Contrary-To-Duties Constraints: from UML to Relational Model

Pedro de Paula Nogueira Ramos
ADETTI /ISCTE Computer Science Department
Av. Forças Armadas, Lisboa, 1649-026 Portugal
Email: Pedro.Ramos@iscte.pt

ABSTRACT

Sometimes, because of an atypical situation, an important mandatory association between classes in a UML Class Diagram must be replaced by an optional one. That semantic and functional impoverishment happens because the mandatory constraint must have a boolean value. In this paper we analyze the use of soft constraints in the UML Class Diagram, and their automatic repercussion in the corresponding Relational Model. The soft (deontic) constraints allow the formal representation of requirements, which ideally should always be fulfilled, but can be violated in atypical situations. In this paper we enrich a previous deontic approach, by introducing the ability to explicitly represent the so called Contrary-To-Duties requirements, i.e., domain integrity requirements that emerge as a consequence of an unfulfilled mandatory constraint. We support our approach with the UML/OCL language.

Keywords: UML, Contrary-To-Duties, Relational Model , Deontic Constraints

1. INTRODUCTION

The paper addresses the formal representation of soft constraints in database applications. More specifically, the paper deals with a relational model enrichment that allows a flexible notion of the mandatory property in foreign key field. We have chosen to have a formal and a high level approach, and for that reason we have adopted the UML [1] notation together with the representation of OCL constraints [2].

In this paper we follow the intuitions and some of the work presented in [3]. In [3] the author has already proposed a relational model enrichment that allows a more flexible notion of mandatory fields (e.g., foreign keys fields). The motivation was to help the database designer in situations where a mandatory foreign key would be the natural solution, but cannot be implemented due to minor, although unavoidable, atypical situations.

The boolean mandatory attribute is adequate for requirements that must hold unavoidably, but is not adequate to deal with requirements that ideally should always be fulfilled, but can be violated in atypical situations. If those violable requirements are explicitly represented it is possible to maintain both the requirement and its violation and, consequently, recur to monitoring procedures for violation warnings. Without that explicit representation, the designer must always choose a non mandatory value if there is a chance, even if a small one, that in an atypical situation the field will not be filled.

Consider the following real example[1] concerning the overtime hours control in an organization where sometimes the employees work for several days outside the organization (in the client's organization).

An organizational rule states that all overtime hours must be authorized by the employee hierarchic superior. Using a workflow system the employee fills in a requirement form that, if authorized by his superior, will be stored in a table record in which the ID of the superior must be filled in. However, sometimes the need for overtime hours is only detected when the employee is working outside the organization. A problem may arise if the employee is outside during a change of month (the problem is related with technical issues regarding the total amount of monthly overtime hours). When the employee is working outside with his superior the usual procedure consists of the employee sending a fax or an email (and then someone fills the requirement form for him) and his superior making a phone call or sending a fax authorizing the overtime hours. Due to security organizational procedures no one can electronically authorize overtime hours without a proper password. When the superior is outside and cannot access the system, the system cannot accept overtime hours. The solution adopted by the organization was to remove the requirement, which stated that all records of the overtime hour authorization table should always have the superior's ID.

The important aspect of the previous example is that, because of one atypical situation (which nevertheless happens several times) an important requirement was abandoned (the overtime authorization control is now made manually, where previously it was validated by the database). That happened because the requirement was inflexible. In [3] the flexible mandatory requirement is introduced in the Unified Modelling Language (UML) Class Diagram. The author also provides some guidelines about the automatic generation of the corresponding relational model (basically suggesting the utilization of triggers, but not providing guidelines on how to do it).

In this paper we strongly enrich the approach presented in [3] (deontic approach), enhancing its ability of application to real systems. The examples presented in [3] only consider situations where the violations of soft constraints (called deontic obligations) don't have consequences to the application behavior. When a soft constraint isn't fulfilled the system only has to report it. If we think in terms of the relational model, in those situations it is only necessary to maintain a set of views that reports the records where the "mandatory" fields aren't fulfilled. In this paper we present three new enrichments:

First we introduce in the UML Class Diagram the ability to explicitly represent the new constraints that emerge as a consequence of the violations of deontic obligations (Contrary-To-Duties constraints). We enrich the Class Diagram in order to allow the representation of sub-ideal states, i.e., situations where deontic obligations are violated and consequently new constraints (obligations) arise to deal with that undesired but tolerated situation.

Secondly we support our approach with the standard UML constraint language, the OCL language. In [3] the OCL representation of the constraints is completely missing and, consequently, a solid automatic mechanism to generate a corresponding

[1] The author worked as a consultant in the organization where the example comes from.

relational model was impracticable.

Finally we provide some rules and objective guidelines for the automatic generation of trigger and views that support the database integrity maintenance.

The reason to focus our attention on the relational model is the fact that Object Databases Management Systems aren't yet an efficient solution for demanding applications. The object-oriented paradigm is becoming the main background for system modeling, but unfortunately the object-oriented databases don't progress at the same speed, making relational databases the standard for data storing. The reason for choosing UML is the fact that it has become a standard language for design and conception of systems.

The paper is organized as follows: in section 2 we present and motivate the deontic approach presented in [3], providing also a corresponding OCL representation. In section 3 we present our extension, e.g., the ability to represent in a UML diagram (and using OCL) the domain integrity requirements that emerge as a consequence of an unfulfilled boolean mandatory constraint. In Section 4 we focus our attention on the Relational Model generation. Some concluding remarks are presented in the last section.

2. Deontic Constraints

In the relational model the attribute mandatory property is specified through a boolean value: *required* or *not required*. That binary representation is enough for most situations. The requirement that states that *all Invoices must have a date* means that the attribute date must be always fulfilled, i.e. it should be impossible to have an invoice without a date. That restriction is also valid in the real world, i.e., invoices without a date are illegal.

However, that simple approach is not always sufficient to capture some properties of the world. For example, let us consider the following requirement: "All students must have a zip code address". That requirement exists due to the fact that the university regularly needs to send correspondence to the student. However, is that a requirement that intends to capture an ideal situation or a requirement that the database should always fulfill? What will be the procedure if one student tries to register himself in the school and has forgotten his zip code? If we want to conditionally accept his registration (telling him that he must supply the zip code as soon as possible) then the requirement is about an ideal situation that sometimes doesn't happen (neither in the real world nor in the database). Blocking the registration could be a wrong choice because, apart from the fact that all data already inserted in the application form will be lost, the school will convey an unpleasant image of unnecessary bureaucracy. The zip code will be indispensable in the future, but during a short period of time the zip code is dispensable.

The previous example intends to distinguish two kinds of requirements:

(i) requirements that ideally should always be fulfilled, but can be violated in atypical situations and;

(ii) requirements that must hold unavoidably.

The boolean mandatory attribute is adequate for the requirements covered by the second situation, but is not adequate to deal with ideal but violable requirements.

Violable requirements, i.e., requirements that should hold but sometimes do not, must be represented differently from other requirements. Otherwise, requirement violations will originate an inconsistent database state. If the violable requirement is explicitly represented it is possible to maintain both the requirement and its violation and, consequently, recur to monitoring procedures for violation warnings.

Consider the student/zipcode example in which the relational model is represented with predicate calculus sentences and zip code is a student attribute:

a) Requirement: $\forall_x \exists y$ (student(x)$\rightarrow$ zipcode(y,x))
b) Facts: student(ann), $\neg\exists$y zipcode(ann,y)

In order to avoid an inconsistent database (i.e., zipcode(ann,x) and $\neg$ zipcode(y,x)) a) should be replaced by:

a') Requirement: *Ideally* ($\forall_x \exists y$ (student(x)$\rightarrow$ zipcode(y,x)))

If we consider that a violation occurs if *Ideally(Φ) and* $\neg\Phi$, we can have a deduction system to automatically infer all the violations.

In the UML graphic notation the mandatory property is only represented in the associations' cardinality. In the UML Class Diagram, the cardinality lowest limit in a relation between classes usually takes the value 0 (Not Mandatory) or 1 (Mandatory). Consider the examples of Figure 1.

In the first example the several zip codes are represented as objects of the Zip Code Class. In the situations described before the constraint may be too strong, i.e., all students must always have a zipcode.

In the second example of Figure 1, regarding a DVD Store, given the cardinality relation, a DVD must always be associated to a Category (in order to be consulted by the clients in a computer). That relation represents a typical situation, i.e., all movies have at least a category. However, sometimes that strong restriction can be inconvenient, or even impossible to keep. Consider the situation where a set of movies has just arrived to the store. A plausible situation will be that an employee immediately registers the new movies in the database. Afterwards, another employee (probably someone with more knowledge about movies) updates the database in order to assign categories to the movies. In that scenario, during a period of time the movie isn't associated to any category. Blocking that situation could have unnecessary and undesirable consequences in the process flow (not allowing the immediate registration of movies in the database).

One may argue that, given the desired functionality, a different association should be chosen, i.e., cardinality with a zero lower limit (0…*). However, that solution can hide important properties of the reality (e.g., that a DVD has at least one category). Also, if that solution was to be adopted, it would be impossible to detect the "violations" of the desired situations.

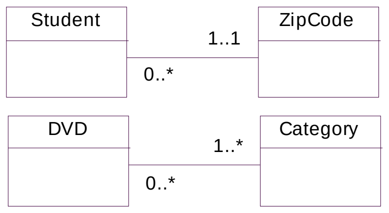

Figure 1 Mandatory Relations

In [3] it is proposed to incorporate the notion of ideality in the UML associations. By ideality we mean a constraint that is not mandatory, but whose violation must be registered in the database.

Going back to the examples of Figure 1, the diagram should explicitly say that all DVDs must have a category (and students a zipcode), but that constraint can be violated. Also, it should become explicit that those violations must be registered.

The reason why the registration of the violations should be explicit in the diagram is because that feature should be a database built-in feature and not implementation dependent. Like integrity constraints are part of the relational model, this notion of ideality became part of our automatically generated relational model mentioned in the last section.

To represent the ideality mandatory constraint (called “Obligation” in the figure label) the diagrammatic representation depicted in Figure 2 is proposed. Notice that no new symbols or concepts are introduced in the UML diagram.

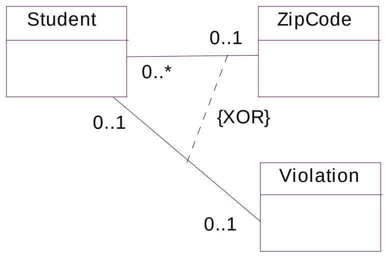

Figure 2 Deontic Constraint (Obligation) Representation

The expression ‘Obligation’ comes from the deontic logic. There are already several works on deontic logic applied to databases [4]. In [5] and [6] the authors introduce the notion of Deontic Constraints (also called Soft Constraints). By deontic constraint they mean “constraints which express norms that indicate how things are in ideal worlds, how things ought to be. […] such constraints are violable”. In their approach all information and knowledge is represented through mathematical logic. They do not consider the relational model or diagrammatic representations of any kind.

In [3] the author doesn’t present a clear semantics to the Class Violation of Figure 2 (it’s only presented as a set of violations). In this paper we adopt the approach first presented in [8]. Deontic rules are represented with violation constants: the previous expression *Ideally(Φ)* (or *Obligation(Φ))* is represented as $\neg V_i \rightarrow \Phi$, in which V_i represents a violation constant. The authors consider a finite set of violation constants, each of them associated to one deontic rule. The semantics is intuitive: $\neg V_i \rightarrow \Phi$ means that if rule i isn’t violated then Φ is a fact (in other words, rule i states that there is an obligation to ensure Φ). We consider that the Class Violation corresponds to the finite violation constants ΔV ($\forall_i \; V_i \in \Delta V$).

The disjunction (XOR) means that if the student does not have a zip code then a violation will be associated with the student[2]. Several classes can be associated with

[2] Notice that this representation is ambiguous. Since the relation is bidirectional, it may represent that there is an obligation to have a zipcode associated to all students (the intended meaning) or, there is an obligation

the Violation Class, as exemplified in Figure 4 (ideally all students should have a zip code and all degrees should have a Coordinator Professor). Each violation object is associated only with one object.

In order to simplify the diagrams, in Figure 3 and Figure 5 an abbreviation is used to represent the Deontic Constraint (Deontic Association): the underlined zero represents the obligation / ideality concept.

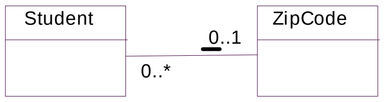

Figure 3: Deontic Constraint Representation Abbreviation (Deontic Association)

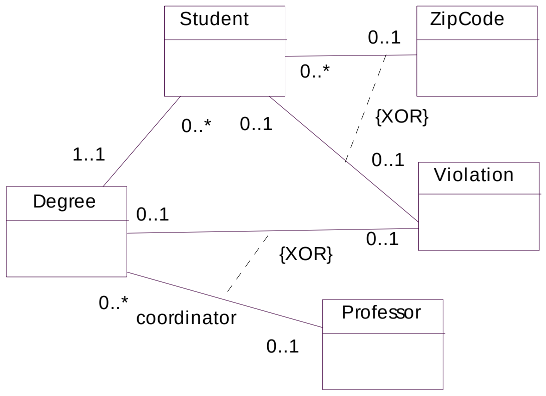

Figure 4 More than one constraint representation in a diagram

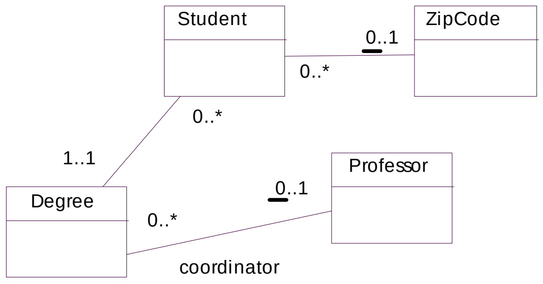

Figure 5 Abbreviation of the diagram of Figure 4

In [3] the OCL representation of the constraints is completely missing and, consequently, a solid automatic mechanism to generate a corresponding relational model is impracticable. Object Constraint Language is a declarative language to

to associate all zipcodes to students (a non intended meaning). This situation can be avoided with the UML navigation association name [1] or with OCL expressions like the one we present further on.

describe constraint rules that apply to UML models. OCL syntax is intuitive enough, therefore we won't present its syntax in this paper.

Each OCL expression is written in the context of an instance of a specific class. Our deontic constraints can be represented as invariant conditions. An OCL expression is an invariant of the class and must be true for all instances of that class at any time.

The following expression represents our proposal to model the obligation depicted in Figure 3 (that is an abbreviation of Figure 2). The context is the Student Class (ST_ZC and Viol-ST_ZC are the role names of the right side of the associations between Student and Zip Code and Student and Violation, respectively)

Inv Oblig-Sudent_ZipCode:
ST_ZC->isEmpty() implies Viol-ST_ZC->notEmpty()

The previous representation is a specific instance of the more general schema: on the left side of the material implication we have the role of the 'deontic association end', and on the right side the violation-side role of the association between the class and the Violation Class.

3. CONTRARY-TO-DUTIES

In this paper we introduce in the UML Class Diagram the ability to explicitly represent the new constraints that emerge as a consequence of the violations of deontic obligations. We enhance the Class Diagram in order to allow the representation of sub-ideal states, i.e., situations where deontic obligations are violated and consequently new constraints arise to deal with that undesired but tolerated situation. In the deontic literature [7] those new constraints are called Contrary-To-Duties constraints.

Consider an application that supports the budget control of a building company[3]. Every building project has its own budget. The budget is disaggregated into several items. When the project leader adjudicates a new work to a supplier, he fulfills a Purchase Order (PO) (to be delivered to the supplier) and also fulfills a Withhold Request (WR) (to ensure that the money will be available when the payment takes place). The Withhold Request (WR) is only allowed by the application if there is enough money in the corresponding budget item (a PO regards a specific item). In order to control the budget the following rule is implemented in the application: **every PO must be associated to a WR**. Consequently, POs are only allowed if there is enough money available in the budget item.

However there are situations where adjudications must take place (the PO must be created) even if there is no budget (for example, unpredictable works). In such situations the standard procedure is to request a budget rearrangement (for example, to exchange values between items). That request takes some time (even days) to be analyzed and sometimes the urgency of the work forces the project leader to *violate* the control rule (for example, an imminent land falling that requires a sustentation

[3] The example illustrates a real implementation done by the author for a building company.

wall). In order to allow the violation of the rule, the application must accept that sometimes a PO is not associated to a WR. Given that flexible interpretation, what the rule really states is that **ideally every PO must be associated to a WR**.

In order to ensure a rigorous budget, when the rule is violated (a PO is not associated to a WR), apart from the Budget Rearrange Request (BRR), the project leader must organize a work meeting to elaborate a formal minute that justifies the decision to adjudicate the new work (in this kind of meeting a third party entity – surveillance company - is always present). Notice that 'new obligation' (to have a minute signed by the three entities: the project leader, the supplier and the third party company) also represents an ideal situation. Sometimes (rare situations) the adjudication must occur before the meeting takes place. So, when the first rule is violated (a PO is not associated to a WR) two new situations arise that can be expressed with two new rules:

1. If a PO is not associated with a WR then a BRR is necessary;
2. If a PO is not associated with a WR then the PO must be associated with a Minute Meeting (MM).

The first rule must always be fulfilled, which means that the application always rejects a PO that is not associated with either a WR or a BRR. The second one represents an ideal situation, which means that the application allows a PO that isn't associated either with a WR or with a MM. This second rule has the same semantics of the initial one; both represent obligations whose violation is accepted by the application.

Contrarily to the first example (Zip Code), in this example, from the violation of one obligation new constraints are derived. Those new constraints are called Contrary-To-Duties in the Deontic Logic Literature. Following this literature we will use the term Necessity to represent the first constraint of the example (situations that must necessarily occur). For the second constrain we will continue to use the term Obligation since it has the same meaning of the original rule (and the rule of the first example).

In the diagram of Figure 6 we propose the use of the UML Dependency Relation to capture the notions of Necessity and Obligation. The stereotypes must be read as follows: "the PO Rearrangement: PO_BRR Necessity and the Minute Meeting PO: PO_MM Obligation depends of the PO_WR violation". The OCL dependency constraint is described in Table 1. As we presented earlier, the diagram of Figure 6 is an abbreviation of the diagram of Figure 7 (in where we explicitly represents the Violation Class).

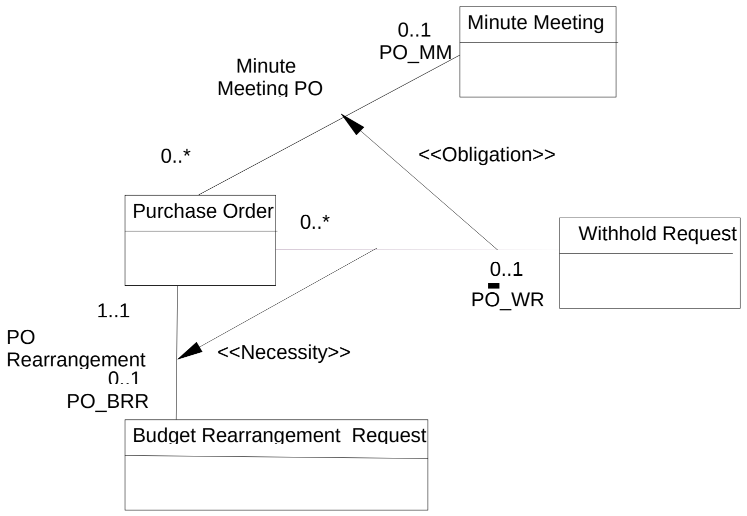

Figure 6 Purchase Order Example

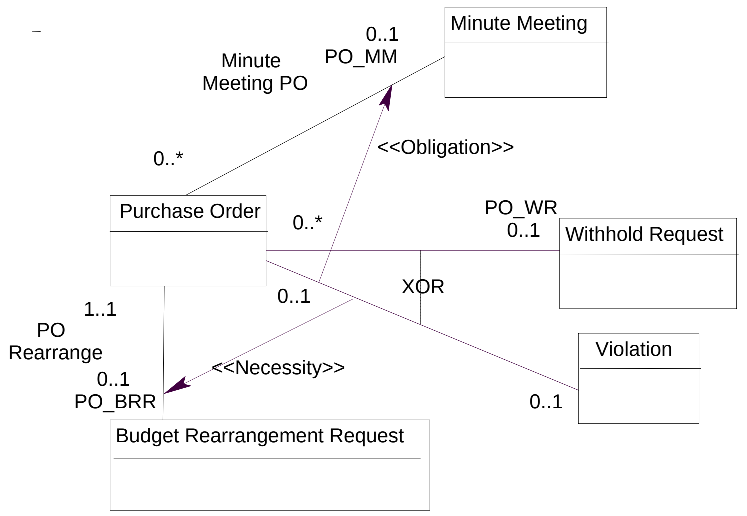

Figure 7 Purchase Order Example (non abbreviated version)

Notice that the derived obligation cannot be represented as illustrated in Figure 8. That representation means the following: **ideally every PO must be associated to an MM**. But that isn’t the case in the example. The association must only ideally occur if the PO isn’t associated with a WR.

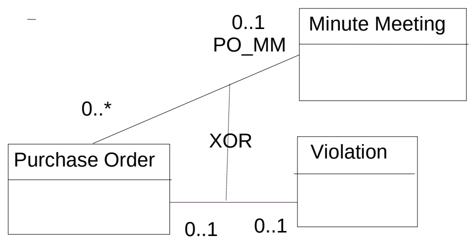

Figure 8 An erroneous representation of the derived obligation

```
Purchase Order
Inv Oblig-PO_WR:
    PO_WR->isEmpty() implies Viol-PO_WR->notEmpty()
Inv CDutie_Necessity-PO:
    Viol-PO_WR->notEmpty() implies PO_BRR ->notEmpty()
Inv CDutie_Obligation-PO:
    Viol-PO_WR->notEmpty() implies  (PO_MM->isEmpty() implies Viol-PO_MM-
>notEmpty())
```

Table 1 OCL Constraints for the Purchase Order Example

In the current database domain, in the presence of Contrary-to-Duties scenarios, a different kind of constraint may arise: constraints, which state that a specific data association is **forbidden** until some constraint (‘obligation’) is fulfilled. We propose using a stereotyped dependency relation called Forbidden. Consider the following example:

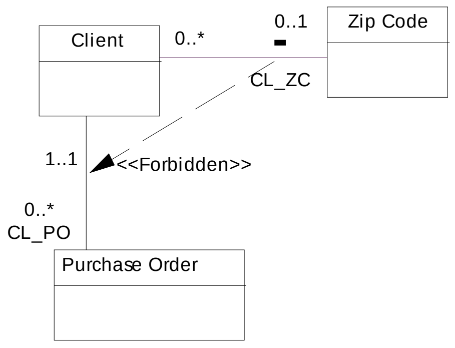

Figure 9 Forbiddingness in a UML Class Diagram

The dependency relation stereotype called “forbidden” means that “the association ‘Client P Order’ cannot occur until the association ‘Zipcode Client’ happen (clients without the zip code cannot place orders). In OCL:

```
Context: Client
Inv CDutie_Forbidden-ZC:
    Viol-CL_ZC->notEmpty() implies  CL_PO->isEmpty()
```

Table 2 OCL Constraints for the Client Purchase Order Example

4. MAPPING UML DEONTIC CONSTRAINTS INTO RELATIONAL ONES

As mentioned earlier, our main goal is to, taking the UML Class Diagram as a starting point, automatically generate the corresponding relational model and its constraints. Since we want to keep the deontic constraints in the database, triggers are the best choice to implement them. Using Before and After action triggers it is possible to ensure that all constraints are fulfilled. The procedure for monitoring the obligations fulfillment can be easily achieved using relational views. For each constraint (OCL invariant) we can generate a SQL view, which retrieves the records that don't satisfy it. That's, for example, what the Dresden Toolkit does (one of the few OCL-SQL generator, [9]). However, trigger and view generations aren't enough if we want to help the database administrator to maintain the data integrity. If the set of deontic constraints is only represented in code (SQL), it becomes very difficult to maintain it and understand it. We consider that, apart from trigger and view generation, a relational table should be created to store the obligations. That table would hold all the deontic associations (obligations) and the new constraints (dependency relations stereotypes) that arise as a consequence of their violation.

The Deontic Table has seven attributes:

- a distinctive identifier (the table primary key);
- the name of each table that implements the association (for example, Student and Zip Code in the first example, or, Client and Purchase Order in the last example);
- the name of the foreign key (two attributes, one for the table name and the other for the foreign key name), which relates the table that implements the class with the deontic constraint[4] (Client in the first example and Purchase Order in last example[5]) to the associated one (Zip Code in what regards the first example and Client in what regards the last example);
- one attribute to classify the kind of deontic relationship (i for Ideal[6], f for Forbidden, o for Obligatory and n for Necessity) and;
- a last one for the dependency relations, i.e., to relate a Forbidden or Obligatory or Necessity with the corresponding Ideal relation (the link is made using the primary key attribute).

[4] The table whose records are subject to constraints.

[5] The table Purchase Order is the owner of the foreign key that has the constraint. New POs can only be assigned to clients that have a zipcode (the constraint regards the Client ID in the PO table).

[6] The primary obligation.

In Figure 10 we illustrate the data structure that supports the table and in Table 3 the previous examples are represented.

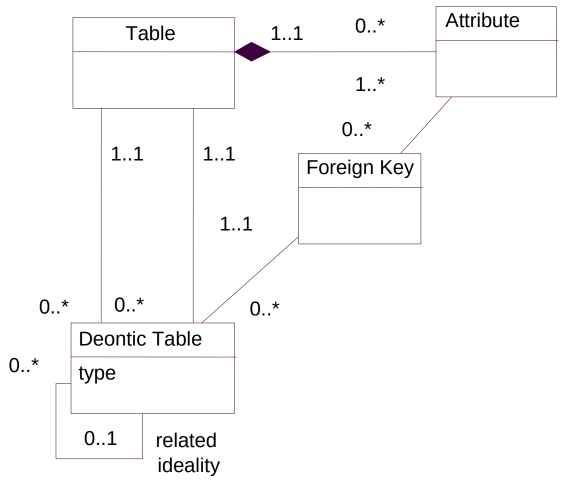

Figure 10 The table structure

1	Student	ZipCode	Student	Zipcode_fk	i	
2	Client	ZipCode	Client	Zipcode_fk	i	
3	Client	Purchase Order	Purchase Order	Client_fk	f	2
4	Purchase Order	Withhold Request	Purchase Order	Withhold_fk	i	
5	Purchase Order	Minute Meeting	Purchase Order	minute_fk	o	4
6	Purchase Order	Budget Rearrange Request	Purchase Order	Budget_fk	n	4

Table 3 Deontic Table: Examples of Deontic Table values

In the example, the second record means that there is an ideality (column 6) deontic relation between the tables Client and ZipCode (columns 2 and 3) that is implemented in the Client (column 4) Zipcode_fk (column 5) foreign key attributes. Notice that columns 2 and 3 are redundant because they can be obtained from columns 4 and 5. However, they are very useful to ensure an efficient and readable generation program.

The third record means that there is a forbidden (column 6) deontic relation between tables Client and Order (columns 2 and 3) associated to the previous ideality deontic relation (column 7).

Tables that implement the UML classes that are arguments of deontic associations (only classes located on the opposite side of the underlying zero abbreviation) will have a foreign key for each deontic association. The foreign key domain is the distinctive identifier of the Deontic Table (the foreign key represents the association between the class and the Violation Class). That foreign key will only assume a Null value when a violation occurs. For example, given the previous diagrams, the tables Student, Client and Purchase Order will have one foreign key each. Furthermore, each of the previous tables will also have one more foreign key for each <<Obligation>> dependency relation that represents a Contrary-To-Duties scenario of the previous deontic association. For that reason the Purchase Order table will have a second foreign key.

Views for violation warnings

OCL Inv Oblig-PO_WR: (Ideal)

```
Select * from Purchase_Order, Deontic_Table where
Deontic_Table.idconstraint=
Purchase_Order.POWR_iddeonticconstraint
```

OCL Inv CDutie_Obligation-PO: (Obligation)

```
Select   *   from   Purchase_Order,   Deontic_Table   As   DT1,
Deontic_Table As DT2 Where
DT1.idconstraint=Purchase_Order.POMM_iddeonticconstraint   and
DT2.idconstraint=DT1.CDTiddeonticconstraint
```

Trigger for integrity maintenance

```
Inv CDutie_Necessity-PO: (Necessity)
Before insert Purchase_Order
if (new.BRRID is null and new.POWR_iddeonticconstraint is
null) then return CancelEvent

Inv CDutie_Fordidden-ZC: (Fordidden)
Before insert Purchase_Order
Select ClientID from Client into ClientFlag where
new.ClientID=Client.ClientID and
Client.ZC_iddeonticconstraint is not nul
if (ClientFlag is not null) then return CancelEvent
```

OCL Inv Oblig-PO_WR: (Ideal)

```
After insert Purchase_Order
if (new.WHID is null) then POWR_iddeonticconstraint=4
```

OCL Inv CDutie_Obligation-PO: (Obligation)

```
After insert Purchase_Order
if (new.WHID is null and new.MMID is null) then
Purchase_Order.POMM_iddeonticconstraint =5
```

Table 4 Examples of Triggers[7] and Views for OCL constraints in Table 1 and Table 2

In order to facilitate the trigger generation and maintenance, the Foreign Key Referential Integrity Strategies for all foreign keys related to the Deontic Table should be Restricted. If we do so, we strongly restrict the number of triggers necessary to ensure all deontic constraints. Throughout the paper we present four different types of deontic constraints: Ideal, Obligatory, Forbidden, and Necessary. The first two constraints don't need any trigger support (Obligatory needs trigger support if arises as a consequence of a Contrary-To-Duties scenario). Views are sufficient to monitor violations.

[7] For the sake of clarity we use an informal and simple trigger syntax

In order to ensure the Necessity constraint it will be necessary to generate a Before Insert trigger. That trigger will cancel the insertion if a violation associated with the new record exists and the ‘necessary record’ in the opposite table doesn’t exist.

In order to ensure the Forbidden constraint it will be necessary to generate a Before Insert trigger. This trigger will cancel the insertion if a violation exists and the new record will be connected with a record of the forbidden table.

5. Final Remarks and related work

In this paper we have extended our previous work in order to introduce in a UML representation the ability to explicitly represent the so called Contrary-To-Duties requirements, i.e., domain integrity requirements that emerge as a consequence of an unfulfilled boolean mandatory constraint. We only used standard UML notation together with UML stereotypes (Obligation, Forbidden, Necessity) to capture all the notions we needed. We supported our approach with the standard UML constraint language, the OCL language.

This approach, together with the deontic table and the SQL guideline generation presented in the last section, provides us with a framework that will support an automatic generation of the extended relational (deontic) model (based on the extended Class Diagram).

Borgida, in his work with exceptions in information systems (originally in [10] with further extensions, e.g., [11]) considers that exceptional situations arises when some constraints are violated, and that exceptions are considered as violations. In his proposal, the occurrence of a violation is signaled by the creation of an object in a class called ANY_VIOLATION. The author proposes an exception handling mechanism to specify failure actions. As Borgida, we also represent explicitly in the object model the constraints violations. Borgida uses two classes, one for the violations itself and another for the violation constraints. We only use the second one because the violations itself are represented by the foreign keys. Apart from these similarities, the approaches are considerably different. Borgida proposes a much more general mechanism to deal with exceptions handling in object oriented programming languages. Our approach, apart from been only oriented on one particular constraint (that Borgidas work doesn’t treat), is focused on the database generation. Borgida, contrary to us, explicitly rejects triggers approached because he wants to maintain the control in a middleware software level. What we call Contrary-To-Duties constraints aren’t treated by Borgidas work.

In the future it will be critical to automatically obtain the views and the triggers. In this paper we have provided some concrete guidelines, but we haven’t yet implemented the automatic procedure.

Furthermore, in the near future it will be necessary to cover more complex situations like relations with more than two arguments and composite obligations like the ones depicted in Figure 11.

Given the diagram we can infer that all courses should be associated to a degree and all courses should be associated to a coordinator. But it could be the case that the two obligations must only be treated as a conjunction and not separately (i.e., the obligation to be assigned to a coordinator disappears if the course isn’t assigned to a degree).

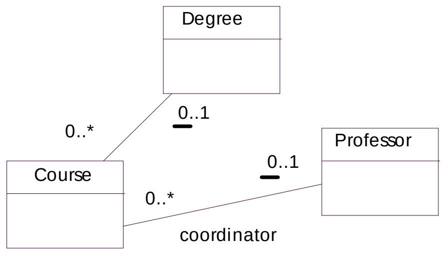

Figure 11 CompositeObligations

6. References

[1] Boock, Grady; Rumbaugh, James; Jacobson, Ivar, 1998. The Unified Modeling Language Reference Manual. In Addison-Wesley object technology series.

[2] Warmer, Jos; Kleppe, Anneke, 2003. The Object Constraint Language: Getting Your Models Ready for MDA (2nd Edition). Ed The Addison-Wesley Object Technology Series .

[3] Ramos, Pedro, 2003. Deontic Constraints: From UML Class Diagram To Relational Model. In *ECEIS* April, France.

[4] Wieringa, R.J.; Meyer, J., 1991. Applications of Deontic Logics in Computer Science: a concise overview In J. Meyer and R.J. Wieringa, , editors, *Procs. First Int. Workshop on Deontic Logic in Computer Science*.

[5] Carmo, José ; Demolombe, Robert; Jones, Andrew, 2001. An Application of Deontic Logic to Information System Constraints. In, *Fundamenta Informaticae* 46, pp 1-17, IOS Press

[6] Meyer, J.; Wieringa, R.J.; Dignum, F., 1998. The Role Of Deontic Logic in the Specifications of Information Systems. In J. chomicki and G. Saake, editors, *logic for Database and Information Systems*. Kluwer.

[7] Carmo, José; Jones, Andrew, 1996. A New Approach to Contrary-To-Duty Obligations. In, *Defeseasible Deontic Logic*, I, Donald Dute (ed.), Synthese Library

[8] Tan, Y.; der Torre, L., Representing Deontic Reasoning in a Diagnostic Framework, in ILCP'94 Workshop on Legal Applications of Logic Programming, Genova, Italy, 1994

[9] Dresden CL Toolkit : http://dresden-ocl.sourceforge.net/

[10]Borgida, Alexander, 1985. Language features for flexible handling of exceptions, in ACM Transactions on Database Systems (TODS), 1985

[11]Borgida, Alexander; Murata, Takahiro, 1999 Tolerating Exceptions in Workflows: A unified framework for Data and Process. in Proc. International Joint Conference on Work Activities Coordination and Collaboration (WACC), USA,1999