Electronic Communications of the EASST
Volume 5 (2006)

Proceedings of the Sixth OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)

Model-Driven Constraint Engineering

Michael Wahler, Jana Koehler and Achim D. Brucker

20 pages

Guest Editors: Dan Chiorean, Birgit Demuth, Martin Gogolla, Jos Warmer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Model-Driven Constraint Engineering

Michael Wahler[1], **Jana Koehler**[2] **and Achim D. Brucker**[3]

[1] wah@zurich.ibm.com, [2] koe@zurich.ibm.com
IBM Zurich Research Laboratory
Saeumerstrasse 4, 8803 Rueschlikon, Switzerland

[3] brucker@inf.ethz.ch
Information Security, ETH Zurich
8092 Zurich, Switzerland

Abstract: Precise specification of meta-models is an important prerequisite for the successful application of a model-driven engineering (MDE) process. One means of precise specification are textual constraints. However, the task of constraint development is time-consuming and error-prone if done manually.

In this paper, we present both a methodology and a tool for developing constraints in a systematic way that can be integrated into a CASE tool. Thus, we provide a semi-automated means for integrating constraints into the MDE process.

Our approach is based on an extensible library of generic constraint patterns. Constraint patterns can be combined to create complex constraints and easily parameterized in a CASE tool. Moreover, we show how these parameterized patterns are transformed into platform-independent or platform-specific constraints by a model transformation.

Keywords: constraint, pattern, model-driven engineering, UML, OCL

1 Introduction

In model-driven engineering (MDE), a model defines the building blocks from which instances can be constructed. The main building blocks in the Meta-Object Facility (MOF, [Obj02]) are classes, their structural features and associations between the classes. Models are usually specified with a concrete graphical syntax, which allows for rough specification only.

The set of possible instances grows with the number of building blocks that are defined in the model. In general, not all possible instances are valid with respect to the semantics of the model. Therefore, textual constraints are used on the model to express details that are either difficult or even impossible to express in a diagrammatic way. Adding constraints to a model usually decreases the number of possible instances unless contradictory constraints are introduced.

Constraints stem from different sources: there may be legal restrictions that a system needs to obey; there may be company policies that grant privileges to certain kinds of customers; there may be technical restrictions on a system [CLW+06]; there may be security restrictions [LBD02]; and there may be facts that are implied by common sense that cannot be expressed diagrammatically. For instance, hundreds of constraints are used in the specification of the Unified Modeling Language (UML) meta-model [Obj05].

Whereas models were solely used for documentation and communication purposes in the past, recent model-centric development approaches use models as first-class artifacts in the development process. For instance, business process models can be transformed to executable code that is run on process execution engines [HK04] or models in a domain-specific security language are transformed to UML [BDW06]. To guarantee correctness of the execution of the generated code, it is crucial that every model instance conform to its defining model and satisfy its constraints. These validity checks can be performed automatically if the constraints are formalized. For instance, tools exist that type-check a set of OCL (Object Constraint Language [Obj03]) constraints and validate a model against them [ÁRF03]. Alternatively, validity checks can be implemented in a programming language, e. g., Java, using a model-access API.

Creation and maintenance of constraints are tedious tasks. In a case study we performed in a business modeling environment, about 80 constraints were necessary to guarantee the executability of a behavioral model for business process monitoring. All constraints are invariants on the model elements and restrict the set of allowed model instances to a set that is executable on a process execution engine. Whereas some of these constraints were rather simple, many complex constraints needed to be formalized, which turned out to be a time-consuming and error-prone task. The formalization resulted in approximately 500 lines of OCL code, which by nature are unlikely to be bug-free. Furthermore, the meaning of formal constraints is often misunderstood by novice users [Cab06].

Even if the constraint expressions and the validation code do not contain any errors, they need to be adapted once the model changes. This usually results in additional time-consuming coding and debugging phases, especially in refactorings [CW04, MB05] where models undergo frequent changes and the attached constraints need to be kept consistent with new versions of the model.

Our contribution to solving the problem of constraint development consists of four parts. Firstly, we introduce the notion of computation-independent constraint patterns and show how to transform them into platform-independent or platform-specific constraints. Secondly, we introduce a library of constraint patterns, separate the patterns into atomic and composite patterns, and add a structure to them to enhance their expressiveness and usability. Thirdly, we provide meta-constraints that restrict the parameter values of the constraint patterns, thus excluding invalid pattern instances. Fourthly, we discuss the requirements for integrating model-driven constraint engineering in a CASE tool and illustrate our prototype for Eclipse/UML2 [ECL].

We believe that a flexible pattern-based approach that is supported by a tool offers an important improvement for constraint engineering. Most syntactic and semantic errors can be avoided because the developer can generate OCL code instead of writing it by hand. Furthermore, our solution promises to decrease development time substantially.

The paper is organized as follows: After presenting some examples motivating the use of patterns in Section 2, we show how patterns can be derived by generalizing a concrete specification in Section 3. In Section 4, we present our library of patterns together with a taxonomy for them. To integrate patterns into an MDE process, we first present the transformation of parameterized patterns to concrete constraints in Section 5. Then, we present how to add support for this approach to a CASE tool in Section 6. We discuss related work in Section 7 and conclude this paper with a summary of our contributions and pointer to future work in Section 8.

2 Example Model and Constraints

In Figure 1 we illustrate a simple model of a company that serves as example throughout the remainder of this paper. The UML class diagram contains five classes, in which Manager and Employee are related by a many-to-many relation. Each instance of Employee is associated with exactly one office, whereas there are no restrictions on the number of inhabitants in one office.

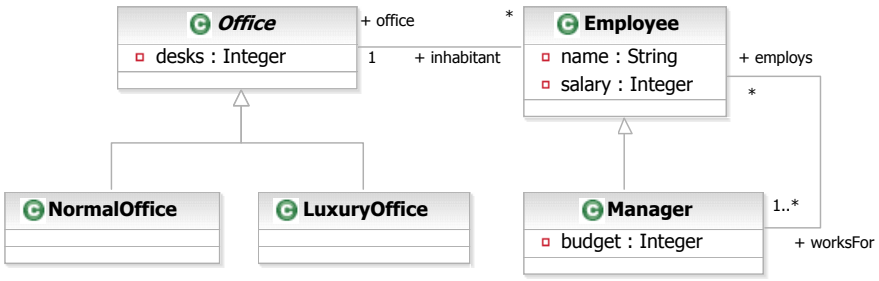


Figure 1: Manager and Employee Class Diagram

Besides the defined classes and associations, instances of this model are not restricted in any way: there may be managers without employees, and employees may have a salary of zero while working for multiple managers. However, there are additional requirements that each company has to comply with. We assume fictitious labor union and company IT requirements that every work environment has to satisfy. The requirements are captured in the following constraints, informally in English and formally as OCL expressions.

Constraint 1. A manager with a budget of more than 100,000 must employ at least one employee with a salary of at least 3000.

This constraint requires that for each instance m of Manager whose budget is greater than 100,000, there exists an instance e of Employee that is related to m by the relation employs. Furthermore, the value of the salary attribute of e must be at least 3000.

```
context Manager
inv: self.budget > 100000 implies self.employs->exists( e | e.salary >= 3000)
```

Constraint 2. A manager may not occur twice within the management hierarchy.

This constraint prevents that a manager m is responsible for him-/herself by being related to him-/herself directly by the worksFor relation or indirectly by other managers $\{m_i, \ldots, m_j\}$ who work for m. Formally, a manager may not be an element of the transitive closure of the worksFor relation. However, OCL does not provide an operator to compute the transitive closure of a relation.

Thus, we need to define an operation closureWorksFor(S) that computes the transitive closure [Baa03] of the worksFor relation. We use the parameter S to ensure the termination of the computation. This parameter stores the elements for which the transitive closure has been computed; it is initially empty. In each step, the set S is deducted from the set of elements in the worksFor relation. Eventually, S contains all elements in the transitive closure, and the computation terminates.

```
context Manager
def: closureWorksFor(S:Set(Manager)) : Set(Manager) =
     worksFor−>union((worksFor − S)−>
       collect (m : Manager | m.closureWorksFor(S−>including(self)))−>asSet())
inv: not self.closureWorksFor(Set{})−>includes(self)
```

Constraint 3. The company may not have more than five organizational layers.

This constraint restricts the depth of the worksFor navigation path. Because a manager can employ another manager, arbitrary hierarchy levels can be instantiated. However, Constraint 3 forbids more than five hierarchy levels.

Therefore, we define the recursive query pathDepthWorksFor(max,counter) that evaluates if the worksFor relation adheres to the maximum path depth. This query has two parameters, max and counter, where max is set to the desired maximum path depth minus 1 and counter is initialized with 0. The query terminates with false if the value of counter is greater than the value of max, i.e., the maximum path depth has been exceeded. Otherwise, the counter is increased and the query recursively evaluated on all elements that are related by worksFor.

```
context Manager
def: pathDepthWorksFor(max:Integer, counter:Integer): Boolean =
     if (counter > max or counter < 0 or max < 0) then false
     else if (self.worksFor−>isEmpty()) then true
          else self.worksFor−>forAll(m:Manager|m.pathDepthWorksFor(max, counter+1))
          endif
     endif
inv: self.pathDepthWorksFor(4,0)
```

3 Deriving Constraint Patterns

Constraint patterns can be identified by analyzing existing constraints for recurring expressions and abstracting from them. In the following, we use the constraints from Section 2 to illustrate how patterns are derived from concrete constraints. Constraint 1 is an implication and thus consists of two parts, a premise and a conclusion. From an abstract point of view, the premise restricts the *value* of an attribute to a constant. In the conclusion, the *existence* of a certain instance related to the context object is required, and there is another value restriction on the attribute salary of the related instance. Thus, we can identify the patterns *Exists* and *AttributeValueRestriction*, corresponding to existential quantification and value restriction respectively.

From Constraint 2, we can derive a pattern *CyclicDependency* that identifies cyclic navigation paths in model instances. Finally, Constraint 3 can generally be seen as a constraint that restricts the maximum length of a navigation path from which we derive the *PathDepthRestriction* pattern.

In general, a constraint pattern is a higher-order function that maps a set of parameters to a constraint. The semantics of a constraint pattern can be provided in any language, e. g., parameterized OCL templates such as in [AT06]. This has the advantage that an OCL constraint can simply be instantiated from such a pattern by providing values for the pattern parameters.

In our solution, which we call *model-driven constraint engineering*, we follow the Model Driven Architecture (MDA) approach [KWB03], which comprises models at different levels

of abstraction. MDA is an MDE variant defined by the Object Management Group, and we use MDA and MDE as synonyms in the remainder of this paper. We consider a constraint pattern a computation-independent model (CIM) of a constraint, because no knowledge of a formal constraint language is required to apply a pattern as long as the informal semantics of the pattern is understood. Such a CIM constraint can be transformed into a platform-independent or platform-specific model (PIM/PSM) by a model transformation.

Following MDA, the (formal) semantics of a pattern is defined within the transformations. Therefore, we define two transformations for the *CyclicDependency* pattern that generate an OCL expression and Java code. First, we need to define a signature for the pattern to specify the parameters and their types. The only parameter for this pattern is an OCL navigation expression, which is a sequence of properties. Thus, the signature is CyclicDependency(navigation:Sequence(Property)).

The transformations for this pattern are simple template–processing functions that replace the placeholders for the parameters with concrete values. The OCL template for the *CyclicDependency* pattern is self.closure<navigation>(Set{})->includes(self), in which we assume the existence of a template function closure<navigation>().

As mentioned, different target platforms can be used instead of generating OCL code. For instance, a template for the transformation to Java validation code can be defined as follows.

```
boolean validateCyclicDependency(List navigation) {
  Set s = this.closure(navigation, new Set());

  if (s.includes(self))
    return true;
    else return false;
}
```

A pattern can be instantiated by providing values for its parameters. However, not all pattern instantiations are meaningful. For instance, the navigation path that is used to parameterize the *CyclicDependency* pattern needs to be reflexive. To exclude meaningless parameter values such as negative values for multiplicity bounds, we define meta-constraints for each constraint pattern. These meta-constraints are usually very simple OCL expressions. For instance, the following meta-constraint ensures that the navigation path that is used to parameterize the *CyclicDependency* pattern is reflexive.

```
self.class.<navigation>.class = self.class
```

4 A Taxonomy of Structured Constraint Patterns

Although the constraint pattern approach as it has previously been introduced [AT06, CGQ+06, MN05] reduces both the development time and error rate for model constraints, it has one important restriction: As each pattern represents a subset of all possible constraint expressions, there will be many constraints that are not expressible in terms of existing constraint patterns. This holds even if an extensive pattern library is used.

Therefore, we introduce the notion of *structured constraint patterns*, which adds a high degree of expressiveness to the constraint pattern approach by two measures. Firstly, we introduce the

logical concepts of implication and negation into the pattern model, which allows a user to create complex constraints from existing patterns. Secondly, we divide constraint patterns into *atomic* and *composite* patterns. The set of atomic patterns represents recurring restrictions that we have identified, and it is extensible by the user. The composite patterns are recursively constructed from atomic patterns and represent higher-order concepts such as quantification.

4.1 Adding Logical Structure to Constraint Patterns

The core of our approach is the class StructuredConstraint, which is a specialization of the UML meta-class Constraint. This class contains the concepts of *negation* and *implication*, allowing instances of each pattern to be inverted and logically combined.

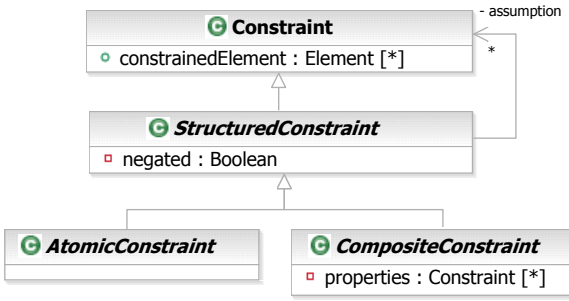


Figure 2: UML Class Diagram of Structured Constraint Concept

The concept of logical implication is implemented as shown in Figure 2. Each structured constraint c can have a finite set A of assumptions that can be any kind of constraint, which is illustrated by the association assumption from StructuredConstraint to Constraint. This allows us to use either arbitrary constraints (e. g., in OCL) or structured pattern instances as assumptions for constraints. The semantics of the assumption relation is defined as follows: Let c be an instance of a structured constraint and A be a finite set of constraints that is related to c with the assumption relation. Then the conjunction of all constraints in A implies c. The concept of logical negation is represented by the attribute negated of the class StructuredConstraint.

Figure 2 also introduces the concepts of AtomicConstraint and CompositeConstraint, which are abstract subclasses of StructuredConstraint. An example for this concept of structured constraints is Constraint 1 from Section 2. This constraint consists of three parts. The first part is the assumption self.budget > 100000, the second part the existential quantification self.employs->exists(e | ...), and the third part the properties of the quantification, e.salary > 3000. We consider the expressions in the assumption and the quantification property as atomic constraints, whereas we consider the quantification itself as a composite constraint. In the following, we elaborate on the concepts of atomic and composite constraint patterns.

4.2 Atomic Constraint Patterns

In this section, we present an extensible library of atomic constraint patterns. The idea of atomic constraint patterns is to identify a relevant set of atomic constraints that covers frequently occurring fundamental restrictions on a model, e. g., restrictions on attribute values or on relations

between objects. The patterns presented in this section originate from a case study in which we formalized constraints for a business process monitoring model [CLW+06].

Furthermore, we relate the patterns using generalization associations. This creates a *taxonomy* of patterns. This taxonomy gives a structure to the set of patterns and helps one to find the right pattern for a specific purpose. Figure 3 illustrates the taxonomy of atomic constraint patterns we have identified. In this figure, the patterns are represented as classes that are related with generalization associations. The parameters of the patterns are specified as attributes, which refer to simple types such as Integer, to UML meta-classes such as Class, and to the OCL meta-class OclExpression.

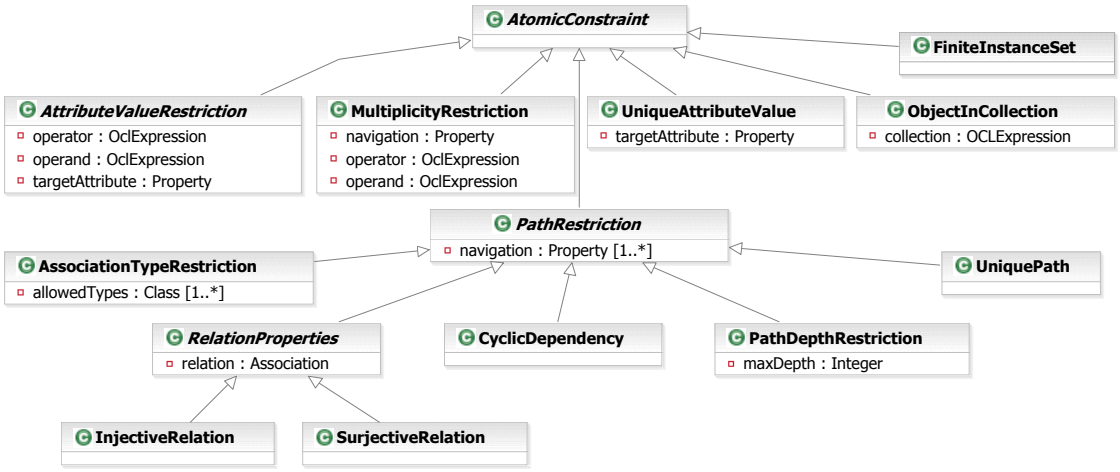


Figure 3: UML Class Diagram of Atomic Constraint Patterns

In the following, we further specify the patterns in Figure 3 with informal and formal semantics. Whereas we use English for the informal semantics, we define the formal semantics in the form of OCL templates. These templates will later be the basis for the model transformation that generates OCL constraints from pattern instances. For each constraint pattern, we also define meta-constraints that ensure the well-formedness of pattern instances, as described in Section 3.

4.2.1 Attribute Value Restriction.

The *AttributeValueRestriction* pattern can be used to restrict the value of an attribute of a class for all instances of the class. It is a very simple pattern and thus well-suited for introducing our syntax for OCL templates to the reader.

```
AttributeValueRestriction ( targetAttribute :Property,operator,term:OclExpression): Boolean
 = self .< targetAttribute > <operator> <term>
```

There is one meta-constraint that instances of this pattern need to satisfy: the parameters targetAttribute and term need to be of the same type.

```
a)  targetAttribute .type = term.type
```

Example: The premise of Constraint 1 from Section 2—the fact that the budget of a manager must be greater than 100,000—can be expressed using an instance of this pattern.

```
context Manager
inv: AttributeValueRestriction (budget, >, 100000)
```

4.2.2 Multiplicity Restriction.

The *MultiplicityRestriction* pattern restricts the multiplicity of an association. Whereas UML class diagrams allow for constraining multiplicities to a fixed interval, this pattern allows a user to define multiplicity restrictions that depend on properties of the model instance, e. g., an attribute value.

```
MultiplicityRestriction (navigation:Sequence(Property),operator,operand:OclExpression): Boolean
= self.<navigation>->size() <operator> <operand>
```

We identified two meta-constraints for this pattern. Firstly, the property navigation needs to evaluate to a collection. Secondly, the operand must be a positive number.

```
a) self.<navigation>.oclIsKindOf(Collection)
b) <operand> >= 0
```

Example: A typical example of this pattern is the association between Office and Employee: The number of employees in an office may not exceed the number of desks in an office.

```
context Office
inv: MultiplicityRestriction (Sequence{inhabitant}, <=, desks)
```

4.2.3 Object in Collection

The *ObjectInCollection* pattern can be used to express that the context element is in a collection of related objects.

```
ObjectInCollection(navigation:Sequence(Property)): Boolean
= self.<navigation>->includes(self)
```

The parameter collection for this pattern needs to evaluate to a Collection.

```
a) self.<collection>.oclIsKindOf(Collection)
```

Example: This constraint pattern can be used to express that a manager needs to work in the same office with at least one employee, using the following pattern instance.

```
context Manager
inv: ObjectInCollection(Sequence{employs.office.inhabitant})
```

4.2.4 Unique Attribute Value

The *UniqueAttributeValue* pattern requires that all instances of the constrained class have distinct values for the target attribute specified. This pattern is also known as “semantic key” [AT06], “primary identifier” [MN05] or “identifier” [CGQ+06] pattern in the literature.

```
UniqueAttributeValue(targetAttribute:Property): Boolean
= self.allInstances()->isUnique(<targetAttribute>)
```

The only meta-constraint that needs to be satisfied is that the specified targetAttribute belongs to the context class.

```
a) targetAttribute.class = self.class
```

Example: Instances of the Employee class are uniquely identifiable by their name attribute.

```
context Employee
inv: UniqueAttributeValue(name)
```

4.2.5 Association Type Restriction

The *AssociationTypeRestriction* pattern can be used to restrict an association a that is defined between the context class and a superclass C_0. Using this pattern, it can be enforced that only instances of certain subclasses $C_1, \ldots, C_n$ of C_0, the allowedTypes, may participate in the relation.

```
AssociationTypeRestriction(navigation:Sequence(Property), allowedTypes:Set(Class))
 = self.<navigation>->forAll( x | <allowedTypes>->exists( c | x.oclIsTypeOf(c)))
```

Again, the property navigation needs to evaluate to a collection.

```
a) self.<navigation>.oclIsKindOf(Collection)
```

Example: Our company model in Figure 1 allows employees to work in any kind of office. This pattern can be used to enforce that managers must work in luxury offices.

```
context Manager
inv AssociationTypeRestriction(Sequence{office},Set{LuxuryOffice})
```

4.2.6 Cyclic Dependency

The *CyclicDependency* pattern can be used to identify cycles in the instance graph of a model. Such a cycle requires a reflexive association or navigation path in the model.

```
CyclicDependency(navigation:Sequence(Property))
 = self.closure<navigation>(Set{})->includes(self)
```

For this pattern, we assume the existence of an operation that computes the transitive closure for each reflexive navigation. We further require that the parameter navigation denote a reflexive association, which we capture in the following meta-constraint.

```
a) self.<navigation>->forAll( x | x.class = self.class)
```

Example: An example instance of this pattern is Constraint 2. As this constraints forbids the existence of a cycle, we need to use the negation feature that each pattern inherits from StructuredConstraint, which can be textually represented as follows.

```
context Manager
inv: not CyclicDependency(Sequence{worksFor})
```

4.2.7 Path Depth Restriction

The *PathDepthRestriction* pattern can be used to restrict the maximum path length for instances of reflexive associations.

```
PathDepthRestriction(navigation:Sequence(Property), maxDepth:Integer)
 = self.pathDepth<navigation>(maxDepth−1, 0)
```

Again, we assume the existence of a function that computes the path depth for each reflexive association. Two meta-constraints need to be satisfied by instances of this pattern. Firstly, navigation needs to be reflexive. Secondly, the value maxDepth needs to be at least one, because of the definition of the path depth function (cf. Section 2).

```
a) self.<navigation>−>forAll( x | x.class = self.class)
b) maxDepth >= 1
```

Example: Constraint 3 is an example instance of this pattern, where the maximum length of the employs association is restricted to 5. Using the pattern, this constraint can be defined as follows.

```
context Manager
inv: PathDepthRestriction(Sequence{worksFor},5)
```

4.2.8 Unique Path

The *UniquePath* pattern ensures that there is not more than one path from the context element to a related element.

```
UniquePath(navigation:Sequence(Property))
 = self.<navigation>−>forAll( x | self.<navigation>−>count(x) = 1)
```

Again, the property navigation needs to evaluate to a collection.

```
a) self.<navigation>.oclIsKindOf(Collection)
```

Example: An infamous example configuration that can be excluded with this pattern was identified in [RC81] and became famous as the "Nixon diamond" in nonmonotonic reasoning and as the "diamond of death" in object-oriented programming languages. In this configuration, four classes A, B, C and D are in the generalization relation $\prec = \{(A,B),(A,C),(B,D),(C,D)\}$. If B and C inherit a structural feature x from A, it is unclear whether D inherits $B :: x$ or $C :: x$. Thus, the path from a class to each superclass of its superclasses should be unique.

```
context Class
inv: UniquePath(Sequence{superClass.superClass})
```

4.2.9 Injective Relation

The *InjectiveRelation* pattern can be used to establish the mathematical concept of an injective relation $R : X \times Y$, i.e., $R(x_1,y) \wedge R(x_2,y) \rightarrow x_1 = x_2$.

```
InjectiveRelation (navigation:Sequence(Property))
 = self.<navigation>−>size() = 1 and
   self.allInstances()−>forAll(x,y | x.<navigation> = y.<navigation> implies x=y)
```

Again, the property navigation needs to evaluate to a collection.

```
a) self.<navigation>.oclIsKindOf(Collection)
```

Example: An intuitive example is the constraint that no two employees may work in the same office. This can be expressed through the following pattern instance.

```
context Employee
inv: InjectiveRelation (Sequence{office})
```

4.2.10 Surjective Relation

The *SurjectiveRelation* pattern can be used to establish the mathematical concept of a surjective relation $R : X \times Y$, i.e., $(\forall y \in Y).(\exists x \in X).R(x,y)$.

```
SurjectiveRelation(navigation:Sequence(Property))
 = self.<navigation>.allInstances()->forAll( y |
      self.allInstances()->exists( x | x.<navigation>->includes(y)))
```

Again, the property navigation needs to evaluate to a collection.

```
a) self.<navigation>.oclIsKindOf(Collection)
```

Having defined patterns for injective and surjective relations, we can deduce a pattern for bijective relations, i.e., one-to-one relations.

```
BijectiveRelation (navigation:Sequence(Property))
 = InjectiveRelation (navigation) and SurjectiveRelation(navigation)
```

Surjectivity and bijectivity can also be expressed using multiplicities in the class diagram. However, these patterns become important if an association is restricted under certain assumptions only (cf. Subsection 4.1) and not globally for all instances of a model.

4.2.11 Finite Instance Set

The *FiniteInstanceSet* pattern can be used to disallow an infinitely large number of instances of a class. This is usually guaranteed because of memory bounds in real systems, but can lead to problems when reasoning about models.

```
FiniteInstanceSet()
 = not self.allInstances()->size().oclIsUndefined()
```

Example: In our company model, we model only real–world entities such as offices or employees. Therefore, each class should be required to have a finite number of instances only.

```
context Employee, Office
inv: FiniteInstanceSet()
```

4.3 Composite Constraint Patterns

Apart from atomic constraint patterns, each of which restricts a basic property of a model, composite constraints can be used to express complex properties by integrating an arbitrary number of other constraints (either atomic or composite). Using such a divide-and-conquer approach, complex constraints can be developed in a structured way by combining several simple constraints.

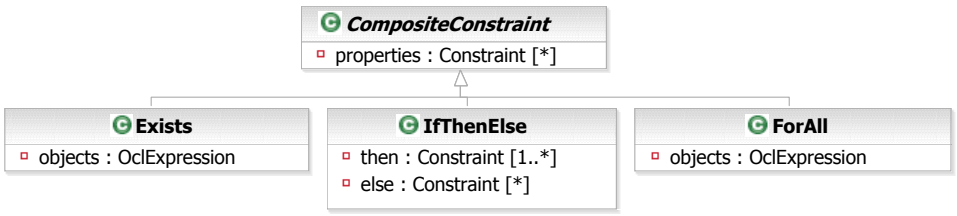


Figure 4: Class Diagram of Composite Constraint Patterns

So far, we have identified three composite constraint patterns, *Exists*, *ForAll* and *IfThenElse*, which we illustrate in Figure 4.

Constraint 1 from Section 2 contains an example instance of the *Exists* pattern: for the context element m of class Manager, there has to exist an element e that is related to m with the navigation employs. This element e must satisfy a set of constraints, the properties of the composite constraint.

In the following, we provide a template for the *Exists* pattern. This pattern cannot be expressed in OCL because it quantifies over a set of predicates, which is a concept of higher-order logic. Therefore, we use the operator $\bigwedge$ to define the pattern in pseudo-OCL. When an instance of a pattern is transformed to an OCL expression, this operator is unfolded to a sequence of conjunctions.

```
Exists(properties:Set(Constraint),objects:OclExpression) =
  objects->exists( o | ⋀_{p∈properties} p(o))
```

The *ForAll* constraint pattern is defined analogously. The *IfThenElse* pattern denotes an if-then-else expression. If the context element of the constraint satisfies all properties, it also needs to satisfy all then constraints, otherwise, it needs to satisfy all else constraints.

```
IfThenElse(properties, then, else:Set(Constraint)) =
  if (⋀_{p∈properties} p(self))
 then (⋀_{p∈then} p(self))
 else (⋀_{p∈else} p(self)) endif
```

5 Transforming CIM to PIM

Having defined a library of CIM constraint patterns, we need to provide model transformations to generate PIM or PSM constraints from the parameterized patterns. As mentioned before, multiple transformations for different target languages can be defined.

In this section, we illustrate a transformation that generates OCL constraints from parameterized CIM constraint patterns. This transformation, transform_OCL(c), uses OCL templates to generate output for a pattern c. We use pseudo code that has the same expressiveness as common programming languages for the definition of the operations.

Three steps are necessary to transform an atomic constraint pattern. First, the code for the assumptions is generated if there are any. Then, the OCL keyword not is inserted into the constraint expression if the pattern attribute negated is true. Finally, the variables in the templates for the constraint patterns are replaced by concrete values from the pattern specification

by replace_parameters(t), which is a simple string replacement and thus not further specified in this paper. Listing 1 shows the complete transformation from CIM to PIM for an atomic pattern.

```
sub transform_OCL(c:AtomicConstraint) {
  # print the assumptions of the constraint
  transform_assumptions_OCL(c);

  # print the OCL keyword 'not' if the constraint is negated
  if (c.negated) print "not ";

  # replace the variables in the template and print constraint
  print replace_parameters(template(c));
}
```

Listing 1: OCL Transformation Function for Atomic Patterns

Two operations are invoked from within transform_OCL(c). Whereas replace_parameters(t) performs simple string replacement, transform_assumptions_OCL(c) is slightly more complicated. In this operation, the set of assumptions is transformed into a conjunction of predicates, followed by the OCL operator implies. Listing 2 shows the definition of this operation.

```
sub transform_assumptions_OCL( c: StructuredConstraint ) {
  # print the conjunction of assumptions
  foreach p in c.assumption
     print transform_OCL(p);
     if (c.assumption.hasNext()) print " and ";

  # print the implication operator if necessary
  if (c.assumption.notEmpty()) print " implies ";
}
```

Listing 2: Transformation Function for Assumptions

Our composite constraints use other constraints as properties for the elements in their object collections. This higher-order use of constraints renders the code generation slightly more complicated than for atomic constraints. In particular, the operator $\bigwedge$, representing a conjunction of predicates in a set, needs to be transformed.

```
sub transform_OCL( c : CompositeConstraint ) {
  transform_assumptions_OCL(c);
  if (c.negated) print "not ";

  # copy the template into a variable 'body'
  body := template(c);

  # generate expressions for the properties and add them to conjuncts
  foreach p in c.properties {
    conjuncts.add(transform_OCL(p).replace("self", "e"));
  }

  # replace "⋀" by the generated conjuncts
  foreach p in conjuncts {
    if (conjuncts.hasNext())
      body.replace("⋀", p+" and "+"⋀");
    else
      body.replace("⋀", p);
  }
```

Listing 3: Transformation Function for Composite Constraints

The transformation of a composite constraint pattern c is shown in Listing 3 and works as follows. Lines 2 and 3 generate the assumptions and the negation flag as usual. In Line 6, the template text is copied into a variable body. In Lines 9–11, the properties associated with the composite constraint are recursively generated and stored in a vector conjuncts. In Lines 14–19, the operator $\bigwedge$ is replaced by an explicit conjunction.

If c is an instance of the *IfThenElse* pattern, its *then* and *else* parts need to be transformed as well. This transformation is analogous to the transformation shown in Lines 9–19 of Listing 3.

6 Tool Support for Model-Driven Constraint Engineering

Tool support is essential for the acceptance and success of model-driven engineering approaches. In this section, we discuss how to integrate our idea of structured constraint patterns in a model-driven development tool and apply the tool to the example from Section 2.

6.1 Technical Solution

As depicted in Figure 2, our concept of a structured constraint is a specialization of the UML meta-class Constraint. There are mainly two commonly accepted approaches for creating variations of the UML meta-model, namely, extending the meta-model itself or adapting the meta-model with a UML Profile [Coo00]. We suggest an implementation of our approach as a UML Profile because we consider it a light-weight approach that simplifies the interoperability between tools.

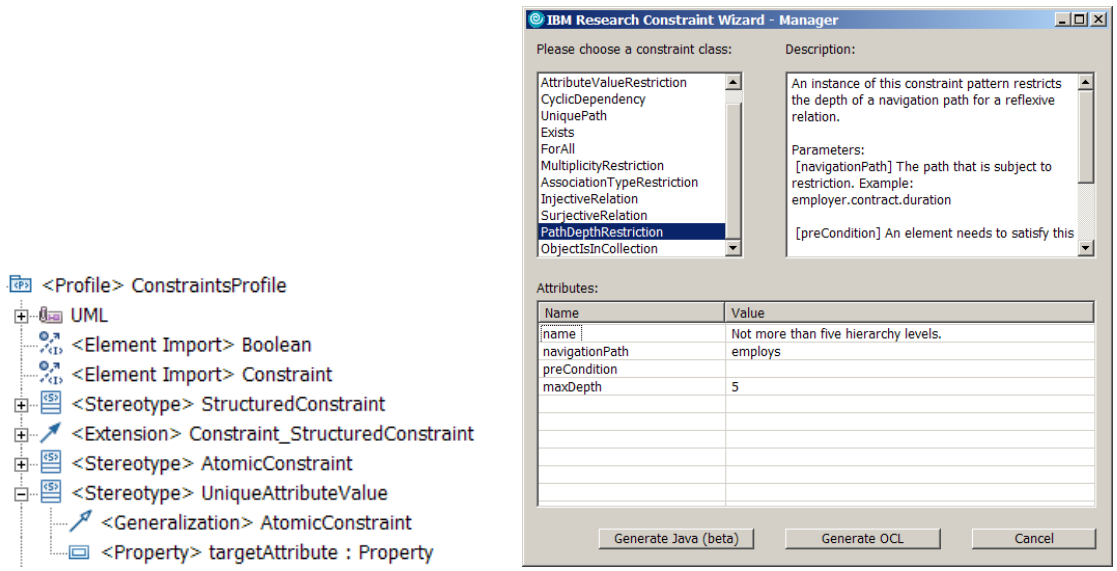
In our profile, each constraint pattern is represented by a UML stereotype. The taxonomy of constraint patterns is established using generalization associations between the stereotypes. The attributes of the constraint patterns become attributes of the stereotypes in the implementation.

In this solution, one limitation of UML 2.0 becomes critical. In UML 2.0, stereotypes may not have associations with meta-classes [Obj05]. Thus, a *UniqueAttributeValue* constraint cannot refer to the UML meta-class Property. Even worse, a composite constraint cannot refer to other constraints as we introduced it in Subsection 4.3. However, this deficiency has been remedied in the UML 2.1 standard [Obj06], in which associations between a stereotype and a meta-class can be defined.

The Eclipse UML2 project [ECL] provides an implementation of the UML 2.1 meta-model based on the Eclipse Modeling Framework [EMF]. This makes Eclipse/UML2 an ideal platform for implementing tool support for structured constraint patterns. In Figure 6(a) we show a screenshot of the UML Profile editor in Eclipse. As can be seen, the taxonomy of structured constraint patterns can be implemented in a straightforward manner as a UML 2.1 profile.

We prototyped a graphical user interface that guides a user during constraint creation and maintenance. In Figure 6(b) we show a screenshot of our "wizard" that we integrated into the graphical modeling tool IBM Rational Software Architect (RSA), based on the UML profile defined in Eclipse/UML2.

In the upper left part of the window, the user can choose a constraint pattern. When a pattern is selected, a description of the pattern and its parameters are shown in the upper right part of the window. In the lower half of the window, the attributes of the pattern selected are shown



(a) Profile Editor (b) Constraint Wizard

Figure 5: Screen Shots of Eclipse Prototype

and attribute values can be entered. In its current state, the wizard implements one CIM-to-PIM transformation that generates OCL expressions and one CIM-to-PSM transformation that creates Java code for run-time model validation. Furthermore, the wizard can also be used to modify previously created structured constraints by invoking it from the context menu of a structured constraint. The user can then adapt the parameter values and regenerate the constraint in question.

6.2 Applying the Tool to the Example

We have argued in this paper that our approach helps to decrease development time and to reduce the rate of syntactic errors. To indicate the practicability of our approach, we revisit the example from Section 2 and apply our method to it. In particular, we use the constraint wizard prototype to implement Constraint 1 by choosing appropriate patterns specifying their parameters.

Constraint 1 is split into three parts, a quantification part, a predicate part and an assumption. This enables a divide-and-conquer approach for formalizing this constraint because each part can be defined separately. The parts can then be linked to form a complex constraint.

When implementing Constraint 1 using our approach, the class Manager is constrained by an instance of the *Exists* pattern. This instance has two parameters: the set of all employees (self.employs) and the quantification predicate, namely, the OCL constraint self.salary >= 3000, which is the only predicate in the properties of the *Exists* pattern instance. Furthermore, this constraint has an assumption that is an *AttributeValueRestriction* on the budget of the manager. Figure 6 shows a visualization of Constraint 1 in RSA.

We have already shown in Subsection 4.2 how Constraint 2 and Constraint 3 can be repre-

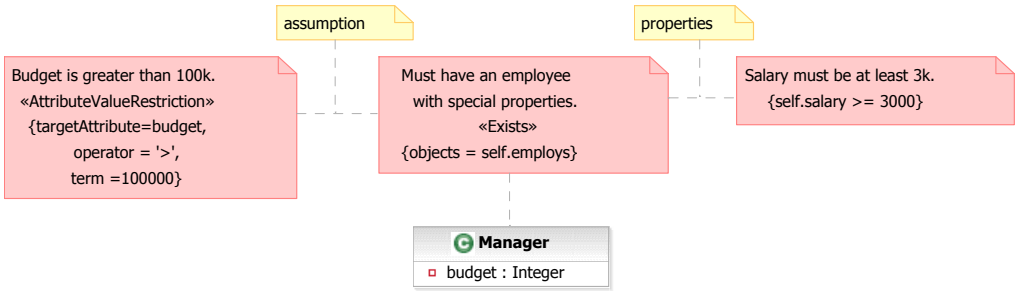


Figure 6: Structured Model of Constraint 1

sented using the *CyclicDependency* and the *PathDepthRestriction* pattern respectively. These two complicated constraints can thus be specified by simply providing a few parameter values each. If requirements change, these constraints can be quickly adapted without reading, adapting, and testing verbose expressions.

We believe that this small example already shows the practicability of our approach. Complicated recursive expressions are replaced by structured, concise, and easy-to-read constraint definitions. In addition, our model-driven approach enables the automatic generation of platform-independent or platform-specific constraints in various languages or modeling frameworks.

7 Related Work

The difficulty of developing concise and correct OCL constraints has been addressed in numerous publications. OCL is considered to be a very important formalism in today's modeling technologies, yet it is difficult to write correct, clear, and efficient OCL expressions [CBC05]. This paper supports the need of using textual constraints in model-driven development, and points out that tool support is critical for the success of OCL.

In [CCBC04], a list of recommendations is presented to improve correctness, clarity and efficiency of OCL expressions, two of which we consider especially important. Firstly, the authors advise to couple an OCL constraint with an informal specification for clarity. Our approach of model-driven constraint engineering, in which a concrete constraint is derived from a CIM constraint pattern, follows this idea. Secondly, it is advised to test constraints for syntactic errors. Our approach avoids syntactic errors by having predefined, syntactically correct templates and a set of meta-constraints for each pattern.

The concrete syntax of OCL has been made responsible for its low acceptance so far. Thus, several publications try to improve the syntax. For instance, a visual concrete syntax for OCL is proposed in [BKPT00], and a mathematical syntax is presented in [Süß06] and [BW06]. The structured constraint patterns we have presented can be regarded as another concrete constraint syntax. However, we elevate the syntax to a more abstract level, which we believe improves conciseness and correctness.

Wrong intuitions about the formal semantics of UML/OCL seem to be a common problem for users unfamiliar with formal specifications [Cab06]. We believe that our approach can help to replace wrong intuitions by precisely defined constraint expressions.

Several publications use the idea of constraint patterns, thus following the general idea of capturing domain knowledge and making it reusable, as for example introduced in [GHJV95] for object orientation. Patterns for *constraints* in model-driven development were first mentioned in [BHSS00], where one pattern—*Singleton*—is introduced. The idea of constraint patterns is further elaborated in [AT06, ABB+05], where a small number of constraint patterns is introduced along with OCL templates.

Two publications present a larger library of constraint patterns [MN05, CGQ+06]. The patterns presented there originate from the data–modeling domain, and partly overlap with the patterns introduced in this paper, even though they are named differently. Some patterns defined in this paper cannot be found in these two papers and vice versa.

Our contribution adds to these approaches in two ways. Firstly, our approach offers composite patterns and, moreover, allows a user to negate patterns and to combine existing patterns using implication. Thus, the user has a higher flexibility in using the patterns. Secondly, our approach is supported by a tool that integrates into existing CASE tools.

8 Conclusion and Future Work

In this paper, we have introduced the notion of *model-driven constraint engineering*. Our approach provides three main contributions for the efficient development and maintenance of concise UML/OCL specifications.

Firstly, we have introduced the notion of computation-independent patterns and transformations to concrete constraint expressions. This allows constraints to be represented in an abstract way, generating platform-independent expressions for precise documentation and platform-specific code for model validation. Secondly, we have introduced an extensible library of patterns. Our patterns originate from a case study in which we formalized about 80 constraints, and first discussions with modeling experts confirmed the relevance of our patterns. We have added a high degree of expressiveness to a pattern-based approach by adding logical structure and by classifying patterns into atomic and composite patterns. Thirdly, we have provided tool support for integrating the concepts of model-driven constraint engineering into a CASE tool. We have presented a wizard that integrates into IBM Rational Software Architect and supports a user in choosing and parameterizing a constraint pattern. Furthermore, the wizard contains transformations to platform-independent and platform-specific code.

We argue that our approach helps to decrease both the time and the error rate for constraint development. For instance, the OCL expression needed for Constraint 3 in Section 2 uses a recursive definition that is not easy to understand. In contrast to the lengthy and complicated OCL statement, the same constraint can be defined as an instance of the *PathDepthRestriction* pattern. Tool support as presented in Section 6 by an initial prototype further reduces the problem of defining a constraint by pointing-and-clicking to relevant model elements.

We would like to emphasize that although we have introduced a *wizard*, we cannot spirit away the complexity inherent in many constraints. However, we believe that our approach offers a powerful tool for dealing with this inherent complexity.

Future work includes the definition of new atomic and composite constraint patterns. We believe that more interesting constraint patterns can be identified in other application domains, such

as model transformations [HKSW05], ontology modeling [CP99] or model refactorings [GR02]. Furthermore, existing and future constraint patterns need to be validated with respect to their relevance in as many case studies as possible.

We are currently working on formalizing the constraint patterns in HOL-OCL [BW06], an interactive proof environment for UML/OCL. Having support for interactive reasoning has two advantages. Firstly, we can formally carry out proofs about the constraint patterns, e. g., redundancy between patterns or parameter values that result in unsatisfiable pattern instances. Secondly, these proofs help to increase the level of automation for consistency proofs of a UML/OCL specification, provided that constraint patterns are used in the specification.

Acknowledgements: We thank David Basin, Jochen Küster, Alexander Pretschner, and Ksenia Ryndina for their valuable feedback on earlier versions of this paper.

Bibliography

[ABB+05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P. H. Schmitt. The KeY Tool. *Software and System Modeling* 4(1):32–54, 2005.

[ÁRF03] J. A. T. Álvarez, V. Requena, J. L. Fernández. Emerging OCL Tools. *Software and System Modeling* 2(4):248–261, 2003.

[AT06] J. Ackermann, K. Turowski. A Library of OCL Specification Patterns to Simplify Behavioral Specification of Software Components. In *Proceedings of Conference on Advanced Information Systems Engineering*. LNCS 4001, pp. 255–269. 2006.

[Baa03] T. Baar. The Definition of Transitive Closure with OCL – Limitations and Applications. In *Proceedings, Fifth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*. LNCS 2890, pp. 358–365. Springer, July 2003.

[BDW06] A. D. Brucker, J. Doser, B. Wolff. A Model Transformation Semantics and Analysis Methodology for SecureUML. In Nierstrasz et al. (eds.), *Models 2006: Model Driven Engineering Languages and Systems*. LNCS, pp. 306–320. Springer-Verlag, 2006.

[BHSS00] T. Baar, R. Hähnle, T. Sattler, P. H. Schmitt. Entwurfgesteuerte Erzeugung von OCL-Constraints. *Softwaretechnik-Trends* 20(3), 2000.

[BKPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, G. Taentzer. Consistency Checking and Visualization of OCL Constraints. Pp. 294–308 in [EKS00].

[BW06] A. D. Brucker, B. Wolff. The HOL-OCL Book. Technical report 525, ETH Zürich, Switzerland, 2006.

[Cab06] J. Cabot. Ambiguity Issues in OCL Postconditions. In *Proceedings of the 6th OCL Workshop at the UML/MoDELS Conference 2006*. Pp. 194–204. 2006.

[CBC05] D. Chiorean, M. Bortes, D. Corutiu. Proposals for a Widespread Use of OCL. In Baar (ed.), *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*. Technical Report LGL-REPORT-2005-001, pp. 68–82. EPFL, Lausanne, Switzerland, 2005.

[CCBC04] D. Chiorean, D. Corutiu, M. Bortes, I. Chiorean. Good Practices for Creating Correct, Clear and Efficient OCL Specifications. In *Proceedings of NWUML'2004 – 2nd Nordic Workshop on the Unified Modeling Language*. Pp. 127–142. 2004.

[CGQ+06] D. Costal, C. Gómez, A. Queralt, R. Raventós, E. Teniente. Facilitating the Definition of General Constraints in UML. In Nierstrasz et al. (eds.), *MoDELS 2006*. LNCS 4199, pp. 260–274. Springer-Verlag, 2006.

[CLW+06] S.-K. Chen, H. Lei, M. Wahler, H. Chang, K. Bhaskaran, J. Frank. A Model Driven XML Transformation Framework for Business Performance Management Model Creation. In *International Journal of Electronic Business*. Volume 4(3/4), pp. 281–301. Inderscience, 2006.

[Coo00] S. Cook. The UML Family: Profiles, Prefaces and Packages. Pp. 255–264 in [EKS00].

[CP99] S. Cranefield, M. Purvis. UML as an Ontology Modelling Language. In *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence*. 1999.

[CW04] A. L. Correa, C. M. L. Werner. Applying Refactoring Techniques to UML/OCL Models. In Baar et al. (eds.), *UML*. LNCS 3273, pp. 173–187. Springer, 2004.

[ECL] The Eclipse UML2 Project. http://www.eclipse.org/uml2/.

[EKS00] A. Evans, S. Kent, B. Selic (eds.). *UML 2000*. LNCS 1939. Springer, 2000.

[EMF] The Eclipse Modeling Framework. http://www.eclipse.org/emf.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA, 1995.

[GR02] M. Gogolla, M. Richters. Expressing UML Class Diagrams Properties with OCL. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*. Pp. 85–114. Springer-Verlag, London, UK, 2002.

[HK04] R. Hauser, J. Koehler. Compiling Process Graphs into Executable Code. In *Third International Conference on Generative Programming and Component Engineering*. LNCS 3286, pp. 317–336. Springer, 2004.

[HKSW05] R. Hauser, J. Koehler, S. Sendall, M. Wahler. Declarative Techniques for Model-Driven Business Process Integration. *IBM Systems Journal* 44(1):47–65, 2005.

[KWB03] A. Kleppe, J. Warmer, W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[LBD02] T. Lodderstedt, D. A. Basin, J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Jézéquel et al. (eds.), *UML 2002*. LNCS 2460, pp. 426–441. Springer, 2002.

[MB05] S. Markovic, T. Baar. Refactoring OCL Annotated UML Class Diagrams. In *MODELS 2005*. LNCS 3713, pp. 280–294. 2005.

[MN05] E. Miliauskaitė, L. Nemuraitė. Representation of Integrity Constraints in Conceptual Models. *Information Technology and Control* 34(4):355–365, 2005.

[Obj02] Object Management Group (OMG). Meta Object Facility (MOF) Specification Version 1.4. April 2002. Available as OMG document formal/2002-04-03.

[Obj03] Object Management Group (OMG). UML 2.0 OCL Final Adopted Specification. 2003. Available as OMG document ptc/03-10-14.

[Obj05] Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0. July 2005. Available as OMG document formal/05-07-04.

[Obj06] Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.1. April 2006. Available as OMG document ptc/2006-04-02.

[RC81] R. Reiter, G. Criscuolo. On Interacting Defaults. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI'81)*, pp. 94–100, 1981.

[Süß06] J. G. Süß. Sugar for OCL. In *Proceedings of the 6th OCL Workshop at the UML/MoDELS Conference 2006*. Pp. 240–251. 2006.