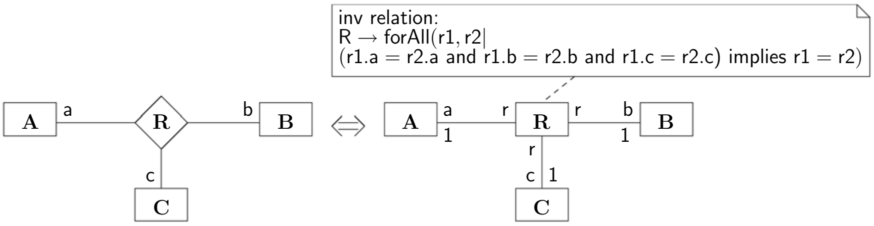
inv relation:
R → forAll(r1, r2|
(r1.a = r2.a and r1.b = r2.b and r1.c = r2.c) implies r1 = r2)
A
a
R
b
B
c
C
⟺
A
a
1
r
R
r
b
1
B
r
c
1
C

Fig. 11. Replacing a ternary association by three binary associations

3 Replacing Ternary Associations

Two equivalence rules are involved in the replacement of a ternary association by three binary associations: In the first step, we replace restricting multiplicities by explicit OCL constraints. The general idea is depicted in Fig. 10. This equivalence must be used up to three times, then the second equivalence rule, depicted in Fig. 11, can be applied to replace the ternary association (now without multiplicity restrictions) by three binary associations: The association R on the left-hand side is replaced by a class R and three new associations on the right-hand side. The additional invariant *relation* is needed to ensure that no two distinct R objects have the same (A,B,C) arms. This behavior of associations is mandatory in UML 1.5 and will be default in UML 2.0.

The corresponding AGG rules 'replace-ternary-multiplicities' and 'ternary-2-binaries' are depicted in Fig. 12 and 13. The sequence in which the rules must be applied is as follows:

replace-ternary-multiplicities*,ternary-2-binaries

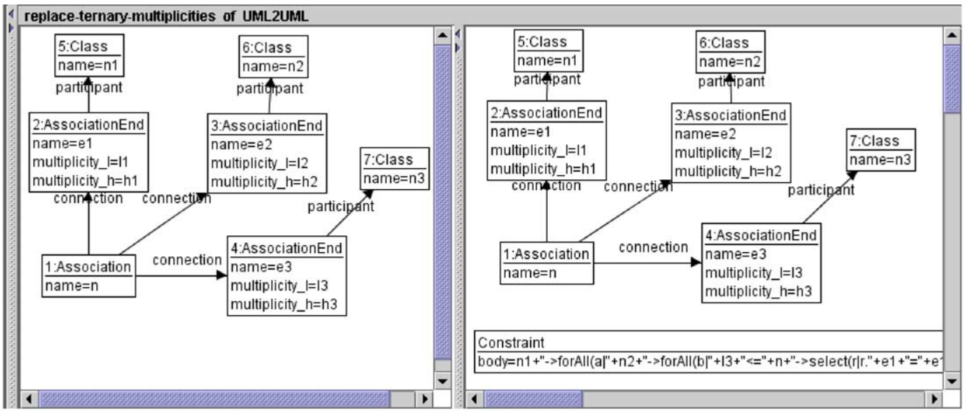

Fig. 12. Graph transformation rule for replacing multiplicities on ternary associations

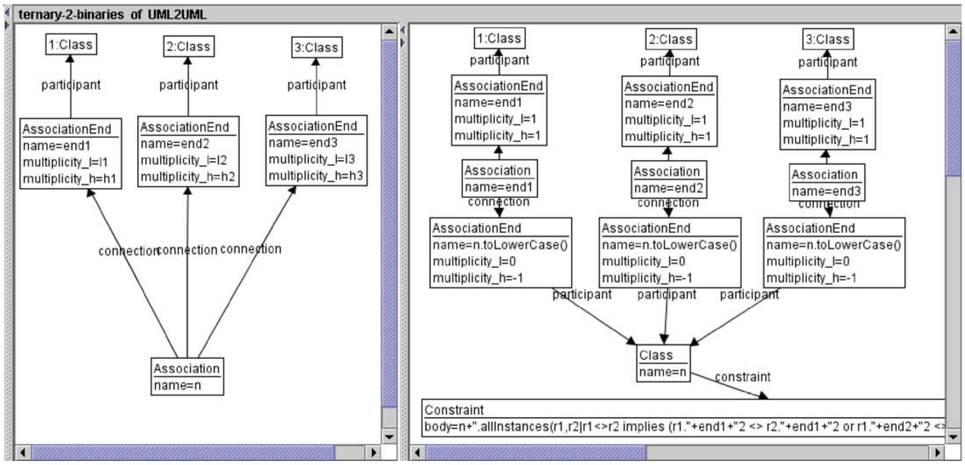

Fig. 13. Graph transformation rule for replacing ternary associations

To apply ‘replace-ternary-multiplicities’, the AssociationEnd node having name $e3$ in the left-hand side of the rule must be matched with an AssociationEnd node in the working graph which has restricted multiplicities.

If we combine the example presented in this section with the first one, we can successively replace association classes by binary associations. In this case, ‘replace-ternary-multiplicities’ must be applied one time, to replace the 0..1 multiplicity introduced by the right-hand side of Fig. 5. Fig. 14 shows the example from Fig. 4 after applying all transformations.

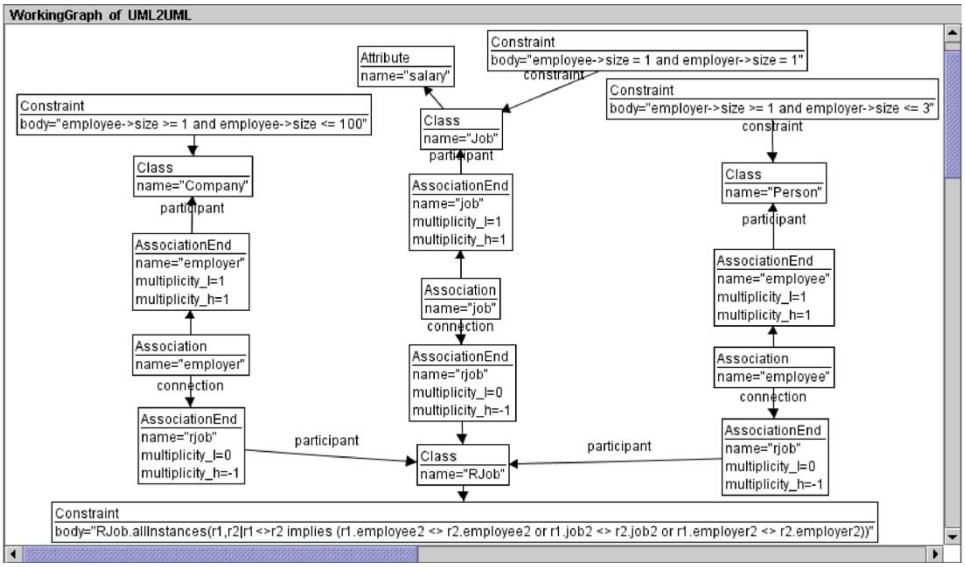
WorkingGraph of UML2UML
Attribute name="salary"
Constraint body="employee->size = 1 and employer->size = 1"
constraint
Constraint body="employer->size >= 1 and employer->size <= 3"
constraint
Constraint body="employee->size >= 1 and employee->size <= 100"
Class name="Job"
participant
Class name="Company"
participant
AssociationEnd name="job" multiplicity_l=1 multiplicity_h=1
Class name="Person"
participant
AssociationEnd name="employer" multiplicity_l=1 multiplicity_h=1
Association name="job"
connection
AssociationEnd name="employee" multiplicity_l=1 multiplicity_h=1
Association name="employer"
connection
AssociationEnd name="rjob" multiplicity_l=0 multiplicity_h=-1
Association name="employee"
connection
AssociationEnd name="rjob" multiplicity_l=0 multiplicity_h=-1
participant
Class name="RJob"
participant
AssociationEnd name="rjob" multiplicity_l=0 multiplicity_h=-1
Constraint body="RJob.allInstances(r1,r2|r1<>r2 implies (r1.employee2 <> r2.employee2 or r1.job2 <> r2.job2 or r1.employer2 <> r2.employer2))"

Fig. 14. Example after applying all transformations

4 Remarks and Future Work

In this paper we have studied the applicability of graph transformations on the UML metamodel. Our metamodel transformations, which represent equivalence rules, may be used in principle in two directions: (A) From the language with 'higher' concepts to the language with 'lower' concepts, e.g., in the direction from left to right in Fig. 1 and (B) from the language with 'lower' concepts to the language with 'higher' concepts, e.g., in the direction from right to left in Fig. 1. In the context of the MDA, the case (A) on which we have concentrated here may be regarded as a PIM-to-PSM transformation, the case (B) may be seen as a PSM-to-PIM transformation.

However, we observe that a sketched, simple transformation like the one in Fig. 1 becomes more involved when being realized in a concrete graph transformation tool like AGG. The single conceptual transformation must be implemented with an additional node kind, i.e., the stub nodes, and with additional transformations like 'stub-associationclass'. It would be nice if graph transformation tools being based on the algebraic approach would able to hide these additional details from the user of the transformation by letting the user have the impression that a complete transformation unit (with additional node kinds and internal rules applied in a certain order) behaves like a single transformation, as in [10].

In our approach, we represent an actual OCL constraint in a naive way as a single String attribute. Thus, without further consideration, constraints can become ill-formed when applying transformation rules. While pre-existing OCL expressions still remain well-formed after applying our first transformation, they may become ill-formed when applying the second one. For example, the OCL expression $a.b$ (a instance of A) is well-formed in the left-hand side in Fig 11, but becomes ill-formed in the right-hand side. Actually, it should have been replaced by $a.r.b$ in the right-hand side. At the current stage, such corrections must be performed by hand after applying certain transformations.

In the upcoming OCL 2.0/UML 2.0 specifications, there will be an OCL metamodel. Using the UML 2.0 metamodel, OCL constraints can be expressed by graphs, and transformations of OCL constraints can be expressed by graph transformations. Thus, the above mentioned problems with OCL expressions can be solved. However, it remains to be seen if model transformations can be still used in an intuitive way when OCL constraints occur as graphs in it.

Nevertheless we think that our approach is already applicable in real world PIM-to-PSM transformations. Tools like the Dresden OCL Compiler [7] may be used to implement OCL constraints generated during the transformation in an imperative programming language. Another approach explained in [17] uses Aspect-Oriented Programming (AOP) to generate 'aspects' which can be weaved with user generated code to validate OCL constraints at runtime.

How does our approach relate to existing work in the area of MDA? The OMG has issued a request for proposals [13] for queries, views, and transformations (QVT) over (MOF 2.0) models. The first revised QVT submission of DSTC, IBM, and CBOP [4] uses a declaritive specification language for transformation, based on a transformation metamodel. Although they propose a textual representation of transformations, they explicitly mention that they are in favour of having many concrete syntaxes, including graphical ones. We have demonstrated that the kind of transformations used in this paper can be very intuitivly expressed in terms of graph transformations. It remains to be seen how a graphical syntax may be mapped to the transformation model of [4].

In order to evaluate the integration of our approach with common UML case tools like Rational Rose, ArgoUML or Poseidon, we are currently working on combining AGG with the Java Metadata Interface (JMI, [8]). This way, we can apply graph transformations directly on UML models. Speaking in technical terms, we define a mapping which allows metamodel instances (represented by JMI) to be treated as AGG graphs. UML models can either be exported and imported from the case tools as XMI files [14] or, in the case of ArgoUML, can be modified in-place. The latter case works because ArgoUML uses a JMI compliant metadata repository. We imagine to include an extendible plug-in into ArgoUML from which several transformations (implemented as graph transformation units) can be applied.

Thanks to the anonymous referees for their helpful comments.

References

[1] The AGG website. http://tfs.cs.tu-berlin.de/agg, November 2003.

[2] Wim Bast Anneke Kleppe, Jos Warmer. *MDA Explained. The Practice and Promise of the Model Driven Architecture*. Addison-Wesley, 2003.

[3] Tony Clark and Jos Warmer, editors. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.

[4] DSTC and IBM and CBOP. MOF query / views / transformations. first revised submission, 2003.

[5] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, Singapore, 1999.

[6] Gonzalo Génova, Juan Llorens, and Paloma Martínez. Semantics of the minimum multiplicity in ternary associations in UML. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 329–341. Springer, 2001.

[GR01] Martin Gogolla and Mark Richters. Expressing UML Class Diagrams Properties with OCL. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 86–115. Springer, Berlin, LNCS 2263, 2001.

[7] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. *Science of Computer Programming*, 44(1):51–69, July 2002.

[8] Java Metadata Interface (JMI) specification, March 2002. Java Specification Request #40 - Specification Lead by Unisys Corporation. http://jcp.org/en/jsr/detail?id=40.

[9] Gabor Karsai, Aditya Agarwal, and Akos Ledeczi. A metamodel-driven mda process and its tools. In Jean Bezivin and Martin Gogolla, editors, *Proceedings 2nd UML Workshop in Software Model Engineering (WiSME'2003)*, 2003. http://www.metamodel.com/wisme-2003/.

[10] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[11] Michael Löwe and Martin Beyer. AGG — an implementation of algebraic graph rewriting. In *Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 451–456. Springer-Verlag, 1993.

[12] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, New York, 1993.

[13] MOF 2.0 query / views / transformations rfp. Technical report, Object Management Group, 2002.

[14] XML Metadata Interchange (XMI) Specification, v1.2. Technical report, Object Management Group, March 2002.

[15] Model Driven Architecture website. http://www.omg.org/mda, March 2003.

[16] Unified Modeling Language Specification version 1.5. Technical report, Object Management Group, March 2003.

[17] Mark Richters and Martin Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In Omar Aldawud, Mohamed Kande, Grady Booch, Bill Harrison, Dominik Stein, Jeff Gray, Siobhan Clarke, Aida Zakaria, Peri Tarr, and Faisal Akkawi, editors, *Proc. UML'2003 Workshop Aspect-Oriented Software Development with UML*. Illinois Institute of Technology, Department of Computer Science, `http://www.cs.iit.edu/~oaldawud/AOM/index.htm`, 2003.

[18] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[19] Gabriele Taentzer. AGG: A tool enviroment for algebraic graph transformation. In M. Nagl, A. Schürr, and M. Münch, editors, *Applications of Graph Transformation with Industrial Relevance: International Workshop, AGTIVE'99, Kerkrade, The Netherlands*, volume 1779, pages 481–488. Springer-Verlag Heidelberg, 2000.

[20] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.