Towards Transformation of Integrity Constraints and Database States

Fabian Büttner, Hanna Bauerdick, Martin Gogolla
Database Group, Computer Science Department
University of Bremen, D-28334 Bremen, Germany

***Abstract*— This paper discusses integrity constraint evolution, from two perspectives: From the perspective of changing the constraint, and from the perspective of consequently changing an existing database state. The paper concentrates on structural models provided by UML class diagrams and OCL invariant constraints. We discuss how changes in association multiplicities can be extended to further integrity constraints attached to a class diagram and how these changes interact with given database states.**

***Index Terms*— Integrity constraint, UML, OCL, cardinality constraint, schema transformation, constraint transformation**

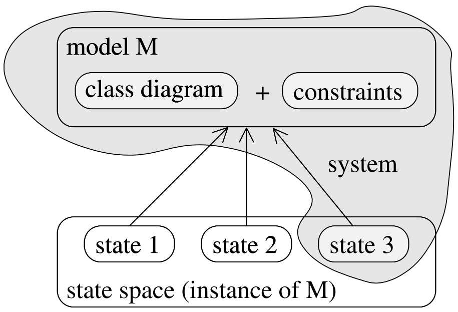


Fig. 1. Different kinds of transformations

I. Introduction

Integrity constraints within information systems are an important contribution for ensuring information system quality, in particular data quality. To mention just a few possible classification viewpoints, the spectrum for constraints ranges from model-inherent to explicit, from static to dynamic, and from declarative to operational. Constraints are regarded as part of a database schema, and it is only natural that with evolving system requirements also the database schema and with this the incorporated integrity constraints should be subject to systematic change and development as newer approaches in software engineering like the MDA [1], [2] propose.

This paper discusses integrity constraint evolution, from two perspectives: From the perspective of the changing the constraint, and from the perspective of (consequently) changing an existing database state. The paper concentrates on structural models provided by UML [3] class diagrams and OCL [4], [5] invariant constraints. However the considerations we make are not restricted to UML and OCL. We think the results could be applied to other data model based on objects (entities), associations (relationships), and integrity constraints described in a first order logic flair like standard SQL.

Figure 1 shows how the central notions in this paper (*model*, *state*, and *system*) are connected and used. A model M is described by a class diagram and a set of OCL integrity constraints, in other words a model can be regarded as a database schema. In general, a model M defines a set of valid (database) states. This is the so-called state space of the model M. By a system we understand the combination of the model M with one particular state. Using this terminology, we explain our understanding of what 'model transformation' could mean; we can imagine three different ways to deal with model transformation:

- An intuitive approach is to describe a model transformation as a modification of a class diagram. This can be done by giving a pair of class diagrams (before and after the transformations) describing the changes to be made. However, applying such a transformation can lead to difficulties, because OCL constraints formulated for the old class diagram may be ill-formed w.r.t. the new class diagram. For example, navigation expressions used in OCL constraints may change their types.
- A more complete approach is to treat models by additionally specifying how to deal with the OCL constraints attached to the old class diagram as well. As we will see, a complete solution for this is not always possible. Some transformations on class diagrams cannot be reflected completely as transformations on the OCL constraints.
- Finally, we can include the states of a model in our consideration. In order to transform a system (i.e., a model having a certain state), we need to specify a transformation on states as well. Unfortunately, it is not always possible to define a state transformation for every such model transformation but one can identify sufficient conditions for this.

One can find interesting transformations in all of the aforementioned categories. We are currently working on a catalogue containing transformations of all three kinds. We always start with a transformation on the class diagram level. Then, we try to extend each transformation to cover constraints and/or states as well, exploiting as much information as possible from the transformation. It is not possible to extend all transformations to cover existing OCL constraints and existing states in every case. Some class diagram transformations are incompatible with existing states, and some transformations are incompatible with attached constraints. However, interesting and relevant suffcient conditions and cases can be identified. In this paper we investigate one transformation from our planned catalogue. We illustrate how a class diagram transformation can be extended to cover OCL constraints and states. The transformation we have chosen are simple but yet complex enough to discuss several important aspects and details in the context of model transformations.

Our ideas are based on preceeding papers on UML class diagrams [6], [7]. Our work is also related to other approaches handling model transformations including constraints. One of them [8] offers a framework to perform transformations on class diagrams and to include potentially attached OCL constraints within this transformation. In [9] a possible automation of model transformations is discussed. Besides work on integrity constraints in general [10], [11] and the role of constraints in databases [12] there have been studies [13] on the transformation and especially the simplification of integrity constraints in databases. [14], [15] deal with multiplicity constraints in particular and discuss them in the context of entity relationship models. UML class diagrams including OCL constraints and their use in relational database design are treated in [16]. Further work includes reasoning aspects [17] and constraint maintenance [18].

The rest of this paper is structured as follows: Section 2 provides class diagram transformations which are investigated with regard to its extensibility: We study association multiplicity weakening and strengthening. Section 3 concludes the paper and shortly describes further ongoing work.

II. Transformations of Static Structure Models

In this section we discuss one kind of class diagram transformation in order to handle complete models (i.e., to include OCL constraints in the transformation) and to handle systems (i.e., to include the states of a model in the transformation).

The transformation used in this section is depicted in Fig. 2. It consists of two class diagram fractions describing how to change a certain part of a (probably larger) class diagram. Fig. 2 changes the multiplicities of the association A. The transformation describes that a multiplicity range (l,h) [low,high] on one association end in the upper class diagram is replaced by a new multiplicity range (l',h') in the transformed class diagram. The identifier rd in both class diagrams is a role name describing the role the D objects play in the association.

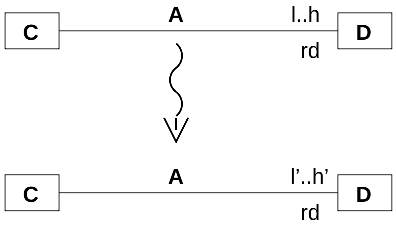
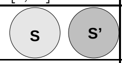
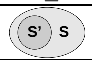
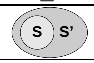
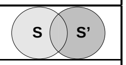


Fig. 2. Transformation of an association multiplicity

TABLE I

Model transformations categorized by state space

relationship of multiplicities	$[l',h'] \cap [l,h] = \emptyset$	$l \leq l'$, $h' \leq h$	$l' \leq l$, $h \leq h'$	otherwise
relationship of state spaces				
valid state transformation	no	for some states	yes	for some states
compatible with constraints	no	yes	no	no

The meaning of the UML multiplicity range (l,h) is as follows: Each instance of class C has a navigable property rd returning at least l and at most h instances of class D when applied to a C instance. Thus, when changing the multiplicities of A, we modify a model-inherent integrity constraint of the respective class. If one considers all states (i.e., all object diagrams) of a class diagram, this modification changes the state space of the class diagram.

Assuming that we consider the class diagram only (without the restricting model-inherent constraint), we can incorporate the states (object diagrams) of a class diagram in the introduced transformation as follows: Every object o of a class C and every link l of an association A in the old class diagram are still allowed in the new class diagram if we neglect the model-inherent constraints. For other class diagram transformations such as renaming a class this would not be the case. However, as we will see later we cannot transform every state in every case. This depends on the way how the range (l,h) is modified.

Table I presents a general classification of such a transformation by describing four categories (in column 2, 3, 4 and 5), which depend on the relationship between the old multiplicity (l,h) and the new multiplicity (l',h'). This is indicated in the first row. The second row illustrates how the state space S of the original class diagram and the state space S' of the modified class diagram are related. Depending on the relationship between the state spaces S and S', the third row states whether there is a state transformation corresponding to the class diagram transformation. The fourth (and last) row explains whether additional OCL constraints attached to the class diagram can be preserved by such a transformation. All this is discussed in more detail below.

If we now look at (l,h) and classify how the transformation works, we can refer to column three of Table I as a strengthening of association multiplicities (the interval is made smaller) and refer to column four as a weakening of association multiplicities (the interval is made larger). In the following, we will further discuss these two cases. Both are part of a general transformation catalogue which we have in mind for future work.

A. Strengthening of Association Multiplicities

Given the situation $l \leq l' \leq h' \leq h$ in Fig. 2, the state space described by the modified class diagram becomes a subset of the state space of the original class diagram. As a consequence, we cannot keep all original states when transforming a model.

TABLE II

TRANSFORMATION OF OCL TERMS

0..* (Multiplicity is unlimited)	Translation to iterate	1 (Multiplicity is one)	0..1 (Multiplicity is at most one)
τ.rd−>forAll(x | φ)	τ.rd−>iterate(x; r = true | r and φ)	φ[x← τ.rd]	τ.rd−>notEmpty implies φ[x← τ.rd]
τ.rd−>exists(x | φ)	τ.rd−>iterate(x; r = false | r or φ)	φ[x← τ.rd]	τ.rd−>notEmpty and φ[x← τ.rd]
τ.rd−>collect(x | φ)	τ.rd−>iterate(x; r = Bag{} | r−>including(φ))	Bag{φ[x← τ.rd]}	if τ.rd−>notEmpty then Bag{φ[x← τ.rd]} else Bag{} endif
τ.rd−>iterate(x; r = α | φ)		φ[x← τ.rd][r← α]	if τ.rd−>notEmpty then φ[x← τ.rd][r← α] else α endif

Some of the original states are not valid w.r.t the modified class diagram. Thus, we cannot extend the class diagram transformation 'strengthening of association multiplicities' to a general transformation of systems.

But we can translate all OCL constraints attached to the class diagram automatically. This is not possible for other arbitrary multiplicity changes as we will see below. Thus any class diagram transformation can be automatically extended to a complete model transformation when strengthening a multiplicity. We do not provide a complete formal proof for the translatability of OCL constraints but illustrate the translation with a typical example.

Figure 3 illustrates an application of our transformation to a class diagram possessing a class Person, a class Company and an association WorksFor. The multiplicity range on the employer end of WorksFor is unlimited $(0, \infty)$ in the original class diagram. This multiplicity range is restricted to a 'one' multiplicity $(1, 1)$ in the modified class diagram.

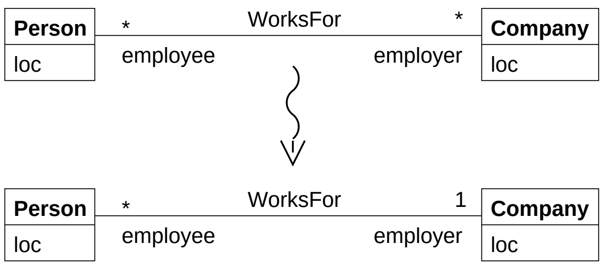


Fig. 3. Example of strengthening an association multiplicity

Let the class Person in the original class diagram have an additional constraint ensuring that each company a person p works for resides in the same location (e.g., a town) as the person p.

```
context Person inv sameLoc:
  self.employer->forAll(c| self.loc=c.loc)
```

When transforming the class diagram as depicted in Fig. 3, the collection-valued property 'employer' becomes an object-valued property due to the upper multiplicity bound 1. Thus the constraint body of the invariant sameLoc is not a well-formed formula w.r.t. the transformed class diagram. Nevertheless, the invariant constraint can be translated by replacing `self.employer` by `Set{self.employer}`:

```
context Person inv sameLoc:
  Set{self.employer}->forAll(c| self.loc=c.loc)
```

Keep in mind that this substitution is easy to perform but it can lead to a very complicated formula. Like in the above constraint there are several OCL collection operations apart from forAll: exists, select, collect and others. In OCL, each expression `self.employer->collOp` using a collection operation collOp can be expressed by using the iterate operation.

Given the expression τ of type $\texttt{Collection(}T_1\texttt{)}$ the syntax of the iterate operation is defined as follows:

$$\tau\texttt{:Collection(}T_1\texttt{)->iterate(y:}T_1\texttt{; result:}T_2\texttt{=}\alpha\texttt{|}\ \varphi\texttt{)}$$

`y` is a control variable that iterates through the given collection τ. The iterate operation returns the result variable `result` of type T_2 which is initialized with α. Within every iteration the result variable is reassigned to the result of the expression φ. Within the transformation the iterate form can be further simplified by taking advantage of the following two equalities:

$$\texttt{Collection\{x\}->iterate(y;r=}\alpha\texttt{|}\ \tau\ \texttt{)} \equiv \tau[\texttt{x}\leftarrow\texttt{y}][\alpha\leftarrow\texttt{r}]$$
$$\texttt{Collection\{\}->iterate(y;r=}\alpha\texttt{|}\ \tau\ \texttt{)} \equiv \alpha$$

The expression $\tau[\texttt{x}\leftarrow\texttt{y}][\alpha\leftarrow\texttt{r}]$ indicates two substitutions which have to be carried out one after the other in the expression τ. Every occurrence of term `y` will be substituted by the term `x` and every occurrence of term `r` will be replaced with the term α.

The two above equalities base on the dimension of the given collections and therefore on the number of iteration that will be performed. In the first equality the collection includes exactly one element, i.e., only one iteration will be made. Thus during the iteration the control variable `y` matches the collection element `x`. The result variable `r` is normally initialized with the expression α. Because we have exactly one iteration the result variable corresponds to α at that step. This leads us to the equivalent term $\tau[\texttt{x}\leftarrow\texttt{y}][\alpha\leftarrow\texttt{r}]$.

The second equality deals with the case of an empty collection. Therefore no iteration will be made and the result variable is only initialized, i.e., the iterate expression can be reduced to the term α.

Table II summarizes how to translate important OCL collection operations when transforming the multiplicity range from 0..* to 1 or to 0..1. As we mentioned above, all collection operations can be mapped to the iterate operation. Therefore only the iterate transformation is required to be transformed. This makes it easier to apply the multiplicity transformation to the constraints.

The table consists of four columns. The first column specifies the source expressions which contain the collection operations `forAll`, `exists`, `collect` and `iterate`. Their translations to the iterate operation are depicted in the second column. Here one can also realize the exact definition of these collection operations. The last two columns include the essential adjustments of the expressions due to the specific multiplicity transformation. These modifications of the OCL expressions result from the application of the iterate transformation listed in the same column to the translation of the collection operation to iterate (specified in column two). Furthermore the resulting expressions are simplified if possible, e.g., τ `and true` is reduced to τ.

Within the first multiplicity transformation described in column three the multiplicity range is restricted to 1, i.e., the transformed expressions will be applied to exactly one element. In terms of the transformed forAll and exists operation it has only to be assured that the condition φ holds for this element. Interestingly both operations result in the same expression, because there is no difference between a condition holding for all elements or just for one, if it is applied to a collection containing exactly one element.

Taking the results of the multiplicity transformation to range 0..1 the forAll and the exists operation offers different outcomes, because the multiplicity case 0 has to be covered. Both operations have distinct descriptions in this case. forAll returns true when applied to an empty collection. However the exists operation returns false in this case, because not at least one element holds for the given condition. Even if it is not obvious, the resulting expressions of forAll and exists are derived from the iterate transformation. For example the forAll transformation is equivalent to the following expression:

```
if τ.rd->notEmpty then φ[x← τ.rd]
  else true endif
```

If we apply the considerations of Table II to the sameLoc constraint this yields the simpler constraint formula:

```
context Person inv sameLoc:
  self.loc=self.employer.loc
```

As we have seen above, the transformation steps listed in Table II can be composed to a transformation of the given OCL collection operations when strengthening an association multiplicity. These considerations can be generalized to any OCL formula containing a navigable association end whose multiplicity range is strengthened by a transformation. More precisely, a class diagram transformation can result in a complete model transformation regarding the transformation steps in the above table.

Although we did not give a formal proof for the translation equivalence, the considerations should illustrate that every constraint that was formulated for the original class diagram can still be expressed under the transformed class diagram if the condition $[l', h'] \subseteq [l, h]$ holds.

B. Weakening of Association Multiplicities

A multiplicity range transformation meeting the condition $l' \leq l \leq h \leq h'$ extends the valid state space for a class diagram. The state space of the modified class diagram becomes a superset of the original one. Thus, an identity transformation on the states is sufficient to transform these states.

However the integrity contraints attached to the class diagram cannot be transformed automatically, e.g., invariants which previously covered only one element have to be expanded to cover a collection of elements. But one cannot guess the intention behind a constraint in order to cover additional elements. This requires decisions based on context knowledge which is not available in a model consisting of a class diagram and OCL constraints only. Nevertheless, we can guide a developer when modifying her or his constraints. This process can even be automated, i.e., an integrity constraint assistent could be implemented to make proposals for the needed constraint transformations. In the following, we discuss this problem in detail and explain general applicable approaches to translate OCL constraints under our multiplicity range expansion.

We take again the `WorksFor` example to demonstrate the various possibilities of invariant transformation. This time the multiplicity range of the employer end changes from a ‘one’ multiplicity (1,1) to unlimited (0,*).

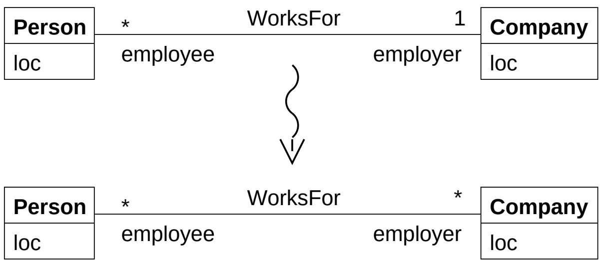


Fig. 4. Example of weakening an association multiplicity

When one weakens an association multiplicity, the class diagram transformation may produce ill-formed OCL expressions due to changing expression types. In Fig. 4, the object-valued property employer of class Person (type Company) is transformed to a collection-valued property (type Set(Company)). Consider an OCL expression τ of type Person which is part of another expression τ`.employer.`p where p refers to a property of Company. Then the complete expression becomes ill-formed after the described transformation, because the type of τ`.employer`, and consequently the type of τ`.employer.`p (the dot-operator can also be a short-hand notation for the collect operation), is changed from an object-valued type to a collection-valued type.

A universal solution to fix an OCL expression containing a term τ whose type is changed from T_1 to Collection(T_1) is the following rule using iterate:

$$\tau\texttt{.p} \quad \rightsquigarrow \quad \tau\texttt{->iterate(y; r:}T_2\texttt{ = }\alpha \mid \texttt{p')}$$

The term on the left-hand side of $\rightsquigarrow$ is meant to be evaluated in the original class diagram, the term on the right-hand side in the transformed class diagram. The right-hand side expression is not yet completely defined because the expression p' has not been explained. But this part cannot be derived automatically. The definition of p' depends on the intended semantics of the transformed expression. The only requirement is that the right-hand side expression models the intended constraint for the states of the original class diagram: all valid original states have to stay valid, all invalid original states (i.e., states that violate at least one invariant constraint) must be rejected. An exception to this are all invalid original states that violate the multiplicity constraint which should be modified within the multiplicity transformation. The validity of those states depends on the semantics of the transformed OCL expressions, i.e., the definition of p'. To illustrate how the class diagram transformation can be reflected in OCL expressions, let us consider a concrete example.

```
context Person inv sameLoc:
   self.employer.loc = self.loc
```

The OCL invariant sameLoc requires the formula `self.employer.loc=self.loc` to be true for all states of the class diagram. The constraint requires that an employee has the same location as her or his company. If we apply the weakening multiplicity range transformation of Fig. 4 to the class diagram, the above formula introduces a type error: self.employer.loc has type Bag(String) and self.loc has type String. And, even on an intuitive level, the meaning of the constraint is not clear anymore. Must all locations where a person is employed coincide with the location of the person? Or is it sufficient if only one location coincides?

This is a decision which requires context knowledge to be made. In any case, we can apply the provided rule employing iterate to translate the expression into a well-formed OCL formula. Using α := oclUndefined(Company) and p' := y, the resulting formula looks as follows:

```
self.employer
  ->iterate(y;r=oclUndefined(Company)|y).loc
    = self.loc
```

The interpretation of this new formula is to compare the loc value of self with the loc value of one chosen element of the employer collection of self. Which element of the employer collection is selected depends on the implementation of the iterate operation on sets.

However, one could use iterate on other places in the formula instead:

1) `self.employer->iterate(y;r=false | r or y.loc=self.loc)`
2) `self.employer->iterate(y;r=true | r and y.loc=self.loc)`

In the first case we chosen α :=`false` and p' :=`(r or y.loc = self.loc)` which means that at least one company location has to conform to the location of the person self. The second case expresses that the persons location has to coincide with all company locations. Thus, we have chosen α :=`true` and p' :=`(r and y.loc = self.loc)`. Comparing both expressions with the formulas listed in Table II one can recognize that these two assignments correspond to the definitions of exists and forAll.

1) `self.employer->exists(y|y.loc=self.loc)`
2) `self.employer->forAll(y|y.loc=self.loc)`

Probably, one of these modified formulas matches the intention of the designer for the transformed class diagram. Thus, when fixing an OCL formula that became ill-formed due to the class diagram transformation, the developer has two choices when applying the iterate rule: (1) where to apply the rule and (2) what formula p' to use in the expression part of the rule employing iterate. Both decisions cannot be made automatically, they both require context knowledge. However an integrity constraint assistent could provide formula proposals based on these considerations.

III. Conclusion

We have discussed two kinds of model transformation, i.e., two kinds of schema transformation: One transformation weakens association multiplicities, the other one strengthens the multiplicities. Both transformations have been considered w.r.t. to further constraints present in the database schema and w.r.t. an existing database state. We think that, when one discusses constraint evolution, one should always also care about the corresponding database state, because in practice it is costly to build a database state.

We feel that other class diagram transformation, for example the ones in [19], could be handled analogously to the approach presented here. However, much more work has to be done w.r.t. to propagating transformations to database states. For example, in special cases one could identify certain safe and uncritical parts in the database states for which one can guarantee that they will not be influenced by the changes made for the constraints. Future work will also discuss how the results from this paper can be applied in the context of standard database languages like SQL.

References

[1] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained. The Practice and Promise of the Model Driven Architecture*. Addison-Wesley, 2003.

[2] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, *MDA Distilled: Principles of Model-Driven Architecture*. Boston: Addison-Wesley, 2004.

[3] "UML 2.0 superstructure final adopted specification," Object Management Group, Tech. Rep., 2004, http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf.

[4] "UML 2.0 OCL final adopted specification," Object Management Group, Tech. Rep., 2004, http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf.

[5] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[6] M. Gogolla and M. Richters, "Equivalence rules for UML class diagrams," in *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, J. Bézivin and P.-A. Muller, Eds., 1998, pp. 87–96. [Online]. Available: citeseer.nj.nec.com/gogolla98equivalence.html

[7] ——, "Development of UML Descriptions with USE," in *Proc. 1st Eurasian Conf. Information and Communication Technology (EURASIA'2002)*, H. Shafazand and A. M. Tjoa, Eds. Springer, Berlin, LNCS 2510, 2002, pp. 228–238.

[8] J. Whittle, "Transformations and software modeling languages: Automating transformations in uml." in *UML*, 2002, pp. 227–242.

[9] E. A. Deba, "Towards automation of model transformation." in *IFIP Student Forum*, 2004, pp. 155–164.

[10] P. Godfrey, J. Grant, J. Gryz, and J. Minker, "Integrity constraints: Semantics and applications." in *Logics for Databases and Information Systems*, 1998, pp. 265–306.

[11] H. Decker, "Translating advanced integrity checking technology to sql." in *Database Integrity*, 2002, pp. 203–249.

[12] M. Morgenstern, "The role of constraints in databases, expert systems, and knowledge representation." in *Expert Database Workshop*, 1984, pp. 351–368.

[13] H. Christiansen and D. Martinenghi, "Simplification of Database Integrity Constraints Revisited: A Transformational Approach." in *LOPSTR*, 2003, pp. 178–197.

[14] M. Lenzerini and G. Santucci, "Cardinality constraints in the entity-relationship model." in *ER*, 1983, pp. 529–549.

[15] S. Ferg, "Cardinality constraints in entity-relationship modeling." in *ER*, 1991, pp. 1–30.

[16] B. Demuth and H. Hußmann, "Using uml/ocl constraints for relational database design." in *UML*, 1999, pp. 598–613.

[17] A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini, "A formal framework for reasoning on uml class diagrams." in *ISMIS*, 2002, pp. 503–513.

[18] E. Baralis, S. Ceri, and S. Paraboschi, "Declarative Specification of Constraint Maintenance." in *ER*, 1994, pp. 205–222.

[19] M. Gogolla and M. Richters, "Expressing UML Class Diagrams Properties with OCL," in *Advances in Object Modelling with the OCL*, T. Clark and J. Warmer, Eds. Springer, Berlin, LNCS 2263, 2001, pp. 86–115.