Towards computer-aided design of OCL constraints

Yves Ledru, Sophie Dupuy-Chessa, and Hind Fadil

Laboratoires LSR et CLIPS - IMAG
B.P. 72 - F-38402 - Saint Martin d'Hères Cedex - France
{Yves.Ledru, Sophie.Dupuy, Hind.Fadil}@imag.fr

Abstract. In UML2.0, significant efforts have been devoted towards a better definition of OCL. Still, the adoption of the language by the software engineers remains a significant challenge. This paper discusses the problem of helping UML analysts to express OCL constraints that link a pair of attributes from different classes. This help consists of finding a navigation path between the classes, and then choosing the best context to express the simplest constraint.

1 Introduction

OCL (Object Constraint Language [1]) is an important part of the UML [2]. It allows analysts to define boolean expressions associated to classes or objects. The most common use of such expressions is the definition of invariant constraints on elements of a class diagram and the specification of methods in terms of pre- and post-conditions. These invariant constraints, pre- and post-conditions complement the information of the class diagram and increase its expressiveness. In UML2.0 [3], significant efforts have been devoted, amongst others in order to

- better integrate OCL in the UML meta-model: an abstract syntax has been defined using the same metamodelisation approach as for the other UML diagrams. As a result, OCL concepts such as classes, methods, attributes or relations now clearly refer to their counterparts in the UML diagrams.
- provide a formal semantics to OCL: a mathematical semantics has been defined which makes it more precise and should allow the development of several tools (e.g. compilers and animators) which take advantage of this semantics.
- facilitate the expression and the synchronisation of actions: new constructs have been incorporated that allow the exchange of messages between objects.

These improvements should favour the use of OCL and make it a significant competitor to formal object oriented specification languages such as JML[4]. Still, a difficulty remains, which is intrinsic to every textual formal modeling language: the expression of a constraint is not always a trivial task. In this paper, we discuss simple techniques that should assist a UML analyst to express OCL constraints.

We focus on two difficult issues:

- navigation and quantification: many OCL constraints relate two attributes from different classes. The expression of the constraint requires the analyst to navigate through the diagram in order to link the relevant objects. This includes the search for a path between these objects, and if several paths exist, the selection of a path which meets the intended semantics of the constraint.
- choice of the right "context" to express the constraint: an OCL constraint is expressed in the "context" of a class which constitutes the starting point for navigation. This class is usually the starting or the end point of the path that links the classes involved in the constraint. Actually, any class of the path can be used as the context of the OCL constraint. In this paper, we show that the choice of the context can significantly simplify the expression of the constraint, amongst others by diminishing the number of quantifiers involved in the constraint.

Unfortunately, such tools have their limits, the most significant one being that tools can not figure out what is the exact semantics of a constraint. Therefore, they can only assist the analyst who should be able to read the produced constraint and understand what the tool helped him to express.

This idea of using tools to help in a modeling or programming process is not new. In CAD/CAM tools (Computer Aided Design/Computer Aided Manufacturing), geometrical modeling involves the definition of geometrical objects and of relations or constraints between these objects. For example, such tools allow the engineer to pick up a circle and a line and to express the constraint that the line should be tangent to the circle. Actually, once the analyst has selected these two objects, a pop-up menu appears and proposes typical constraints that apply to a line and a circle: e.g. tangency or perpendicularity. The engineers, who know these notions can then select the most appropriate one for the item under construction. Similarly, commercial spreadsheets provide "wizards" to help you define a complex function for a given cell. Here the wizard usually starts by prompting the user for the function he wants to define (e.g. standard deviation) and then asks him to click on the cells that provide the argument(s) to the function.

This paper explores the design of similar tools to assist the definition of OCL constraints. First, section 2 will present an example that illustrates our approach. The following section will show how to find navigation paths in a class diagram. Then, we will address the problem of choosing the best context in order to facilitate the constraint expression. Finally we will discuss the limitations and future investigations of this work.

2 Example

In order to illustrate this paper, let us consider the information system of an airline company. The system manages the commercial offer of the company:

it defines the routes covered by the company, the schedules of the flights that correspond to these routes, the price of the flights depending on the season of the year. The information system also includes aspects linked to the management of airplanes in the company. It records which airplane is intended to perform a given flight at a given date.

A simple version of the airline company database is described by the following UML class diagram (Fig. 1):

– "ROUTE" corresponds to the routes served by the company. They are described by the departure and arrival airports and the distance between them.
– "FLIGHT" represents the regular flights proposed for a given route. Several flights may correspond to the same route. Each of them is characterized by its scheduled departure and arrival times.
– "FL_INST" describes the flight instances. In this simple specification, we only record the date of a given flight instance.
– "AIRPLANE" models the airplanes ensuring flight instances. Each airplane has an identification number and a range that defines the maximum distance it can fly.
– "SEASON" defines the periods during which flights occur. So it has a start and an end dates.
– The price of a flight is represented by an association attribute between "FLIGHT" and "SEASON" as it is calculated according to the season and the flight.

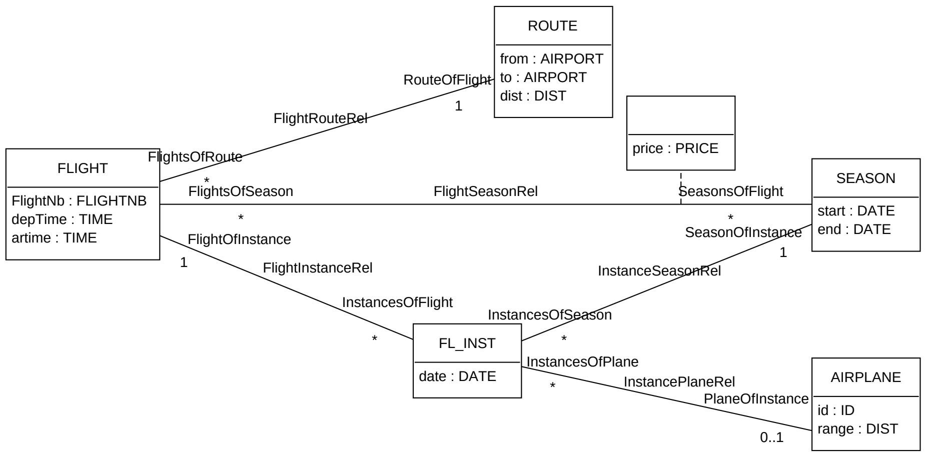

Fig. 1. UML class diagram for an airplane database

This class diagram cannot express all the characteristics of the airline database. For instance, the following constraints complete the model description:

– a flight instance must occur between the start and the end dates of its season;

- the range of an airplane must be greater than the distances of the routes that it flies;
- an airplane cannot fly two flight instances at the same time;
- the season of a flight instance is included in the seasons of the flight of this instance.

UML suggests to express these constraints in OCL [5]. For example, the first constraint that specifies that a flight instance occurs between the start and the end dates of its season can be written:

```
context FL_INST inv:
    self.date ≥ self.SeasonOfInstance.start and
    self.date ≤ self.SeasonOfInstance.end
```

This constraint is an invariant (keyword *inv*) expressed in the context of the class "FL_INST". So *self* represents an object of "FL_INST" and the expression "self.SeasonOfInstance" is the season of the object *self* obtained by navigating through the association "InstanceSeasonRel". The first line of the constraint expresses that the date of a flight instance must be greater than or equal to the start date of its season and the second line expresses that the date of a flight instance must be less than or equal to the end date of its season.

The expression of the second constraint is more complex. It involves the classes "AIRPLANE" and "ROUTE" that are not directly linked by an association. So we must find a path between "AIRPLANE" and "ROUTE". Here we choose to go through "InstancePlaneRel", "FlightInstanceRel", and "FlightRouteRel". Once a path is identified, we have to choose among the classes of the path the context of the constraint. According to the context chosen, the constraint will be more or less easy to write and to understand. If we use AIRPLANE as the context of this constraint, we get the following invariant:

```
context AIRPLANE inv:
    self.InstancesOfPlane ->
        forAll (IN | self.range ≥ IN.FlightOfInstance.RouteOfFlight.dist)
```

The first line of the constraint expresses that from *self* we can get the set of flight instances corresponding to the airplane. The second line means that for all flight instances of this set, the distance of the route of the corresponding flight must be smaller than or equal to the range of the airplane.

These two constraints give examples of the kind of constraints that we will consider in this paper i.e. constraints on attributes of different classes. We will use them to illustrate the questions to answer in order to make the constraints on attributes as simple as possible:

- what is the best navigation path between two classes whose attributes must be compared in the constraint?
- what is the best context to easily express the constraint?

3 Finding Navigation Paths

Many OCL constraints relate different classes. So their expression requires to navigate through the class diagram to link classes i.e. to find navigation paths between classes in the diagram. We define a path as a set of contiguous associations which link the selected classes. For example, a path which links PLANE and ROUTE is:

$$\{FlightRouteRel, FlightInstanceRel, InstancePlaneRel\}$$

According to the number of paths, the work of the constraint writer will be different.

Existence of only one navigation path If there is only one path between the classes, the constraint writer has simply to check that this path semantically corresponds to the link he wants between the classes.

Absence of navigation path If there is no path between the classes, the constraint cannot be expressed in the diagram. This often reveals a modelisation flaw. For example, some association is missing in the diagram.

A similar problem may arise if there is a path between the classes but there are too many restrictions on the navigability of the roles. In this case, the constraint writer may have to add navigable links between the classes in order to be able to write his constraint.

Existence of several navigation paths If there are several paths between the classes, the constraint writer has to identify paths that are semantically correct and choose one that will facilitate the constraint expression. For instance, two paths can be found between "FL_INST" and "SEASON" to express the constraint on the date of a flight instance (a flight instance occurs between the start and the end dates of its season):

- the first one is direct through the association "InstanceSeasonRel";
- the second one goes through the associations "FlightInstanceRel" and "FlightSeasonRel". But it does not correspond to the relevant semantics for the constraint. As a matter of fact, the constraint implies to find a single season which corresponds to the flight instance (a flight instance must occur between the start and the end dates of its season). But if the navigation path goes from "FL_INST" to "SEASON" through the roles "FlightOfInstance" and "SeasonsOfFlight", we find several seasons corresponding to the seasons of the flight of the instance.

So here, there is only one semantically correct path between "FL_INST" and "SEASON", the direct one through the association "InstanceSeasonRel". But this is not always the case. For example, let us modify the class diagram by introducing the concept of "FARE" instead of "SEASON" (Fig. 2): unlike

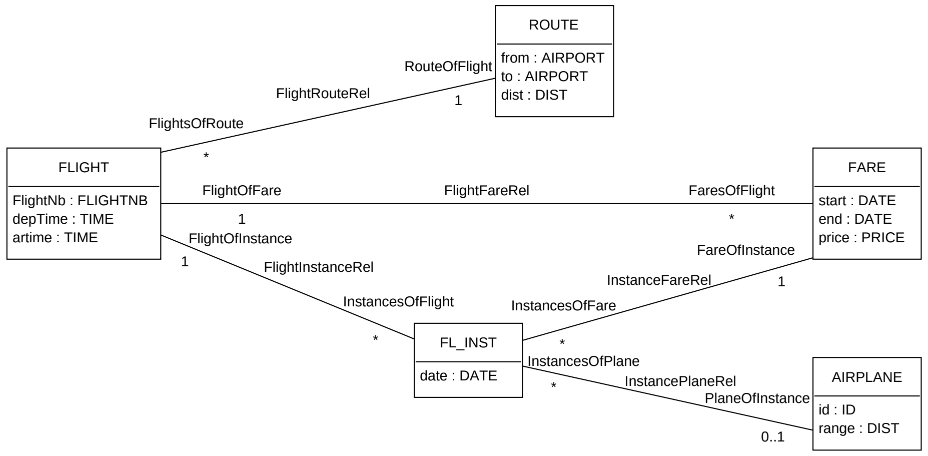

Fig. 2. The airplane database with FARE

seasons, each fare is specific to one flight. So "price" becomes an attribute of "FARE" and the role "FlightOfFare" is monovaluated.

In this model, there are two ways to find the flight corresponding to a given flight instance. The shortest path is to follow "FlightOfInstance", but another path goes through "FareOfInstance" and "FlightOfFare". As a result, this diagram also includes two paths between "AIRPLANE" and "ROUTE" to express the second constraint. For example, the second constraint of Sect. 2 (range of an airplane greater than the distance of the corresponding routes) expressed in the context of AIRPLANE along this new path is:

```
context AIRPLANE inv:
    self.InstancesOfPlane ->
        forAll (IN | self.range ≥ IN.FareOfInstance.FlightOfFare.RouteOfFlight.dist)
```

4 Choosing The Constraint Context

Even when there is only one semantically correct path, there remains several ways to express a constraint. If we consider the constraint that specifies that a flight instance occurs between the start and the end dates of its season (constraint related to Fig. 1), the previous section concludes that the direct path between "FL_INST" and "SEASON" is the only semantically correct path. Then the constraint can be written in two different contexts: "FL_INST" and "SEASON".

```
context FL_INST inv:
    self.date ≥ self.SeasonOfInstance.start and
    self.date ≤ self.SeasonOfInstance.end
```

```
context SEASON inv:
    self.InstancesOfSeason ->
        forAll (IN | IN.date ≥ self.start and IN.date ≤ self.end)
```

In this simple example with a direct path, the constraint is much simpler to express in the context of "FL_INST". This is due to the fact that in the first expression, the role from "FL_INST" to "SEASON" is monovaluated while the role from "SEASON" to "FL_INST" is multivaluated. The multivaluation of a role implies the use of a *forAll* expression which makes constraints more complex. So one criterion to evaluate a constraint complexity can be the number of multivaluated roles in the path: the lesser are multivaluated roles in a path, the simpler will be the path.

Let us now consider the example of the range of an airplane (the range of an airplane must be greater than the distances of the routes that it flies). In Fig. 1, there are two possible paths between "AIRPLANE" and "ROUTE". But the one that goes through "SEASON" does not correspond to the intended semantics. The remaining path includes the following associations: InstancePlaneRel, FlightInstanceRel and FlightRouteRel. Classes "AIRPLANE", "FL_INST", "FLIGHT" and "ROUTE" are potential contexts to state the constraint. We end up with four equivalent expressions of the same constraint:

```
context AIRPLANE inv:
    self.InstancesOfPlane ->
        forAll (IN | self.range ≥ IN.FlightOfInstance.RouteOfFlight.dist)        (C-1)

context FL_INST inv:
    self.PlaneOfInstance.range ≥ self.FlightOfInstance.RouteOfFlight.dist        (C-2)

context FLIGHT inv:
    self.InstancesOfFlight ->
        forAll (IN | IN.PlaneOfInstance.range ≥ self.RouteOfFlight.dist)         (C-3)

context ROUTE inv:
    self.FlightsOfRoute ->
        forAll (FL | FL.InstancesofFlight ->
            forAll (IN | IN.PlaneOfInstance.range ≥ self.dist))                  (C-4)
```

Although these constraints are equivalent, they involve different numbers of quantifiers, depending on the context. Actually, the number of quantifiers can be predicted before writing the constraint. For each context, we can navigate towards the ends of the path and count the number of multivaluated roles. This is summarized in the following table.

Navigation (context in bold face)	Number of multivaluated roles
ROUTE (n) FLIGHT (n) FL_INST (1) AIRPLANE	2
AIRPLANE (n) FL_INST (1) FLIGHT (1) ROUTE	1
FLIGHT (n) FL_INST (1) AIRPLANE / **FLIGHT** (1) ROUTE	1
FL_INST (1) FLIGHT (1) ROUTE / **FL_INST** (1) AIRPLANE	0

So according to this table, the best context to express the range constraint is the context of "FL_INST". By looking at the constraint expressed in OCL, it seems to be a good choice as the understanding of constraint in the context of "FL_INST" is simplified by the fact that there is no *forAll* expression.

But in the OCL constraints above, we can also note that constraints with shorter navigation expressions are easier to read. So the length of the navigation expressions can also be used as a criterion to compare constraints. For the example, this criterion gives the following results:

Navigation (context in bold face)	Maximum number of roles
ROUTE (n) FLIGHT (n) FL_INST (1) AIRPLANE	3
AIRPLANE (n) FL_INST (1) FLIGHT (1) ROUTE	3
FLIGHT (n) FL_INST (1) AIRPLANE / **FLIGHT** (1) ROUTE	2
FL_INST (1) FLIGHT (1) ROUTE / **FL_INST** (1) AIRPLANE	2

According to this criterion, the simplest constraint expressions are written in the contexts of "FLIGHT" and "FL_INST". If the importance of the length of navigation path is not really highlighted by this example, this criterion can be a every good one to compare contexts between classes of paths that are composed of different numbers of classes. For instance, we apply the two criteria to the path between "AIRPLANE" and "ROUTE" that go via "FARE" in Fig. 2: "InstancePlaneRel", "InstanceFareRel", "FlightFareRel", and "FlightRouteRel". The path here includes one more class than the previous one that do not go through "FARE".

	Number of multivaluated roles	Maximum number of roles
ROUTE (n) FLIGHT (n) FARE (n) FL_INST (1) AIRPLANE	3	4
AIRPLANE (n) FL_INST (1) FARE (1) FLIGHT (1) ROUTE	1	4
FLIGHT (n) FARE (n) FL_INST (1) AIRPLANE / **FLIGHT** (1) ROUTE	2	3
FL_INST (1) FARE (1) FLIGHT (1) ROUTE / **FL_INST** (1) AIRPLANE	0	3
FARE (1) FLIGHT (1) ROUTE / **FARE** (n) FL_INST (1) AIRPLANE	1	2

According to this table, two contexts should be considered. "FL_INST" is the best context to avoid the use of quantifiers, while "FARE" leads to the shortest navigation expressions. The resulting constraints are:

```
context FL_INST inv:
    self.PlaneOfInstance.range ≥ self.FareOfInstance.FlightOfFare.RouteOfFlight.dist

context FARE inv:
    self.InstancesOfFare ->
        forAll (IN | IN.PlaneOfInstance.range ≥ self.FlightOfFare.RouteOfFlight.dist)
```

Although we feel that none of this constraints is better than constraint `(C-2)` expressed on the shorter path, it is interesting to compare these constraints to the other constraints from the shorter path. It is our personal feeling that these are probably simpler than `(C-4)` which corresponds to a shorter path but involves two quantifiers.

Actually, it is up to the analyst to choose the constraint that appears the simplest, but we believe that applying these criteria will help him reduce his choice to the most interesting expressions.

5 Conclusion

This paper has discussed the problem of helping a user to express an OCL constraint for a given class diagram. Starting from the selection of a pair of attributes by the user, it is possible to find out all possible paths which link these attributes. For each of these paths, the constraint may be expressed in a variety of contexts, corresponding to the classes that appear on the paths. It is possible to automate these tasks (find the possible paths, and then express the constraint in every context). A tool has been developed in our laboratory [6], which takes a pair of attributes and a comparison operator and lists the possible expressions of the constraints for all paths and all contexts. The user has then to choose amongst the proposed constraints the one that: (a) corresponds to the intended semantics of the paths if several paths exist between the attributes and (b) appears the "simplest" to understand.

Limits of this approach The approach presented in this paper can only be applied to a specific kind of constraint: it must link a pair of attributes from different classes. Sect. 2 has listed four constraints for Fig. 1. The last two constraints do not correspond to this kind. One of them ("an airplane cannot fly two instances at the same time") links the same attributes for a pair of objects of the same class. The difficulty of this constraint is to select the set of flight instances corresponding to an airplane and then to compare these flight instances in a pairwise manner. The last one ("the season of a flight instance is included in the seasons of the flight of this instance") expresses the inclusion of a set of objects in another one.

Perspectives The major limit of the approach is the specific kind of constraints it addresses: linking a pair of attributes from different classes. Although we believe it encompasses a significant number of the constraints one would like to express, further work is needed to establish a classification of possible constraints and then design specific helps for some of them.

Another perspective is the development of automated tools to support the approach. The experience we got from our prototype tells us that such a support is feasible and in particular that OCL is a flexible language that eases the automatic construction of these constraints from class diagram information. An interesting challenge is to help the analyst choose amongst the possible constraints that can be expressed automatically. Sect. 4 has shown that syntactical criteria (shortest path, number of multivaluated roles) can be used to sort these constraints. Tool support can take advantage of these criteria to help the analyst. Moreover, the idea of evaluating complexity on the sole basis of syntactical issues may lead to numerous discussions [7]. Still, it is the analyst's responsibility to understand the constraints and pick up the right one.

OCL constraints can definitely help produce more precise and more expressive class diagrams. Although the language has been designed specifically for class diagrams, the expression of OCL constraints is still a complex task and slows down the adoption of the language. We hope that this paper brings some light on the potential for automated support that can be dedicated to constraint expression, and that such tools will contribute to the adoption of such a language.

References

1. Warmer, J., Kleppe, A.: The Object Constraint Language (Second Edition) - Getting your models ready for MDA. Addison-Wesley (2003)
2. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language- User Guide. Addison-Wesley (1998)
3. Group, O.M.: Unified Modeling Language 2.0 proposal. (2003)
4. Leavens, G., Baker, A., Ruby, C.: JML: A notation for detailed design. In Kilov, H., Rumpe, B., Simmonds, I., eds.: Behavioral Specifications of Businesses and Systems. Kluwer Academic Publishers (1999) 175–188
5. Group, O.M.: Response to the UML 2.0 - OCL RfP (ad/2000-09-03. (2003) Revised submission, version 1.6 - OMG document ad/2003-01-07.
6. Fadil, H.: Intégration et génération de contraintes dans la spécification des systèmes d'information. Rapport de DEA, Univ. Joseph Fourier, Grenoble, France (2003)
7. Edmonds, B.: Syntactic Measures of Complexity. PhD thesis, University of Manchester (1999) http://bruce.edmonds.name/thesis/.