MDE-driven OCL Specification Patterns

Dan Chiorean* Vladiela Petraşcu*
Ileana Ober**

**Babeş-Bolyai University, Cluj-Napoca, Romania (e-mail: {chiorean, vladi}@cs.ubbcluj.ro)*
*** Université Paul Sabatier, Toulouse, France (e-mail: ileana.ober@irit.fr)*

Abstract: Detailed and unequivocal model specifications are a prerequisite for attaining the automated software development goal as promoted by the Model Driven Engineering (MDE) paradigm. The use of assertions, as promoted by the Design by Contract approach, assists in creating such model specifications. However, writing from scratch a large amount of assertions can be tedious, time-consuming, and error-prone. Consequently, a number of *constraint patterns* have been identified in the literature, and corresponding OCL specifications have been proposed. Automating their use in tools should speed the writing task and increase its correctness. Yet, no attention has been paid to the influence of such specifications in the area of error detection and diagnosis. We approach this topic by proposing new *OCL specification patterns* for some of the existing constraint patterns. Our proposal should increase the efficiency of testing and debugging processes performed for models and applications. Relevant examples and tool-support are used in order to explain and validate our approach.

Keywords: OCL, constraint patterns, MDE, testing, model compilability, model testing

1. INTRODUCTION

Nowadays, the large scale use of modeling is undoubtedly the most promising approach to software development automation. The recently emerged paradigm that promises attainment of such a goal is referred as MDE (Model Driven Engineering) (Schmidt 2006). However, this new approach can only deliver its promises when provided with means of creating detailed and rigorous models. Such models can be achieved by complementing the graphical modeling languages with appropriate assertion languages, such as OCL (Object Constraint Language (OMG 2010a)) for the MOF (Meta Object Facility (OMG 2006)) based family of modeling languages. The purpose and value of using assertions in software specification have been argued, among others, by the *Design by Contract* approach (Meyer 1997).

The emergence of MDE has triggered a major shift with respect to the usage of models and assertions in software development. Traditionally, modeling has been primarily targeted at facilitating problem understanding, assisting the client-developer communication, and guiding a mostly-manual implementation process. In this context, the purpose of writing model assertions (pre/post-conditions and invariants) was threefold. Firstly, they were meant to assist in writing correct programs, their explicit definition being regarded as a precondition of their enforcement in software. Secondly, they were the "oracles" against which to mentally assess software correctness during program testing. Thirdly, they were used as documentation artifacts, along with the complemented models.

With the advent of MDE however, models are upgraded from helpers to "first class citizens" of software development. Consequently, the emphasis is now laid on the creation of comprehensive correct models, intended to be automatically turned into code. Moreover, models are used at various abstraction levels (user-model, metamodel, meta-metamodel), reflected in metamodeling architectures. These new requirements of MDE call for new means of using assertions. Ensuring model correctness requires model compilability checks and model testing; the former activity is based on the evaluation of metamodel-level assertions (called Well Formedness Rules or WFRs), while the latter involves the evaluation of user model-level assertions (called Bussines Constraint Rules of BCRs). Moreover, following a natural MDE process, the model-level assertions should be turned into program-level routines, along with the model itself. This provides for runtime assertion monitoring, enabling the automatic use of assertions in program testing (as opposed to the traditional mental reasoning), as well as an aid to debugging and the creation of fault tolerant systems (Meyer 1997).

We claim that these new means of using assertions should enforce a change with respect to the style in which assertion specification is done. While complying with the role of assertions in traditional software development required them to be specified in the shortest, most intuitive manner, conformance to the requirements of MDE demands specifications to facilitate efficient error detection and diagnosis.

Widespread use of constraints has triggered the identification of several constraint patterns. However, the existing solutions to specify them do not seem targeted at fulfilling the aforementioned conditions. Through this paper, we aim at filling this gap, by proposing OCL specification patterns

driven by the MDE requirements on using assertions. Our proposal has a great automation potential, and, as such, it does not come with the price of penalizing constraints' readability.

The rest of the paper is organized as follows. Section 2 sets the context by introducing constraint patterns and summarizing related work that we rely on. Our specification approach is detailed in Section 3 and validated in Section 4. The paper ends with conclusions and hints on future work in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 Constraint Patterns

In Software Engineering, *patterns* are used to identify recurring development problems, for which they provide a common description and solution in a given context.

***Definition* [*Pattern*].** A *pattern* describes a generic solution to a recurring problem in a certain domain that can be reapplied to instances of the same problem (Wahler 2008).

In software modeling, *constraint patterns* (Wahler et al. 2006) may be used to capture frequently occuring restrictions imposed on models. This paper is focused on describing and specifying such constraint patterns. We work with UML (Unified Modeling Language (OMG 2010b)) as the model specification language and we use OCL for specifying constraints. Throughout the paper, we will clearly distinguish among the concept of *constraint pattern* and that of *OCL specification pattern*, although the phrases appear to be used interchangeably in the literature (see (Ackermann 2005b); (Wahler 2008)). For the purpose of this work, a *constraint pattern* denotes a logical constraint (restriction) on a model, while an *OCL specification pattern* refers to a proposed way of specifying constraints, patterns included. We describe solutions to the *constraint patterns* of interest as *OCL specification patterns*.

We will mainly focus on three constraint patterns that have been identified in the literature - *Attribute Value Restriction*, *Unique Identifier*, and *For All*. In a pattern taxonomy proposed by Wahler (2008), the first two are classified as *atomic* or *elementary* constraint patterns, while the last one is regarded as a *composite* constraint pattern. Elementary patterns abstract basic constraints on a model, while composite patterns allow expressing arbitrarily complex restrictions, by integrating any number of constraints, either atomic or composed (Wahler 2008). *Attribute Value Restriction* (Wahler 2008) is a basic pattern used to express various constraints on the value of a given class attribute. *Unique Identifier* (Wahler 2008) captures the situation in which an attribute (a group of attributes) of a class plays the role of an identifier for the class, i.e. the class' instances should differ in their value for that attribute (group). This is probably the best known constraint pattern, being referred under different names in the literature - *Semantic Key* by Ackermann (2005a), *Primary Identifier* by Miliauskaite et al. (2005), or simply *Identifier* by Costal et al. (2006). Finally, the *For All* constraint pattern requires every object of a certain collection to fulfill a number of specified restrictions.

In the following subsection, we present existing OCL specification patterns for the three mentioned constraint patterns. The OCL specification patterns are given in terms of OCL parameterized templates, as described by Wahler (2008). Using this representation, each OCL specification pattern is a template (macro) that depends on a number of parameters which are typed by elements from the UML and OCL metamodels. Replacing them with actual model elements generates a pattern instantiation. In order to keep consistency and allow an easy comparison of our work with existing related approaches, we will use the same representation means (OCL templates) when introducing our proposals in Section 3.

2.2 Existing Approaches to Specifying Constraint Patterns

Wahler (2008) uses OCL templates and HOL-OCL functions to represent the OCL specification patterns and to formalize their semantics. The atomic patterns are described both ways. However, since composite patterns are higher order constructs, representing constraints over constraints, their semantics cannot be expressed naturally using OCL parameterized templates, as in case of atomic patterns. Therefore, composite patterns are only described in terms of HOL-OCL.

Ackermann (2005a) introduces a detailed pattern description scheme exposing all properties of a pattern: name, parameters, restrictions for pattern use, as well as type, context and body of the resulting constraint.

Since we want to keep all descriptions consistent and intelligible, we will only use OCL templates when introducing the OCL specification patterns of interest from the two mentioned papers. Therefore, the specification pattern for *Semantic Key* given by Ackermann (2005a) will be translated into this representation, without altering its core semantics. We also provide an OCL template sketch for the composite *For All* pattern from (Wahler 2008). All the other specifications faithfully reproduce the ones in (Wahler 2008).

In (Wahler 2008), the following template representation is given for the OCL specification pattern associated to *Attribute Value Restriction*.

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

This template depends on three parameters (given in italics), namely *property* - which stands for the attribute that is to be constrained, *operator*, and *value* - which are used to restrict the attribute's value. Such a pattern may be instantiated to generate an invariant in the context of an UML class, by providing as actual parameters an attribute of the class, a concrete operator and a value expression.

Following, there is the OCL template for *Unique Identifier*, as proposed in (Wahler 2008) and (Wahler et al. 2006). The template employs one parameter, *property*, standing as a

placeholder for any tuple of class attributes on which we want to impose the uniqueness constraint.

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

The specification given in (Ackermann 2005a) for the same pattern matches the following OCL template. The template uses two parameters, *class* and *property*, the first standing for the class on whose instances we want to impose the uniqueness constraint, and the second for the attribute under consideration (whose values should be unique).

```
pattern SemanticKey(class:Class, property:Property)=
 class.allInstances()->forAll(i1, i2 | i1 <> i2 implies
                              i1.property <> i2.property)
```

Finally, we give below the OCL template-like description for the specification pattern associated to the *For All* constraint pattern.

```
pattern ForAll(collection:OclExpression,
               properties:Set(OclExpression))=
 collection->forAll(y | oclAND(properties, y))
```

The template-like description above uses two parameters. The first one, *collection*, denotes the collection of model elements which is iterated. The second parameter, *properties*, stands for the set of constraints that should be fulfilled by all objects in the collection. Each such constraint is, in fact, an instantiation of some constraint pattern. Since the OCL standard does not include a construct for expressing the conjunction of an arbitrary number of boolean expressions, we introduce the `oclAND` notation in this purpose. Consequently, `oclAND`(*properties*, *y*) denotes the expression resulting from the conjunction of the boolean expressions obtained by replacing `self` with `y` in each of the constraints from *properties*.

3. PROPOSED OCL SPECIFICATION PATTERNS

The majority of OCL specifications found in the literature (those for constraint patterns included) are focused exclusively on the clearness of expressions. During testing and debugging however, the mere information that a system state is inconsistent or that a method pre/post-condition is not fulfilled is not enough. Identifying the exact failure reasons is of utmost importance for error correction. This is the core-idea of the OCL specification approach that we promote. In this respect, in the following we give improvements of existing OCL specification patterns, considering both the case of invariants and that of pre/post-conditions.

3.1 The case of invariants

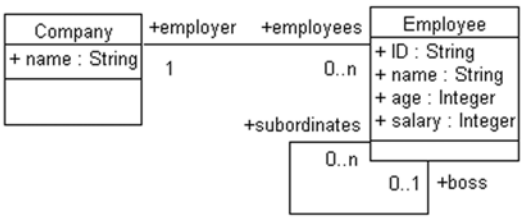

Fig. 1. A sample class model.

1) *The "For All" constraint pattern*: Let us consider the UML model in Figure 1, together with a business constraint rule stating that "All employees of a company should be aged at most 65". We will refer to this particular constraint as *Complying with Retirement Age Limit* (*CRAL*) in the following.

Most of the OCL specifications encountered in the literature for such a constraint consist of an invariant of the following shape.

```
context Company inv CRAL_E:
 self.employees->forAll( e | e.age <= 65)
```

The specification above is quite compact and intelligible, due to the fact that the OCL `forAll()` operation directly corresponds to the mathematical logic's universal quantifier $(\forall)$. However, such an invariant shape is not the most useful in a testing/debugging-related context. The feedback it offers in case of failure is tool-dependent, consisting in either a simple `false` message, or an implementation-dependent tree showing the result of evaluating the age constraint on each employee of the company. In the former case, we have no useful hint regarding the identity of those employees which are over the age limit. In the latter, the task of searching within the employees tree may be disturbing and time-consuming. This turns the debugging of an object model containing a large number of employees into a difficult task.

As an answer to the above problem, we suggest using any of the following two specifications. These are convenient not only when modeling, but also at runtime, allowing a user to easily get the information he needs in fixing a possible error.

```
context Company inv CRAL_P1:
 self.employees->reject(e | e.age <= 65)->isEmpty()

context Company inv CRAL_P2:
 self.employees->select(e | e.age > 65)->isEmpty()
```

As proved in the Appendix, `CRAL_E`, `CRAL_P1`, and `CRAL_P2` are semantically equivalent. However, the newly proposed invariants have the advantage of allowing evaluations of the `reject()`/`select()` subexpressions, which return exactly the set of employees violating the age constraint. A proof of this will be provided in Subsection 4.1, using the OCLE tool (LCI 2005).

A basic reasoning on the kind of restriction imposed by *CRAL* leads to the conclusion that this particular constraint is, in fact, an instantiation of the *For All* composite constraint pattern described in Subsection 2.1. The constraint used by the composite is, at its turn, an instantiation of an atomic constraint pattern, namely *Attribute Value Restriction*. Furthermore, the OCL specification employed by the `CRAL_E` invariant (a widely-encountered style in the literature) is an instantiation of the `ForAll` specification pattern, as proposed in (Wahler 2008) (see Subsection 2.2). The latter uses an instantiation of the `AttributeValueRestriction` specification pattern proposed by the same reference. In light of these statements, the `CRAL_E` invariant reads as follows.

```
context Company inv CRAL_E:
 ForAll(self.employees,
        Set{AttributeValueRestriction(age,<=,65)})
```

Above, we have emphasized the benefits gained by replacing the CRAL_E specification with its CRAL_P counterparts. In order to be able to exploit those benefits in case of all constraints instantiating the *For All* composite pattern, we propose the following two equivalent OCL specification patterns for it.

```
pattern ForAll_Reject(collection:OclExpression,
                      properties:Set(OclExpression))=
collection->reject(y | oclAND(properties,y))->isEmpty()

pattern ForAll_Select(collection:OclExpression,
                      properties:Set(OclExpression))=
collection->select(y | not oclAND(properties,y))->isEmpty()
```

In this context, it is obvious that the CRAL_P1 and CRAL_P2 specifications are instantiations of the ForAll_Reject and ForAll_Select specification patterns respectively, as shown below.

```
context Company inv CRAL_P1:
 ForAll_Reject(self.employees,
               Set{AttributeValueRestriction(age,<=,65)})
context Company inv CRAL_P2:
 ForAll_Select(self.employees,
               Set{AttributeValueRestriction(age,<=,65)})
```

As previously stated, we claim that the use of the newly proposed OCL specification patterns (ForAll_Reject and ForAll_Select) has a significant impact on the efficiency of testing/debugging activities, by increasing it in case of large collections of objects.

2) *The "Unique Identifier" constraint pattern*: There are basically two possible contexts for applying the *Unique Identifier* constraint pattern, although little emphasis has been put on distinguishing among them in the literature. One of them refers to the so-called "global" uniqueness - certain models or applications may require all possible instances of a class to differ in their value for a particular attribute. The other captures a "container-relative" uniqueness - a model/application constraint may state that each instance of a class accessible starting from a given "container" should be uniquely identifiable by the value of a particular attribute among all instances of the same class from within the same container. The existing OCL specification patterns for the *Unique Identifier* constraint pattern (those reproduced in Subsection 2.2) concern the "global" uniqueness case exclusively. Therefore, they should not be used for the "container-relative" one, as generally done in the literature. Moreover, the existing specification patterns have drawbacks, even when used in the appropriate context. In the following, we will analyze both uniqueness cases and propose appropriate OCL specification patterns for each.

"Global" uniqueness case (*GUID*): Let us consider a census application whose model contains a Person class. Suppose this class has an ID attribute and there is a business constraint stating that persons should be uniquely identifiable by their IDs. This is a classical case of a "global" uniqueness constraint. For such a constraint, the majority of OCL specification proposals in the literature have one of the following shapes.

```
context Person inv GUID_E1:
Person.allInstances()->forAll(p,q| p<>q implies p.ID<>q.ID)

context Person inv GUID_E2:
Person.allInstances()->isUnique(ID)
```

It may be easily noticed that these are instantiations of the SemanticKey and UniqueIdentifier specification patterns, as reproduced in Subsection 2.2. Therefore, they can be written as:

```
context Person inv GUID_E1: SemanticKey(Person, ID)
context Person inv GUID_E2: UniqueIdentifier(Tuple{x=ID})
```

As given above, these specifications have two drawbacks, which have been confirmed by the tests that we have performed using OCLE.

1) The worst is that GUID_E1 breaks the semantics of invariants, as promoted by Design by Contract. To acknowledge this, consider the case when at least two persons have the same ID value. Then, the evaluation of this invariant would return false for ALL persons (even for those having an unique ID). According to the semantics of an invariant, this should evaluate to false only for those instances violating the constraint that it formalizes. Such a semantics is clearly disregarded by GUID_E1 invariant. This drawback is triggered by the use of allInstances() uncorrelated with the contextual instance.

2) Due to the use of forAll() and isUnique() respectively, the two specifications do not provide appropriate debugging support.

Since we have shown that the two specifications are, in fact, instantiations of the OCL specification patterns UniqueIdentifier and SemanticKey, it logically follows that the OCL patterns themselves are incorrect. As a replacement of GUID_E1 and GUID_E2, we propose the following invariant

```
context Person inv GUID_P:
 Person.allInstances()->select(p|p.ID = self.ID)->size()=1
```

from which we deduce the general OCL specification pattern that should be applied in case of "global" uniqueness constraints.

```
pattern GloballyUniqueIdentifier(class:Class,
                                 attribute:Property)=
class.allInstances()->select(i |
            i.attribute = self.attribute)->size() = 1
```

It is obvious that GUID_P is an instantiation of this pattern.

```
context Person inv GUID_P:
 GloballyUniqueIdentifier(Person,ID)
```

The specification pattern that we have introduced above is both correct with respect to the semantics of invariants and useful from a debugging perspective. However, as stated by the UML 1.5 Specification (OMG 2003, p. 6-19): "The use of allInstances has some problems and its use is discouraged in most cases. The first problem is best explained by looking at the types like Integer, Real and String. For these types the meaning of allInstances is undefined. ... The second problem with allInstances is that the existence of objects must be considered within some overall context, like a system or a model. ... **A recommended style is to model the overall contextual system explicitly as an object within the system and navigate from that object to its containing instances without using allInstances.**"

"Container-relative" uniqueness case (*CUID*): According to the emphasized phrase from the above quotation, we claim that this is how the majority of uniqueness constraints should be imposed. Let us start from the model in Figure 1 and suppose there is a constraint requiring that employees of a company should be uniquely identified by their `ID`s. This is a classical case of an uniqueness requirement in the context of a "container", the "container" being represented by a `Company` object.

A correct and efficient specification for the considered constraint can be given in the context of `Company`, as follows.

```
context Company inv CUID_P1:
 self.employees->reject(e |
  self.employees.ID->count(e.ID)=1)->isEmpty()
```

This has a minor efficiency issue, however, due to the repeated computation of the employees' id collection. In order to avoid it, we propose to isolate this computation by means of an OCL let statement.

```
context Company inv CUID_P2:
 let allIDs:Bag(String) = self.employees.ID in
 self.employees->reject(e|allIDs->count(e.ID)=1)->isEmpty()
```

Analyzing the `CUID_P2` invariant, we discover a potentially new atomic constraint pattern requiring that "A given class attribute has exactly one occurence in a given bag of elements of the same type". We use the phrase *Unique Occurence in Bag*, with the acronym *UOB*, to denote the newly proposed atomic constraint pattern. For *UOB*, we propose the following OCL specification pattern.

```
pattern UniqueOccurenceInBag(bag:OclExpression,
                             attribute:Property)=
 bag.count(self.attribute) = 1
```

Based on this, it can be easily noticed that the proposed `CUID_P2` invariant specification uses an instantiation of our previously proposed OCL specification pattern `ForAll_Reject`, composed with an atomic `UniqueOcurrenceInBag` instance. Therefore, we propose the following OCL specification pattern, that is to be applied in all cases of "container-relative" uniqueness.

```
pattern ContainerRelativeUniqueIdentifier(
                navigation:Property, attribute:Property)=
 let bag:Bag(OclAny) = self.navigation.attribute in
 ForAll_Reject(navigation,
               Set{UniqueOccurenceInBag(bag,attribute)})
```

It is possible to generalize the `ContainerRelativeUniqueIdentifier` pattern, such that the uniqueness constraint, instead of applying to all contained elements, would rather apply to a conveniently selected subset. Below, we give the OCL pattern that we propose in this respect. Within it, *navigation* stands for the feature used to access the objects on which we want to impose the uniqueness constraint, *properties* denotes the set of constraints used to filter them, *class* stands for their classifier and *attribute* for the id-like attribute.

```
pattern GenContainerRelativeUniqueIdentifier(
     navigation:Feature, properties:Set(OclExpression),
     class:Class, attribute:Property)=
 let subset:Set(class) = self.navigation->select(e |
                         oclAND(properties,e)) in
 let bag:Bag(OclAny) = subset.attribute in
 ForAll_Reject(subset,
               Set{UniqueOccurenceInBag(bag,attribute)})
```

An instantiation of this pattern will be provided within the validation section.

3.2 The case of pre/post-conditions

The existing OCL specification patterns have been mostly related to the use of invariants for specifying constraint patterns. And, indeed, the use of invariants is the only alternative in case static model evaluation is required. From a dynamic perspective however, it is better to prevent than to cure. At runtime, instead of evaluating invariants after each call of a method that may modify the properties they depend on, it is preferable to prevent constraint breaking by means of appropriate pre/post-condition pairs. Due to space constraints of the paper, we only exemplify this for the particular cases considered previously, omitting the general pattern specifications that may be easily deduced.

The uniqueness constraint imposed on `ID`s of employees within a company can be prevented from breaking by means of appropriate pre/post-condition pairs for the two modifiers that may violate this constraint. The two modifiers concern adding an employee to a company and setting the name of an employee, respectively. The corresponding specifications are given below.

```
context Company::addEmployee(emp:Employee)
pre CUID_preAdd:
 self.employees->reject(e | e.ID <> emp.ID)->isEmpty()
post CUID_postAdd:
 self.employees = self.employees@pre->including(emp)

context Employee::setID(id:String)
pre CUID_preSet:
 self.employer.employees->reject(e | e.ID <> id)->isEmpty()
post CUID_postSet:
 self.ID = id
```

In case the uniqueness constraint is imposed in a global fashion, the corresponding contract for the `ID` setter is the following.

```
context Person::setID(id:String)
pre GUID_preSet:
 Person.allInstances->reject(p | p.ID <> id)->isEmpty()
post GUID_postSet:
 self.ID = id
```

4. TOOL SUPPORT AND VALIDATION

As already pointed out at the beginning of this paper, the primary goal of any modeling activity leaded under the umbrella of MDE should be the production of correct models. Except for the abstraction level, the question of determining the correctness of a model can be thought by analogy to that of establishing the correctness of a program. Namely, ensuring program correctness requires carrying out two mandatory tasks, *compilability checks* and *program testing*, a successful accomplishment of the former coming as a precondition of the latter. Compilability checks are meant to establish the correctness of a program with respect to the programming language used to define it; specifically, they test whether the programs conforms of not to the syntactical and static semantics' rules of the language in question. Program testing aims at deciding whether the program is correct or not with respect to its requirements specification.

The same two steps should be undertaken when judging the correctness of a model; their specifics in a modeling context are detailed in the following.

In modeling, *compilability checks* encompass both the model itself and its associated assertions, being regarded as a mandatory prerequisite of any model transformation task (code generation included). The model should be compilable with respect to the modeling language used to describe it, while its assertions should be compilable with respect to the constraint language employed. Failure to fulfill the first requirement may trigger inability to accomplish the second. Compilability of a model with respect to its modeling language is judged by its conformance to the abstract syntax and static semantics of the language (Chiorean et al. 2010). The abstract syntax of a modeling language is described by means of its metamodel, while the static semantics is given by metamodel-level invariants called Well Formedness Rules (WFRs). Conformance to the WFRs is checked by evaluating all such WFRs (or their programming language equivalents) on the model. Starting from the assumption that all WFRs are correct (since the modeling language must have been extensively tested prior to release), failure to fulfill any of the WFRs indicates a bug in the model. Writing the WFRs with model debugging-support in mind (as promoted by our proposed specification patterns) considerably facilitates this task, thus speeding up the development process.

Model testing aims at deciding whether the model conforms or not to the domain/reality that it represents and the rules that govern it. Such rules are referred as Business Constraint Rules (BCRs), being represented by means of model-level invariants (as opposed to WFRs which are given by metamodel-level invariants). Testing is performed using snapshots (domain model instantiations), which are meant to detect faults in the model itself (e.g. missing concepts, missing/wrong relationships, attributes having wrong types), as well as bogus model BCRs. Assuming that the model itself is correct, faults in the BCRs are identified by evaluating them on the test snapshots. The detection of any false-positive (wrong snapshot that is accepted, i.e. all BCRs evaluate to true) or false-negative (good snapshot that is denied, i.e. fails to fulfill a particular BCR) points to a logical bug in the BCR expressions. Designing them with debugging support in mind may ease the task considerably. This is especially useful when the model to test is, in fact, a metamodel, since, given their reuse potential, metamodels require extensive testing on sizable models.

Within this section, we aim at giving proof of the testing/debugging potential of the proposed specification patterns, covering both compilability checks and model testing.

4.1 Tool Support

All software development-related activities (testing and debugging included) require tools supporting both the employed formalisms and the pursued goals. The advantages derived from using the specification patterns described in this work, instead of those already proposed in the literature, may be highlighted by means of an appropriate tool, such as OCLE (Object Constraint Language Environment (LCI 2005)). We have chosen this particular OCL tool over others, since it best supports the proposed specification approach.

OCLE is a CASE (Computer Aided Software Engineering) tool that allows the specification of OCL assertions at two abstraction levels (metamodel-level and user model-level). Assertions are stored in ascii files whose extensions denote the abstraction level employed: ".ocl" for metamodel-level assertions (WFRs) and ".bcr" for user model-level assertions (BCRs). Once compiled, these assertions can be evaluated using any of the following three model validation approaches:

1) Validation of the entire model, with respect to all specified constraints. Each object is validated against all constraints specified for its class and its ancestors.

2) Validation of all instances of one or several classes belonging to a given package, with respect to either all constraints specified for those classes or a specific subset.

3) Validation of a particular object, with respect to a particular constraint.

Information regarding the errors identified during the validation of a model or of a set of objects is exposed in a tree-like manner: each broken constraint is represented by a node having as a direct ancestor its context class and as direct descendants rule failure messages pointing at the "responsible" instances. Starting from such an error message, the user can access/edit:

- the assertion (constraint) whose assessment has failed,

- the diagram pointing out the bogus instance (which is automatically set as the contextual instance),

- the model excerpt containing the base class of the problematic instance.

By means of the textual editor, the user can then evaluate any of the constraints's subexpressions on the contextual instance, with the aim of identifying the exact failure reasons.

4.2 Validation

1) *Model compilability checks*: In order to emphasize the benefits that our proposal brings in ensuring model compilability, let us start from the UML 1.5 metamodel excerpt in Figure 2 and from a sample WFR included in the UML 1.5 specification document (OMG 2003). The WFR in question concerns the name uniqueness of model elements within namespaces. In (OMG 2003), the rule is located in the `Namespace` context, having the following informal statement "If a contained element, which is not an Association or Generalization has a name, then the name must be unique in the Namespace." Below, there is the OCL equivalent of this informal WFR, as provided by the same

reference. As may be seen, it is stated as an instantiation of the classical `ForAll` OCL pattern.

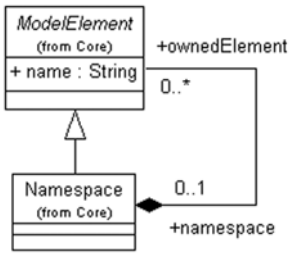

Fig. 2. UML 1.5 metamodel excerpt.

```
self.allContents->forAll(me1, me2 : ModelElement |
 (not me1.oclIsKindOf(Association) and
  not me2.oclIsKindOf(Association) and
  me1.name <> '' and me2.name <> '' and
  me1.name = me2.name
 ) implies me1 = me2)
```

Assume that there is the need of creating an UML model for components, whose syntactic description is given as follows: "From a syntactic perspective, a component is a named entity that offers services through a set of provided interfaces, for whose accomplishment it needs to use the services provided by a set of required interfaces.

Each such interface has itself a name and consists of a collection of operations. An operation is a typed entity that owns a number of parameters. Each parameter is itself a typed element, which additionally specifies data flow direction (input, output, or both)." The model part created for this syntactic description is illustrated within the model browser (top left) and diagram (top right) panels of the OCLE screenshot in Figure 3. We may assume that this is a part of a larger model covering all aspects related to the specification of components. The syntactic part of the model is rooted in the `SyntacticSpec` package.

OCLE allows checking the compilability of UML 1.5 models, by evaluating the metamodel WFRs against them. In order to ensure maximum flexibility, the WFRs are not hard coded, but stored explicitely as OCL expressions that may be conveniently edited.

Suppose that the OCL WFR used by OCLE for prohibiting names clashes within namespaces has been written as above. When checking the compilability of the component model (prior to code generation, for example), the tool reports that the WFR concerning name clashes is violated by the `SyntacticSpec` package, as shown by the bottom evaluation panel.

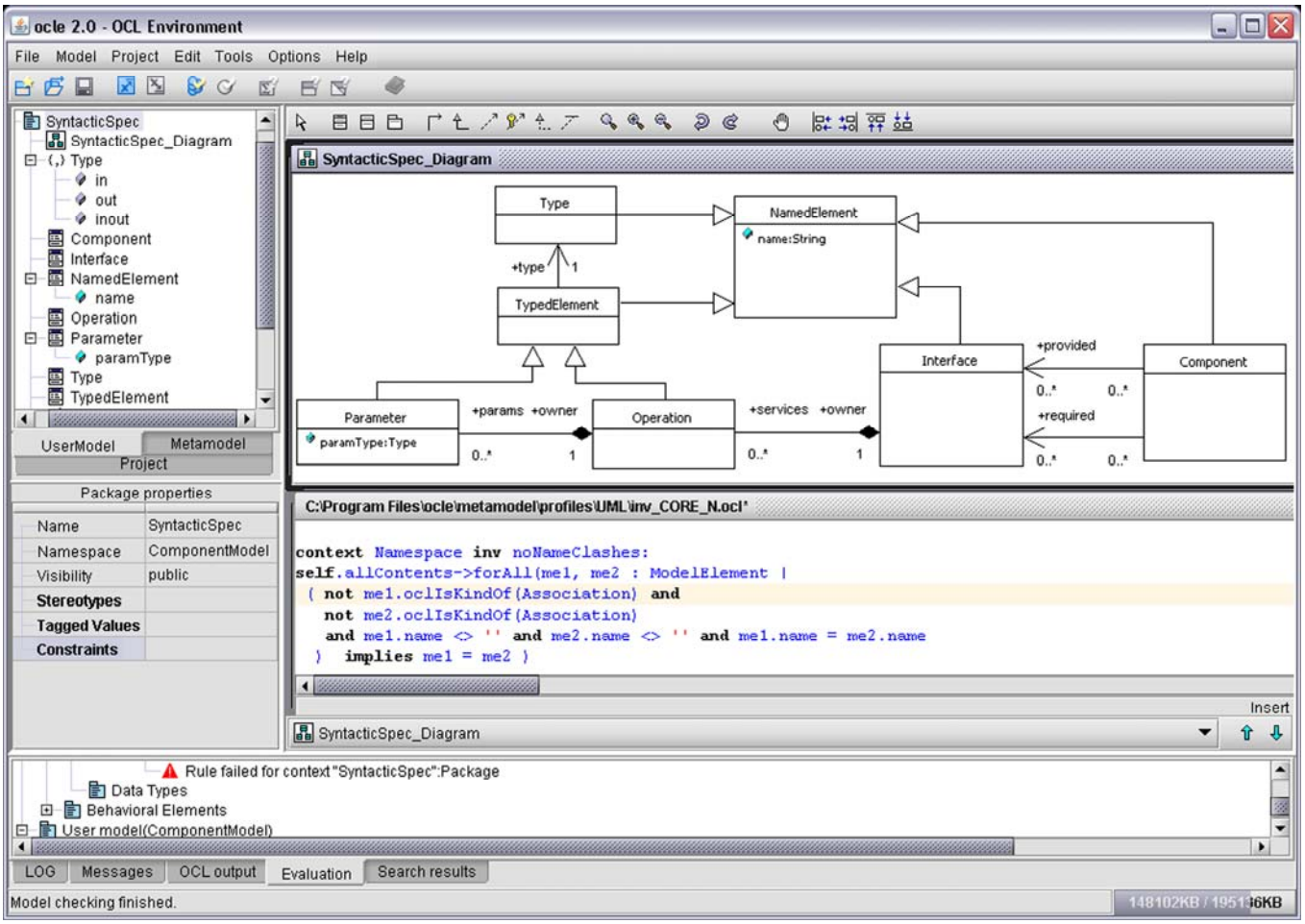

Fig. 3. OCLE screenshot illustrating the sample component model.

However, given the shape of the constraint, there are only two partial evaluations that could be performed on it in the attempt of discovering the model fault. First, there is the evaluation of the `allContents` additional operation, that basically returns the entire contents of the `SyntacticSpec` package; this does not offer extra debugging support, since it is also entirely visible in the model browser. Secondly, there is the evaluation of the `forAll` expression, which simply returns a `false` value, indicating constraint violation by `SyntacticSpec` - an already known information bearing no debugging help.

Yet, the WFR that we are dealing with can be stated as an instantiation of the generalized form of the "container-relative" unique identifier pattern given in Subsection 3.1, as shown below.

```
context Namespace inv noNameClashes:
 GenContainerRelativeUniqueIdentifier(allContents,
   Set{AttributeValueRestriction(name,<>,''),
       not self.oclIsKindOf(Association),
       not self.oclIsKindOf(Generalization)
      }
   ModelElement, name)
```

The literal OCL expression resulting from instantiation is given in the middle-right panel of the OCLE screenshot in Figure 4.

By replacing the previous WFR definition with the new one within the metamodel constraints file, and retrying the model checking operation, this will fail again, as expected, due to the violation of the WFR in question by the `SyntacticSpec` package. However, this time, the invariant shape enables subexpression evaluations that would directly lead to the fault causing the failure, allowing to efficiently rectify it. The three lines in the bottom panel of the OCLE screenshot in Figure 4 correspond to the subexpression evaluations performed. The first gives the result of evaluating the `subset` helper variable, namely the subset of model elements from within the `SyntacticSpec` package to which the name uniqueness constraint applies. The second line provides the names of all those elements, as a result of evaluating the `bag` subexpression. Finally, the evaluation of the `reject` subexpression provides, on the third line, (hyperlinks to) the model elements causing the failure. As indicated by the latter, the model is faulty since it contains both a class and an enumeration having the same name, `Type`. The class has been introduced to represent the intuitive concept of type, while the enumeration is used to classify the values indicating possible data flow directions of parameters (input, output, or both). Switching to a less general, more meaningfull name for the enumeration, such as

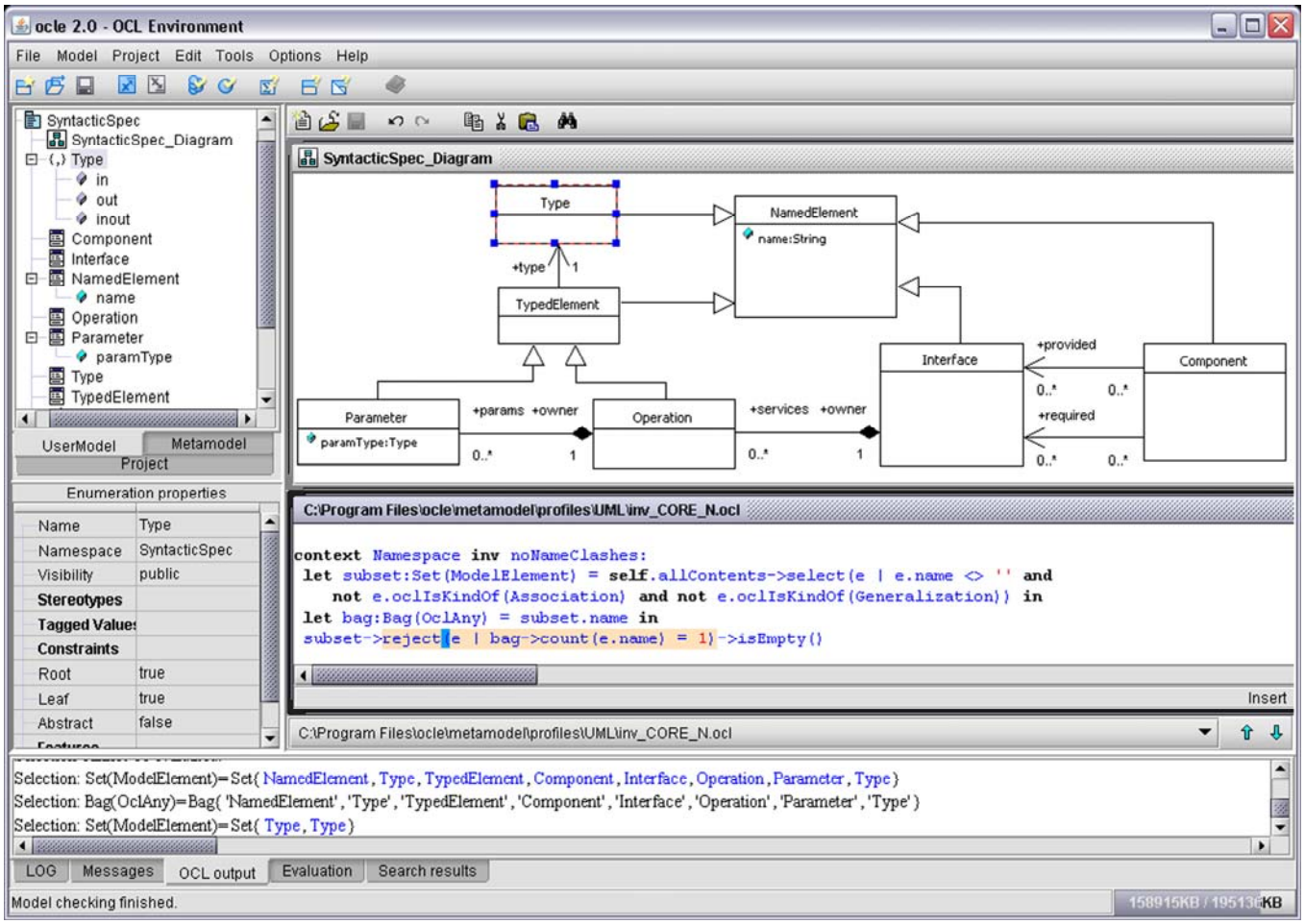

Fig. 4. Approach validation in OCLE - model compilability checks.

`ParamDirection`, would eliminate the model error. In this respect, one of the hyperlinks made available by the last evaluation provides immediate access to the properties of the enumeration object, allowing to change its name so as to ensure compilability.

The component model employed in the previous demonstration is, in fact, a metamodel, that can be seen as an instantiation of MOF 1.4 (the metalanguage of UML 1.5). Moreover, both the UML 1.5 metamodel excerpt in Figure 2 and the WFR used have equivalents in the MOF 1.4 specification. Therefore, in this particular context, it would have been more natural to carry the compilability discussion above at a higher abstraction level (compilability of a metamodel with respect to its meta-metamodel). The fact that we have decreased the level, by treating the component metamodel as an ordinary UML 1.5 model and checking its compilability with respect to an UML 1.5 WFR, can be justified by tool constraints. Namely, OCLE, the only available tool supporting the approach that we promote, only works with an UML 1.5 repository in its current version.

2) *Model testing*: To prove the advantages of our approach with respect to model testing, let us return to the model from Figure 1. Suppose that the domain experts have stated a business rule requiring that within a company, each boss should have a better income than any of its subordinates, and the modelers have hastily coded it in OCL as follows:

```
context Employee inv bossHasBetterIncome:
 self.subordinates->reject(e |
  e.salary > self.salary)->isEmpty()
```

The OCL specification above is obviously an instantiation of the `ForAll_Reject` pattern, reading as:

```
context Employee inv bossHasBetterIncome:
 ForAll_Reject(subordinates,
  AttributeValueRestriction(salary,>,self.salary))
```

Assume that one of the snapshots used during the model testing phase is the one figured in the top right-most panel of the screenshot in Figure 5. The snapshot consists of a `Company` object with four employees, the employee named Mike being the boss of the others. Since the salaries of all subordinates are smaller than the one of the boss, the test is intended to be a positive one with respect to the above constraint (it is expected to pass). However, constraint evaluation fails for the boss employee, which turns the test into a false-negative (it crashes, although it shouldn't). As indicated by the bottom OCL output panel, the two partial evaluations performed for the `subordinates` reference and `reject` subexpression return the same set of employees. Therefore, all subordinates break the rule, which immediately leads to the assumption that the invariant may have been written on the reverse. And indeed, the swapping of `e` with `self` in the relational expression used by `reject` corrects the invariant, making the test succeed. If the bogus invariant had been stated as an instatiation of the classical `ForAll` pattern

```
context Employee inv bossHasBetterIncome:
 self.subordinates->forAll(e | e.salary > self.salary)
```

fault identification might have been slower, in the absence of any debugging hint. When a `forAll` fails, the resultis the same (`false`), irrespective of whether the failure has been triggered by a single element in the collection or by all. Therefore, identifying the failure's cause generally involves a detailed and time-consuming examination of both the snapshot and constraint.

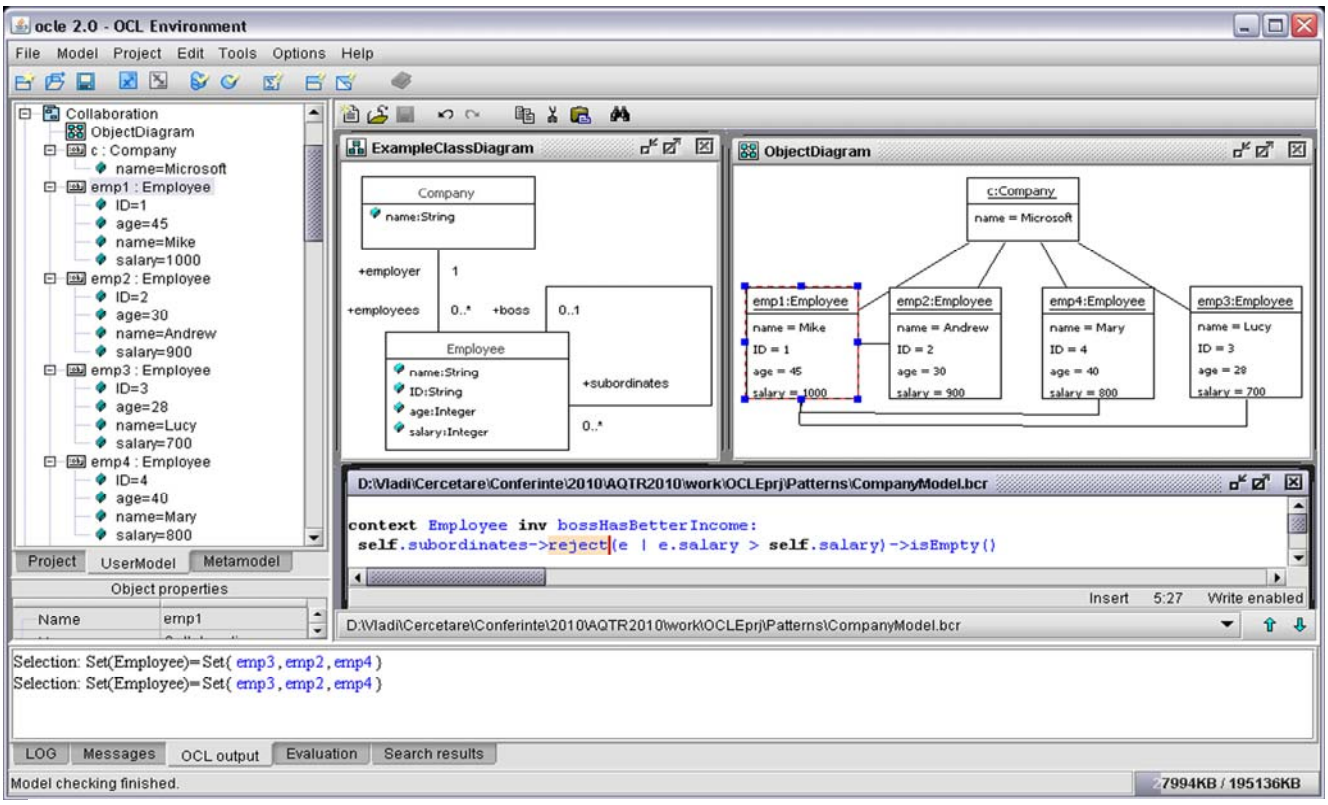

Fig. 5. Approach validation in OCLE - model testing.

In this respect, our approach brings an improvement in the efficiency, by providing useful hints for error diagnosis.

5. CONCLUSIONS AND FUTURE WORK

Within this paper, we have focused on specification patterns that increase the efficiency of error diagnosing tasks for models and applications. In this respect, we have used OCL, the standard constraint language for the MOF-based family of modeling languages. Related to the best known approaches in the field, (Ackermann 2005a); (Ackermann 2005b); (Wahler et al. 2006); (Wahler 2008), our contribution consists of: 1) proposal of a pair of OCL specification patterns for the *For All* constraint pattern, efficient with respect to our pursued goals; 2) a deeper analysis of the *Unique Identifier* constraint pattern, with regard to the particular type of uniqueness imposed ("global" vs. "container-relative"); 3) proposal of appropriate OCL specifications patterns for each uniqueness context; 4) approaching the constraint patterns' problem from both a static and a dynamic perspective, with appropriate specification patterns for each case; 5) validation of our proposed approach using appropriate tool-support. To our knowledge, this is the only approach to constraint patterns' specification aimed at maximizing the amount of relevant testing/debugging related information.

Future work targets at: 1) identifying new constraint and OCL specification patterns, along with improving some of the existing ones; 2) automating the instantiation of proposed patterns by means of an appropriate tool; 3) developing an automated test-data generator; 4) a detailed study on run-time exception handling.

ACKNOWLEDGEMENTS

This work was supported by CNCSIS-UEFISCSU, project number PNII-IDEI 2049/2008.

APPENDIX

We aim at proving that the invariants `CRAL_E` and `CRAL_P1`, described in Subsection 3.1 are semantically equivalent (in case of `CRAL_P2` the proof is quite similar). Thus, we provide a translation of the two constraints, together with the corresponding part of the diagram, into a mathematical model that uses predicate logic and set theory.

Let us denote by C the set of all `Company` instances and by E the set of all `Employee` instances. Then the `age` attribute from `Employee` may be formalized using a function *age*: $E \to \square$ and the `employees` reference using a function *emp*: $C \to \mathcal{P}(E)$, where $\mathcal{P}(E)$ denotes the power set of E.

In the above context, the `CRAL_E` invariant reduces to the following predicate

$$(\forall) self \in C \cdot (\forall) e \in emp(self) \cdot age(e) \le 65, \tag{1}$$

while `CRAL_P` reduces to

$$(\forall) self \in C \cdot \\ emp(self) - \{e \in emp(self) \mid age(e) \le 65\} = \varnothing \tag{2}$$

The predicate (2) is equivalent to

$$(\forall) self \in C \cdot \\ emp(self) \subseteq \{e \in emp(self) \mid age(e) \le 65\} \tag{3}$$

Since (3) is obviously equivalent to (1), our proof is complete.

REFERENCES

Ackermann, J. (2005a). Formal description of OCL specification patterns for behavioral specification of software components. In Thomas Baar (ed.), *Proceedings of the MoDELS' 05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, Technical Report LGL-REPORT-2005-001, EPFL, 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.

Chiorean, D. and Petraşcu, V. (2010). Towards a conceptual framework supporting model compilability. In *Proceedings of the Workshop on OCL and Textual Modelling* (*OCL 2010*), vol. 36 of ECEASST, 14 pages. EASST.

Costal, D., Gómez, C., Queralt, A., Raventós, R., and Teniente, E. (2006). Facilitating the definition of general constraints in UML. In Nierstrasz, O., Whittle, J., Harel, D., and Reggio, G. (ed.), *Model Driven Engineering Languages and Systems*, 260–274. Springer Berlin/ Heidelberg.

LCI (Laboratorul de Cercetare în Informatică) (2005). *Object Constraint Language Environment* (*OCLE*). <http://lci.cs.ubbcluj.ro/ocle/>

Meyer, B. (1997). *Object-oriented software construction*, 2nd ed., Prentice Hall.

Miliauskaite E. and Nemuraite, L. (2005). Representation of integrity constraints in conceptual models. *Information Technology and Control*, 34 (4), 355–365.

OMG (Object Management Group) (2003). *Unified Modeling Language* (*UML*) *Specification, Version 1.5*.

OMG (Object Management Group) (2006). *Meta Object Facility* (*MOF*) *Core Specification, Version 2.0*.

OMG (Object Management Group) (2010a). *Object Constraint Language* (*OCL*), *Version 2.2*.

OMG (Object Management Group) (2010b). *Unified Modeling Language* (*UML*) *Superstructure*, *Version 2.3*.

Schmidt, D. C. (2006). Model-Driven Engineering. *Computer*, 39 (2), 25–31.

Wahler, M., Koehler, J., and Brucker, A.D. (2006). Model-driven constraint engineering. *Electronic Communications of the EASST*, 5, 20 pages.

Wahler, M. (2008). *Using patterns to develop consistent design constraints*, Ph.D. dissertation, ETH Zurich, Switzerland.