Object Constraint Language (OCL): A Definitive Guide

Conference Paper *in* Lecture Notes in Computer Science · June 2012

DOI: 10.1007/978-3-642-30982-3_3

CITATIONS
173

READS
13,098

2 authors:

Jordi Cabot
Luxembourg Institute of Science and Technology (LIST)
420 PUBLICATIONS **9,233** CITATIONS
SEE PROFILE

Martin Gogolla
University of Bremen
401 PUBLICATIONS **6,585** CITATIONS
SEE PROFILE

Object Constraint Language (OCL): A Definitive Guide

Jordi Cabot[1] and Martin Gogolla[2]

[1] INRIA / École des Mines de Nantes, France
jordi.cabot@inria.fr
[2] University of Bremen, Germany
gogolla@informatik.uni-bremen.de

Abstract. The Object Constraint Language (OCL) started as a complement of the UML notation with the goal to overcome the limitations of UML (and in general, any graphical notation) in terms of precisely specifying detailed aspects of a system design. Since then, OCL has become a key component of any model-driven engineering (MDE) technique as the default language for expressing all kinds of (meta)model query, manipulation and specification requirements. Among many other applications, OCL is frequently used to express model transformations (as part of the source and target patterns of transformation rules), well-formedness rules (as part of the definition of new domain-specific languages), or code-generation templates (as a way to express the generation patterns and rules).

This chapter pretends to provide a comprehensive view of this language, its many applications and available tool support as well as the latest research developments and open challenges around it.

1 Introduction

The Object Constraint Language (OCL) appeared as an effort to overcome the limitations of UML when it comes to precisely specifying detailed aspects of a system design. OCL was first developed in 1995 inside IBM as an evolution of an expression language in the Syntropy method [26]. The work on OCL was part of a joint proposal with ObjectTime Limited presented as a response to the RFP for a standard object-oriented analysis and design language issued by the Object Management Group (OMG) [26]. That standard came to be what we now know as UML and OCL became integrated in it in 1997.

Initially, OCL was only used as a constraint language for UML but quickly expanded its scope and now OCL has become a key component of any model-driven engineering (MDE) technique as the default language for expressing all kinds of (meta)model query, manipulation and specification requirements. Among many other applications, OCL is frequently used to express model transformations (as part of the source and target patterns of transformation rules), well-formedness rules (as part of the definition of new domain-specific languages, or code generation templates (as a way to express the generation patterns and rules).

M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.): SFM 2012, LNCS 7320, pp. 58–90, 2012.

To adapt the language to these new applications, several new (sub)versions of the language have been released. At the moment of writing this chapter, the current version of the OCL language is version 2.3.1 [20].

This chapter pretends to provide a comprehensive view of this language, its many applications and available tool support as well as the latest research developments and open challenges around it. The rest of this chapter is structured as follows. Section 2 motivates the need for OCL. Section 3 gives a brief overview of the language, while Section 4 provides a more precise language description. Then, Section 5 classifies existings OCL tools. Finally, Section 6 outlines a possible research agenda for OCL and Section 7 provides some final conclusions.

2 Motivation

Graphical modeling languages are the preferred choice for many designers when it comes to define the structural aspects of a domain (i.e., its main concepts, their properties and the relationships between them). The most typical example of a graphical notation is UML [21], specially its class diagram which is by far the most used UML diagram [13].

Nevertheless, this facility of use comes with a price. In order to keep the number of notational elements manageable, language designers must limit the expressiveness of the language. This means that graphical notations can only express a limited subset of all the relevant information of a domain. This is where OCL (and in general, any other textual language) comes into play. They are a necessary complement of the UML (or other graphical languages) notation in order to be able to precisely specify all detailed aspects of a system design.

As an example, take a look at the class diagram of Figure 1 that will be used as running example throughout the chapter. This diagram is an excerpt of the EU-Rent Car Rentals Specification [14], an in-depth specification of the EU-Rent case study, which is a widely known case study being promoted as a basis for demonstration of product capabilities. EU-Rent presents a car rental company with branches in several countries that provides typical rental services. EU-Rent was originally developed by Model Systems, Ltd.

This excerpt contains information about the rentals of the company (*Rental* class), the company branches (*Branch* class), the rented cars (*Car*), the category to which they belong (*CarGroup*) and the customers (*Customer*) that at some point in time may become blacklisted (*BlackListed*) due to delayed car returns, unpaid rentals, etc. Each rented car has one or more registered drivers and a pickup and drop off branch assigned.

This may look like a quite complete definition of the problem but in reality it is just the tip of the iceberg. Many important details cannot be defined just using the notation available for UML class diagrams. Just to mention some aspects that the UMl diagram does not answer:

1. Can black listed people rent new cars? (common sense may suggest answering no to this question but in fact this is not specified anywhere in the diagram so different people may assume different answers)

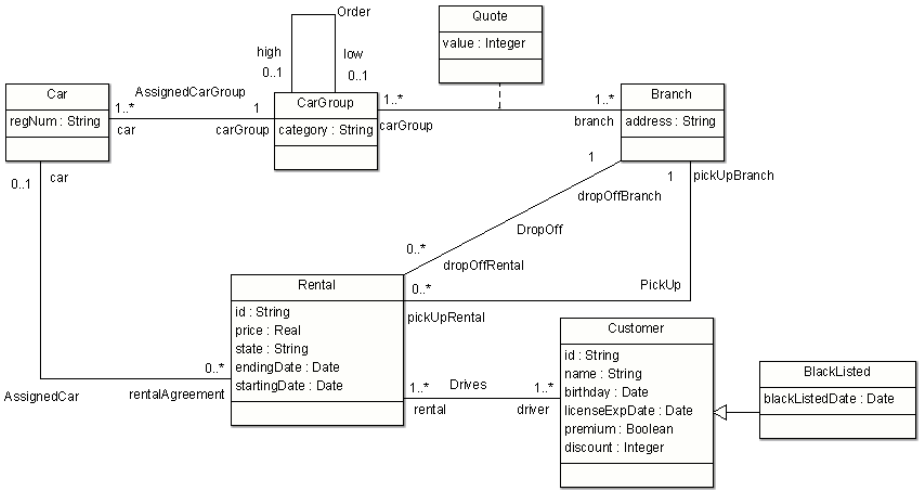


Fig. 1. Running Example - Partial Class Diagram of the EU-Rent company

2. How is the price of a rental calculated?
3. What are the conditions to be able to extend an existing rental?
4. Should the driving license of all drivers be valid throughout the full rental period? Is there a minimum driving seniority required? Can the same driver have two active rentals?
5. Can the pickup and drop off branches differ?
6. Can I choose a car already assigned to another rental?

The next section will show how OCL can be used to express all these additional concerns.

3 OCL in a Nutshell

The goal of this section is to give you an informal short description of the OCL and show its usefulness by exemplifying how it can be used to solve the open questions left at the end of the last section.

OCL is a general-purpose (textual) formal[1] language adopted as a standard by the OMG (see the current version of the OCL specification [20]) used to define several kinds of expressions that complement the information of (UML) models.

OCL is a typed, declarative and side-effect free specification language. *Typed* means that each OCL expression evaluates to a type (either one of the predefined OCL types or a type in the model where the OCL expression is used) and must conform to the rules and operations of that type. *Side-effect free* implies that OCL expressions can query or constrain the state of the system but not modify

[1] The degree of formality of OCL is under discussion but we could agree that at least it can be considered a semi-formal language.

it. *Declarative* means that OCL does not include imperative constructs like assignments. And finally, *specification* refers to the fact that the language definition does not include any implementation details nor implementation guidelines.

Among the many applications of OCL, it can be used to define the following kinds of expressions [2]:

- Invariants to state all necessary condition that must be satisfied in each possible instantiation of the model.
- Initialization of class properties.
- Derivation rules that express how the value of derived model elements must be computed.
- Query operations
- Operation contracts (i.e., set of operation pre- and postconditions)

In the following we briefly introduce each expression type and explain some basic OCL construct along the way. The next section will present the full details of the language.

3.1 Invariants

Integrity constraints in OCL are represented as invariants defined in the context of a specific type, named the *context type* of the constraint. Its body, the boolean condition to be checked, must be satisfied by all instances of the context type.

Invariants are without a doubt the most common OCL expression since they allow designers to easily specify all kinds of conditions that the system must comply with.

Invariants can restrict the value of single objects, like the following *QuoteOverZero*:

```
context Quote inv QuoteOverZero: self.value > 0
```

stating that all quotes must have a positive value. Note that the *self* variable represents an arbitrary instance of the *Quote* class and the dot notation is used to access the properties of the *self* object (as the *value* attribute in the example). As stated above, all instances of *Quote* (the context type of the constraint in this case) must evaluate this condition to true.

Nevertheless, many invariants express more complex conditions limiting the possible relationships between different objects in the system, usually related through association links. For instance, this *NoRentalsBlackListed* constraint forbids BlackListed people of renting cars:

```
context BlackListed inv NoRentalsBlackListed:
  self.rental->forAll(r | r.startDate < self.blackListedDate)
```

[2] For the sake of simplicity, we focus on the kinds of expressions useful for class diagrams; e.g., OCL can also be used to define guards in state machines.

where we first retrieve all rentals linked to a blacklisted person and then we make sure that all of them were created before the person was blacklisted. This is done by iterating on all related rentals and evaluating the date condition on each of them; the *forAll* iterator returns true iff all elements of the input collection evaluate the condition to true.

3.2 Initialization Expressions

OCL can be used to specify the initial value that the properties of an object must take upon the object creation. Obviously, the type of the expression must conform to the type of the initialized property (this must also take into account cases where the property to be initialized is a collection).

For instance, the following OCL expression initializes to false the value of the *premium* attribute of Customers (we are assuming that customers can only promote to the premium status after renting several cars).

```
context Customer::premium: boolean init: false
```

3.3 Derived Elements

Derived elements are elements whose value/population can be inferred from the value/population of other model elements as defined in the element's derivation rule. OCL is a popular choice for specifying these derivation rules.

OCL derivation rules follow the same structure as init expressions (see above) although their interpretation is different. An *init* expression must be true when the object is created but the restriced property may change its value afterwards (i.e., customers start as non-premium but may evolve to premium during their time in the system). Instead, derivation rules constrain the value of a derived element throughout all its life-span. Note that this does not imply that the value of a derived element cannot change, it only means that it will always change according to the evaluation of its derivation rule.

As an example, consider the following rule for the derived element *discount* in class *Customer*, stating that premium members get a 30% discount while non-premium members get 15% if they have at least rented high category cars five times while the rest of the customers get no discount at all.

```
context Customer::discount: integer
derive:
  if not self.premium then
    if self.rental.car.carGroup->
         select(c|c.category='high')->size()>=5
    then 15
    else 0 endif
  else 30 endif
```

The *select* iterator in the expression returns the subcollection of elements from the input collection that satisfy the condition. Then, the *size* collection operator returns the cardinality of the output subcollection and this value is compared with the '5' threshold. Note that in this example, the input collection (self.rental.car.carGroup) is not a set but a bag (i.e., a collection with repeated elements) since a user may have rented the same car twice in different rentals or two cars belonging to the same car group.

3.4 Query Operations

As the name indicates, query operations are a *wrapped* OCL expression that queries the system data and returns the information to the user.

As an example, the following query operation returns true if the car on which the operation is executed is the most popular in the rental system.

```
context Car::mostPopular(): boolean
body: Car::allInstances()->forAll(c1|c1<>self implies
      c1.rentalAgreement->size()<=self.rentalAgreement->size())
```

3.5 Operation Contracts

There are two different approaches for specifying an operation effect: the *imperative* and the *declarative* approach [27]. In an imperative specification, the designer explicitly defines the set of structural events (inserts/updates/deletes) to be applied when executing the operation. Instead, in a declarative specification, a contract for each operation must be provided. The contract consists of a set of pre- and postconditions. A precondition defines a set of conditions on the operation input and the system state that must hold when the operation is issued while postconditions state the set of conditions that must be satisfied by the system state at the end of the operation. OCL is usually the language of choice to express pre- and postconditions for operation contracts at the modeling level.

As an example, the following newRental operation describes (part of) the business logic behind the creation of a new rental in the EU-rent system:

```
context Rental::newRental(id:Integer, price:Real, startingDate:Date,
                  endingDate:Date, customer:Customer, carRegNum:String,
                  pickupBranch: Branch, dropOffBranch: Branch)
pre: customer.licenseExpDate>endingDate
post: Rental.allInstances->one(r |
        r.oclIsNew() and r.oclIsTypeOf(Rental) and
        r.endingDate=endingDate and r.startingDate=startingDate and
        r.driver=customer and r.pickupBranch=pickupBranch and
        r.dropOffBranch=dropOffBranch and
        r.car=Car.allInstances()->any(c | c.regNum=carRegNum))
```

The precondition checks that the customer has a valid license for the duration of the rental[3] while the postcondition states that by the end of the operation a new object r of type *Rental* must have been created and initialized with the set of values passed as parameters [4].

4 Language Description

Figure 2 gives an overview on the OCL type system in form of a feature model. Using a tree-like description[5], feature models allow to describe mandatory and optional features of a subject, and they allow to specify alternative features as well conjunctive features. In particular, the figure pictures the different kinds of available types. Before explaining the type system in a systematic way, let us discuss OCL example types which are already known or which can be deduced from the class diagram of our running example in Fig. 3. Attributes types, as for example in `Car::regNum:String`, are *predefined basic, atomic* types. Classes which are defined by the class diagram are *atomic, user-defined class* types. If we already have an expression `cg` of type `CarGroup`, then the OCL expression `cg.car` has the type `Set(Car)` due to the multiplicity `1..*`. The type `Set(Car)` is a *flat, concrete collection* type. `Set(Car)` is a reification of the *parametrized collection* type `Set(T)` where `T` denotes an arbitrary type parameter which can be stubstituted. The type `Sequence(Set(Car))` is a *nested* collection type being a reification of the parametrized, nested collection type `Sequence(Set(T))`. If `cg:CarGroup` is given, then the expression `Tuple{cat:cg.category, cars:cg.car}` has type `Tuple(cat:String, cars:Set(Car))` which is a *tuple* type.

4.1 OCL Types

Let us now consider the types in Fig. 2 in a systematic way. An OCL type is either an atomic type or a template type. Atomic types are either predefined basic types or user-defined types. Predefined basic types are `Integer`, `Real`, `String`, and `Boolean`. User-defined types are either class types (e.g., `Customer`) or enumeration types (e.g., `BranchKind=#airport, #downtown, #onTheRoad`). A template type is a type which uses at least one of the six predefined type constructors: `Set`, `Bag`, `Sequence`, `OrderedSet`, `Collection`, and `Tuple`. A parametrized template type has one or more parameters (e.g., `Bag(T)` or `Tuple(part1:T1, part2:T2)`)

[3] There are several styles when writing preconditions, some people choose to include in the preconditions the verification of all integrity constraints that may be affected by the operation while others consider this redundant.

[4] Note that postconditions are underspecifications, i.e., they only specify part of the system state at the end of the execution which leads to the frame problem [4] and other similar issues; this problem is not OCL-specific and thus it is outside of the scope of this chapter.

[5] The actual structure of the feature model is a dag (directed, acyclic graph).

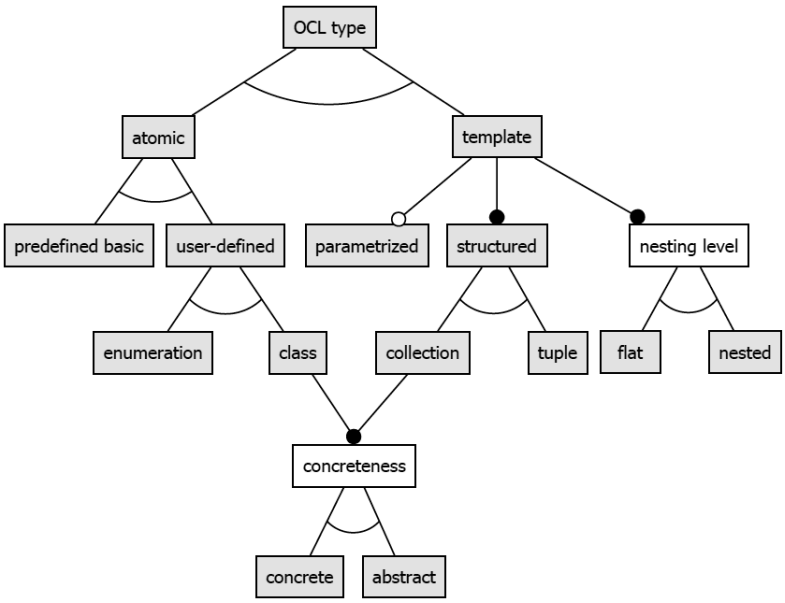


Fig. 2. OCL Types as a Feature Model

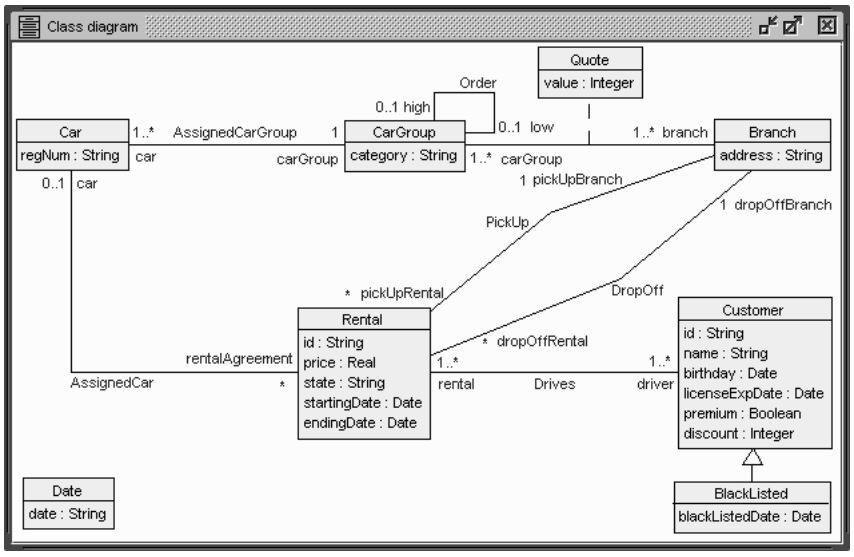


Fig. 3. Example Class Diagram

and can be applied to another type in order to obtain a more complex type. A template type is either a structured collection type or a structured tuple type. In OCL there are four parametrized, concrete collection types, namely `Set(T)`, `Bag(T)`, `Sequence(T)`, and `OrderedSet(T)`. As shown in the left side of Fig. 4, there is one abstract, parametrized collection type, namely `Collection(T)` which is the supertype of each of these four parametrized types and which cannot be instantiated directly but only through its subtypes. Collection and tuple types may be flat when they have a nesting level equal to 1 or they may be nested when they have a nesting level greater than 1. Figure 2 summarizes the OCL type structure which is formally represented as part of the OCL metamodel in the OCL standard.

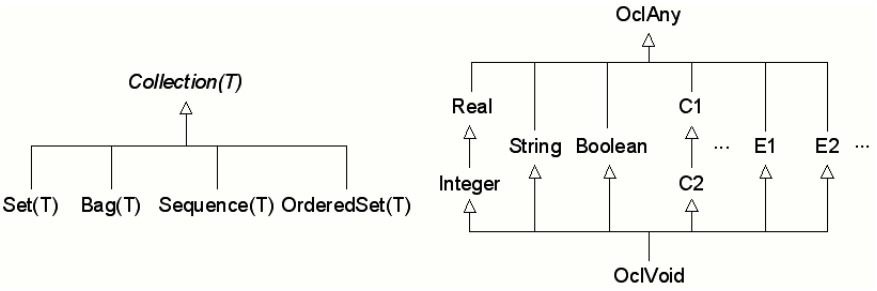


Fig. 4. OCL Type Inheritance Hierarchy

Let us give some more examples. The expression `Set{11, 12}` has type `Set(Integer)`, whereas `Bag{42, 42}` has type `Bag(Integer)`. Both, `Set(Integer)` and `Bag(Integer)`, are subtypes of the abstract collection type `Collection(Integer)`. An example expression where this type occurs is `Sequence{Set{11, 12}, Bag{42, 42}}` which is typed by the nested collection type `Sequence(Collection(Integer))`. Tuple and collection types may be used orthogonally. Thus the expression

```
Set{Tuple{name:'Ada', emails:Set{'ada@acm.org','ada@ieee.org'}},
    Tuple{name:'Bob', emails:Set{'bob@acm.org'}}}
```

has the type `Set(Tuple(name: String, emails: Set(String)))`.

Apart from the peculiarities for types discussed above, OCL has a type inheritance relationship `<` defining subtypes and supertypes. Type inheritance occurs in connection with the predefined basic types (`Integer < Real`), the defined classes (e.g., `BlackListed < Customer`), the collection types (e.g., `Set(Bag(String)) < Collection(Bag(String))`) and two special types for the top and the bottom of the type hierarchy, namely `OclAny` for the top type and `OclVoid` for the bottom type. A general overview is shown in Fig. 4. On the right side, the subtypes of `OclAny` being at the same time the supertypes

of `OclVoid` are displayed. The subtypes can be categorized into the predefined basic types, the class types, and the enumeration types. Please note that neither `Collection(T)` nor any of its reifications (e.g., `Set(String)`) is a subtype of `OclAny`. However, any type from the right side may be substituted for the type parameter `T` in the left side, and any subtyping relationship is carried over from the right side to the left side, e.g., `C2 < C1` induces `Bag(C2) < Bag(C1)` and `Set(C2) < Collection(C2) < Collection(C1)`.

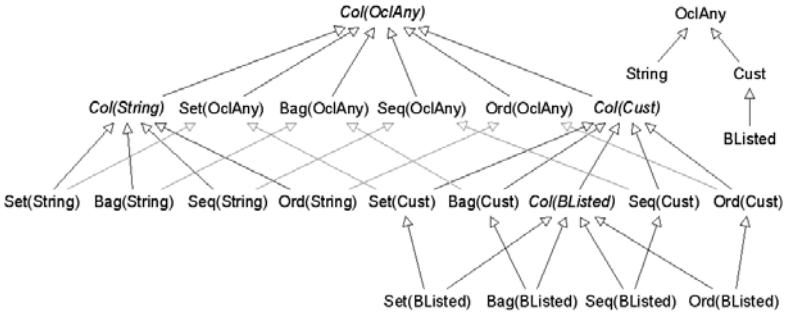


Fig. 5. OCL Example Types and Induced Collection Types

For our running example, we obtain `Set(Integer) < Collection(Real)` or `Bag(BlackListed) < Collection(Customer)`. Please also be aware of the fact, that, for example, `Set(OclAny)` is a valid type and, therefore, the expression `Set{42, true, 'ABBA', @Car42}` including an `Integer` constant, a `Boolean` constant, a `String` constant, and an object of class `Car` is a valid OCL expression of type `Set(OclAny)`. However, a construct like `Set{42,Sequence{42}}` is invalid, because the types `Integer` and `Sequence(Integer)` do not have a common supertype. The upper right part of Fig. 5 shows a subset of the example types and its induced first level collection types where the collections are applied only once and therefore no nested collection types arise. We have used the abbreviations `Seq[uence]`, `Ord[eredSet]`, `Col[lection]`, `Cust[omer]`, `B[lack]Listed` in the Figure, and we will use the shortcuts of the collections also further down, if we need it. Please note, that, for example, `Set(BListed)` has five supertypes: `Set(Cust)`, `Col(BListed)`, `Set(OclAny)`, `Col(Cust)`, `Col(OclAny)`. The type relationships would become even richer when all example types (e.g., `Integer` and `Branch`) would have been used. In principle, there is an infinite number of induced collection types, because the nesting level may be arbitrarily deep, but the used maximal nesting level is always finite: every class model and every OCL term will use only a finite fraction of all possible types.

4.2 OCL Values

As you see from the feature model, the OCL type system is involved, but for an introductory paper, we want to offer to the reader a way through all possible combinations by means of a clear, manageable number of categories. The OCL type feature model gives rise to nine different categories of available types. We label the categories with letters from (A) to (I) and show examples for OCL expressions representing values in each of the nine categories.

- atomic
  - (A) predefined basic
  - (B) enumeration
  - (C) class
- template
  - (D) parametrized
  - structured collection
    - (E) concrete flat
    - (F) concrete nested
    - (G) abstract nested
    - abstract flat: unpopulated
  - structured tuple
    - (H) flat
    - (I) nested

We will first go through the categories (A) to (I), explain each single category and show positive examples. Afterwards, we will explain why we consider the category *abstract, flat structured collection* as being unpopulated.

The OCL expressions for category (A) are straight forward and need not be explained. The enumeration values in category (B) show that an enumeration literal can be introduced by the hash sign as required in early OCL versions and that they can be written down preceeding their type name and separated by double colons in later OCL versions. As shown in category (C), a literal for an object can be any allowed identifier. Often the literal indicates somehow the class type for the object, but this is not a requirement. For small examples often well choosen object literals (like `ibm,sun:Company`) support intuition about the use of the object. In category (D), the parametrized types always possess at least one type parameter. Parametrized types can be nested arbitrarily deep, but the nesting in each type is always finite. The six keywords `Set`, `Bag`, `Sequence`, `OrderedSet`, `Collection`, and `Tuple` occur in connection with parametrized types.

```
(A) 42 : Integer
    43.44 : Real
    'fortytwo' : String
    false : Boolean
```

```
(B) #airport, #downtown, #onTheRoad : BranchKind
    BranchKind::airport : BranchKind

(C) car42, Herbie, OFP857 : Car
    branch42, SunsetStrip77 : Branch

(D) Set(T)
    Sequence(T)
    Tuple(part1:T1,part2:T2)
    Sequence(OrderedSet(T))
    Sequence(Collection(T))
```

Category (E) contains values for flat, concrete collections, category (F) displays nested, concrete collections, and category (G) involves nested, abstract collections. As the examples point out, values for collection types can be built with constructor operations denoted by `Set`, `Bag`, `Sequence` and `OrderedSet`. A type is considered to be *abstract* if its type expression involves `Collection` or an abstract class type from the underlying class diagram. Note that collections may contain different values which have different least types, but in any case the values inside a single collection must have a common supertype. For example, the last expression in category (E) `Set{car42,'fortytwo'}` involves values of type `Car` and `String`. The example for ordered sets shows that ordered sets are not neccessarily sorted. Please note that the top-most example set in category (F) contains three elements which are pairwise distinct. The last example in category (G) shows a degree of flexibilty gained through the possibility of having abstract collections: A collection of email adresses can be a set or a sequence of strings, depending on whether a priority for email adresses is required to be stated or not.

```
(E) Set{42,43} : Set(Integer)
    Bag{42,42,43} : Bag(Integer)
    Sequence{43,42,44,42} : Sequence(Integer)
    OrderedSet{43,42,44} : OrderedSet(Integer)
    Sequence{'Steve','Jobs'} : Sequence(String)
    Bag{backlisted42,blacklisted43} : Bag(BlackListed)
    Set{42,43.44} : Set(Real)
    Sequence{blacklisted42,customer43} : Sequence(Customer)
    Set{car42,'fortytwo'} : Set(OclAny)

(F) Sequence{Bag{42,42},Bag{42,43}} : Sequence(Bag(Integer))
    Set{Sequence{Set{7},Set{8}},
        Sequence{Set{8},Set{7}},
        Sequence{Set{7,8}}} : Set(Sequence(Set(Integer)))

(G) Sequence{Set{7,8},Bag{7,7}} : Sequence(Collection(Integer))
    Set{Set{Set{7}},Bag{Bag{7}}} : Set(Collection(Collection(Integer)))
    Set{Tuple{name:'Ada',emails:Set{'ada@acm','ada@ibm'}},
        Tuple{name:'Bob',emails:Sequence{'bob@omg','bob@sun'}}} :
      Set(Tuple(emails:Collection(String),name:String))
```

Categories (H) shows flat tuples and category (I) nested tuples. Tuples can be constructed with the keyword `Tuple` and by additionally specifying part names and part values. In category (H), the first two examples explain that tuple parts may be assigned by using the colon or alternatively by the equality symbol. Tuples may contain parts of arbitrary types, e.g., class types and predefined types as in the third example. The order of tuple parts does not matter in OCL. Thus we have for example `Tuple{first:'Steve',last:'Jobs'} = Tuple{last:'Jobs',first:'Steve'}`. The tool employed here (USE) [17] sorts the tuple parts by their names, and therefore the shown type to the right of the colon may exhibit a different part order than the input expression. In category (I), four tuple values are presented. The first one is a nested tuple with two parts having a tuple type. The second one is a tuple with two parts having type `String` and `Set(String)`, respectively. The third one is an OCL representation of a simple relational database state in first normal form. The fourth one represents the same state information in a non-first normal form style in which the second relation has a set-valued, non-atomic part type.

```
(H) Tuple{first:'Steve',last:'Jobs'} : Tuple(first:String,last:String)
    Tuple{first='Steve',last='Jobs'} : Tuple(first:String,last:String)
    Tuple{carRef:Herbie,year:1963,manufRef:VW} :
      Tuple{carRef:Car,manufRef:Company,year:Integer}

(I) Tuple{name:Tuple{first:'Steve',last:'Jobs'},
          adr:Tuple{street:'Infinite Loop',no:1}} :
      Tuple(adr:Tuple(no:Integer,street:String),
            name:Tuple(first:String,last:String))

    Tuple{name:'Ada',emails:Set{'ada@acm','ada@ibm'}} :
      Tuple{emails:Set(String),name:String}

    Tuple{Customers:Set{Tuple{name:'Ada',birth:1962},
                        Tuple{name:'Bob',birth:1962}},
          Rentals:Set{Tuple{name:'Ada',start:'2012-01-01'},
                      Tuple{name:'Ada',start:'2002-01-01'},
                      Tuple{name:'Cyd',start:'2002-01-01'}}} :
      Tuple(Customers:Set(Tuple(birth:Integer,name:String)),
            Rentals:Set(Tuple(name:String,start:String)))

     Customers | name  | birth   Rentals | name  | start
    -----------+-------+-------  ---------+-------+--------------
               | 'Ada' | 1962             | 'Ada' | '2012-01-01'
               | 'Bob' | 1962             | 'Ada' | '2002-01-01'
                                          | 'Cyd' | '2002-01-01'

    Tuple{Customers:Set{Tuple{name:'Ada',birth:1962},
                        Tuple{name:'Bob',birth:1962}},
          Rentals:Set{Tuple{name:'Ada',
                            starts:Set{'2012-01-01','2002-01-01'}},
                      Tuple{name:'Cyd',starts:Set{'2002-01-01'}}}} :
```

```
Tuple(Customers:Set(Tuple(birth:Integer,name:String)),
      Rentals:Set(Tuple(name:String,starts:Set(String))))
```

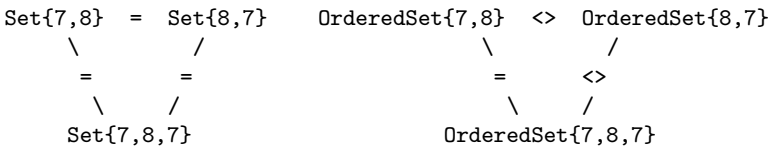
The category *abstract, flat structured collection* cannot be populated because, for example, you cannot build a value for `Collection(Integer)` which is not also a value for `Set(Integer)` or `Bag(Integer)` or `Sequence(Integer)` or `OrderedSet(Integer)`. Of course we have: `Set{42}: Set(Integer)` and `Set{42}: Collection(Integer)` because `Set(Integer) < Set(Collection)`. But there is no *proper* value in `Collection(Integer)` which is only in that type and not also in one its subtypes. The statement can be expressed formally as follows.

```
VALUES[Collection(Integer)] - VALUES[Set(Integer)]
                            - VALUES[Bag(Integer)]
                            - VALUES[Sequence(Integer)]
                            - VALUES[OrderedSet(Integer)] = EMPTY
```

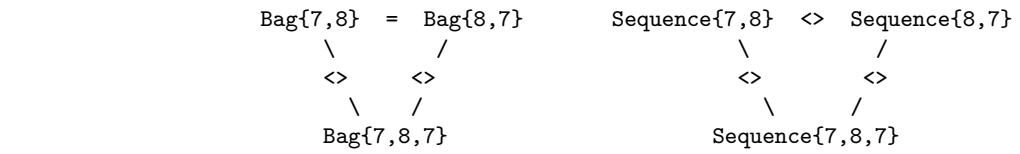
This is different for the combination *abstract and nested*. For example, we have `Sequence{Set{42},Bag{42}}` has type `Sequence(Collection(Integer))`. Note however, that all abstract types, which have `Collection` as its top type and an arbitrarily nested, but concrete type as its inner type, cannot be (in the above sense) *properly* populated. For example, we have `Set{Sequence{42}}` has type `Set(Sequence(Integer))` and as a consequence `Set{Sequence{42}}` has type `Collection(Sequence(Integer))`. And we have `Bag{Sequence{42}}` has type `Bag(Sequence(Integer))` and as a consequence `Bag{Sequence{42}}` has type `Collection(Sequence(Integer))`. But there are no values in `Collection(Sequence(Integer))` which are at the same time not in `Set(Sequence(Integer))` or `Bag(Sequence(Integer))` or `Sequence(Sequence(Integer))` or `OrderedSet(Sequence(Integer))`.

4.3 OCL Collection Properties

OCL denotes equality and inequality with the operations `=` und `<>`, respectively. Let us consider equality and inequality on collection values in more detail. This will also lead us to an explanation of the similarities and the differences between the four different collection kinds.[6]



[6] Collection kind VS collection type: In our view each collection kind is manifested by many collection types. For example, the collection kind set is manifested by `Set(String)` or `Set(Sequence(Integer))`.



Above we have displayed twelve different collection expressions: three sets, three ordered sets, three bags, and three sequences. There are three element insertion orders: (A) first 7 and second 8, (B) first 8 and second 7, and (C) first 7, second 8, third 7. We have also displayed whether the respective collection expressions are equal or inequal. The four collection kinds can be distinguished by their equal-inequal pattern: sets show `(=,=,=)`, ordered sets display `(<>,=,<>)`, bags give `(=,<>,<>)`, and sequences have `(<>,<>,<>)`. Using these examples one can also check general properties which collections may possess: insensibility to element insertion order and insensibility to element insertion frequency. The four collection kinds can be distinguished nicely on the basis of these two criteria.

	insertion order	
	insensible	sensible
insertion frequency insensible	Set	OrderedSet
insertion frequency sensible	Bag	Sequence

Both criteria can formally be defined in an OCL-like style with predicates as stated below. Here, we already use three operations on collections which will be explained later. The operation `forAll` checks whether a boolean expression evaluates to `true` on all collection elements. The operation `including` inserts an element into a collection and (possibly) constructs a *new* collection. The operation `includes` checks whether an item is part of a collection.

```
orderInsensible(c:Collection(OclAny),witness:Bag(OclAny)):Boolean=
  witness->forAll(e,f |
    c->including(e)->including(f)=c->including(f)->including(e))
frequencyInsensible(c:Collection(OclAny),witness:Bag(OclAny)):Boolean=
  witness->forAll(e |
    c->includes(e) implies c->including(e)=c)
```

The operation `orderInsensible` checks whether for a parameter collection the order in the addition of two further elements does not matter. The operation `frequencyInsensible` checks whether the addition of an already present collection element does not matter. Both operations have an additional parameter determining a collection of test witnesses with which the respective property is checked. The actual OCL definitions are a bit more complicated because the operation `including` does not work on collections, but on the concrete subtypes only. We do not show them here. Using these two operations we can build the following OCL evaluations which demonstrate the distinctive features of the four different OCL collections. The OCL construct `let` allows us to define a name for an expression which can be used later.

```
? let C=Set{7} in let W=Bag{7,8,9} in
    Sequence{orderInsensible(C,W),frequencyInsensible(C,W)}
> Sequence{true,true} : Sequence(Boolean)

? let C=OrderedSet{7} in let W=Bag{7,8,9} in
    Sequence{orderInsensible(C,W),frequencyInsensible(C,W)}
> Sequence{false,true} : Sequence(Boolean)

? let C=Bag{7} in let W=Bag{7,8,9} in
    Sequence{orderInsensible(C,W),frequencyInsensible(C,W)}
> Sequence{true,false} : Sequence(Boolean)

? let C=Sequence{7} in let W=Bag{7,8,9} in
    Sequence{orderInsensible(C,W),frequencyInsensible(C,W)}
> Sequence{false,false} : Sequence(Boolean)
```

The OCL evaluations emphasize what was presented in the above table: Sets are order insensible and frequency insensible; bags are order insensible but frequency sensible; sequences are order sensible and frequency sensible; ordered sets are order sensible but frequency insensible.

We must mention some further details concerning equality and inequality on collections. We have seen that equality and inequality can be checked between two expressions possessing the same collection kind, e.g., we obtain `(Set{7,8} = Set{8,7,7}) = true`. But equality and inequality can also be applied between expressions between different collection kinds. For example, we obtain `(Set{7,8} = Bag{7,8}) = false` and `(OrderedSet{8,9} <> Sequence{8,9}) = true`. Note that although left and right-hand side of the collection comparisions contain the same values (even in the same order) the collections are different because their types are different. In particular, although bags possess the potential to contain one element twice, they are not forced to do so. In ordered sets, the first insertion of an element dominates over following insertions, e.g., we obtain `(OrderedSet{7,8,8} = OrderedSet{7,8,7}) = true` and `(OrderedSet{7,8,8} <> OrderedSet{8,7,8}) = true`. And, ordered sets are not *sorted*: `(OrderedSet{7,9,8} <> OrderedSet{9,7,8}) = true`

4.4 OCL `null` Value

As we have seen before, the OCL type system knows a top type, namely `OclAny`, which includes all atomic values (but not the structured values). We have mentioned also a bottom type, namely `OclVoid`. This type is populated by one extra value denoted by `null`. As in the database language SQL, this value can be used to express that some particular information is not available. Because, `OclVoid` is a subtype of any other atomic type, the value `null` is present in all atomic types and can be used in collections and tuples. The literal `null` was introduced in a newer OCL version. Formerly, there was the check `oclIsUndefined` on `OclAny` with which it is still possible to test for this value. Let us consider some uses of `null`.

```
ada.discount=null : Boolean
branch42.address=null : Boolean
1/0=null : Boolean
Tuple{name:'Jobs, Steve',telno:null} : Tuple(name:String,telno:String)
(1/0).oclIsUndefined=true : Boolean
42.oclIsUndefined=false : Boolean
Set{'800-275-2273','800-694-7466',null} : Set(String)
```

The first two example express that the discount of customer `ada` and the address of branch `branch42` are currently undefined. In the third example `null` is used to express the partiality of a function. The fourth example shows a tuple whose part `telno` is undefined. The last example shows the `null` value in a collection. As in SQL, the value `null` is an exceptional value distinct from all ordinary values. For example, in OCL we have that the following propositions are true: `0<>null`, `''<>null`, `'null'<>null`, `'NULL'<>null`, and `'Null'<>null`.

4.5 Navigation in OCL

Given an object diagram, i.e., a system state, OCL allows us to access objects and their properties, e.g., attributes, and to navigate between them by using opposite side role names from associations. This navigation is syntactically denoted by a dot. Consider the object diagram in Fig. 6 which shows a valid system state where all classes and associations are instantiated through objects and links and where all association multiplicities are satisfied. Then the following attribute accesses and navigation possibilities exist.

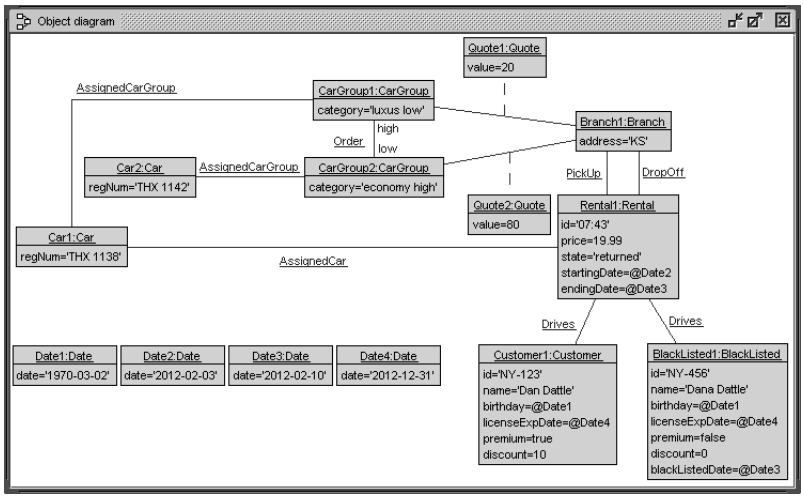


Fig. 6. Example Object Diagram

```
? Car1
> @Car1 : Car

? Car2.regNum
> 'THX 1142' : String

? Car2.carGroup
> @CarGroup2 : CarGroup

? Car2.carGroup.category
> 'economy high' : String

? Branch1.carGroup
> Set{@CarGroup1,@CarGroup2} : Set(CarGroup)

? Branch1.carGroup.car
> Bag{@Car1,@Car2} : Bag(Car)

? Rental1.driver
> Set{@BlackListed1,@Customer1} : Set(Customer)

? Rental1.driver.birthday
> Bag{@Date1,@Date1} : Bag(Date)

? CarGroup2.high
> @CarGroup1 : CarGroup

? CarGroup2.high.low
> @CarGroup2 : CarGroup

? CarGroup2.high.high
> null : OclVoid

? Car2.rentalAgreement
> Set{} : Set(Rental)
```

Navigation from one object with an opposite side role name results either in a single-valued return type (as in `Car2.carGroup : CarGroup`) or in a set-valued valued return type (as in `Branch1.carGroup : Set(CarGroup)`) depending on the multiplicity of the role name (here, `0..1` VS `1..*`). The multiplicities `0..1` and `1` yield single-valued return types whereas other multiplicities, for example `0..*` and `1..*`, give set-valued return types. In the set-valued case, the result is empty (as in `Car2.rentalAgreement`) if no object connection exists, whereas in the single-valued case the result is `null` (as in `CarGroup2.high.high`) if no object connection exists. Further navigation through a second dot is possible in a single expression and can yield a bag-valued result (as in `Rental1.driver.birthday = Bag{@Date1,@Date1}`). In this example, the preservation of duplicates with a bag-valued result reflects the fact that the two different objects in `Rental1.driver` evaluate identical with respect

to the second navigation `birthday`. A flat bag will also be the result in the case of two succesive set-valued navigations (as in codeBranch1.carGroup.car).

4.6 Logic-Related Operations in OCL

Because OCL has the `null` value and `Boolean` is a predefined type, the `null` value is also a `Boolean` value. This leads to a three-valued logic. Apart from the standard `Boolean` operations `and`, `or`, and `not`, OCL knows a (binary) exclusive `xor` and the implication `implies`. The truth tables for these operation are shown in the tables below. Of course, all `Boolean` operations coincide with the standard two-valued interpretation if one leaves out the `null` value.

```
    b     | not(b)
    ------+-------
    null  | null
    false | true
    true  | false
```

```
          |       b2                          |        b2
 b1 or b2 | null  false true       b1 and b2 | null  false true
 ---------+-----------------       ----------+------------------
     null | null  null  true            null | null  false null
 b1 false | null  false true        b1 false | false false false
     true | true  true  true            true | null  false true
```

```
          |       b2                          |        b2
b1 xor b2 | null  false true    b1 implies b2 | null  false true
----------+------------------   --------------+------------------
     null | null  null  null            null | null  null  true
 b1 false | null  false true        b1 false | true  true  true
     true | null  true  false           true | null  false true
```

```
          |       b2                          |        b2
  b1 = b2 | null  false true        b1 <> b2 | null  false true
----------+-----------------       ---------+------------------
     null | true  false false           null | false true  true
 b1 false | false true  false       b1 false | true  false true
     true | false false true            true | true  true  false
```

With respect to equality and inequality, the `null` value is treated like any other value. Equality and inequality do not return `null` as a result and operate as equality and inequality on `Set{null, false, true}`.

Apart from the usual, above discussed `Boolean` connectives, OCL has a universal quantifier `forAll` and an existial quantifier `exists`, both in the spirit of first order logic. However, both quantifiers range over finite collections only and cannot be used, for example, on all instances of the type `Integer` or `String`. We show examples for using the quantifiers. We here employ the not yet mentioned OCL feature to define collections of integers with range expressions taking the form `low..high`.

```
? Set{1,2,3,4,5,6,7,8,9,10,11,12}=Set{1..12}
> true : Boolean

? Set{1..12}->exists(n|n.mod(2)=0 and n.mod(3)=0)
> true : Boolean

? Bag{1..12}->exists(n|n.mod(3)=0 and n.mod(7)=0)
> false : Boolean

? Sequence{1..12}->forAll(n|0<=n*n and n*n<=255)
> true : Boolean

? OrderedSet{1..12}->forAll(n|0<=n*n and n*n<=127)
> false : Boolean

? Set{}->exists(n|n.mod(2)=0 and n.mod(3)=0)
> false : Boolean

? Bag{}->exists(n|n.mod(3)=0 and n.mod(7)=0)
> false : Boolean

? Sequence{}->forAll(n|0<=n*n and n*n<=255)
> true : Boolean

? OrderedSet{}->forAll(n|0<=n*n and n*n<=127)
> true : Boolean

? not(Set{1..12}->forAll(n|not(n.mod(2)=0 and n.mod(3)=0)))
> true : Boolean

? not(OrderedSet{1..12}->exists(n|not(0<=n*n and n*n<=127)))
> false : Boolean
```

4.7 OCL Collection Operations

The basic OCL operations for collection construction are the already mentioned constructor operations `Set`, `Bag`, `Sequence` and `OrderedSet`. In addition, OCL knows the constructor `including` which (possibly) adds an element to a collection. The strongly related, but not inverse operation is `excluding` that removes *all* occurrences of an element from the collection. Note that a law like `c = c->including(e)->excluding(e)` does not hold in OCL for all collections `c` and all elements `e`. We will observe the following evaluations.

```
? Set{7,8}=Set{}->including(8)->including(7)
> true : Boolean

? Bag{7,8,7}=Bag{8}->including(7)->including(7)
> true : Boolean
```

```
? Sequence{7,8,7}=Sequence{7,8}->including(7)
> true : Boolean

? OrderedSet{7,8}=OrderedSet{7}->including(8)->including(7)
> true : Boolean

? Set{7,8}->excluding(8)=Set{7}
> true : Boolean

? Bag{7,8,7}->excluding(7)=Bag{8}
> true : Boolean

? Sequence{7,8,7}->excluding(7)=Sequence{8}
> true : Boolean

? OrderedSet{9,6,7,8,6,7}->excluding(6)=OrderedSet{9,7,8}
> true : Boolean
```

In order to test membership in collections the operations `includes` and `excludes` testing on single elements as well as `includesAll` and `excludesAll` for testing element collections are available. The following examples explain the use of the operations.

```
? Set{7,8}->includes(9)
> false : Boolean

? Bag{7,8}->excludes(9)
> true : Boolean

? Sequence{7,9,8,7}->includesAll(Sequence{7,8,8})
> true : Boolean

? OrderedSet{7,9,8,7}->excludesAll(OrderedSet{3,2,4,2})
> true : Boolean
```

The operations `isEmpty`, `notEmpty` and `size` check for the existence of elements and determine the number of elements in the collection, respectively.

```
? Set{7}->excluding(7)->isEmpty()
> true : Boolean

? Bag{7}->excluding(8)->notEmpty()
> true : Boolean

? Set{7,8,7,8,9}->size()
> 3 : Integer

? Bag{7,8,7,8,9}->size()
> 5 : Integer
```

```
? Sequence{7,8,7,9}->size()
> 4 : Integer

? OrderedSet{7,8,7,9}->size()
> 3 : Integer
```

In order to filter collection elements the operations `select` and `reject` apply and in order to construct new collections from existing ones the operations `collect` and `collectNested` can be employed. Please note that `collect` applied to a set has to return a bag, because the functional term inside the collect may map two different source elements to the same target value. An analogous mechanism applies to ordered sets when a `collect` is used, because then the result will be a sequence. When applying `collect`, a possibly nested result is automatically converted into a flat collection. When you want to obtain the nested result you have to use `collectNested`. The following examples use a conditional `if then else endif` which is available in OCL on all types.

```
? Set{7,8,9}->select(i|i.mod(2)=1)
> Set{7,9} : Set(Integer)

? Bag{7,8,7,9}->reject(i|i.mod(2)=1)
> Bag{8} : Bag(Integer)

? Sequence{7,8,9}->collect(i|i*i)
> Sequence{49,64,91} : Sequence(Integer)

? Set{-1,0,+1}->collect(i|i*i)
> Bag{0,1,1} : Bag(Integer)

? OrderedSet{-1,0,+1}->collect(i|i*i)
> Sequence{1,0,1} : Sequence(Integer)

? Set{7,8,9}->collect(i|if i.mod(2)=0 then 'Even' else 'Odd' endif)
> Bag{'Even','Odd','Odd'} : Bag(String)

? Set{7,8}->collectNested(i|Sequence{i,i*i})
> Bag{Sequence{7,49},Sequence{8,64}} : Bag(Sequence(Integer))

? Set{7,8,9}->collect(i|Sequence{i,i*i})
> Bag{7,8,9,49,64,81} : Bag(Integer)
```

Another group of OCL collection operations are the operations `one`, `any`, `isUnique`, and `sortedBy`. `one` is a variation of the `exists` quantifier which yields true if exactly one element in the collection meets the specified predicate. `any` is a non-deterministic choice from the collection elements which satisfy the specified predicate. A deterministic use of this operation is when it is applied to a collection with exactly one value. Such a call realizes a coercion from the collection type `Collection(T)` to the parameter type `T`. `isUnique` checks whether

the mapping achieved by applying the functional inner expression to each collection element is a one-to-one mapping. `sortedBy` converts a collection into a sequence using the specified collection element properties. `union` builds the union of the two specified collections. For sequences and ordered sets, it results in the concatenation.

```
? Set{7,8,9}->one(i|i.mod(2)=0)
> true : Boolean

? Set{7,8,9}->one(i|i.mod(2)=1)
> false : Boolean

? Set{7,8,9}->any(true)
> 7 : Integer -- implementor's decision
> 8 : Integer -- also allowed
> 9 : Integer -- also allowed

? Set{7,8,9}->any(i|i.mod(2)=0)
> 8 : Integer

? let C=Set{7} in if C->size()=1 then C->any(true) else null endif
> 7 : Integer

? let C=Set{7,8,7} in if C->size()=1 then C->any(true) else null endif
> null : OclVoid

? Set{7,8,9}->isUnique(i|i*i)
> true : Boolean

? Set{7,8,9}->isUnique(i|i.mod(2)=0)
> false : Boolean

? Bag{8,7,8,9}->sortedBy(i|i)
> Sequence{7,8,8,9} : Sequence(Integer)

? Set{7,8,9}->sortedBy(i|if i.mod(2)=0 then 'Even' else 'Odd' endif)=
> Sequence{8,7,9} : Sequence(Integer)

? Sequence{-8,9,-7}->sortedBy(i|i.abs)
> Sequence{-7,-8,9} : Sequence(Integer)

? Set{7,8}->union(Set{9,8})
> Set{7,8,9} : Set(Integer)

? Bag{7,8}->union(Bag{9,8})
> Bag{7,8,8,9} : Bag(Integer)

? Sequence{7,8}->union(Sequence{9,8})
> Sequence{7,8,9,8} : Sequence(Integer)
```

```
? OrderedSet{7,8}->union(OrderedSet{9,8})
> OrderedSet{7,8,9} : OrderedSet(Integer)
```

OCL offers the possiblity to convert one collection kind into any of the other three collection kinds by means of the operations `asSet`, `asBag`, `asSequence`, and `asOrderedSet`. Please be aware of the fact that some these conversions must make an implementation dependent decision, for example, the conversion that takes sets and returns sequences. In order to flatten a nested collection the operation `flatten` can be used to obtain a flat collection having the same elements as the source nested collection. Thus the operation `collect` can be seen as a shortcut for `collectNested` and a following `flatten`. `flatten` returns the top-most collection kind of the source expression. `flatten` also must make implementation dependent decisions. Such decisions must be taken, if the conversion goes from an order insensible collection kind to an order sensible collection kind.

```
? Sequence{8,7,7,8}->asSet()
> Set{7,8} : Set(Integer)

? Sequence{8,7,7,8}->asBag()
> Bag{7,7,8,8} : Bag(Integer)

? Set{8,7,7,8}->asSequence()
> Sequence{7,8} : Sequence(Integer) -- implementor's decision
> Sequence{8,7} : Sequence(Integer) -- also allowed

? Sequence{8,7,7,8}->asOrderedSet()
> OrderedSet{8,7} : OrderedSet(Integer)

? Set{8,7,9}->asSequence()
> Sequence{7,8,9} : Sequence(Integer) -- implementor's decision
> Sequence{9,8,7} : Sequence(Integer) -- also allowed
> Sequence{7,9,8} : Sequence(Integer) -- also allowed

? Set{7,8}->collectNested(i|Sequence{i,i*i})->flatten()
> Bag{7,8,49,64} : Bag(Integer)

? Sequence{Set{7,8},Set{8,9}}->flatten()
> Sequence{7,8,9,8} : Sequence(Integer) -- implementor's decision

? Set{Bag{7,8},Bag{8,9}}->flatten()
> Set{7,8,9} : Set(Integer)

? OrderedSet{Bag{7,9},Bag{8,7}}->flatten()
> OrderedSet{7,9,8} : OrderedSet(Integer)

? OrderedSet{Set{7,8,9}}->flatten()
> OrderedSet{7,9,8} : OrderedSet(Integer) -- implementor's decision
> OrderedSet{9,7,8} : OrderedSet(Integer) -- also allowed
> OrderedSet{7,9,8} : OrderedSet(Integer) -- also allowed
```

Concerning the above decision which the implementor has to take, one might argue that the order on type `Integer` is pretty well determined. But please recall that ordered sets are not sorted sets. And, there ist no natural *single* order on object collections for user-defined class types: one natural order on objects is determined by their object identity often being an identifier, and a second natural order is the order in which the objects are created.

4.8 OCL Collection Operation `iterate`

The last collection operation `iterate` is the most complicated one, but also the most powerful collection operation, because, basically, all other collection operation are special cases of `iterate`. The syntax of the basic form of an iterate expression is represented as follows.

```
COLEXPR->iterate(ELEMVAR:ELEMTYPE; RESVAR:RESTYPE=INITEXPR | ITEREXPR)
```

An iterate expression is based on other expressions, on variables and on types: a collection expression `COLEXPR` for the argument collection, an element variable `ELEMVAR` for an iteration variable, an element type `ELEMTYPE`, a result variable `RESVAR`, a result type `RESTYPE`, an initialization expression `INITEXPR` for the result variable, and an iteration expression `ITEREXPR` for the result variable. The iterate expression is applied to an argument collection `COLEXPR`. Within a loop, each argument collection element having the type `ELEMTYPE` is considered once with the variable `ELEMVAR`. Thus the number of steps in the loop is equal to the number of elements in the argument collection. The result of the iterate expression is of type `RESTYPE` and fixed by the variable `RESVAR` which is initialized with the expression `INITEXPR` before the loop is entered. Within the loop, the expression `ITEREXPR` is evaluated for each element of the argument collection once and the intermediate result is again assigned to `RESVAR`. `ITEREXPR` may use `ELEMVAR` and `RESVAR` as free variables, but `ITEREXPR` is not forced to do so. The overall result of the iterate expression is determined by the last value of `RESVAR`.

We consider the following examples for `iterate`. These examples will use the relational database state which we have expressed above as a nested tuple value. The OCL examples will use the abbreviations `C` and `R`, for all customer and rental tuples, respectively. For the respective example, we will show also its SQL counterpart and a formulation without `iterate`.

(A) Show names in Customers together with names in Rentals. Two `iterate` expressions are employed: one over the `Customer` relation, one over the `Renatls` relation. The formulation without `iterate` employs two `collect` enpressions.

```
let dbs=Tuple{Customers:Set{Tuple{name:'Ada',birth:1962},
                            Tuple{name:'Bob',birth:1962}},
              Rentals:Set{Tuple{name:'Ada',start:'2012-01-01'},
                          Tuple{name:'Ada',start:'2002-01-01'},
                          Tuple{name:'Cyd',start:'2002-01-01'}}} in
let C=dbs.Customers in let R=dbs.Rentals in
C->iterate(c;R1:Bag(String)=Bag{}|R1->including(c.name))->union(
```

```
  R->iterate(r;R2:Bag(String)=Bag{}|R2->including(r.name)))

Bag{'Ada','Ada','Ada','Bob','Cyd'} : Bag(String)

SELECT name FROM Customers UNION SELECT name FROM Rentals

C->collect(c|c.name)->union(R->collect(r|r.name))
```

(B) Retrieve the earliest rentals. The formulation with `iterate` uses two nested `iterate` expression with different result types. The outer `iterate` corresponds to a `select` call, the inner `iterate` to a universal quatification.

```
R->iterate(r1;R1:Set(Tuple(name:String,start:String))=Set{}|
     if R->iterate(r2;R2:Boolean=true|R2 and r1.start<=r2.start)
       then R1->including(r1) else R1 endif)

Set{Tuple{name='Ada',start='2002-01-01'},
    Tuple{name='Cyd',start='2002-01-01'}} :
    Set(Tuple(name:String,start:String))

SELECT * FROM Rentals WHERE start <= ALL (SELECT start FROM RENTALS)

R->select(r1|R->forAll(r2|r1.start<=r2.start))
```

(C) Show names in Rentals from year 2002. This OCL expression uses the operation `substring` which is applied to a `String` value with two parameters indicating the first and the last position of the substring to be retrieved. Note that the same effect in the two calls to `select` and `collect` in the second formulation is achieved in the first formulation with a single `iterate` call.

```
R->iterate(r;R1:Bag(String)=Bag{}| if r.start.substring(1,4)='2002'
     then R1->including(r.name) else R1 endif)

Bag{'Ada','Cyd'} : Bag(String)

SELECT name FROM Rentals WHERE start.substring(1,4)='2002'

R->select(r|r.start.substring(1,4)='2002')->collect(r|r.name)
```

5 Tool Support

Though still limited, OCL tool support has been considerably growing in the last years. The goal of this section is to present a sorted (non-exhaustive) list of tools that can help in your OCL learning process. Other reports of OCL tools are [10] and [12].

5.1 OCL Parsers and IDEs

The two main OCL Parsers available today are MDT/OCL [7] and Dresden OCL[8].

MDT/OCL is part of the official Model Development tools Eclipse project whose goal is to provide an implementation of industry standard metamodels and to provide exemplary tools for developing models based on those metamodels. MDT/OCL provides a set of APIs for parsing and evaluating OCL constraints and queries on Ecore or UML models, support for serializing parsed OCL expressions (avoiding the need for reparsing them every time we load the model), and a visitor API for the abstract syntaxt tree to allow their transformation.

DresdenOCL provides a set of tools to parse and evaluate OCL constraints on various types of models thanks to its Pivot Model strategy [18]. The pivot model decouples the OCL parser and interpreter from a specific metamodel and thus enables connecting the tool to every meta-model supporting basic object-oriented concepts. DresdenOCL can be executed as an independent tool or integrated in the EMF Eclipse framework.

Due to the relevance of OCL in other areas, we can also find OCL parsers embedded in other kinds of MDE components. This is specially true in the case of model transformations where each transformation engine (e.g., the ATL[9] one) comes with its own OCL parser. This is not an ideal situation since each differs on the kind of OCL expressions supported (and even worse, sometimes also on how they interpret them). In this sense, SimpleOCL [10] looks like a step in the right direction for MDE tools that do not need/want to integrate the full OCL language. SimpleOCL is intended as an embeddable OCL implementation for inclusion in transformation languages.

5.2 UML Tools with OCL Support

Unfortunately only a handful of UML modeling tools are equipped with OCL support. By "OCL support" we mean that the UML tool is able to at least understand (i.e., parse) the OCL expressions attached to the model and not treat them just as plain text strings (same as they were just natural language).

Some exceptions are:

- ArgoUML[11] provides syntax and type checking of OCL expressions thanks to the integration of DresdenOCL
- Rational Rose thanks to the OClarity plug-in[12] offers syntax,type and some semantic checkings for OCL Expressions (e.g., detecting that a non-navigable association is traversed as part of the expression)
- Enterprise Architect[13] allows users to add and validate OCL constraints

[7] http://www.eclipse.org/modeling/mdt/?project=ocl
[8] http://www.dresden-ocl.org/index.php/DresdenOCL
[9] http://www.eclipse.org/atl/
[10] http://soft.vub.ac.be/soft/research/mdd/simpleocl
[11] http://argouml.tigris.org/
[12] http://www.empowertec.de/products/rational-rose-ocl/
[13] http://www.sparxsystems.com/

- MagicDraw [14] includes an OCL execution engine that can be used to write, validate (models vs metamodels, instances vs models) and execute (e.g., querying) OCL expressions
- Borland Together [15] offers syntax highlighting and checking of OCL expressions
- Several UML tools in Eclipse like Papyrus [16] integrate the MDT/OCL component introduced in the previous section.

We believe that the increasing quality and availability of OCL parsers and evaluators ready to be embedded in other tools will help to improve this situation in the near future.

5.3 Verification and Validation Tools for OCL

OCL is a very expressive language that allows designer to write complex constraint, derivation rules, pre/postconditions,etc. Therefore, it is easy to make mistakes while writing OCL expressions. Tools mentioned in the previous section take care of the syntactic errors (i.e., they make sure that the expressions are "grammatically" correct). Nevertheless, syntactic correctness is not enough. This section introduces some tools to validate and verify OCL expressions. With these tools designers may check that the expressions are a valid representation of the domain and that there are no inconsistencies, redundancies, ... among them.

The tool USE (UML-based Specification Environment) [16,19] can be employed to validate and partly to verify a model. System states (snapshots of a running system) can be created semi-automatically and manipulated. For each snapshot the OCL constraints are automatically checked and the results are given to the designer using graphical or textual views. This simulation of the system allows designers to identify if the model is overconstrained (i.e., some valid situations in the domain are not allowed by the specification) or underconstrained (some invalid scenarios are evaluated as correct in the specification). With USE properties like constraint consistency or independency [17] can be checked. USE supports UML class, object, sequence and statechart diagrams.

Advanced correctness properties may require a more complete reasoning on the expressions and the system states that each constraint restricts. At least, we must ensure that the constraints are *satisfiable*, i.e., there are finite and non-empty system states that evaluate to true all model constraints at the same time (obiously, if the model constraints are unsatisfiable, the model is useless since users will never be able to create valid instantiations of it). Unfortunately, reasoning on OCL is undecidable in general. Therefore, current verification tools either require user interaction with the verification procedure (e.g., HOL-OCL [6], based on the Isabelle theorem prover), restrict the OCL constructs that can be used when writing OCL expressions (e.g., [23], based on query containment checking techniques) or follow a bounded verification approach, where the search

[14] `https://www.magicdraw.com/`
[15] `http://www.borland.com/us/products/together/index.aspx`
[16] `http://www.eclipse.org/modeling/mdt/papyrus/Papyrus`

space is finite in order to guarantee termination. The bounds in the verification are set by limiting the number of instances and the restricting the attribute domains to explore during the verification. Examples of tools in this category are UML2Alloy [1] (based on a translation of the UML/OCL models into Alloy), [24] (OCL constraints reexpressed as a boolean satisfiability problem) and UMLtoCSP [8] and EMFtoCSP [17](UML/OCL and EMF models, respectively, are reexpressed as a Constraint Satisfaction Problem (CSP)).

Other correctness properties can be defined in terms of this basic satisfiability property.

5.4 Code Generation from OCL Expressions

Constraints at the model level state conditions that the "data" ot the system must satisfy at runtime. Therefore, the implementation of a system must guarantee that all operations that modify the system state will leave the data in a consistent state (by consistent we mean a state that evaluates to true all model invariants). Clearly, the best way to achieve this goal (and to reuse the effort put by the designers when precisely specifying the models) is by providing code-generation techniques that take the OCL constraints and produce the appropriate checking code in the target platform where the system is going to be executed.

Typically, OCL expressions are translated into code either as database triggers or as part of the method bodies in the classes corresponding to the constraint context types. Roughly, in the database strategy each invariant is translated as a SQL SELECT expression (or a view) that returns a value if the data does not satisfy that given constraint (usually, this value returned by the SELECT is the set of rows that are inconsistent). This SELECT expression is called inside the body of a trigger so that if the SELECT returns a non-empty value then the trigger raises an exception. Triggers are fired after every change on the data to make sure that the system is always in a consistent state. When implementing the constraints as part of an object-oriented language, constraints are usually embedded in the method bodies of the classes. There are several ways to embed them. For instance, we could add them as if-then conditions at the beginning of the method or, if the language offers this possibility, as assertion expressions.

In both scenarios, the efficiency of the integrity checking process can be improved a lot if we follow an incremental checking strategy [11]. The idea is to minimize the amount of data that must be reevaluated after every update on the system state by determining at design-time, when and how each constraint must be checked at runtime to avoid irrelevant verifications. Clearly, the *NoRentalsBlackListed* invariant can become violated when adding a rental to a BlackListed person but not when changing the name of that person, nor when we remove one of his rentals or change its rental price. Therefore, instead of checking this constraint after each state change we can just check it (and only for the affected pieces of data) after assignments of new rentals, a blacklisting of a Customer or

[17] `http://code.google.com/a/eclipselabs.org/p/emftocsp/`

changes on the involved dates and forget about it for all the other events. This "knowledge" can be used to decide which triggers must call the SELECT expression corresponding to this constraint or on which method bodies the if-then condition for the constraint must be added.

Despite the usefulness of these code-generation techniques for OCL, most MDD tools do not include them as part of their code-generation features (in fact, for this particular aspect the survey in [10] is still valid nowadays). Some prefer to provide more limited (in terms of expressiveness) DSLs that allow users to define simple validation rules to be implemented in the interface layer (as part of form validation conditions.

6 Research Agenda for OCL

This section hints at some research lines we belive are important challenges for the evolution and continued success of OCL.

6.1 Modularization and Extensibility

OCL is a very expressive language with an extensive standard library. In fact, the large number of operators in the library and their overlappings (many expressions can be written using alternative combinations of operators) may be confusing for users only interested in writing simple expressions.

On the other hand, the library is missing some relevant operators, like basic statistical functions [9] that make cumbersome using OCL in some domains.

Therefore, we believe there is a clear need of adding modularization constructs to the language that enable users to select the exact set of OCL *modules* they need, including, when necessary, the import of external OCL libraries created by OCL experts to extend the language.

The need of OCL libraries has also been raised by other researchers [2], [28] but it is still an open problem with many issues to be solved: how to make the libraries available?, who validates them?, strategies to solve conflicts when importing several libraries?, how are the libraries defined?, how to express the semantic of each individual operation?, etc.

6.2 Language Improvements

Even if OCL is already in its version 2.3 the language itself offers plenty of opportunities for improvement both at the concrete and abstract syntax levels.

At the concrete level, users still have problems with some notational aspects like the overlapping of the dot notation and the arrow notation for collections with a single element. Besides, OCL expressions involving iterators become quite verbose quickly so a few shortcuts have been proposed [18]. Moreover, at the abstract syntax level, several issues regarding the OCL type system (e.g., [7]) and undefinedness semantics [5] have been detected. Not to mention that OCL is still missing a complete definition of its formal semantics.

[18] `http://eclipsemde.blogspot.com/2010/05/acceleo-ocl-made-simple.html`

6.3 Efficient Reasoning on OCL Expressions

The application of MDE to more complex problems (like model-driven reverse engineering where very large models are automatically obtained from source code) requires efficient evaluation and reasoning techniques for OCL. Right now, OCL analysis techniques exhibit scalability issues when dealing with large models (e.g., when verifying them or when identifying matching submodels as part of a model transformation).

Some initial results in the area have focused on the incremental [11,3] or lazy evaluation of OCL expressions [25]. In the former, we aim to minimize the number of instances that are accessed every time we evaluate the expression while in the latter we delay the evaluation of the OCL expressions to the last possible moment, i.e., only when the user wants to access an element that it is computed by an OCL expression (e.g., a target element in a model transformation), that expression is evaluated.

Nevertheless much work needs to be done. One area worth exploring is the use of a cloud computing environment as an execution infrastructure for OCL-related analysis services. The model to be evaluated could be sliced and processed in parallel in a network of virtual nodes in the cloud.

6.4 Establishing an OCL Community

One aspect hindering the adoption (and as a consequence the evolution) of OCL is the lack of an established community of OCL practitioners that pushes the language forward.

The *OCL and Textual Modeling Languages Workshop*[19]is the most important (and basically the only) annual meeting point for researchers. Even though the organizers (among them the authors of this chapter) always try to bring industrial practitioners, the success is limited.

The OMG OCL RTF (Revision Task Force) who maintains the OCL specification could lead the creation of a professional community around OCL but given the *closed*[20] nature of the OMG, its impact is rather limited. For instance, it has been proven very difficult for researchers to influence the evolution of the OCL standard (of course, this is not only OMG's fault but also due to the nature of the research work; researchers have very limited time and resources to actively participate in standardization committees).

Some online forums, like the Eclipse OCL community forum[21] facilitate a joint discussion between researchres and practitioners but they focus on specific tools. The OCL Portal [22] was born with the goal of collecting all information about OCL but unfortunately the activity level is low. OCL is also a topic discussed in the Modeling Languages portal [23].

[19] See `http://gres.uoc.edu/OCL2011/` for information on its latest edition

[20] The results of the task force are public but participation for non-OMG members is restricted.

[21] `http://www.eclipse.org/forums/index.php?t=thread&frm_id=26`

[22] `http://st.inf.tu-dresden.de/ocl/`

[23] `http://modeling-languages.com`

We hope that with the increasing adoption of OCL, the number of practitionesr reaches the critical mass needed to create a real community around the language where researchres and practitioners work and discuss together.

7 Conclusions

This chaper has provided a broad overview of the OCL language including its main usage scenarios, a precise overview of the language constructs and the current tool support available to those interested in using OCL in their new software development projects.

Of course, OCL is far from perfect. We have identified several research challenges that the community must address in order to facilitate the adoption of OCL among practitioners. We hope by now you are convinced that, given the important role of OCL in the model-driven engineering paradigm, these challenges are worth pursuing.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MoDELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)
2. Baar, T.: On the need of user-defined libraries in OCL. ECEASST 36 (2010)
3. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MoDELS 2010, Part I. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010)
4. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Trans. Software Eng. 21(10), 785–798 (1995)
5. Brucker, A.D., Krieger, M.P., Wolff, B.: Extending OCL with null-references. In: Ghosh [15], pp. 261–275
6. Brucker, A.D., Wolff, B.: The HOL-OCL book. Technical Report 525, ETH Zurich (2006)
7. Büttner, F., Gogolla, M., Hamann, L., Kuhlmann, M., Lindow, A.: On better understanding OCL collections *or* an OCL ordered set is not an OCL set. In: Ghosh [15], pp. 276–290
8. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: ASE, pp. 547–548. ACM (2007)
9. Cabot, J., Mazón, J.-N., Pardillo, J., Trujillo, J.: Specifying aggregation functions in multidimensional models with OCL. In: Parsons, et al. [22], pp. 419–432
10. Cabot, J., Teniente, E.: Constraint Support in MDA Tools: A Survey. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 256–267. Springer, Heidelberg (2006)
11. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software 82(9), 1459–1478 (2009)
12. Chimiak-Opoka, J.D., Demuth, B., Awenius, A., Chiorean, D., Gabel, S., Hamann, L., Willink, E.D.: OCL tools report based on the ide4OCL feature model. ECEASST 44 (2011)

13. Dobing, B., Parsons, J.: How UML is used. Commun. ACM 49, 109–113 (2006)
14. Frias, L., Queralt, A., Olivé, A.: Eu-rent car rentals specification. Technical Report LSI Research Report. LSI-03-59-R, UPC (2003)
15. Ghosh, S. (ed.): MoDELS 2009. LNCS, vol. 6002. Springer, Heidelberg (2010)
16. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling 4(4), 386–398 (2005)
17. Gogolla, M., Büttner, F., Richters, M.: Use: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3), 27–34 (2007)
18. Heidenreich, F., Wende, C., Demuth, B.: A framework for generating query language code from OCL invariants. ECEASST 9 (2008)
19. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
20. Object Management Group. OCL 2.3.1 Specification (2010)
21. Object Management Group. UML 2.4.1 Superstructure Specification (2011)
22. Parsons, J., Saeki, M., Shoval, P., Woo, C.C., Wand, Y. (eds.): ER 2010. LNCS, vol. 6412. Springer, Heidelberg (2010)
23. Queralt, A., Rull, G., Teniente, E., Farré, C., Urpí, T.: Aurus: Automated reasoning on UML/OCL schemas. In: Parsons, et al. [22], pp. 438–444
24. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using boolean satisfiability. In: DATE, pp. 1341–1344. IEEE (2010)
25. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Lazy Execution of Model-to-Model Transformations. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS 2011. LNCS, vol. 6981, pp. 32–46. Springer, Heidelberg (2011)
26. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley (2003)
27. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Comput. Surv. 30(4), 459–527 (1998)
28. Willink, E.D.: Modeling the OCL standard library. ECEASST 44 (2011)