Object Constraint Language

Object Constraint Language (OCL) is a formal language used to express constraints. OCL typically specify the invariant conditions that must hold for the system being modeled.

Expressions can be used in a number of places in the UML model:

- To specify the initial value of an attribute or association end.
- To specify the derivation rule for an attribute or association end.
- To specify the body of an operation.
- To indicate an instance in a dynamic diagram.
- To indicate a condition in a dynamic diagram.
- To indicate the actual parameter values in a dynamic diagram.

There are four types of constraints:

- An *invariant* is a constraint that states a condition that must always be met by all instances of the class, type, or interface. The invariant is described using an expression that evaluates to true if the invariant is met. Invariants must be true all the time.
- A *precondition* to an operation is a restriction that must be true at the moment the operation is going to be executed. The obligations are specified by the postconditions.
- A *postcondition* to an operation is a restriction that must be true at the moment the operation has just been executed.
- A *guard* is a constraint that must be true before a state transition discharged.

You can use derived properties in the OCL expressions. Scripts having multiple parameters can now be defined using OCL in structured expressions or Opaque Behaviors

Invariants on attributes

The simplest constraint is an invariant on an attribute. Suppose a model contains a class *Customer* with an attribute age, then the following constraint restricts the value of the attribute:

```
context Customer inv :
age >= 18
```

Invariants on associations

One may also put constraints on the associated objects. Suppose a model contains the class Customer that has an association to the class Salesperson with the role name salesrep and multiplicity 1, then the following constraint restricts the value of the attribute knowledge level of the associated instance of Salesperson:

```
context Customer inv :
salesrep.knowledgelevel >= 5
```

Collections of objects

In most cases the multiplicity of an association is not 1, but more than 1. Evaluating a constraint in these cases will result in a collection of instances of the associated class. Constraints can be put on either the collection itself, e.g. limiting the size, or on the elements of the collection. Suppose in a model the association between
Salesperson and Customer has the role name clients and multiplicity 1..* on the side of the Customer class, then we might restrict this relationship by the following constraints:

```
context Salesperson inv :
clients->size() <= 100 and clients->forAll(c: Customer | c.age >= 40)
```

Pre- and postconditions

In the pre- and postconditions the parameters of the operation can be used. Furthermore, there is a special keyword result which denotes the return value of the operation. It can be used in the postcondition only. For example an operation sell was added to the Salesperson class.

```
context Salesperson::sell( item: Thing ): Real
pre : self.sellableItems->includes( item )
post: not self.sellableItems->includes( item ) and result = item.price
```

Derivation Rules

Models often define derived attributes and associations. A derived element does not stand alone. The value of a derived element must always be determined from other (base) values in the model. Omitting the way to derive the element value results in an incomplete model. Using OCL, the derivation can be expressed in a derivation rule. In the following example, the value of a derived element usedServices is defined to be all services that have generated transactions on the account:

```
context LoyaltyAccount::usedServices : Set(Services)
derive : transactions.service->asSet()
```

Initial Values

In the model information, the initial value of an attribute or association role can be specified by an OCL expression. In the following examples, the initial value for the attribute points is 0, and for the association end transactions, it is an empty set:

```
context LoyaltyAccount::points : Integer
init: 0
context LoyaltyAccount::transactions : Set(Transaction)
init: Set{}
```

Body of Query Operations

The class diagram can introduce a number of query operations. The query operations are operations that have no side effects, i.e. do not change the state of any instance in the system. The execution of a query operation results in a value or set of values without any alterations in the state of the system. The query operations can
be introduced in the class diagram, but can only be fully defined by specifying the result of the operation. Using OCL, the result can be given in a single expression, called a body expression. In fact, OCL is a full query language, comparable to SQL. The use of body expressions is an illustration thereof.

The next example states that the operation *getCustomerName* will always result in the name of the card owner associated with the loyalty account:

```
context LoyaltyAccount::getCustomerName() : String
body: Membership.card.owner.name
```

To check OCL syntax according to OCL grammar

---

1. In the constraint Specification window, click the Specification property value.
2. Click the **...** button in the property value cell. The **Edit Specification** window opens.

3. In the Language list, click **OCL** and select the **Check OCL syntax** check box. In the Body box, incorrect expression will be underlined in red.