Facilitating the Definition of General Constraints in UML[1] (extended version)

Dolors Costal, Cristina Gómez, Anna Queralt, Ruth Raventós and Ernest Teniente

Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
{dolors, cristina, aqueralt, raventos, teniente}@lsi.upc.edu

Abstract. One important aspect in the specification of conceptual schemas is the definition of general constraints that cannot be expressed by the predefined constructs provided by conceptual modeling languages. In general this is done by means of general-purpose languages, like OCL. In this paper we propose a new approach to facilitate the definition of such general constraints in UML. More precisely, we define a profile that extends the set of UML predefined constraints with some types of constraints that are used very frequently in conceptual schemas. We also study the application of our ideas to the specification of two real-life applications and we show how results in constraint-related problems may be easily incorporated to our proposal.

1. Introduction

An information system maintains a representation of the state of a domain in its information base (IB). The conceptual schema of an information system must include all relevant knowledge about the domain. Hence, the structural conceptual schema defines the structure of the IB while the behavioral conceptual schema defines how the IB changes when events occur. In UML, structural conceptual schemas are represented by means of class diagrams [RJB05].

A complete structural conceptual schema must include the definition of all relevant integrity constraints [ISO82]. The form of the definition of such constraints depends on the conceptual modeling language used [Oli03]. Some constraints are inherent in the conceptual model in which the language is based. This is the case, for example, of referential integrity constraints in UML class diagrams. Nevertheless, almost all constraints require an explicit definition. Most conceptual modeling languages offer a number of special constructs for defining some of them. In particular, UML offers graphical constructs for constraints such as multiplicity and also provides a set of predefined constraints which includes, for instance, association "xor" constraints and "disjoint" constraints.

However, there are many types of constraints that cannot be expressed using those special constructs provided by conceptual modelling languages. These are general constraints whose definition requires the use of a general-purpose sublanguage. With this objective, UML provides OCL [OMG05]. The use of OCL is not mandatory and

[1] This work has been partially supported by the Ministerio de Ciencia y Tecnologia under project TIN2005-06053.

the UML designer may use other languages for writing general constraints such as Java or C++ or even natural language.

There are some problems associated to the definition of general constraints through general-purpose languages. Constraints defined in natural language are often imprecise and ambiguous. Editing OCL constraints manually, although providing the means to write constraints with a precise semantics, is time-consuming and error-prone and OCL expressions may be difficult to understand for non-technical readers. Moreover, an automatic treatment of those constraints (either for reasoning or for automatic code generation) may be difficult to achieve.

For these reasons, it becomes necessary to reduce the extent of cases in which constraints must be defined through general-purpose languages. In this sense, we propose to extend the set of UML predefined constraints with some types of constraints that are used very frequently in conceptual schemas. We make the extension by defining a UML profile, the standard mechanism that UML establishes to incorporate new constructs to the language. The application of this profile has been studied in the specification of two real-life applications: the EU-Rent Car Rentals system [FQO03] and a conceptual schema for e-marketplaces [QT05].

Our proposal facilitates the definition of general constraints in UML since it decreases significantly the number of constraints that must be defined and, consequently, it reduces the scope of the problems associated to their use.

Our approach provides also important advantages regarding the automatic treatment of integrity constraints. In particular, our profile allows incorporating easily previous results on reasoning about constraint-related problems (such as satisfiability or redundancy) and facilitates obtaining an automatic implementation of the constraints defined in the conceptual schema. In this way, another contribution of our work is to show the significant advantages provided by the use of constraint stereotypes in conceptual modelling.

The rest of the paper is organised as follows. Next section illustrates the problems regarding the definition of general constraints. Section 3 presents our profile, whose application to two case studies is discussed in Section 4. Section 5 shows how to reason about the constraints specified in our profile. Section 6 reviews related work while, finally, Section 7 presents our conclusions and points out future work.

2. Problems in the Definition of General Constraints

There are some problems associated to the definition of general constraints. We will illustrate them according to the example in Figure 1 which refers to a fragment of a system that supports teaching activities of a University. The structural schema shows the definition of courses and their sections. It also contains information on teachers, their course of expertise and their assignment to sections. The structural schema includes eight general constraints, whose specification as OCL invariants is given in Figure 1:

1) Courses are identified by their name
2) Courses are identified by their code
3) Teachers are identified by the union of their name and last name

4) Each section is identified by its number within each course
5) There are no cycles in the recursive association *IsPrerequisiteOf*, i.e., a course cannot be directly or indirectly prerequisite of itself.
6) Teachers assigned to sections of a course must be experts in that course
7) The size of sections, in any case, cannot be greater than 80
8) Courses must have at least a lecturer or a professor

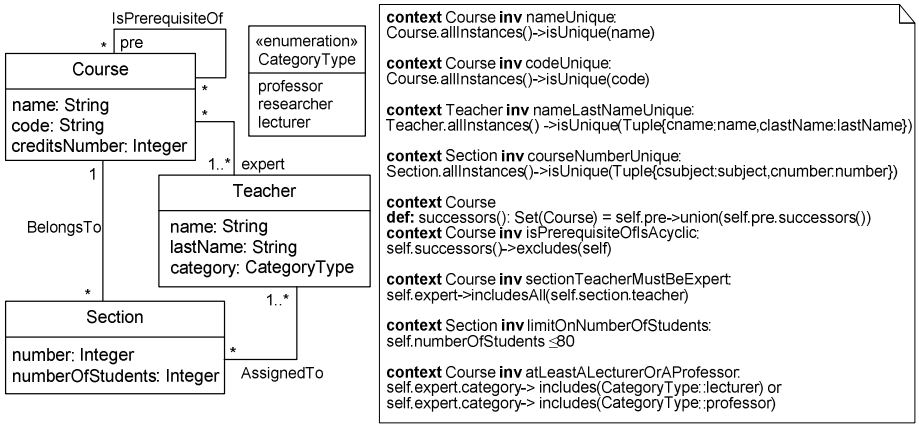

Fig. 1. Fragment of the class diagram for the example application

If general constraints are defined in natural language, they are often imprecise and ambiguous and their interpretation and treatment remain as a human responsibility. In our example, the previous descriptions of the constraints may be subject to wrong interpretations because they do not establish unambiguously their precise meaning.

This problem may be avoided by using formal general-purpose languages such as OCL. Formal languages provide the means to write constraints with precise semantics. Nevertheless, we can also identify some disadvantages of using them:

- Difficulty of understanding for non-technical readers. For example, previous constraints would not be easy for readers not familiar with OCL.
- Time-consuming definition: the designer must define explicitly the underlying semantics of each particular constraint. Additionally, in the frequent case in which there are groups of constraints that can be considered of the same type and that share common semantic aspects, the complete semantics of all the constraints of the same type must be defined for each individual constraint. This happens, for example, in the definition of textual constraints *nameUnique*, *codeUnique*, *nameLastNameUnique* and *courseNumberUnique* of Figure 1.
- Error-prone definition: formal languages are sometimes difficult to use for the designers inducing the possibility of mistakes. For instance, the constraint *isPrerequisitOfIsAcyclic* is not easily defined in OCL. Moreover, the designer could use '`includes`' instead of '`includesAll`' in constraint *sectionTeacherMustBeExpert* and then the expression would be wrong.

– Difficulty of automatic treatment: constraints expressed by means of general-purpose languages are very difficult to interpret automatically since they do not have a pre-established interpretation that can be easily incorporated to CASE tools. The lack of easy automatic interpretation has the following consequences on the automatic treatments that may be performed:
  - Some well-studied rules that allow reasoning about the constraints cannot be automatically applied.
  - Constraint semantics are difficult to incorporate to subsequent models generated automatically and, in particular, to code generation. This is a drawback towards obtaining one of the goals of the MDA, i.e., making the transformation from platform-independent models (PIMs) to platform-specific models (PSMs) as automatic as possible [OMG03].

From the above listed difficulties, we can conclude that it is interesting to reduce the extent of cases in which UML constraints must be defined using general-purpose languages. Next section presents our proposal in this direction.

3. Predefining Constraints

A constraint is a condition expressed in natural language or in a machine readable language to add some semantics to an element. UML offers a number of special constructs to define some common constraints, such as multiplicity. In addition, certain kinds of constraints, such as a disjoint constraint, are predefined in UML, but there are many others that cannot be expressed using these constructs and their definition requires the use of a specific language, such as OCL.

There are, however, some kinds of user-defined constraints that occur very frequently in conceptual schemas. For instance, a very prominent kind of constraint is the *identifier constraint* [Hal01, MB02], which may have several realizations such as *nameUnique* or *codeUnique* for a given class, i.e., either the attribute *name* or the attribute *code* uniquely identify instances of said class.

In this section we present our proposal to extend the set of predefined constraints offered by UML. We use the standard extension mechanism provided by UML, the definition of a profile [RJB05, OMG05], to achieve this goal. In particular, we define a set of stereotypes that provide some additional semantics to UML constraints that play the role of invariants.

We have defined stereotypes for some of the most frequent generic kinds of constraint, namely uniqueness, recursive association, path comparison and value comparison constraints. The use of these stereotypes prevents the designer from having to define explicit expressions to specify the corresponding constraints every time they appear. Instead, these constraints can be graphically represented in the class diagram, and, optionally, to be generated automatically in OCL.

Once applied our stereotypes to the example in Figure 1, seven out of the eight textual constraints (all except the last one) could be expressed graphically in the class diagram, making unnecessary their definition in natural language or in OCL.

3.1 UML Profile for Predefined Constraints

Our profile contains a set of stereotypes that extend the semantics of a constraint. Thus, the metaclass *Constraint* of the UML metamodel is extended by means of several stereotypes representing generic kinds of constraints, divided in four groups according to their semantics. The metaclass *Constraint* refers to a set of *constrainedElement*, i.e. those elements required to evaluate the constraint. The *context* of *Constraint* may be used as a *namespace* for interpreting names used in the expression. Each constraint has an associated *OpaqueExpression* that includes the constraint expression and the language used to define it. Each instance of *Constraint* represents a user-defined constraint, which may play the role of invariant, precondition, postcondition or body condition of an operation.

Figure 2 shows the abstract stereotype *PredefinedConstraint,* with six subtypes: *Uniqueness, RecursiveAssociation, PathComparison, ValueComparison, MandatoryDisjoint and CardinalityAssoc* stereotypes.

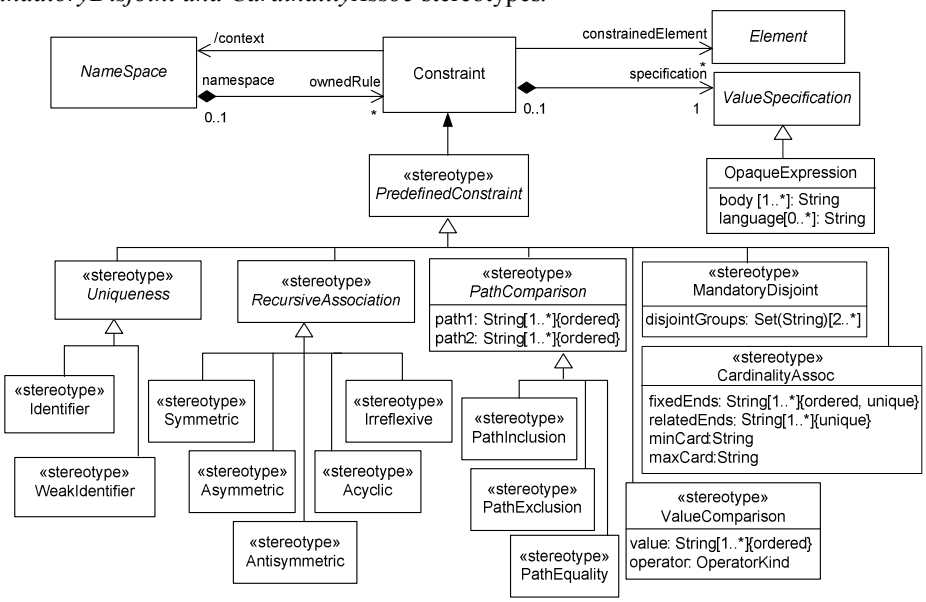

Fig. 2. UML Profile for Predefined Constraints

In order to describe each stereotype defined in this profile, we adopt a uniform and consistent template which includes the following sections: (1) Name of the constraint. (2) Semantics: describes the meaning of the constraint. (3) Stereotype Description: description of the relevant structural aspects of the stereotype proposed as solution to the constraint. (4) Attributes: contains a list of the attributes that are defined for the stereotype. (5) Constraints: defines well-formedness rules that apply to the stereotype. (6) Notation: gives the basic notational forms used to represent and use the stereotype in class diagrams. (7) Example: includes additional illustrations of the application of the stereotype and its notation.

3.1.1 PredefinedConstraint

Semantics
PredefinedConstraint defines those features shared by all constraints that are instances of this stereotype. Instances of this stereotype are the union of instances of the stereotypes presented in next sections. Each instance of this stereotype represents a constraint over a set of elements that can be generated automatically in OCL

Stereotype Description
PredefinedConstraint is an abstract stereotype of *Constraint*.. The *constrainedElement* of this stereotype is a set of elements and the language of the associated *OpaqueExpression* may be either OCL (if the designer chooses to represent the constraint also in this language) or it is left empty (if only the graphical representation is selected).

Constraints
[1] An instance of this stereotype cannot be a precondition or a postcondition of an operation.

3.1.2 Uniqueness Constraint

Semantics
A uniqueness constraint defines a uniqueness condition over the population of a class. It captures some conditions that apply to the values of attributes of ints instances. In particular, we distinguish two types of uniqueness constraints: the *identifier constraint* i.e., a set of attributes of a class that uniquely identifies it; and the *weak identifier, constraint,* i.e., a set of attributes of a class combined with another class associated to the former that uniquely identifies the instances of such a class.

Stereotype Description
Uniqueness is an abstract stereotype of *Constraint* and a subclass of *PredefinedConstraint*. An instance of this stereotype represents a constraint over a class that defines a uniqueness condition over its population.

3.1.3 Identifier Constraint

Semantics
An *identifier* is a set of one or more attributes of a class that uniquely distinguishes each instance of such a class.
Let *A* be a class with a set of attributes *{a1,...,an}*. An *identifier constraint* specifies that a subset *{ai,...,aj}* of those attributes uniquely identifies the instances of *A*. This constraint may be expressed in OCL as follows:

context A **inv**: A.allInstances()->isUnique(Tuple{cai:ai,...,caj:aj})

Stereotype Description
Identifier is a concrete stereotype of *Constraint* and a subclass of *Uniqueness*. An instance of this stereotype represents a uniqueness condition over a set of attributes of a class. The *constrainedElement* must be of type *Property.*

Constraints
[1] None of the *constrainedElement* has the lower bound of their multiplicity equal to zero.
[2] There cannot be two instances of *Identifier* such that *constrainedElement* be a subset of the other *constrainedElement*.
[3] There cannot be an instance of *WeakIdentifier* such that *constrainedElement* be the same as *constrainedElement* of *Identifier*.

Notation
The notation for an *identifier constraint* is a constraint with stereotype «Identifier». There is a dashed line between the stereotyped constraint and its constrained elements.

Example
An example of the *identifier constraint* is *nameLastNameUnique,* shown in Figure 1. It states that instances of *Teacher* are identified by the union of their *name* and *lastName*. Figure 3 shows the use of the «Identifier» stereotype to represent the constraints *nameLastNameUnique.*

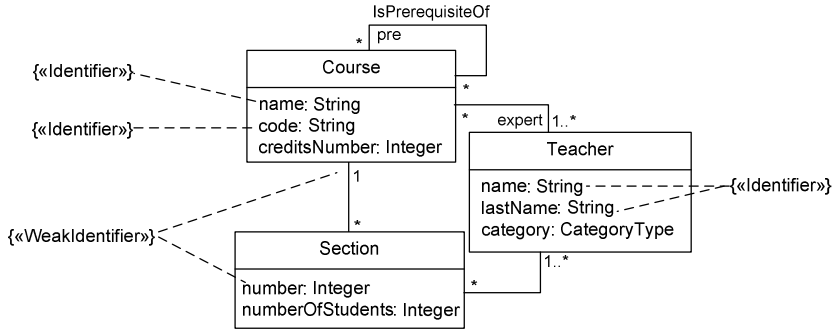

Fig. 3. Example of the use of *Identifier* and *WeakIdentifier* stereotypes

3.1.4 Weak Identifier Constraint

Semantics
A weak identifier is a set of one or more attributes of a class that with the combination of another associated class uniquely distinguishes each instance of the former class.
Let *A* be a class with a set of attributes *{a1,...,an}* and associated, via the member end *b*, to a class *B*. A w*eak identifier* constraint specifies that a subset *{ai,...,aj}* of those attributes, combined with *B*, uniquely identifies the instances of *A*. This constraint may be formally expressed in OCL as follows:

context A **inv**: A.allInstances()->isUnique(Tuple{cb:B,cai:ai,...,caj:aj})

Stereotype Description
WeakIdentifier is a concrete stereotype of *Constraint* and a subclass of *Uniqueness*. An instance of this stereotype represents a constraint over a class that defines which set of attributes combined with an associated former class uniquely identifies the instances of such a class.

Constraints
[1] None of the *constrainedElement* has the lower bound of their multiplicity equal to zero.
[2] One of the *constrainedElement* corresponds to a property associated to the class that owns the other constrained elements and multiplicity 1.

Notation
The notation for a *weak identifier* constraint is a constraint with stereotype «WeakIdentifier».There is a dashed line between the stereotyped constraint and its constrained elements.

Example
An example of the *weak identifier* constraint is *courseNumber* shown in Figure 1. It states that each instance of *Section* is identified by its *number* within each instance of *Course*. Figure 3 shows the use of «WeakIdentifier» stereotype to represent the constraints *courseNumberUnique* stated above. There is a dashed line between the constraint with the corresponding stereotype and its constrained elements.

3.1.5 Recursive Association Constraint

Semantics
Recursive association constraints, called ring constraints in [Hal01], are a type of constraints that apply over a recursive binary association, guaranteeing that the association fulfills a certain property. We consider five types of those constraints: *irreflexive*, *symmetric*, *antisymmetric*, *asymmetric* and *acyclic* constraints.

Stereotype Description
RecursiveAssociation is an abstract stereotype of *Constraint* and a subclass of *PredefinedConstraint*. An instance of this stereotype represents a constraint over a binary and recursive association that defines a condition over its population.

Constraints
[1] A recursive association invariant has as *constrainedElement* an association.
[2] The *constrainedElement* association must be binary and recursive.
[3] An instance of recursive association constraint must have as a namespace one of the member end class of the association.
[4] There cannot be two instances of the same recursive association constraint for the same association.

3.1.6 Irreflexive Constraint

Semantics

An irreflexive constraint represents a constraint over a recursive association. This restriction constrains the extension of the association defining it as irreflexive. Let *A* be a class and *R* a recursive association over *A,* with *r1* and *r2* member ends. An irreflexive constraint over *R* guarantees that if *a* is instance of *A* then *a* is never R-related to itself. This constraint may be formally expressed in OCL as follows:

context A **inv:** self.r2->excludes(self)

Stereotype Description

Irreflexive is a concrete stereotype of *Constraint* and a subclass of *RecursiveAssociation*. Each instance of the stereotype represents the constraint related to a recursive association that defines it as irreflexive.

Constraints

[1] There cannot be another instance of asymmetric nor acyclic constraint for the same association[2].

Notation

The notation for an *irreflexive* constraint is a constraint with stereotype «Irreflexive».

Example

See the example in Figure 4. The recursive association *playMatch* between two football teams is restricted by an irreflexive constraint. This means that local team *t1* cannot play a match with visitor team *t1*.

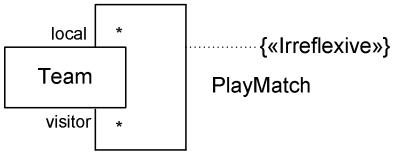

Fig. 4. Example of a Recursive Association with stereotype «Irreflexive»

3.1.7 Symmetric Constraint

Semantics

A symmetric constraint represents a constraint over a recursive association. This restriction constrains the extension of the association defining it as symmetric. Let *A* be a class and *R* a recursive association over *A,* with *r1* and *r2* member ends. A symmetric constraint over *R* guarantees that if *a* and *b* are instances of *A* and *a* is R-related to *b* then *b* is R-related to *a*. Formally, in OCL:

context A **inv:** self.r2.r2->includes(self)

[2] This constraint is defined to avoid redundancy between recursive association invariants

Stereotype Description
Symmetric is a concrete stereotype of *Constraint* and a subclass of *RecursiveAssociation*. Each instance of the stereotype represents the constraint related to a recursive association that defines it as symmetric.

Constraints
[1] There cannot be another instance of acyclic, asymmetric nor antisymmetric constraint for the same association[3].

Notation
The notation for a *symmetric* constraint is a constraint with stereotype «Symmetric».

Example
See the example in Figure 5. The recursive association *isMarriedTo* between two people is constrained by a symmetric constraint. This means that if person *p1* is married to person *p2* then person *p2* is married to person *p1*.

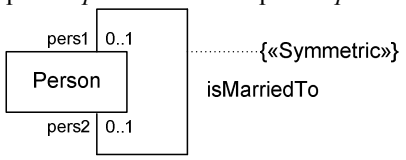

Fig. 5. Example of a Recursive Association with stereotype «Symmetric»

3.1.8 Antisymmetric Constraint

Semantics
An antisymmetric constraint represents a constraint over a recursive association. This restriction constrains the extension of the association defining it as antisymmetric. Let *A* be a class and *R* a recursive association over *A,* with *r1* and *r2* member ends. An antisymmetric constraint over *R* guarantees that if *a* and *b* are instances of *A*, *a* is R-related to *b* and *b* is R-related to *a*, then *a* and *b* are the same instance. In OCL:

context A **inv:** self.r2->excludes(self) implies self.r2.r2->excludes(self)

Stereotype Description
Antisymmetric is a concrete stereotype of *Constraint* and a subclass of *RecursiveAssociation*. Each instance of the stereotype represents the constraint related to a recursive association that defines it as antisymmetric.

Constraints
[1] There cannot be another instance of symmetric constraint for the same association[3].

[3] This constraint is defined to validate interactions between recursive association invariants

[2] There cannot be another instance of asymmetric nor acyclic constraint for the same association[2].

Notation
The notation for an *antisymmetric* constraint is a constraint with stereotype «Antisymmetric».

Example
See the example in Figure 6. The recursive association *follows_or_hasSamePosition* between two people in a classification is constrained by an antisymmetric constraint. This means that if person *p1* and person *p2* are different then if person *p1* follows or has the same position in a classification as person *p2* then person *p2* cannot follows or has the same position as person *p1*.

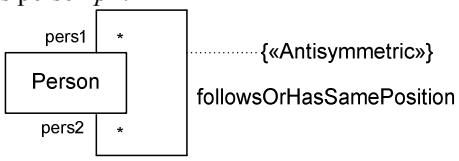

Fig. 6. Example of a Recursive Association with stereotype «Antisymmetric»

3.1.9 Asymmetric Constraint

Semantics
An asymmetric constraint represents a constraint over a recursive association. This restriction constrains the extension of the association defining it as asymmetric. Let *A* be a class and *R* a recursive association over *A,* with *r1* and *r2* member ends. An asymmetric constraint guarantees that if *a* and *b* are instances of *A* and *a* is R-related to *b* then *b* is not R-related to *a*. Observe that this constraint is equivalent to the union of antisymmetric an irreflexive constraints. It may be expressed in OCL as follows:

context A **inv:** self.r2.r2->excludes(self)

Stereotype Description
Asymmetric is a concrete stereotype of *Constraint* and a subclass of *RecursiveAssociation*. Each instance of the stereotype represents the constraint related to a recursive association that defines it as asymmetric.

Constraints
[1] There cannot be another instance of symmetric constraint for the same association[3].
[2] There cannot be instances of antisymmetric nor irreflexive nor acyclic constraints for the same association[2].

Notation
The notation for an *asymmetric* constraint is a constraint with stereotype «Asymmetric».

Example
See the example in Figure 7. The recursive association *supervises* between a supervisor and a supervised employee is constrained by an asymmetric constraint. This means that if employee *e1* is a supervisor of employee *e2* then employee *e2* cannot be a supervisor of employee *e1*.

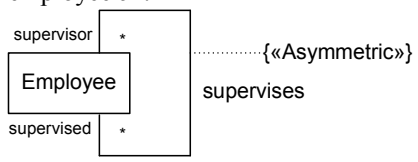

Fig. 7. Example of a Recursive Association with stereotype «Asymmetric»

3.1.10 Acyclic Constraint

Semantics
An acyclic constraint represents a constraint over a recursive association. This restriction constrains the extension of the association defining it as asymmetric. Let *A* be a class and *R* a recursive association over *A,* with *r1* and *r2* member ends. An acyclic constraint guarantees that if *a* and *b* are instances of *A* and *a* is R-related to *b* then *b* or instances R-related directly or indirectly to *b* are not R-related to *a*. Formally, in OCL:

```
context A
def: successors(): Set(A) = self.r2->union(self.r2.successors())
inv: self.successors()->excludes(self)
```

Stereotype Description
Acyclic is a concrete stereotype of *Constraint* and a subclass of *RecursiveAssociation*. Each instance of the stereotype represents the constraint related to a recursive association that defines it as acyclic.

Constraints
[3] There cannot be another instance of symmetric constraint for the same association[3].
[4] There cannot be instances of asymmetric nor antisymmetric nor irreflexive constraints for the same association[2].

Notation
The notation for an *acyclic* constraint is a constraint with stereotype «Acyclic».

Example
In the example of Figure 1, *isPrerequisiteOfIsAcyclic* is a constraint of this type that applies to the recursive association *isPrerequisiteOf*. Figure 8 shows the definition of this constraint as an instance of the *Acyclic* stereotype.

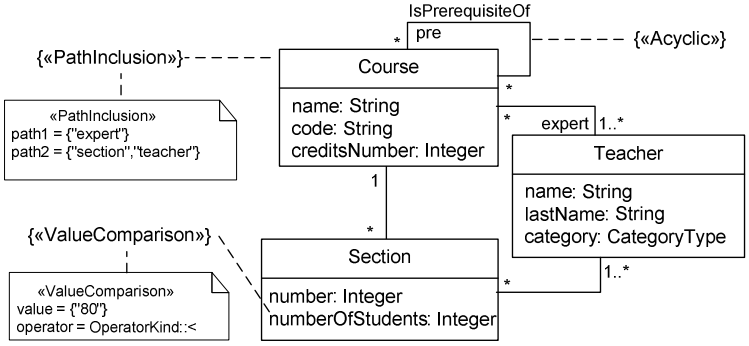

Fig. 8. Applying Acyclic, PathInclusion and ValueComparison stereotypes

3.1.11 Path Comparison Constraint

Semantics
Path comparison constraints restrict the way the population of one role or role sequence (path for short) relates to the population of another [Hal01]. Constraints belonging to this type are *path inclusion, path exclusion* and *path equality*. They all apply to a class *A* related to a class *B* via two different paths *r1...ri*, *rj...rn*. We consider three types of those constraints: *path inclusion, path exclusion* and *path equality* constraints.

Stereotype Description
PathComparison is an abstract stereotype of *Constraint* and a subclass of *PredefinedConstraint*. An instance of this stereotype represents a constraint that defines a set comparison over two paths. The *constrainedElement* of this stereotype is the start class of both paths.

Attributes

- path1, path2: String [1..*] {ordered} Specify the paths to be compared.

Constraints

[1] The *constrainedElement* associated to an instance of *PathsComparison* is an element of type *Class.*
[2] *path1* and *path2* correspond to valid paths.
[3] The first element of each path corresponds to a property of the constrained class.

[4] The classes reached by each path are the same.
[5] There cannot be two instances of *PathComparison* with the same paths.

3.1.12 Path Inclusion Constraint

Semantics
Let *A* and *B* be two classes related via two different paths *r1...ri* and *rj...rn.* A path inclusion constraint guarantees that if *a* is an instance of *A,* the set of instances of *B* related to *a* via *r1...ri* includes the set of instances of *B* related to *a* via *rj...rn.* It can be expressed in OCL as follows:

context A **inv:** self.r1...ri->includesAll(self.rj...rn)

Stereotype Description
PathInclusion is a concrete stereotype of *Constraint* and a subclass of *PathComparison*. Each instance of the stereotype represents a constraint that defines an inclusion relationship between the sets of instances at the end of two paths.

Notation
The notation for path inclusion constraints is a constraint with stereotype «PathInclusion». Values of *path1* and *path2* attributes are shown in a note attached to the constraint.

Example
See the example in Figure 8. Class *Course* has a *PathInclusion* constraint meaning that the teachers assigned to sections of a course must be expert in that course.

3.1.13 Path Exclusion Constraint

Semantics
Let *A* and *B* be two classes related via two different paths *r1...ri* and *rj...rn.* A path exclusion constraint guarantees that if *a* is an instance of *A,* the set of instances of *B* related to *a* via *r1...ri* does not contain any of the instances of *B* related to *a* via *rj...rn.* Formally, in OCL:

context A **inv:** self.r1...ri->excludesAll(self.rj...rn)

Stereotype Description
PathExclusion is a concrete stereotype of *Constraint* and a subclass of *PathComparison*. Each instance of the stereotype represents a constraint that defines an exclusion relationship between the sets of instances at the end of two paths.

Notation
The notation for path exclusion invariants is a constraint with stereotype «PathExclusion». Values of *path1* and *path2* attributes are shown in a note attached to the constraint.

Example
See the example in Figure 9. Class *Course* has a *PathExclusion* constraint meaning that the teachers assigned to sections of a course cannot be beginners in that course.

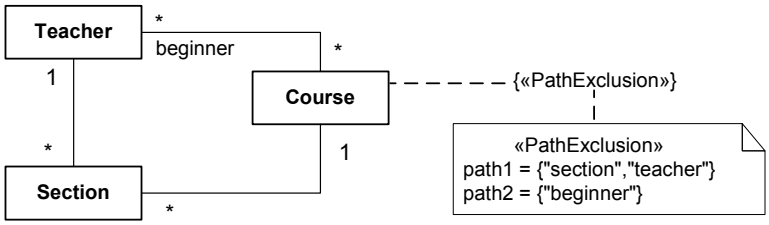

Fig. 9. Example of a Class with stereotype «PathExclusion»

3.1.14 Path Equality Constraint

Semantics
Let *A* and *B* be two classes related via two different paths *r1...ri* and *rj...rn.* A path equality invariant guarantees that if *a* is an instance of *A,* the set of instances of *B* related to *a* via *r1...ri* coincides with the instances of *B* related to *a* via *rj...rn.* In OCL, this constraint is expressed as follows:

context A **inv:** self.r1...ri = self.rj...rn

Stereotype Description
PathEquality is a concrete stereotype of *Constraint* and a subclass of *PathComparison*. Each instance of the stereotype represents a constraint that defines an equality relationship between the sets of instances at the end of two paths.

Notation
The notation for path equality invariants is a constraint with stereotype «PathEquality». Values of *path1* and *path2* attributes are shown in a note attached to the constraint.

Example
See the example in Figure 10. Class *Course* has a *PathEquality* constraint meaning that all the teachers that belong to a course must be assigned to a section of that course.

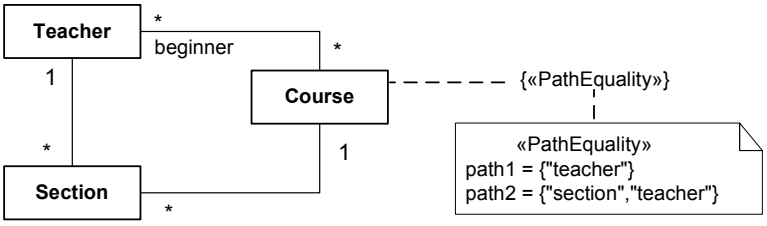

Fig. 10. Example of a Class with stereotype «PathEquality»

3.1.15 Value Comparison Constraint

Semantics
Value comparison constraints restrict the possible values of an attribute, either by comparing it to a constant or to the value of another attribute [Ack05].

Let *A* be a class, let *a1* be an attribute of *A,* let *v* be either a constant or the value of an attribute accessible from *A* and let *op* be an operator of kind <, >, =, <>, ≤, or ≥. A value comparison constraint restricts the possible values of *a1* regarding the value of *v*. This constraint can be formally expressed in OCL as follows:

context A **inv:** self.a1 *op* v

Stereotype Description
ValueComparison is a concrete stereotype of *Constraint* and a subclass of *PredefinedInv*. An instance of this stereotype represents a constraint that restricts the value of an attribute.

Attributes

- operator: OperatorKind — Specifies the operator to be used in the comparison.
- value: String[1..*] {ordered} — Specifies the value to be compared to the attribute. It can be either a path or a constant.

Constraints

[1] The constrainedElement associated to an instance of *ValueComparison* is an element of type *Property,* not belonging to an association.
[2] The constrained attribute has multiplicity 1.
[3] The attribute *value* represents either a valid path starting from the class that owns the constrained attribute, or a constant.
[4] If *value* is a path, its last element is an attribute with multiplicity 1.
[5] The type of the value represented in *value* conforms to the type of the constrained attribute.
[6] If the constrained attribute is of type Boolean, the operator can only be = or <>
[7] There cannot be two instances of *ValueComparison* referring to the same attribute and value.

Notation
The notation for value comparison constraints is a constraint with stereotype «ValueComparison» attached to an attribute. Values of *operator* and *value* attributes are shown in a note attached to the constraint.

Example
See the example in Figure 8. Attribute *age* has a *ValueComparison* constraint meaning that the number of students of a section must be lower than 80.

3.1.17 Mandatory Disjoint Constraint

Semantics
A mandatory disjoint constraint restricts that the disjunction of a set of attributes is mandatory {{158 Halpin, T. 2001; }}.
Let *A* be a class with a set of attributes, *{at1,...,atn}*. Mandatory disjoint invariant allows to specify that at least values of one of disjoint set of attributes *{{ati,...,atj},...,{atp,...,atq}}*, where *{ati,...,atj},...,{atp,...,atq}* are disjoint subsets of *{at1,...,atn}*, are mandatory. This constraint may be expressed, formally, in OCL as follows:

context A **inv**:
(ati->notEmpty() and … and atj->notEmpty()) or …or (atp->notEmpty() and … and atq->notEmpty())

Stereotype Description
MandatoryDisjoint is a concrete stereotype of *Constraint*. An instance of this stereotype represents a constraint over a class that defines for any instance of a class which values of disjoint set of attributes are mandatory.

Attributes

• disjointGroups:Set(String)[2..*]	Specifies two or more disjoint groups of sets of strings.

Constraints
[1] The *constrainedElement* associated to *MandatoryDisjoint* is an element of type *Class*.
[2] There cannot be two instances of *MandatoryDisjoint* with the same *disjointGroups*.
[3] The *disjointGroups* attribute corresponds to the name of disjoint subsets of attributes of the *constrainedElement* class.
[4] None of groups of *disjointGroups* can be empty.
[5] At least, one of the attributes of each group of *disjointGroups* has 0 as minimum multiplicity.
[6] An instance of mandatory disjoint constraint must have as a namespace the same class as *constrainedElement* class.

Notation
The notation for a *mandatory disjoint constraint* is a constraint with stereotype «MandatoryDisjoint». Values of *disjointGroups* attribute that is a set of 2 or more sets of 1 or more strings are shown in a note attached to the constraint.

Example
See the example in figure 11. The class *Person* has a mandatory disjoint invariant which ensures that either *name*, *lastname* or *SSNumber* are mandatory.

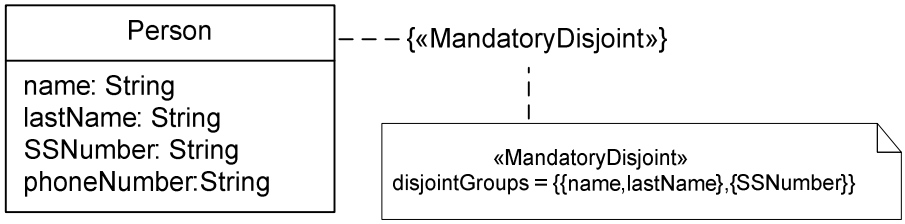

Fig. 11. Example of a class with stereotype «MandatoryDisjoint»

3.1.16 Cardinality Association Constraint

Semantics
A cardinality association constraint represents a restriction over an *n*-ary association, i.e. association that has three or more ends.

Let *R* be an association and let *fixedEnds* and *relatedEnds* be disjoint subsets of the end properties related by *R* shown in Figure 12. A cardinality association constraint for *fixedEnds* and *relatedEnds* restricts the minimum and maximum number of instances of *R* that relate any combination of objects, such that the combination includes an object that is instance of each one of the owner end classes of the properties in *fixedEnds*, with a different combination of objects that are instance of each one of the owner end classes of the properties in *relatedEnds*.

Observe that when *relatedEnds* has a single property and *fixedEnds* has the rest of properties related by the association, the corresponding cardinality constraint can be expressed graphically in UML class diagrams by means of the multiplicity of the association ends. Nevertheless, the rest of cardinality constraints of an *n*-ary association can not be expressed graphically in UML class diagrams.

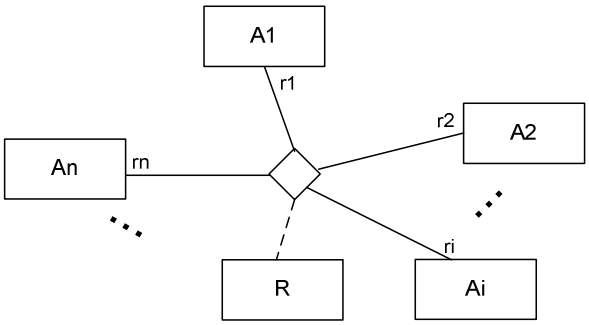

Fig. 12. Example of an *n*-ary association

Assume that association *R* of Figure 12 has a cardinality association constraint where:

- fixedEnds = $\{r_1,r_2\ldots,r_i\}$.
- relatedEnds = $\{r_{i+1},\ldots,r_k\}$.
- min is the minimum cardinality established by the constraint.
- max is the maximum cardinality established by the constraint.

Then, the corresponding cardinality association constraint can be expressed, formally, in OCL as follows:

context A_1 **inv:**

A_1.allInstances()->forAll(vr_1|A_2.allInstances()->forAll(vr_2| …A_i.allInstances()->forAll(vr_i| R.allInstances()->select(t| t.r_1=vr_1 and t.r_2=vr_2 and … and t.r_i=vr_i))…))->collect(t| Tuple{cr_{i+1}=t.r_{i+1}, …, cr_k=t.r_k})->asSet()->size()>= min and

A_1.allInstances()->forAll(vr_1|A_2.allInstances()->forAll(vr_2| …A_i.allInstances()->forAll(vr_i| R.allInstances()->select(t| t.r_1=vr_1 and t.r_2=vr_2 and … and t.r_i=vr_i))…))->collect(t| Tuple{cr_{i+1}=t.r_{i+1}, …, cr_k=t.r_k}) ->asSet()->size()<= max

Stereotype Description

CardinalityAssoc is a concrete stereotype of *Constraint*. Each instance of this stereotype represents constraint related to an *n*-ary association that restricts its cardinality.

Attributes

fixedEnds: String[1..*]{ordered,unique}
: Specifies a subset of the end properties of the association that constitute the *fixedEnds* of the cardinality constraint.

relatedEnds: String[1..*]{unique}
: Specifies a subset of the end properties of the association that constitute the *relatedEnds* of the cardinality constraint.

minCard: String
: Specifies the minimum cardinality permitted by the cardinality association constraint.

maxCard: String	Specifies the maximum cardinality permitted by the cardinality association constraint.

Constraints

[1] A cardinality association constraint has as *constrainedElement* an association.
[2] The *constrainedElement* association must be *n*-ary (association that has three or more ends).
[3] *minCard* must represent a non-negative integer.
[4] *maxCard* must be '*' and, otherwise, must represent a positive integer.
[5] If *maxCard* is different from '*' then the value it represents must be greater than the value represented by *minCard*.
[6] Properties represented by values in *fixedEnds* must be end properties of the *constrainedElement* association.
[7] Properties represented by values in *relatedEnds* must be end properties of the *constrainedElement* association.
[8] *fixedEnds* and *relatedEnds* must be disjoint.
[9] If *relatedEnds* has a single element then the number of elements of *relatedEnds* must be less than the number of ends of the *constrainedElement* association minus 1. Otherwise the constraint could be graphically represented in UML.
[10] There cannot be another instance of cardinality association constraint such that has as *constrainedElement* the same association and that properties represented in fixedEnds and relatedEnds are the same, respectively.
[11] A cardinality association constraint must have as *namespace* the end class that owns the property that corresponds to the first element in *fixedEnds*.

Notation

The notation for a cardinality association constraint is a constraint with stereotype «CardinalityAssoc». Values of *fixedEnds*, *relatedEnds*, *minCard* and *maxCard* attributes are shown in a note attached to the constraint.

Examples

See the example in Figure 13. The n-ary association *Supply* has a cardinality association constraint which establishes that for any building under construction and day, the maximum number of supplies of different products and suppliers is 50.

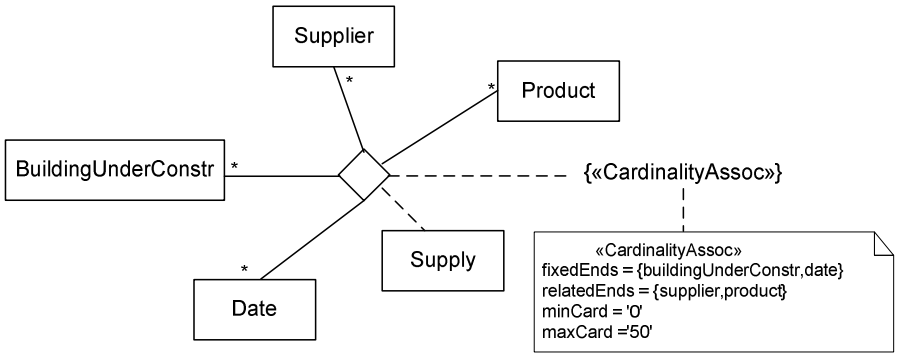

Fig. 13. Example of a n-ary association with stereotype «CardinalityAssoc»

See the example in Figure 14. The n-ary association *Uses*, which represents the skills used by employees in the projects where they participate, has a cardinality constraint invariant which establishes that the maximum number of projects where an employee may participate is 5.

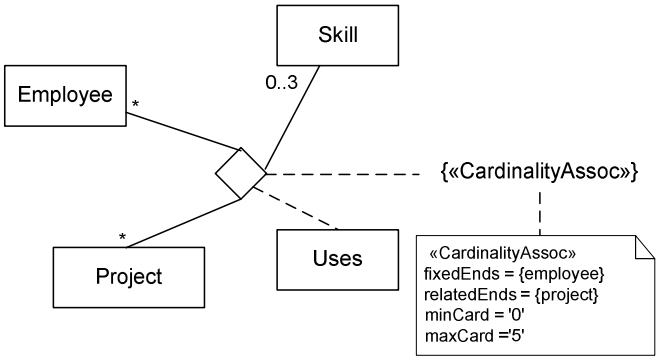

Fig. 14. Example of a n-ary association with stereotype «CardinalityAssoc»

3.2 Creating the Instances of an Stereotype

To be able to specify new predefined constraints, for each stereotype we have also defined an operation that allows creating its instances. This operation associates to each new instance its corresponding context, constrained elements and specification. This specification has an empty *body* attribute if the designer only desires a graphical representation. Otherwise, if the designer also requires the definition of the OCL expression, the operation assigns to the *body* attribute the expression automatically generated according to the type of constraint.

The specification of the operations to create instances of the stereotypes defined above is given in next subsections.

3.2.1 NewIdentifier Operation

The operation *newIdentifier* results in the creation of an instance of the stereotype *Identifier*.

The parameters needed are a class, the set of attributes that identify each of its instances, the name of the constraint and the way to represent this constraint in the schema which is an enumeration of two values ocl and graphically. The value ocl indicates that the constraint will be represented graphically and textually in OCL and the value *graphically* indicates that the representation will be only graphical. The postconditions guarantee that a new instance of *Identifier* will be created, the constrained elements will be the set of properties and the namespace will be the indicated class. This operation can be defined in OCL as follows:

```
context Identifier::newIdentifier(c:Class, a:Set(Property), name:String[0..1],
                                  representation:RepresentationType)
 let ident = 'Tuple{'.concat(Sequence{1..a->size()}-> iterate(pn; s: String = '' | s.concat((if
       (pn>1) then ', ' else'' endif).concat ('c').concat(a->at(pn). name).concat (': ').concat
       (a-> at(pn). name)))).concat('}') in
 post: id.oclIsNew() and id.oclIsTypeOf(Identifier) and id.name=name and
       id.constrainedElement -> includesAll(a.name) and
       c.ownedRule->includes(id) and
       expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
       id.specification = expr and
       representation=RepresentationType::ocl implies
             expr.language = 'OCL' and expr.body = 'context '.concat(id.context.name).
             concat(' inv ').concat(name).concat(': ').concat(c.name).concat('.allInstances()->
             isUnique('). concat(ident).concat(')')
```

3.2.2 NewWeakIdentifier Operation

The operation *newWeakIdentifier* results in the creation of an instance of the stereotype *WeakIdentifier*.

The parameters needed are a class, the set of properties (attributes and the associated class) that identify each instances of the former class, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *WeakIdentifier* will be created, the constrained elements will be the set of properties and the namespace will be the indicated class. This operation can be defined in OCL as follows:

```
context WeakIdentifier::newWeakIdentifier(c:Class, a:Set(Property), name:String[0..1],
                                  representation:RepresentationType)
 let ident = 'Tuple{'.concat(Sequence{1..a->size()}-> iterate(pn; s: String = '' | s.concat((if
       (pn>1) then ', ' else'' endif).concat ('c').concat(a->at(pn). name).concat (': ').concat
       (a-> at(pn). name)))).concat('}') in
 post: id.oclIsNew() and id.oclIsTypeOf(WeakIdentifier) and id.name=name and
       id.constrainedElement -> includesAll(a.name) and c.ownedRule->includes(id) and
       expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
       id.specification = expr and
       representation=RepresentationType::ocl implies
             expr.language = 'OCL' and expr.body = 'context '.concat(id.context.name).
             concat(' inv ').concat(name).concat(': ').concat(c.name).concat('.allInstances()->
             isUnique('). concat(ident).concat(')')
```

3.2.3 NewIrreflexive Operation

The operation *newIrreflexive* results in the creation of an instance of the stereotype *Irreflexive.*

The parameters needed are an association, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *Irreflexive* will be created, the constrained elements and the namespace will be the association. This operation can be defined in OCL as follows:

```
context Irreflexive:: newIrreflexive (a:Association, name:String[0..1],
                                      representation:RepresentationType)
let rol:String =  if a.memberEnd->last().name->isEmpty()
                  then a.memberEnd-> last().class.name
                  else a.memberEnd->last().name
in
post: irref.oclIsNew() and irref.oclIsTypeOf(Irreflexive) and irref.name=name and
      irref.constrainedElement->includes(a) and
      a.memberEnd->last().class.ownedRule->includes(irref) and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      irref.specification=expr and
      representation=RepresentationType::ocl implies
            expr.language='OCL' and
            expr.body='context  '.concat(irref.context.name).concat('  inv').  concat(name).
            concat(': self.').concat(rol).concat('<>self')
```

3.2.4 NewSymmetric Operation

The operation *newSymmetric* results in the creation of an instance of the stereotype *Symmetric.*

The parameters needed are an association, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *Symmetric* will be created, the constrained elements and the namespace will be the association. This operation can be defined in OCL as follows:

```
context Symmetric:: newSymmetric (a:Association, name:String[0..1],
                                  representation:RepresentationType)
let rol:String = if a.memberEnd->last().name->isEmpty()
                 then a.memberEnd-> last().class.name
                 else a.memberEnd->last().name
in
post:  sym.oclIsNew() and sym.oclIsTypeOf(Symmetric) and sym.name=name and
       sym.constrainedElement->includes(a) and
       a.memberEnd->last().class.ownedRule->includes(sym) and
       expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
       sym.specification=expr and
       representation=RepresentationType::ocl implies
             expr.language='OCL' and
             expr.body='context '.concat(sym.context.name). and
             concat(' inv').concat(name).concat(': self.'). concat(rol).
             concat('. ').concat(rol).concat('->includes(self) ')
```

3.2.5 NewAntisymmetric Operation

The operation *newAntisymmetric* results in the creation of an instance of the stereotype *Antisymmetric*.

The parameters needed are an association, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *Antisymmetric* will be created, the constrained elements and the namespace will be the association. This operation can be defined in OCL as follows:

```
context Antisymmetric:: newAntisymmetric (a:Association, name:String[0..1],
                                representation:RepresentationType)
let rol:String = if a.memberEnd->last().name->isEmpty()
                 then a.memberEnd-> last().class.name
                 else a.memberEnd->last().name
in
post: antisym.oclIsNew() and antisym.oclIsTypeOf(Antisymmetric) and
      antisym.name=name and
      antisym.constrainedElement->includes(a) and
      a.memberEnd->last().class.ownedRule->includes(antisym) and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      antisym.specification=expr and
      representation=RepresentationType::ocl implies
            expr.language='OCL' and
            expr.body='context '.concat(antisym.context.name). concat(' inv').
            concat(name).concat(': self.').concat(rol).concat(<>self implies
            self.).concat(rol).concat('.'). concat(rol).concat('->excludes(self)')
```

3.2.6 NewAsymmetric Operation

The operation *newAsymmetric* results in the creation of an instance of the stereotype *Asymmetric*.

The parameters needed are an association, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *Asymmetric* will be created, the constrained elements and the namespace will be the association. This operation can be defined in OCL as follows:

```
context Asymmetric:: newAsymmetric (a:Association, name:String[0..1],
                                representation:RepresentationType)
let rol:String = if a.memberEnd->last().name->isEmpty()
                 then a.memberEnd-> last().class.name
                 else a.memberEnd->last().name
in
post: asym.oclIsNew() and asym.oclIsTypeOf(Asymmetric) and asym.name=name and
      asym.constrainedElement->includes(a) and
      a.memberEnd->last().class.ownedRule->includes(asym) and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      asym.specification=expr and
      representation=RepresentationType::ocl implies
            expr.language='OCL' and
            expr.body='context '.concat(asym.context.name). concat(' inv'). concat(name).
            concat(': self.').concat(rol).concat('.').concat(rol).concat('->excludes(self)')
```

3.2.7 NewAcyclic Operation

The operation *newAcyclic* results in the creation of an instance of the stereotype *Acyclic.*

The parameters needed are an association, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *Acyclic* will be created, the constrained elements and the namespace will be the association. This operation can be defined in OCL as follows:

```
context Acyclic:: newAcyclic (a:Association, name:String[0..1],
                                    representation:RepresentationType)
let rol:String = if a.memberEnd->last().name->isEmpty()
                 then a.memberEnd-> last().class.name
                 else a.memberEnd->last().name
in
post: acyc.oclIsNew() and acyc.oclIsTypeOf(Acyclic) and acyc.name=name and
      acyc.constrainedElement->includes(a) and
      a.memberEnd->last().class.ownedRule->includes(acyc) and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      acyc.specification=expr and
      representation=RepresentationType::ocl implies
            expr.language='OCL' and
            expr.body='context '.concat(acyc.context.name).concat(' def:
            successors():Set(').concat(acyc.context.name).concat(') =self.').concat(rol).
            concat('->union(self.').concat(rol).concat('.successors()) context').
            concat(acyc.context.name).concat(' inv'). concat(name).
            concat(': self.successors()->excludes(self)')
```

3.2.8 NewPathInclusion Operation

Additional Operations

We define an additional operation that, given a sequence of properties, returns a String representing the corresponding path. This operation is defined in the abstract stereotype *PathComparison* and will be used when constructing the OCL expressions of the concrete path comparison constraints.

```
context PathComparison
def: givePath(path: Sequence(Property)): String = 'self'.concat(path->
iterate(p:String; expr: String | '. '.concat(expr.concat(p.name)))
```

Creation Operation

The operation *newPathInclusion* results in the creation of an instance of the stereotype *PathInclusion*.

The parameters needed are a class, two sequences of properties that represent the paths, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *PathInclusion* will be created, and that the constrained element and the namespace will be the given class. This operation can be defined in OCL as follows:

```
context PathInclusion::newPathInclusion (start: Class, path1: Sequence(Property),
path2: Sequence(Property), name:String[0..1], representation: RepresentationType)
post: pi.oclIsNew() and pi.oclIsTypeOf(PathExclusion) and pi.name=name and
      pi.constrainedElement->includes(start) and
      start.ownedRule->includes(pi) and
      pi.path1 = path1.name and pi.path2 = path2.name and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      pi.specification=expr and
      representation= RepresentationType::ocl implies
            expr.language=’OCL’ and
            expr.body='context '.concat(start.name).concat(' inv: ').
            concat(givePath(path1)).concat('-> includesAll(').concat(givePath(path2)).
            concat(')')
```

3.2.9 NewPathExclusion Operation

The operation *newPathExclusion* results in the creation of an instance of the stereotype *PathExclusion*.

The parameters needed are a class, two sequences of properties that represent the paths, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *PathExclusion* will be created, and that the constrained element and the namespace will be the given class. This operation can be defined in OCL as follows:

```
context PathExclusion::newPathExclusion (start: Class, path1: Sequence(Property),
path2: Sequence(Property), name:String[0..1], representation: RepresentationType)
post: pe.oclIsNew() and pe.oclIsTypeOf(PathExclusion) and pe.name=name and
      pe.constrainedElement->includes(start) and
      start.ownedRule->includes(pe) and
      pe.path1 = path1.name and pe.path2 = path2.name and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      pe.specification=expr and
      representation= RepresentationType::ocl implies
            expr.language=’OCL’ and
            expr.body='context '.concat(start.name).concat(' inv: ').
            concat(givePath(path1)).concat('-> excludesAll(').concat(givePath(path2)).
            concat(')')
```

3.2.10 NewPathEquality Operation

The operation *newPathEquality* results in the creation of an instance of the stereotype *PathEquality*.

The parameters needed are a class, two sequences of properties that represent the paths, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *PathEquality* will be created, and that the constrained element and the namespace will be the given class. This operation can be defined in OCL as follows:

```
context PathEquality::newPathEquality (start: Class, path1: Sequence(Property),
path2: Sequence(Property), name:String[0..1], representation: RepresentationType)
post: pe.oclIsNew() and pe.oclIsTypeOf(PathExclusion) and pe.name=name and
      pe.constrainedElement->includes(start) and
      start.ownedRule->includes(pe) and
      pe.path1 = path1.name and pe.path2 = path2.name and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      pe.specification=expr and
      representation= RepresentationType::ocl implies
            expr.language='OCL' and
            expr.body='context '.concat(start.name).concat(' inv: ').
            concat(givePath(path1)). concat('=').concat(givePath(path2))
```

3.2.11 NewValueComparison Operation

Additional Operations

We define an additional operation that, given a sequence of properties, returns a String representing the corresponding path:

```
context ValueComparison
def: giveExpression(value: Sequence(TypedElement)): String =
if value->size() = 1 then value.name
else 'self'.concat(value->
      iterate(v:String; expr: String | '. '.concat(expr.concat(v.name)))
```

Creation Operation

The operation *newValueComparison* results in the creation of an instance of the stereotype *ValueComparison*.

The parameters needed are a property, the operator, the value, the name of the constraint and the way to represent this constraint in the schema. Note that value is of type *Sequence(TypedElement)* including either a constant or a path. The postconditions guarantee that a new instance of *ValueComparison* will be created, that the constrained element will be the indicated property and the namespace will be the class that the property belongs to. This operation can be defined in OCL as follows:

```
context ValueComparison:: newValueComparison (attr: Property, op: ValueOperator,
      value: Sequence(TypedElement), name:String[0..1], representation:
      RepresentationType)
post: vc.oclIsNew() and vc.oclIsTypeOf(ValueComparison) and vc.name=name and
      vc.constrainedElement->includes(attr) and
      attr.class.ownedRule->includes(vc) and
      vc.operator = op and vc.value=value.name and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      vc.specification=expr and
      representation= RepresentationType::ocl implies
            expr.language='OCL' and
            expr.body='context '.concat(attr.class.name).concat(' inv: ').
            concat(attr.name).concat(op).concat(giveExpression(value))
```

3.2.12 NewMandatoryDisjoint Operation

The operation *newMandatoryDisjoint* results in the creation of an instance of the stereotype *MandatoryDisjoint.*

```
context MandatoryDisjoint:: newMandatoryDisjoint(c: Class, a:Set(Set(Property)),
representation:RepresentationType)
post: manDis.oclIsNew() and manDis.oclIsTypeOf(MandatoryDisjoint) and
      manDis.name=name and
      manDis.constrainedElement -> includes(c) and
      manDis.disjointGroups -> includes(a) and
      c.ownedRule->includes(manDis) and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      manDis.specification = expr and
      representation=RepresentationType::ocl implies
            expr.language = 'OCL' and
            expr.body= 'context '.concat(manDis.context.name).
            concat(' inv:').concat(Sequence{1.a->size()}-> iterate (pn; s: String = '' |
            s.concat((if (pn>1) then ') or ' else'' endif).
            concat(a->at(pn)->Sequence{1.a->at(pn)->size()}->
            iterate (qn; s2: String = '(' |s2.concat((if (qn>1) then ' and '
            else'' endif).concat(a->at(pn)->at(qn).name).
            concat('->notEmpty() ')))))).concat(')')
```

3.2.13 NewCardinalityAssoc Operation

Additional Operations

We define an additional operation that, given a property, returns a String that represents its name or, in case it is empty, the name of its owner class.

```
context CardinalityAssoc
def: giveName(p:Property):String=if p.name->isEmpty()
                                 then p.class.name
                                 else p.name endif
```

Creation Operation

The operation *newCardinalityAssoc* results in the creation of an instance of the stereotype *CardinalityAssoc*.

The parameters needed are an association, the name of the constraint and the way to represent this constraint in the schema. The postconditions guarantee that a new instance of *CardinalityAssoc* will be created, the constrained elements and the namespace will be the association. This operation can be defined in OCL as follows:

```
context CardinalityAssoc:: newCardinalityAssoc (a:Association, fe:OrderedSet(Property),
re:Set(Property), min:String, max:String, name:String[0..1],
 representation:RepresentationType))
let forAllfe:String=fe->iterate(e:Property;acc:String=''|
acc.concat(a.memberEnd-> select(m|m=e).class.name).
concat('.allInstances()->forAll(v').concat(giveName(e)).concat('|'))
let assoc:String=a.name.concat('.allInstances()')
```

```
let sel1:String='->select(t|t. '.concat(giveName(fe.first())).concat('=v').
concat(giveName(fe.first())).concat(' ')
let sel2:String=fe->subOrderedSet(2,fe->size())->
iterate(e:Property;acc:String=' '|acc.concat('and t. ').concat(giveName(e)).
concat('=v').concat(giveName(e)).concat(' '))
let pars:String=fe->iterate(e:Property;acc:String=' '|acc.concat(') '))
let collect1:String='->collect(t|Tuple{'
let auxcollect2:String=re->iterate(e:Property;acc:String='' |
acc.concat('c').concat(giveName(e)).concat('=t. ').concat(giveName(e)).
concat(', '))
in
let collect2:String=auxcollect2.substring(1,auxcollect2.size()-1)
let collect3:String='})'
in
post: cainv.oclIsNew() and cainv.oclIsTypeOf(CardinalityAssocInv)
      and cainv.name=name and
      cainv.constrainedElement->includes(a) and
      a.memberEnd->select(m|m=fe.first()).class.ownedRule->includes(cainv) and
      cainv.fixedEnds=fe->iterate(e:Property acc:OrderedSet(String)=
      OrderedSet{}| acc.append(giveName(e))) and
      cainv.relatedEnds=re->iterate(e:Property acc:Set(String)=
      Set{}| acc.including(giveName(e))) and
      cainv.minCard=min and
      cainv.maxCard=max and
      expr.oclIsNew() and expr.oclIsTypeOf(OpaqueExpression) and
      cainv.specification=expr and
      representation=RepresentationType::ocl implies
            expr.language='OCL' and
            expr.body='context '.concat(cainv.context.name).concat(' inv: ').
            concat(forallfe).concat(assoc).concat(sel1).concat(sel2).concat(pars).
            concat(collect1).concat(collect2).concat(collect3).
            concat('->asSet()->size()>=').concat(min).concat(' and ').
            concat(forallfe).concat(assoc).concat(sel1).concat(sel2).concat(pars).
            concat(collect1).concat(collect2).concat(collect3).
            concat('->asSet()->size()<=').concat(max)
```

4. Case Study

This section summarises the results obtained from the application of our profile to the specification of two real-life applications. The analysis of both schemas allows us to stress the advantages of using the profile. In particular, we have analysed a conceptual schema for the well-known EU-Rent Car Rentls system [FQO03] and we have also studied a generic conceptual schema for the e-marketplace domain [QT05].

EU-Rent is a (fictitious) car rental company with branches in several countries. The company rents cars to its customers who may be individuals or companies. Different models of cars are offered, organized into groups and cars within a group are charged at the same rates. The class diagram we have studied consists of 59 classes, 50 associations and 40 constraints that require an explicit definition. Our profile prevents us from specifying in OCL a considerable amount of said constraints. Only 14 out of 40 do not correspond to any of our stereotypes and, thus, a specific OCL expression needs to be constructed to specify them.

Figure 15 shows a small fragment of the EU-Rent class diagram (10 classes and 16 constraints) to further illustrate the conclusions we have drawn from the development of this case study. The first seven constraints may be specified by applying the *Identifier* stereotype since they state the attributes that identify each class. Constraints 8 and 9 may be specified by applying the *ValueComparison* stereotype.

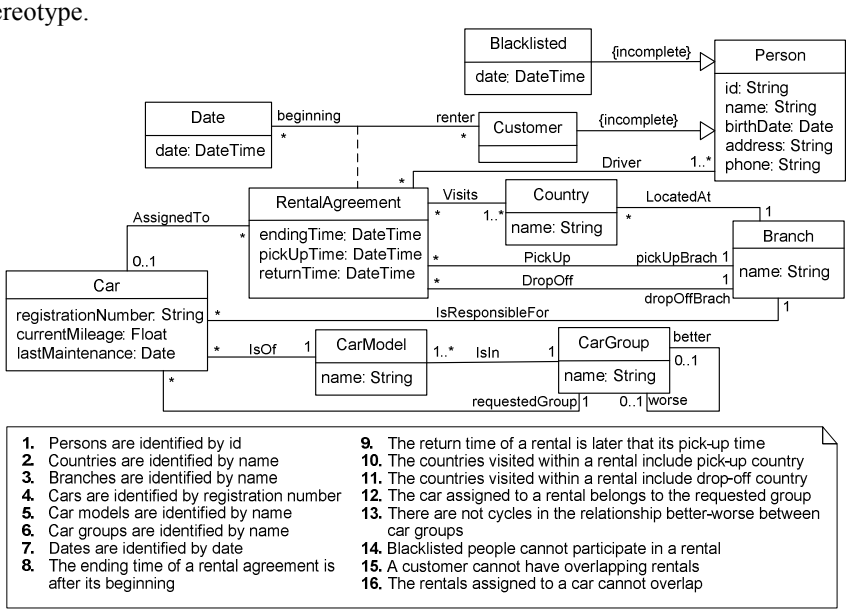

Fig. 15. Fragment of EU-Rent class diagram

Constraints 10 and 11 correspond to the *PathInclusion* stereotype; 12 corresponds to the *PathEquality* stereotype and 13 corresponds to the *Acyclic* stereotype. Finally, constraints 14, 15 and 16 do not match any of our predefined constraints and thus an ad-hoc OCL expression must be built to specify them.

The second case study consists of the specification of a generic conceptual schema for the e-marketplace domain [QT05] which covers the main functionalities provided by an e-marketplace: determining product offerings, searching for products and price discovery. The whole specification includes 40 classes, 15 associations and 41 constraints that require an explicit definition. After analysing the constraints, the results obtained are quite similar to those obtained with EU-Rent. In this case, the success rate is a bit lower, about 54% instead of 65% as before, but still interesting. This means that we have to specify manually only 19 out of 41 OCL constraints.

From the results of both case studies, we see that it has been possible to use our stereotypes almost in 60% of the constraints, by reducing the number of OCL expressions from 81 to 33.

5. Reasoning and Generating Code

One of the main benefits of the proposed profile is the ability of reasoning about constraints represented as instances of our stereotypes and their automatic code generation into a given technological platform. In the following we show how our profile facilitates reasoning about constraint satisfiability and constraint redundancy. We outline also how to use the profile to generate code for checking those constraints in a relational database.

5.1 Constraint Satisfiability

A conceptual schema is *satisfiable* if it admits at least one legal instance of the IB. For some constraints it may happen that only the empty or non-finite IBs satisfy them. In conceptual modeling, the IBs of interest are finite and may be populated. We then say that a schema is *strongly satisfiable* if there is at least one fully populated (i.e each class and association has at least one instance) instance of the IB satisfying all the constraints [LN90]. Otherwise, the schema is incorrect.

Constraint satisfiability has received a lot of attention in conceptual modeling. For instance, [Hal01] presents in the Euler diagram in Figure 16 the relationships between recursive association constraints. Some satisfiability rules can be deduced from the figure. For instance, a recursive association with an acyclic and a symmetric invariant is not strongly satisfiable because there can not exist instances in the IB of the corresponding association that satisfy, at the same time, both invariants.

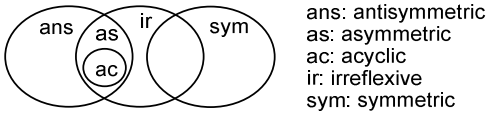

Fig. 16. Relationships between recursive association constraints

Unfortunately, and as a consequence of problems associated to the definition of general constraints, known results in constraint satisfiability checking cannot be applied to the definition of constraints by means of general-purpose languages. For example, in Figure 1, the designer could define another constraint that defines the association *isPrerequisiteOf* as symmetric. As explained before, this new invariant makes the schema incorrect.

Our proposal allows us incorporating easily some of these results. In fact, the definition of predefined constraints as stereotypes permits to attach new constraints that represent well-studied satisfiability rules that detect if a set of constraints is strongly satisfiable. Table 1 summarizes the stereotypes and the constraints we have attached to them to incorporate the results presented in [Hal01]. Other known results for constraint satisfiability can be incorporated in the same way.

Table 1. Validation of recursive association constraints

Stereotype	**Constraint attached to the stereotype**
Symmetric	There cannot be another instance of acyclic, asymmetric nor antisymmetric constraint for the same association
Antisymmetric	There cannot be another instance of symmetric constraint for the same association
Asymmetric	There cannot be another instance of symmetric constraint for the same association
Acyclic	There cannot be another instance of symmetric constraint for the same association

5.2 Constraint Redundancy

A conceptual schema is *redundant* if an aspect of the schema is defined more than once [CST02]. For instance, a constraint is redundant with respect to another constraint if in each state of the IB that violates the latter, the former is also violated.

We may also draw from the diagram shown in Figure 6 some rules that permit to detect some redundancies between recursive association constraints. For example, an acyclic constraint is redundant with respect to an asymmetric constraint of the same association because asymmetric associations are always acyclic.

Our proposal also allows incorporating easily results on constraint redundancy. Table 2 summarizes the stereotypes and the constraints we have attached to them to incorporate rules that detect redundancies between recursive association constraints. Other results can be incorporated in a similar way.

Table 2. Redundancy of recursive association constraints

Stereotype	**Constraint attached to the stereotype**
Irreflexive	There cannot be another instance of asymmetric nor acyclic constraint for the same association
Antisymmetric	There cannot be another instance of asymmetric nor acyclic constraint for the same association
Asymmetric	There cannot be another instance of antisymmetric nor irreflexive nor acyclic constraint for the same association
Acyclic	There cannot be another instance of asymmetric nor antisymmetric nor irreflexive constraints for the same association

5.3 Automatic Code Generation

Many UML CASE tools offer code generation capabilities. However, most of them do not generate the code required to check whether constraints defined in general-purpose languages are violated by the execution of a transaction. We outline in this section how our profile may be used to facilitate such important task.

As we have seen, each stereotype explicitly states a precise semantics for the type of constraints it defines. Semantics may be taken into account during code generation to determine the most adequate translation from the conceptual schema to a particular technology. Thus, assuming an implementation on a relational database, identifier constraints could be translated into primary key or unique constraints; weak identifiers into foreign key plus primary key constraints; value comparisons into check constraints and other constraints by means of triggers or stored procedures. For example, classes Course, Section and their constraints would be translated as follows:

```
CREATE TABLE Course (
   name    char(30)    PRIMARY KEY,
   code    char(30)    UNIQUE,
   creditsNumber  int   NOT NULL)
```

```
CREATE TABLE Section (
   nameCourse    char(30),
   number        int,
   numbOfStud    int,
   PRIMARY KEY (nameCourse, number),
   CONSTRAINT fkSect  FOREIGN KEY (nameCourse)
                      REFERENCES Course(name) )
```

6. Related Work

In this section, we analyze other works that contribute to facilitating the definition of general constraints in UML.

Executable UML (xUML) is a profile of UML that allows defining an information system in sufficient detail that it can be executed [MB02]. As part of its proposal, xUML extends the set of constraints that can be graphically specified. In particular, it covers our uniqueness constraints and some kinds of path comparison constraints, i.e. path equality and path inclusion. Considering the EU-Rent and the e-marketplace case studies, xUML would cover only 28% of the constraints instead of the 60% covered by our proposal. Moreover, we provide the profile definition in terms of the UML 2.0 metamodel including the definition of the creation operations that permit to add instances to the stereotypes.

Ackermann [Ack05] proposes a set of OCL specification patterns that facilitate the definition of some UML integrity constraints, namely what we call identifier constraints and a subset of value comparison constraints. When applied to our case studies it covers only 26% of the constraints. This approach is based on the automatic generation of OCL expressions from a set of patterns and, thus, it does not extend the language via a profile definition as we propose. Consequently, it does not extend the set of UML predefined constraints which facilitates their graphical representation. Furthermore, it does not use the established mechanisms to extend the language and, thus, it can not be directly incorporated to UML CASE tools.

In [MN05] a large taxonomy of integrity constraints (which includes constraints that are inherent, graphical and user-defined in UML) is analyzed. The authors

advocate the definition of stereotypes for some of them. They leave the stereotype definition for future work but propose that model elements such as associations and attributes should be taken as base class for their definition. We think instead that all the proposed stereotypes should be stereotypes of *Constraint*. The reasons are that the semantics of *Constraint* corresponds to the purpose of the stereotypes, it permits to graphically represent the incorporated constraints similarly as predefined constraints and, finally, it facilitates a uniform treatment of the incorporated constraints together with the rest of constraints of a UML class diagram.

In addition to already stated drawbacks of previous proposals, we must note that none of them deals with the ability of reasoning about the general constraints they may handle.

7. Conclusions and Future Work

We have proposed a new approach to facilitate the definition of general constraints in the UML. Our approach is based on the use of constraint stereotypes in conceptual modelling and it allows defining as predefined UML constraints some types of general constraints that are frequently used, instead of having to specify them by means of a general-purpose sublanguage such as OCL.

Being able to specify general constraints as predefined constraints we overcome the limitations of having to define them manually which may usually imply a time-consuming and error-prone definition, difficulty of understanding (since the reader may not be familiar with the formal language used to define the general constraint) and difficulty of automatic treatment (since general constraints do not have a pre-established interpretation while predefined ones do).

We have applied our approach to the specification of two real-life applications: the EU-Rent Car Rentals system [FQO03] and a conceptual schema for the e-marketplace domain [QT05], and we have seen that 60% of the general constraints of those case studies may have been defined as predefined by means of our stereotypes.

Finally, we have also incorporated into our stereotypes previous results regarding constraint satisfiability and constraint redundancy checking. This has been easily done by attaching to our stereotypes well-established rules that detect whether a set of constraints is strongly satisfiable [Hal01] and redundancies between recursive association constraints. We have also outlined how to automate code generation from our profile to check integrity constraints in a relational database.

Since one of the main goals of our paper has been to illustrate the advantages provided by the use of constraint stereotypes, we have not intended to be exhaustive in the extent of predefined constraints considered. Future work may involve the definition of other types of frequent general constraints. We also plan to incorporate into our stereotypes other known results for reasoning about constraints and to further develop the automatic code generation from our stereotypes.

References

[Ack05] Ackermann, J. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components, MoDELS Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, EPFL - Technical Report LGL-REPORT-2005-001, pp. 15-29, 2005.

[CST02] Costal, D., Sancho, M. R., Teniente, E., Understanding Redundancy in UML Models for Object-Oriented Analysis. 14th Int. Conf. on Advanced Information Systems Engineering (CAiSE'02), LNCS 2348, 659-674.

[FQO03] Frías, L., Queralt, A., Olivé, A., EU-Rent Car Rentals Specification. Departament de LSI, UPC, Technical Report LSI-03-59-R, 2003.

[Hal01] Halpin, T. Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design. Morgan Kaufmann. 2001.

[ISO82] ISO/TC97/SC5/WG3, J.J. van Griethuysen (Ed.). Concepts and Terminology for the Conceptual Schema and the Information Base, 1982.

[LN90] Lenzerini, M., Nobili, P.: On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. Information Systems(4), pp. 453-461 1990.

[MB02] Mellor, S.J; Balcer, M.J. Executable UML: A Foundation for Model-Driven Architecture. Object Technology Ed. Addison-Wesley. 2002.

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

[Oli03] Olivé, A. "Integrity Constraints Definition in Object-Oriented Conceptual Modeling Languages", In Proc. ER'03, LNCS 2813, pp.349-362.

[OMG03] OMG. "MDA Guide Version 1.0.1", OMG, omg/2003-06-01, 2003.

[OMG05] OMG. "UML2.0 OCL Specification", OMG Adopted Specification, 2005.

[QT05] Queralt, A., Teniente, E., A Platform Independent Model for the Electronic Marketplace Domain. Departament de LSI, UPC, Technical Report LSI-05-9-R, 2005.

[RJB05] Rumbaugh, J., Jacobson, I., Booch, G., The Unified Modeling Language Reference Manual, Second Edition, Addison-Wesley, 2005.

[WK03] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. 2nd edn. Addison-Wesley Professional, 2003.