Transformation-Based Structure Model Evolution

Fabian Büttner

University of Bremen, Computer Science Department, Database Systems Group

Abstract. This paper summarizes an approach to support evolution of software models by means of a transformation catalogue. These transformations treat UML class diagram models, OCL constraints, and existing states of the models in a coherent and consistent way. By implementing the catalogue in the UML tool USE, we effectively support developers in an iterative and incremental process for model development.

1 Introduction

Models have become more and more important in software development. The Model Driven Architecture (MDA) currently addressed by a large number of people, research groups, and tools documents this emphasis on having models as central artifacts in software engineering.

Typically, a model undergoes many changes in its lifetime, in the same way as an end product (software) does. If it does not, the model might have been a throw away artifact which is no longer connected to the software developed from it. To prevent having throw-away models, we believe that one needs strong tool support for the evolution of models which allows us to continuously develop and refine existing models.

In our work, we formulate a catalogue of transformations to static structure models which are given by UML class diagrams and attached OCL constraints. Changing static structure model elements may render existing OCL expressions invalid. Thus, each transformation has to be accomplished by a number of change rules for existing OCL expressions. Furthermore, there typically exist states (object diagrams) of the model as well. Such states may exist as analysis artifacts, for example to refine use case descriptions. It is therefore important to investigate each model transformation w.r.t. its capability to represent existing states under a changed model. Thus, we have to distinguish between state preserving model transformations and those which are not, or only partly state preserving.

In our work, each model transformation is organized as a set of *transformation steps*. Each of these steps describes a single change to a model in terms of its metamodel representation, providing an operational characterization of the transformation. Furthermore, each transformation is classified w.r.t. the context under which it is state preserving. If a transformation is state preserving, a corresponding state transformation is defined.

On the tool side we are currently implementing the complete catalogue as an extension of the USE (“UML Specification Environment”) tool. This extended version of

USE allows us to interactively change a model while keeping OCL constraints and existing state in sync. An evolution browser documents all changes through the life cycle of a model.

There is related work in many areas: refactorings and design patterns aim to provide automatic changes and solutions to typical design problems to programmers. For models, there are several approaches to employ graph transformations to define model transformations. Other transformation languages follow a relation-based approach. There also exist a number of model transformation language proposals as a respond to the QVT RFP of the OMG. Due to the paper format, we consequently do not include any references except our own previous work. The final work will of course include a detailed discussion of related work.

We have already studied a couple of transformations to UML class diagrams in [BG04b,BBG]. Particular aspects of class diagram semantics are discussed in [BG04a].

This paper is structured as follows: Sect. 2 shows how the transformation catalogue is structured and how the transformations are presented. It particularly picks one transformation from our catalogue and goes a little bit deeper into details. Section 3 shortly discusses the state of our work.

2 The Transformation Catalogue

The catalogue under development will contain a selection of carefully chosen transformations that are essential to the evolution of structure models. This includes changing the multiplicity of association ends, changes to the generalization hierarchy, moving attributes along associations, splitting and joining classes, and other transformations. The idea is that as many as possible of the changes to be made during the lifetime of a model can be made consistently and automatically within the USE tool (i.e., without manually rewriting existing OCL constraints or throwing away existing states). All transformations of our catalogue are (or will be) implemented in the aforementioned extension to the USE tool.

2.1 Structure of transformations

Each transformation is motivated by a single change to a class diagram. Each of these changes is realized by a number of transformation steps. These steps are provided mainly by UML collaborations showing how an instance of the UML metamodel (for class diagrams) and the OCL expression trees (given as an instance of the OCL metamodel) have to be modified. A step control sequence states the order in which the states have to be applied. If a transformation is state preserving, meaning that states of the former model can be represented under the new model, a state transformation is included.

2.2 Example

In the following, we briefly introduce one transformation “replace generalization by composition” taken from our catalogue which is commonly applied in the evolution of a static structure model (described by a class diagram). Due to lack of space, only certain aspects are illustrated. The main change of this transformation is depicted in Fig. 1 and is motivated by the fact, that generalization is commonly introduced inadequately in a model in the sense that it does not model the intended "real world" domain. Typically, this can be recognized as inadequate object identities. Consider the class di-

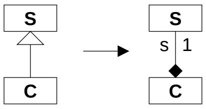

Fig. 1. Replace generalization by composition

agram in Fig. 2. In several cases, this is not good model: a student can never become a professor, because a (UML) object cannot change its class during its lifetime. A much better design is to represent being a professor or a student as *roles* of persons. This can be achieved by replacing the two generalizations by composition.

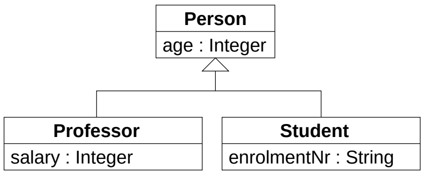

Fig. 2. Motivation: Roles confused with generalization

However, if the model in which these generalizations are embedded contains OCL expressions, simply replacing the generalization may render these expressions invalid. In general, one has to change some constraints as well.

This is exemplified in Fig. 3. The generalization between the classes Student and Person is replaced by a composition in the left-hand side of the class diagram. The two OCL invariant constraints attached to the class diagram are changed as well. In the first invariant (requiring students to be at least 18 years old), the query to the age attribute (in the forAll clause) must be delegated to the person object which is now related to each student. This is necessary as 'age' is not longer present in the full class descriptor of the Student class. In the second invariant (stating students have to study in their home town), the association from a student to its town must now be delegated, for the same reason.

In general, OCL expression have to be modified not only at places where features (attributes, navigable associations, operations) of a former superclass are used. One also has to deal with implicit and explicit applications of the subtype substitution principle (casts). Among other places, this can happen in invocations of operations. Figure 4 illustrates this. The left-hand side class diagram defines four operations which check for equality. The operations eq2 and eq3 require an implicit type conversion from C to

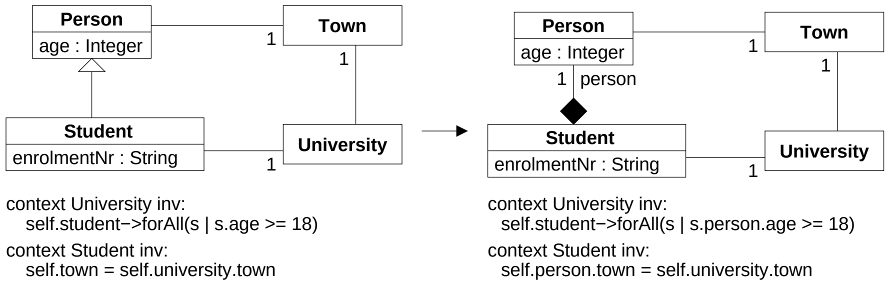

Fig. 3. Replacing generalization by composition: Example

S and use the equality defined by C::=(C). When transforming the expressions used to define eq[1-4], each type conversion from C to S must be replaced by a navigation from C to S. Similarly, when transforming the four invariant constraints in Fig. 4, even more conversions are implicit to the constraint formulas. All these replacements are necessary to ensure that each valid system state of the former class diagram can be represented as a valid system state of the modified class diagram (all invariants still hold).

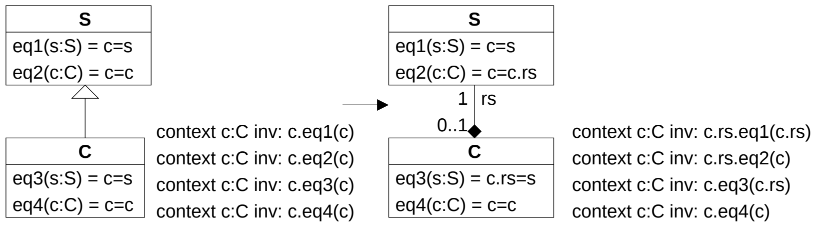

Fig. 4. Example showing several changes induced by subtype polymorphism

To provide a short insight, one of the transformation steps that realizes the described transformation is depicted in the collaboration diagram in Fig. 5. This step named 'gentocomp.navigation' deals with navigations going out from the former superclass. The composition *toParent* that will replace the generalization is already present in this diagram (it is created by another step which has to be applied earlier). In this step, the argument *obj* of a navigation expression *nav* is delegated through another, newly created navigation expression which navigates from the former subclass to the former superclass. This step is applicable if the object expression *obj* belong to to former subclass (or a subclass of it) and the navigation goes out from the former superclass (or a superclass of it). This application conditions are formulated as OCL constraints, enclosed in brackets in Fig. 5.

We do not go deeper into details at this point. The complete description of "replace generalization by composition" deals with representing existing states under the modified class digram (this transformation is state preserving) and includes a lot more consideration w.r.t to which preconditions must be met. It consists of 13 transformation steps (seven for the model and six for existing states).

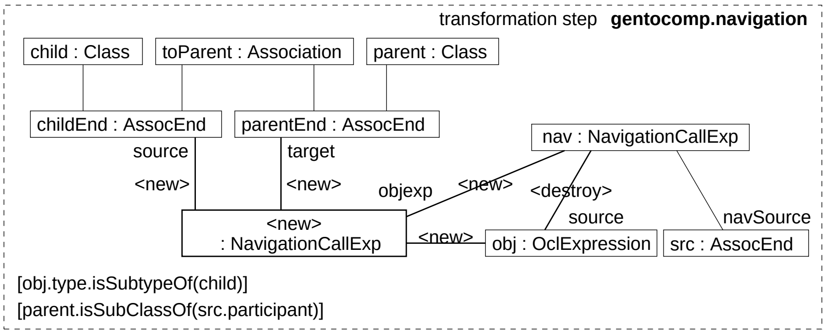

Fig. 5. One transformation step

3 State of Work

At the time of writing, the transformation catalogue is partly implemented in USE. We have gained much feedback on our transformations by experimenting with several models. In particular, the transformations for changing the multiplicities of association end, replacing generalizations by compositions, and moving attributes along associations are already applicable. Implementing the collaborations in Java was really simplified by the help of our 'dynamic dispatcher' [BRLG04] component (a very slim implementation of multi-methods tailored for applying the visitor design pattern). The way the transformations are implemented follows closely the way they are presented in our work, giving some first evidence that the transformations, organized in transformation steps, are sound and consistent. A more formal discussion will be included in our final work.

References

[BBG] Fabian Büttner, Hanna Bauerdick, and Martin Gogolla. Towards transformation of integrity constraints and database states. In Danielle Martin, editor, *Proc. 16th International Conference and Workshop on Database and Expert Systems Applications (DEXA 2005),* to appear. IEEE, Los Alamitos.

[BG04a] Fabian Büttner and Martin Gogolla. On Generalization and Overriding in UML 2.0. In Nuno Jardim Nunes, Bran Selic, Alberto Rodrigues da Silva, and Ambrosio Toval Alvarez, editors, *UML'2004 Modeling Languages and Applications. UML'2004 Satellite Activities.*, pages 67–67. Springer, Berlin, LNCS 3297, 2004.

[BG04b] Fabian Büttner and Martin Gogolla. Realizing UML Metamodel Transformations with AGG. In Reiko Heckel, editor, *Proc. ETAPS Workshop Graph Transformation and Visual Modeling Techniques (GT-VMT'2004)*. Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier, 2004.

[BRLG04] Fabian Büttner, Oliver Radfelder, Arne Lindow, and Martin Gogolla. Digging into the Visitor Pattern. In Frank Maurer and Günther Ruhe, editors, *Proc. IEEE 16th Int. Conf. Software Engineering and Knowlege Engineering (SEKE'2004)*. IEEE, Los Alamitos, 2004.