chapter 5 Object Constraint Language (OCL)

Soumia LAYACHI

October 2025

1 Introduction

UML (Unified Modeling Language) provides mechanisms for expressing different types of constraints, which can be classified as follows :

- **Structural constraints :** These include attributes within classes, different types of relationships between classes, multiplicity (cardinality), and navigability of structural properties.
- **Type constraints :** These involve the data types assigned to properties.
- **Miscellaneous constraints :** This category includes visibility constraints (e.g., public, private), as well as the definition of abstract methods and classes.

2 Constraints

2.1 Writing Constraints

A constraint is a semantic condition or restriction expressed as a statement in a textual language. This language can be natural or formal. The text string itself constitutes the body of the constraint, which can be written in various languages :

- A natural language (e.g., English, French).
- A dedicated constraint language, such as Object Constraint Language (OCL).
- Or even syntax directly from a programming language.

2.2 Representation of Constraints and Predefined Constraints

UML offers several ways to associate a constraint with a model element, as illustrated in the figures below.

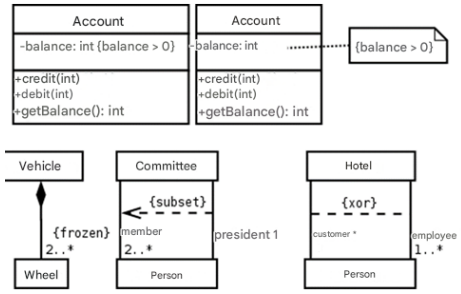

FIGURE 1 – This figure shows different methods of applying constraints in UML. In the top two diagrams, a constraint specifies that an attribute must be positive. In the bottom left, the frozen constraint indicates that the number of wheels on a vehicle cannot change. In the center, the subset constraint specifies that the president is also a member of the committee. In the bottom right, the xor (exclusive or) constraint states that hotel employees are not allowed to book a room at the same hotel where they work.

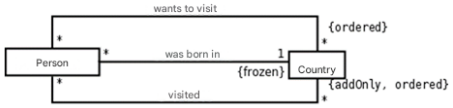

FIGURE 2 – This diagram expresses the following constraints : a person is born in a country, and this association cannot be changed. They have visited several countries, in a specific order, and the number of countries visited can only increase. They have a list of countries they want to visit, and this list is ranked (probably by preference). As we have just seen, UML includes several predefined constraints (frozen , subset , xor , ordered and addOnly). To overcome these limitations, the Object Constraint Language (OCL) offers an elegant and more efficient solution.

3 Types of OCL Constraints

3.1 Invariant Constraints

A condition that must always be true for all instances of a class.

Syntax :

```
context ClassName
inv InvariantName : boolean_expression
```

Example :

```
context BankAccount
inv PositiveBalance : balance >= 0
inv OverdraftLimit  : balance >= -authorizedOverdraft
```

3.2 Precondition Constraints

A condition that must be true before an operation is performed.

Syntax :

```
context ClassName :: operationName(parameters) : ReturnType
pre PreconditionName : Boolean_expression
```

Example :

```
context BankAccount :: withdraw(amount : Real)
pre PositiveAmount     : amount > 0
pre SufficientBalance  : balance - amount >= -authorizedOverdraft
```

3.3 Postcondition Constraints

A condition that must be true after an operation is performed.

Syntax :

```
context ClassName :: operationName(parameters) : ReturnType
post PostconditionName : Boolean_expression
```

Example :

```
context BankAccount :: withdraw(amount : Real)
post Updated : balance = balance@pre - amount
```

4 Class Diagram with OCL

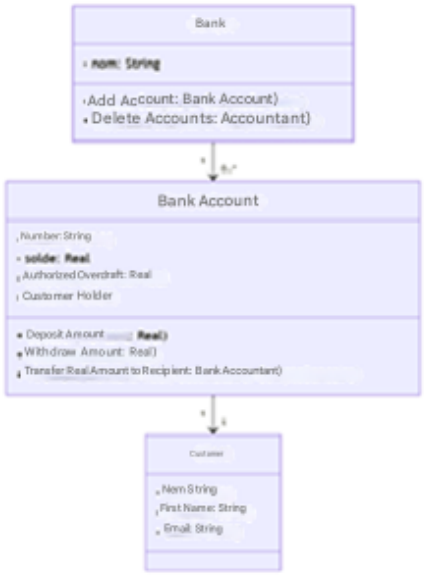

FIGURE 3 – Class diagram for a simple banking system.

4.1 Associated OCL Constraints

Bank Account Constraints

Class Context :

```
context BankAccount
inv NonNegativeBalance : balance >= -authorizedOverdraft
inv ValidOverdraft     : authorizedOverdraft >= 0
```

Deposit Operation :

```
context BankAccount :: deposit(amount : Real)
pre ValidAmount : amount > 0
post            : balance = balance@pre + amount
```

Withdraw Operation :

```
context BankAccount :: withdraw(amount : Real)
pre ValidAmount        : amount > 0
pre SufficientBalance  : balance - amount >= -authorizedOverdraft
post Decreased         : balance = balance@pre - amount
```

Constraint on the Bank

```
context Bank
inv UniqueAccounts : account->forAll(c1, c2 | c1 <> c2 implies c1.number <> c2.number)
```

5 Data Types and Collections in OCL

5.1 Basic Types

OCL Type	Description	Example
Boolean	Logical values	`true`, `false`
Integer	Whole numbers	`5`, `-10`, `0`
Real	Real numbers	`3.14`, `-2.5`
String	Character sequences	`"Good morning"`

5.2 Collections

Types of collections in OCL :
— **Set :** Unordered collection without duplicates.
— **OrderedSet :** Ordered collection without duplicates.
— **Sequence :** Ordered list that allows duplicates.
— **Bag :** Unordered collection that allows duplicates (multiset).

Examples :

```
-- Collection statements
Set{1, 2, 3, 2}       -- Result: {1, 2, 3}
Sequence{1, 2, 3, 2}  -- Result: [1, 2, 3, 2]
Bag{1, 2, 3, 2}       -- Result: Bag{1, 2, 2, 3}
```

5.3 Basic Operations on Collections

— `size()` – Returns the number of elements in the collection.
— `includes(x)` – Checks whether element `x` is part of the collection.
— `isEmpty()` – Returns `true` if the collection contains no elements.
— `sum()` – Returns the sum of all numerical elements (for numeric collections).

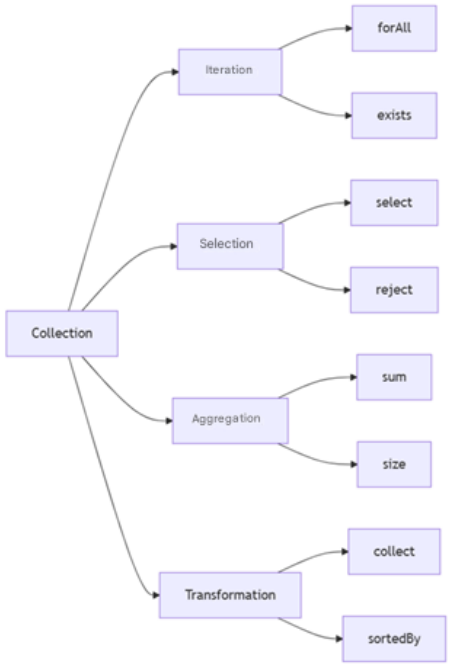

FIGURE 4 – Flowchart summarizing common OCL collection operations.

5.4 Examples of Operations

Assuming a collection : `clients : Set(Client)`

Selection

```
context Bank
def : majorClients = clients->select(age >= 18)
```

Rejection

```
context Bank
def : minorClients = clients->reject(age >= 18)
```

Existential Quantification

```
context Bank
inv atLeastOneRichClient : clients->exists(balance > 100000)
```

Universal Quantification

```
context Bank
inv allValidClients : clients->forAll(c | c.name.size() > 0)
```

Aggregation

```
context Bank
def : totalDeposits = accounts.balance->sum()
```

For each bank, total deposits are defined as the sum of the balances of all its accounts.

Transformation

```
context Bank
def : customerNames = customers.name->asSet()
```

For each bank, the list of customer names is defined as the set of names of all its customers after removing duplicates.

6 Navigation in Associations

6.1 Simple Navigation

Navigation in OCL allows accessing related objects through associations defined in the UML model. It enables the traversal of links between classes to express constraints and derived attributes.

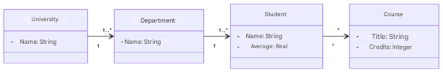

FIGURE 5 – Class diagram showing associations between university, department and student.

OCL Constraints

University Constraints

```
context University
-- All departments must have at least 10 students
inv : departments->forAll(d | d.students->size() >= 10)

-- At least one student must have an average above 15
inv : departments.students->exists(average > 15)
```

Complex Navigation *For each department, total student credits are defined as the sum of all credits for all courses taken by all its students.*

```
context Department
def : totalStudentCredits = students.courses.credits->sum()
```

6.2 Navigation with Complex Conditions

```
context University
-- Students enrolled in at least 5 courses
def : activeStudents = departments.students->select(s | s.courses->size() >= 5)

-- Average student grade by department
def : averagePerDepartment = departments->collect(d | d.students.average->avg())
```

7 Case Study : Reservation System

7.1 Class Diagram

The reservation system case study demonstrates the use of UML class diagrams combined with OCL constraints to specify precise business rules and relationships between model elements.

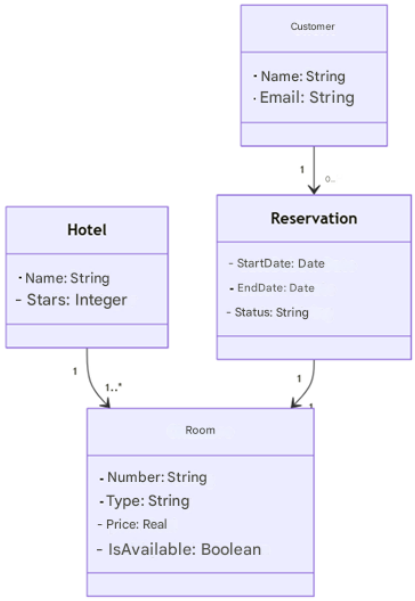

FIGURE 6 – Class diagram for a hotel reservation system.

7.2 Business Constraints in OCL

Hotel Constraints *Invariant validStars :* "The number of stars in a hotel must be between 1 and 5."

Invariant uniqueRooms : "All rooms in a hotel must have different numbers."

```
context Hotel
```

```
inv validStars  : stars >= 1 and stars <= 5
inv uniqueRooms : rooms->forAll(r1, r2 | r1 <> r2 implies r1.number <> r2.number)
```

Room Constraints *Invariant positivePrice :* "The price of a room must be strictly positive."

Invariant validType : "The type of room must be 'Single', 'Double', or 'Suite'."

```
context Room
inv positivePrice : price > 0
inv validType     : type = 'Single' or type = 'Double' or type = 'Suite'
```

Reservation Constraints *Invariant validDates :* "The start date of a reservation must be before its end date."

Invariant maxDuration : "The duration of a reservation cannot exceed 30 days".

```
context Reservation
inv validDates : startDate < endDate
inv maxDuration : (endDate - startDate) <= 30 -- Maximum 30 days
```

Preconditions and Postconditions for Reservation Creation :

Precondition roomAvailable : "To make a reservation, the room must be available."

Precondition datesInFuture : "To create a reservation, the start date must be in the future."

Postcondition roomReserved : "After making a reservation, the room is no longer available."

Postcondition statusConfirmed : "After creating a reservation, its status is 'Confirmed'."

```
context Reservation::create(startDate : Date, endDate : Date, room : Room)
pre roomAvailable   : room.isAvailable = true
pre datesInFuture   : startDate > Date::now()
post roomReserved   : room.isAvailable = false
post statusConfirmed: status = 'Confirmed'
```

Customer Constraints *Invariant validEmail :* "A customer's email address must follow the standard email format."

Invariant non-overlapping reservations : "No "a customer's reservations should overlap another reservation of the same customer."

```
context Customer
inv validEmail : email.matches('^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$')

inv nonOverlappingReservations :
```

```
reservations->forAll(r1, r2 |
    r1 <> r2 implies
    (r1.endDate < r2.startDate or r2.endDate < r1.startDate)
)
```

8 Conclusion

OCL is proving to be an indispensable component in the MDA modeling ecosystem, complementing UML by providing the precision and formal rigor that visual diagrams alone lack. Through its three main types of constraints—**invariants** to guaranty the permanent integrity of objects, **preconditions** to validate operation input, and **postconditions** to specify their effects— OCL allows the unambiguous expression of even the most complex business rules.

Its powerful navigation capabilities through model associations, combined with its rich set of collections operations, allow the formulation of sophisticated constraints that would otherwise be difficult to express. Finally, its declarative and side-effect-free nature makes it a reliable tool for specification, validation, and code generation, ensuring consistency between design and implementation throughout the model-driven development process.