IADIS International Journal on Computer Science and Information Systems
Vol. 8, No. 1, pp. 1-13
ISSN: 1646-3692

CONSTRAINT SPECIFICATIONS USING PATTERNS IN OCL

Ali Hamie. *University of Brighton, Brighton, UK*

ABSTRACT

Constraint patterns are very useful for specifying OCL constraints on UML class models. They potentially shorten the development time and reduce the errors for constraint development by providing predefined templates that can be instantiated in particular contexts. Constraint patterns can be identified by analyzing existing constraints for recurring expressions and abstracting from them. This paper extends the collection of published constraint patterns by identifying further patterns as well as making some improvements for existing patterns and their description. These are derived from an example model described in UML and augmented with OCL constraints.

KEYWORDS

Constraint, Pattern, UML, OCL, Specification

1. INTRODUCTION

In the Unified Modeling Language UML (OMG, 2011), model states can only be partially constrained by means of class diagrams. The modelling notation allows model developers to define classes, attributes, and operations. Properties and operations have types, and they may have multiplicity constraints, i.e., they can be sets with a predefined minimum or maximum size. However, the diagrammatic notation is not expressive enough to describe details that frequently occur in systems and need to be expressed in the model. Such limited expressiveness typically requires refinement with textual constraints. In order to express complex relations and restrictions in a model, the textual constraint language OCL has been introduced (Warmer & Kleppe, 2003) (OMG, 2012) as part of UML. OCL is based on three-valued logic with an explicit element denoting undefinedness and typed set theory. It provides basic data types and a library of collection types such as sets, bags and sequences. OCL can be used to specify *invariants* for classes, and *preconditions* and *postconditions* for operations. An invariant can be defined as a predicate that holds for all objects of the constrained class. In this paper, we use the terms invariant and constraint synonymously.

The development of constraint specifications is a difficult and error-prone task (Ackermann, 2005b) (Wahler, et al., 2006). This is partly due to the fact that class models can express complicated relations between concepts, including subtyping, reflexive relations, or potentially dealing with infinitely large instances, and specifying such facts requires addressing this complexity. The process of writing constraints can be simplified by using *constraint patterns* to shorten the development time and avoid syntax errors. A constraint pattern captures and generalizes frequently used logical expressions. It is a parameterizable constraint expression that can be instantiated to solve a class of specification problems. Constraint patterns have been introduced for object-oriented programming (Horn, 1992) and been adopted in the literature for UML/OCL (Ackermann & Turowski, 2006) (Wahler, et al., 2006) (Wahler, 2008) (Dolors, et al., 2006).

In this paper we use a model for a video rental store in order to identify additional constraint patterns and provide equivalent formulations of some existing constraint patterns. In particular we use a formulation that also provides the context for applying a constraint pattern. This makes it possible to formulate the conditions under which the pattern can be applied. Also some pattern formulations may be more useful for debugging purposes when checking the satisfiability of the constraints. In order to document the various formulations of a constraint pattern we extend the template used for describing constraint patterns by adding an optional clause named *alternative*.

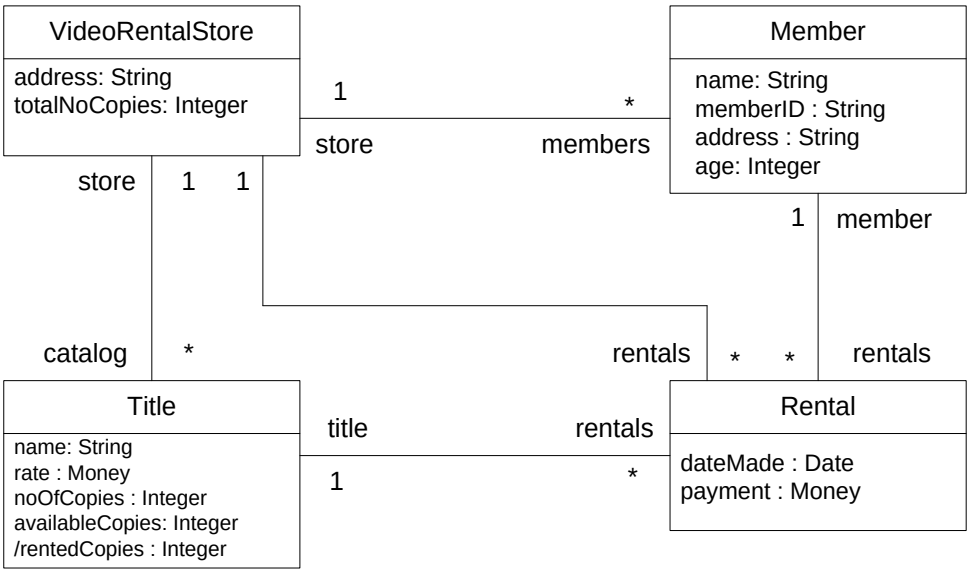

Figure 1. Class Diagram for Video Rental Store.

The remainder of this paper is structured as follows. In Section 2 we briefly explain OCL using an example and introduce constraint patterns. In Section 3 we present some constraint patterns based on the video rental store model. In Section 4 we provide a concluding remark.

2. BACKGROUND AND RELATED WORK

2.1 The Object Constraint Language

The Object Constraint Language (OCL) (Warmer & Kleppe, 2003) (OMG, 2012) is a textual, declarative notation that can be used to specify constraints or rules that apply to UML models. OCL plays an important role in model driven software development because UML diagrams are not precise enough to enable the transformation of a UML model to code. In fact, it is an essential component of OMG's standard for model transformation for the model driven architecture (Frankel, 2003).

A UML diagram alone cannot express all relevant constraints for an application. The diagram in Figure **1**, for example, is a UML class diagram modelling the services of a video rental store. The store has some members that can rent copies of video titles. The main classes are VideoRentalStore, Title, Member and Rental. The association between VideoRentalStore and Member indicates that a store has many members and a member belongs to exactly one store. The association between VideoRentalStore and Title indicates that a store has many titles and a title belongs to exactly one store. The association between Title and Rental indicates that a title has many rentals and a rental belongs to one title. The association between Member and Rental indicates that a member has many rentals and a rental is only for one member. However, the class diagram does not express the fact that the number of available copies of a title is less than or equal to the number of copies of that title. It is very likely that a system built based on class diagram alone will be incorrect. These additional constraints on objects and entities within a UML model can be precisely described using OCL. It is a textual language based on mathematical set theory and predicate logic. It supplements UML by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. The above constraint, for example, can be expressed in OCL as follows.

```
context Title
  inv: self.availableCopies <= noOfCopies
```

This constraint, called an *invariant*, states a fact that should always be true in the model. The context of the invariant is the class Title. The variable self refers to an instance of class Title.

It is also possible to specify the behaviour of an operation in OCL. For example, the following OCL constraints specify the behaviour of an operation Title::addCopy(n:Integer) using a pair of predicates called *preconditions* and *postconditions*.

```
context Title::addCopy(n: Integer)
 pre:  n > 0
 post: self.noOfCopies = self.noOfCopies@pre + n and
       self.availableCopies = self.availableCopies@pre + n
```

The above precondition and postcondition state that if invoked with n greater than zero, then the operation increases both the number of copies and available copies by n. In the postcondition, @pre refers to the value of the expression at the precondition time.

2.2 Constraint Patterns

In software design, patterns are used to identify recurring design problems, for which they provide a common description and solution in a given context. In software modeling the notion of *constraint patterns* was introduced to capture frequently occurring restrictions imposed on models. Wahler (Wahler, 2008) defined a *pattern* as a description of a generic solution to a recurring problem in a certain domain that can be reapplied to instances of the same problem.

There are several ways for describing constraint patterns. Wahler (Wahler, 2008) uses OCL templates and HOL-OCL functions to describe the specification patterns for OCL and to formalize their semantics. In that approach, the patterns are classified as *atomic* or *composite*. The atomic patterns are described as OCL templates and HOL-OCL functions. However, since composite patterns are higher order constructs (representing constraints over constraints) their semantics cannot be described naturally using OCL parameterized templates, as in the case with atomic patterns. Consequently, composite patterns are only described in terms of higher order functions in HOL-OCL.

In (Ackermann, 2005a) a detailed pattern description scheme is provided exposing all properties of a pattern: name, parameters, restrictions for pattern use, type, context and body of the resulting constraint. In order to keep the description of patterns consistent, this paper shall use the OCL template approach when introducing a specification pattern.

In (Wahler, 2008), a pattern named AttributeValueRestriction is defined in order to restrict the values of attributes. The template for representing the pattern is given as follows.

```
pattern AttributeValueRestriction  (property : Property, operator, value : OclExpression) =
  self.property  operator  value
```

The pattern has three parameters: property stands for the attribute that is to be restricted, operator and value which are used to restrict the value of the attribute. Such a pattern can be applied in the context of a class to generate an invariant by providing actual values for the parameters.

Using the above pattern, an invariant stating that the number of copies for a title is always greater than zero, can be stated as follows.

```
context  Title
  inv: AttributeValueRestriction(noOfCopies, >, 0)
```

That is the parameters property, operator and value have been replaced by the actual values noOfcopies, >, and 0 respectively.

3. ATERNATIVE FORMULATION FOR CONSTRAINT PATTERNS

In this section we consider alternative formulations for some constraint patterns that can be useful for testing and debugging the model. For this purpose, the template for describing patterns is extended by adding an alternative clause that provides a semantically equivalent expression to the body of the pattern. In this way one description may concentrate on the clarity of the expression defining the pattern while the other may be useful for testing the model.

3.1 Restricting the Multiplicity of Associations

The multiplicities of properties (associations) can only be roughly constrained in a diagrammatical way in class models. However, there are situation where the multiplicity of an association depends on the value of an attribute. For example, an object of class Title can have an arbitrary number of rentals which cannot exceed the total number of copies for that title. So there is a dependency between the association with role name rentals and the attribute noOfCopies. Here we are assuming that the model deals with current rentals rather than past rentals. The following OCL constraint captures this dependency.

```
context Title inv RentalsRestriction:
 self.rentals->size() <= noOfCopies
```

A constraint pattern named MultiplicityRestriction is defined in (Wahler, 2008) to capture this kind of constraints. This pattern is presented as an OCL template as follows.

```
pattern MultiplicityRestriction (navigation : Sequence(Property), operator : OclExpression,
                                 value : OclExpression) =
  self.navigation->asSet()->size() operator value
```

This pattern has three parameters: navigation, represented as a sequence of properties, thus allowing the use of OCL navigation expressions such as self.catalog.rentals, operator, and value, which can be arbitrary OCL expressions. value can be the name of an attribute or an arbitrary OCL expression. Since self.navigation may result in a bag, the OCL operator asSet() is used to convert the resulting collection into a set.

Using the MultiplicityRestriction pattern, we can define the constraint RentalsRestriction as follows.

```
context Title inv RentalsRestriction:
 MultiplicityRestriction(rentals, <=, noOfCopies)
```

This is done by replacing the parameters navigation, operator and value by rentals, <=, and noOfCopies respectively.

The constraint patterns as defined in (Wahler, 2008) leaves the context class implicit. This makes it difficult to state the conditions for applying the pattern. A variant of this pattern is obtained by including the context class as a parameter.

```
pattern MultiplicityRestriction (class: Class, navigation : Sequence(Property),
                                 operator: OclExpression, value:OclExpression) =
  self.navigation->asSet()->size() operator value
```

The class parameter does not appear in the expression defining the pattern, however it can be used to state the meta-level conditions for applying the pattern, i.e. the first property in navigation belongs to instances of class. In some cases the class parameter may be used in the body of the pattern, in particular when the operation allInstances is used.

For invariants this formulation of the pattern can be regarded as equivalent to the following where the parameters or placeholders of the template are written with angle brackets (<>). The place holders inside the pattern definition are identified with actual type names and properties when the pattern is applied.

```
context <class>
  inv: self.<navigation>->asSet()->size() <operator> <value>
```

3.2 Unique Identification

The unique identification constraint occurs very frequently. In the video rental store model it is required that the ID for members is unique. That is any two members, m1 and m2 should be distinguishable by their membership identities. In OCL such constraint is expressed as follows.

```
context Member
 inv UniqueID: Member.allInstances->isUnique(memberID)
```

This constraint can be generalized to composite primary keys by using the OCL tuple type.

The *Unique Identifier* pattern (Wahler, 2008) (referred to *Semantic Key* in (Ackermann & Turowski, 2006)) captures the situation where an attribute (or a group of attributes) of a class plays the role of an identifier for the class. That is the instances of the class should differ in their values for that attribute (group). The corresponding OCL template as defined in (Wahler, 2008) is given as follows.

```
pattern UniqueIdentifier (property : Tuple(Property)) =
  self.allInstances->isUnique(property)
```

This pattern has one parameter property, which denotes a tuple of properties that have to be unique for each object of the context class. However, self.allInstances is not well defined in OCL since the operation allInstances applies to classes only. This can be overcome by taking the context class as a parameter for the pattern. Thus the pattern is to be instantiated in the context of class.

```
pattern UniqueIdentifier (class: Class, property : Tuple(Property)) =
 class.allInstances->isUnique(property)
```

In order to ensure the syntactic correctness of the resulting OCL expression the following conditions are needed. The pattern application should be performed in the context of class, and that the properties in the tuple should be among the attributes of class.

The constraint that requires instances of the Member class are uniquely identifiable by their ID, can be expressed using the Unique Identifier pattern as follows:

```
context Member
 inv  UniqueID: UniqueIdentifier(memberID)
```

The Unique Identifier pattern provides a *global* uniqueness since it refers through the OCL operation allInstances to all instances of the class. However, there are situations where the constraint may state that each instance of a class accessible starting from a given collection should be uniquely identifiable by the value of a particular attribute. For example the constraint UniqueID can be expressed as follows.

```
context VideoRentalStore
 inv  UniqueID:  self.members->isUnique(memberID)
```

In this case we navigate from the class VideoRentalStore to get the collection of members for that store. That is the members within a video store may have unique identities. A variant of the Unique Identifier pattern can be defined as the following OCL template.

```
pattern UniqueIdentifier (navigation: Sequence(Property), property : Tuple(Property)) =
  self.navigation->isUnique(property)
```

Using this pattern we can express the uniqueness constraints for members as follows.

```
context VideoRentalStore
 inv  UniqueID:  UniqueIdentifier(members,memberID)
```

The formulation of the pattern using the operation allInstances can be problematic since the operation is not defined on the basic types such as Integer. This can be overcome by having a class representing the system and navigating from this class to obtain all the existing instances of the other class as shown above.

Some constraint patterns may have different formulations with some more suitable than others for testing and debugging the model. For this we extend the template for describing constraint patterns by adding a new clause as follows.

```
pattern  UniqueIdentifier  (class: Class, property : Tuple(Property)) =
    class.allInstances->isUnique(property)
   alternative:  class.allInstances->select(m | m.property = self.property)->size() = 1
```

The constraint in the alternative clause provides better support for debugging as it will provide information about those objects that violates the constraint rather than just returning *true* or *false*.

Another useful clause to include for the patterns is a ‘derive’ clause which states some consequences that follow from the definition of the pattern. This can be used for reasoning about the model.

3.3 Restricting Attribute Values

The values of attributes of one or more classes cannot be related to each other using the diagrammatic modeling language. In this subsection, we illustrate this by an example.

Attribute Sum Restriction. The class model of the video rental store has an attribute totalNoCopies representing the number of copies in the store. The number of copies in the store is the sum of the number of copies of all titles in that store. Therefore, the attributes totalNoCopies and noOfCopies must be related. However such relations cannot be modelled in terms of the UML meta-model. Using OCL we can write the following constraint.

```
context VideoRentalStore
 inv AllCopies: self.totalNoCopies = self.catalog.noOfCopies->sum()
```

A constraint pattern named AttributeSumRestriction is defined in (Wahler, 2008) to capture similar kind of constraints. However, this pattern is defined using only the comparison operator <=. This pattern is presented as an OCL template as follows.

```
pattern AttributeSumRestriction(navigation: Sequence(Property),
                                summand: Property, summation: Property) =
 self.navigation.summand->sum() <= summation
```

In order to express the above constraint we extend the AttributeSumRestriction pattern by adding a new parameter for the comparison operator as follows.

```
pattern AttributeSumRestriction(navigation: Sequence(Property), summand: Property,
                                operator: OclExpression, summation: Property) =
 self.navigation.summand->sum() operator summation
```

This pattern has four parameters. navigation represents a path expression to a related class, summation refers to the property in the context class that denote the value that provides a limit, and summand refers to the property in the related class that is accumulated, operator denotes a comparison operator. The original pattern defined in (Wahler, 2008) uses <= for the operator and does not include the parameter operator. This definition provides more flexibility as the operator can be taken as =, <= or >=.

Employing this constraint pattern, the constraint AllCopies can be expressed in a more concise way as follows:

```
context VideoRentalStore inv AllCopies:
 AttributeSumRestriction(catalog, noOfCopies, =, totalNoCopies)
```

Attribute Relation. A simple form of this constraint is when two properties can be related by a binary operator such as less-than or equal (<=). For example the value of attribute noOfCopies is always less than or equal to totalNoCopies. The following OCL constraint describes such relationship.

```
context VideoRentalStore inv lessCopies:
 self.catalog->forAll(t | t.noOfCopies <= totalNoCopies)
```

The derived constraint pattern from this constraint is given by (Wahler, 2008) as follows:

```
pattern AttributeRelation(navigation: Sequence(Property), remoteAttribute: Property,
                          operator: OclExpression, contextAttribute: Property)=
  self.navigation->forAll(x | x.remoteAttribute  operator  contextAttribute)
```

Using this constraint pattern, the constraint lessCopies can be expressed as follows:

```
context  VideoRentalStore  inv  lessCopies:
 AttributeRelation(catalog,noOfCopies, <=, totalNoCopies)
```

The constraint pattern can be modified by an additional argument in order to indicate whether the whole constraint will be negated or not. This results in the following pattern.

```
pattern AttributeRelation(navigation: Sequence(Property), remoteAttribute: Property,
                operator: OCLExpression, contextAttribute: Property, neg : Boolean) =
  let  b: boolean  = self.navigation->forAll(x | x.remoteAttribute  operator  contextAttribute)
  in if  not(neg)  then  b  else  not(b)
```

3.4 Commutativity Constraints

For the video rental store model (Figure **1**), there are three possible ways to find all the rentals. One way is to navigate using the association between the VideoRentalStore and the Member classes, and then navigate along the association between Member and Rental. The other way is similar but the navigation is via the associations between VideoRentalStore and Title, and Title and Rental respectively. These two ways should result in the same collection. Thus we have the following OCL constraint.

```
context VideoRentalStore
 inv AllRentals: self.catalog.rentals = self.members.rentals
```

This constraint can be generalized with the following NavigationCommutativity pattern defined as an OCL template.

```
pattern NavigationCommutativity(navigation1, navigation2 : Sequence(Property)  =
  self.navigation1 = self.navigation2
```

This pattern has two parameters.navigation1 and navigation2 represent paths expression to related classes respectively.

Using this constraint pattern, the constraint AllRentals can be expressed as follows:

```
context  VideoRentalStore  inv:
 NavigationCommutativity(Sequence{catalog, rentals}, Sequence{members, rentals})
```

There are situations where the values of the navigation expressions results in values of different collection types. A typical example is where one navigation expression results in a set and the other expression results in a bag. In this case the OCL operator asSet can be used to convert a bag into a set. This leads to the following formulation of the pattern.

```
pattern  NavigationCommutativity(navigation1, navigation2 : Sequence(Property)   =
  self.navigation1->asSet() = self.navigation2->asSet()
```

This also works in cases where the result of one navigation expression is a bag and the result of the other is a set.

The formulation of the commutativity pattern is useful for debugging the model. That is when the constraint fails (returning *false*) one can use a tool for evaluating the two expressions and compare their results for identifying the objects causing the problem.

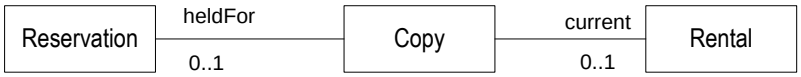

Figure 2. Exclusive Associations.

3.5 Exclusive Property Constraints

A common constraint is where one object of class A can be linked to an object of class B or to an object of class C but not both. For example if we include copies of titles as well as reservations then the following diagram in Figure **2** may be part of the video rental store class model. The constraint in this context is that a copy is on hold for a reservation or is rented but not both. The OCL invariant that captures this constraint is given as follows.

```
context Copy inv :
 self.heldFor->notEmpty()  xor  self.current->notEmpty()
```

Since the associations are optional (partial) we can check for a link by treating the navigation expressions as sets and applying the OCL operation notEmpty.

This constraint can be generalized with the following ExclusiveProperty pattern defined as an OCL template.

```
pattern ExclusiveProperty(property1: Property,  property2: Property) =
  self.property1->notEmplty()  xor   self.property2->notEmplty()
```

Using this constraint pattern, the above exclusive property constraint can be expressed as follows:

```
context  Copy inv:
  ExclusiveProperty(heldFor, current)
```

3.6 Preconditions and Postconditions Constraints

Most of the existing OCL specification patterns have been related to the use of invariants for specifying constraints patterns. This is due to the fact that reasoning about the model mainly involves the static aspects which require the specification and evaluation of invariants. However, constraint patterns may be useful for preconditions and postconditions, in particular those constraints that prevent the breaking of invariants. For example, the uniqueness

constraint imposed on the ids of members within the video rental store can be prevented from breaking by means of suitable precondition and postcondition pair for the operations that may violate this constraint.

Consider the operation enrolls which adds a new member to the video store. The corresponding OCL specification of enrolls is given below.

```
context VideoRentalStore::enrol(name: String, id : String)
 pre:  self.members.memberID->excludes(id)
 post: self.members->exists(m : Member | m.oclIsNew() and
                                         m.name = name and
                                         m.memberID = id)
                and  self.members->size() = self.members@pre->size()+1
```

The precondition states that the parameter id is not yet used as a unique identity for existing members, and the postcondition states that a new object of class Member was created with the attribute memberID equals to id and with attribute name equals to name. The size of the new members set after executing the operation is increased by one.

From the precondition the constraint pattern NotInBag can be derived. This pattern takes two parameters, one is a bag and the other is an element, and asserts that the element is not in the bag. It is defined by the following template.

```
pattern NotInBag(bag: OclExpression, e: OclExpression) =
   bag->excludes(e)
 alternative: bag->count(e) = 0
```

In the pattern we added a clause that provides an alternative formulation of the body of the pattern. In this case the two expressions bag->excludes(e) and bag->count(e)=0 are semantically equivalent. This will be useful because some formulations may be better for readability purposes while others are better for testability purposes.

Using the pattern NotInBag the precondition of the operation enroll can be specified as follows.

```
pre: NotInBag(members.memberID, id)
```

For the postcondition we have the following pattern.

```
pattern NewObject(navigation: Sequence(Property),
                  attribute1, attribute2: Property, value1, value2: OclExpression) =
 self.navigation->exists(o | o.oclIsNew() and
                             o.attribute1 = value1 and
                             o.attribute2 = value2)  and
 self.navigation->size() = self.navigation@pre->size()+1
```

The pattern NewObject takes five parameters. navigation represents a path expression to a related class which evaluates to a set. attribute1 and attribute2 represents the properties of the new object to be set to the values value1 and value2 respectively. For simplicity, this pattern is defined with two properties (attributes), however, for the general case the pattern would be defined by taking a list of properties and a list of corresponding values as parameters. An auxiliary operation can be easily defined to set the values of the properties using the two lists.

Using the pattern NewObject the postcondition of the operation enroll can be specified as follows.

```
post: newObject(members, memberID, name, id, name)
```

The operation enroll creates a new member object and sets its attributes to some values provided as parameters. However if the object is created beforehand then one can define an operation addMember that takes the member as a parameter and adds it to the collection of members. The following is an OCL specification for addMember.

```
context VideoRentalStore::addMember(m: Member)
 pre:  self.members->excludes(m)
 post: self.members->includes(m)
```

In this case new patterns can be defined that correspond to an element in a collection and an element not in a collection.

4. CONCLUSION

In this paper, we have introduced additional constraints patterns that can be used for the development of constraint specifications in UML and OCL. In particular we have adapted the representation of some constraint patterns so that the conditions for applying them can be formalized. This is done by including the context of the constraint as a parameter for the pattern. This is also needed in contexts where the OCL operation allInstances is used. We have also presented two variants of the unique identification pattern. The template used for describing specification patterns was extended by adding a new clause that provides an alternative formulation of the pattern. This is useful since some constraints may have various formulations some of which provide better support for checking and debugging their satisfiability within the model. Further research will identify further constraint patterns for invariants, preconditions and postconditions, incorporate these new constraints within tools and using an appropriate tool for the instantiation of proposed patterns.

REFERENCES

Ackermann, J., 2005a. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Component. In Thomas Barr (ed.). *Proceedings of the MoDELS' 05 Conference Workshop on Tool Support for OCL and Related Formalisms- Needs and Trend,* Technical Report LGL-REPORT-2005-001, EPFL, pp. 15-29.

Ackermann, J., 2005b. Frequently Occurring Patterns in Behavioral Specification of Software Components. In Klaus Turowski and Johannes Maria Zaha (ed.), *COEA 2005*, 41–56. GI, Erfurt, Germany.

Ackermann, J. & Turowski, K., 2006. A Library of OCL Specification Patterns for Behavioral Specification of Software Components. *Lecture Notes in Computer Science,* pp. 255-272.

Dolors, C. et al., 2006. Facilitating the Definition of General Constraints in UM*L. Lecture Notes in Computer Science*, Springer, pp. 260-274.

Frankel, D. S., 2003. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley.

Horn, B., 1992. Constraint patterns as a basis for object oriented programming. *Proceedings of the European Conference on Object-oriented programming systems, languages, and applications (OOPSLA'92)*. Vancouver, British Columbia, Canada, ACM, pp. 218-233.

OMG, 2011. *Unified Modeling Language (UML) Superstructure Version 2.4.1*.

OMG, 2011. *Unified Modeling Language Infrastrucure Version 2.4.1*.

OMG, 2012. *Object Constraint Language (OCL) Version 2.3.1*.

Wahler, M., 2008. *Using patterns to develop consistent design constraints*, PhD dissertation, ETH Zurich, Switzerland.

Wahler, M., Koehler, J. & Brucker, D. A., 2006. Model-Driven Constraint Engineering. *Electronic Communications of the EASST*.

Warmer, J. & Kleppe, A., 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*. Reading, MA: Addison-Wesley.