Appendix B. The Object Constraint Language

The Object Constraint Language 2.0 (OCL) is an addition to the UML 2.0 specification that provides you with a way to express constraints and logic on your models. For example, you can use OCL to convey that a person's age must always be greater than 0 or that a branch office must always have one secretary for every 10 employees.

OCL isn't new to UML 2.0; it was first introduced in UML 1.4. However, as of UML 2.0, it was formalized using the Meta-Object Facility and UML 2.0. From a user's perspective the language has been updated and refined but the fundamentals remain the same. This appendix introduces the basic concepts of OCL. For more detailed information, consult the OCL specification available from the Object Management Group's web site (http://www.omg.org/).

PREV NEXT

B.1. OCL Basics

The Object Constraint Language is just that: a language. It obeys a syntax and has keywords. However, unlike other languages, it can't be used to express program logic or flow control. By design, OCL is a *query-only* language; it can't modify the model (or executing system) in any way. It can be used to express preconditions, postconditions, invariants (things that must always be `TRue`), guard conditions, and results of method calls.

OCL can be used virtually anywhere in UML and is typically associated with a classifier using a note. When an OCL expression is evaluated, it is considered to be instantaneous, meaning the associated classifier can't change state during the evaluation of an expression.

B.1.1. Basic Types

OCL has several built-in types that can be used in OCL expressions:

Boolean

Must be either `true` or `false`. Supports the logical operators`and`,`or`,`xor`,`not`,`implies`, and`if-then-else`.

Integer

Any integer value (e.g., 100, -12345, 5211976, etc.). Supports the operators `*`,`+`,`-`,`/`, and`abs( )`.

Real

Any decimal value (e.g., 2.222003, -67.76, etc.). Supports the operators `*`,`+`,`-`,`/`, and `floor( )`.

String

A series of letters, numbers, or symbols interpreted as a string (e.g., "All writing and no play make Dan..."). Supports the operators `concat( )`,`size( )`, and `substring( )`.

In addition to the built-in types, any classifier used in your UML model is recognized as a type by OCL. Because OCL is a strongly typed language, you can't compare values of one type directly with values of another type.

B.1.2. Casting

OCL does support *casting* objects from one type to another as long as they are related through a generalization relationship. To cast from one type to another use the operation `*oldType*.oclAsType(*newType*)` . For example, to cast a Java `List` to an `ArrayList` to call the `size( )` operation, use the expression:

```
    List.oclAsType(ArrayList).size(  )
```

If the actual object isn't an instance of the new type, the expression is considered undefined.

PREV NEXT

PREV NEXT

B.2. OCL Syntax

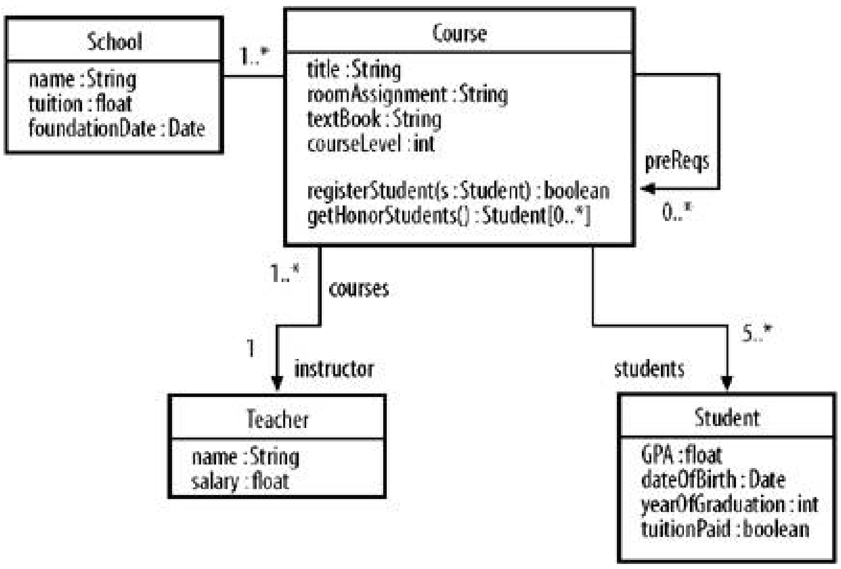
The remainder of this chapter uses examples from the class diagram shown in Figure B-1.

B.2.1. Constraints on Classifiers

Each OCL expression must have some sense of *context* that an expression relates to. Often the context can be determined by where the expression is written. For example, you can link a constraint to an element using a note. You can refer to an instance of the context classifier using the keyword `self`. For example, if you had a constraint on `Student` that their GPA must always be higher than 2.0, you can attach an OCL expression to `Student` using a note and refer to the GPA as follows:

```
self.GPA > 2.0
```

Figure B-1. Example class diagram used in this chapter



It's important to realize that this OCL expression is an invariant, meaning the system would be in an invalid state if a student's GPA dropped to less than 2.0. If you want to allow a GPA of less than 2.0 and send out a letter to the student's parents in the event such a low GPA is achieved, you would model such behavior using a UML diagram such as an activity or interaction diagram.

You can follow associations between classifiers using the association end names as though they were attributes of the originating classifier. The following invariant on `Course` ensures that the instructor is being paid:

```
self.instructor.salary > 0.00
```

If an association has a multiplicity of 0..1, you can treat the association end as a `Set` and check to see if the value is set by using the built-in `notEmpty( )` operation. To call the `notEmpty( )` operation on a set you must use an arrow (`->`) rather than a dot (`.`). See "Collections" for more information on sets. The following invariant on `Course` enforces that a course has an instructor:

```
self.instructor->notEmpty(  )
```

If an association role name isn't specified, you can use the classifier name. The following invariant on `School` checks that each course has a room assignment:

```
self.Course->forAll(roomAssignment <> 'No room!')
```

Comments can be inserted into an expression by prefixing each comment with two dashes (), like this:

```
-- make sure this student could graduate
self.GPA > 2.0
```

If you can't determine the context of an expression by looking at the UML model, or if you want to be explicit with the context, use the OCL keyword `context`, followed by the classifier name. If you use this notation, you should say what the expression represents. In the following case, we're showing an invariant, so we use the keyword `inv`. Other types of expressions are explained in later sections.

```
context Student
inv: self.GPA > 2.0
```

Instead of using the keyword `self`, you can assign a name to a classifier that you can use in the expression. Write the name you want to use, followed by a colon (`:`) and then the classifier name. For example, you can name the instance of `Student` as `s`:

```
context s : Student
inv: s.GPA > 2.0
```

Finally, you can name an expression by placing a label after the expression type but before the colon (`:`). The label is purely decorative; it serves no OCL function.

```
context s : Student
inv minimumGPARule: s.GPA > 2.0
```

B.2.2. Constraints on Operations

Beyond basic classifiers, OCL expressions can be associated with operations to capture preconditions and postconditions. Place the signature of the target operation (classifier, operation name, parameters, etc.) after the `context` keyword. Instead of the invariant keyword, use either `pre` or `post` for preconditions and postconditions, respectively.

The following expression ensures that any student who will be registered for a course has paid their tuition:

```
context Course::registerStudent(s : Student) : boolean
pre: s.tuitionPaid = true
```

When writing postconditions, you can use the keyword `result` to refer to the value returned by an operation. The following expressions verify that a student's tuition was paid before registering for a course and that the operation `registerStudent` returned `true`:

```
context Course::registerStudent(s : Student) : boolean
pre: s.tuitionPaid = true
post: result = true
```

As you can with invariants, you can name preconditions and postconditions by placing a label after the `pre` or `post` keywords:

```
context Course::registerStudent(s : Student) : boolean
pre hasPaidTuition: s.tuitionPaid = true
post studentWasRegistered: result = true
```

Postconditions can use the `@pre` keyword to refer to the value of some element *before* an operation executes. The following expression ensures that a student was registered and that the number of students in the course has increased by 1. This expression uses the `self` keyword to reference the object that owns the `registerStudent` operation.

```
context Course::registerStudent(s : Student) : boolean
pre: s.tuitionPaid = true
post: result = true AND self.students = self.students@pre + 1
```

You may specify the results of a *query* operation using the keyword `body`. Because OCL doesn't have syntax for program flow, you are limited to relatively simple expressions. The following expression indicates that honors students are students with GPAs higher than 3.5. The collection syntax used in this example is described in the "Collections" section.

```
context Course::getHonorsStudents( ) : Student
body: self.students->select(GPA > 3.5)
```

B.2.3. Constraints on Attributes

OCL expressions can specify the initial and subsequent values for attributes of classifiers. When using OCL expressions with attributes, you state the context of an expression using the classifier name, two colons (`::`), the attribute name, another colon (`:`), and then the type of the attribute. You specify the initial value of an attribute using the keyword `init`:

```
context School::tuition : float
init: 2500.00
```

You can specify the value of attributes after their initial value using the `derive` keyword. The following example increases the tuition value by 10% every time you query it:

```
context: School::tuition : float
derive: tution * 10%
```

PREV NEXT

B.3. Advanced OCL Modeling

Like any other language, OCL has an order of precedence for operators, variable declarations, a
logical constructs (only for evaluating your expressions, not for program flow). The following
sections describe constructs that you can use in any OCL expression.

B.3.1. Conditionals

OCL supports basic boolean expression evaluation using the `if-then-else-endif` keywords. The
conditions are used only to determine which expression is evaluated; they can't be used to
influence the underlying system or to affect program flow. The following invariant enforces that
student's year of graduation is valid only if she has paid her tuition:

```
context Student inv:
if tuitionPaid = true then
  yearOfGraduation = 2005
else
  yearOfGraduation = 0000
endif
```

OCL's logic rules are slightly different from typical programming language logic rules. The boole
evaluation rules are:

1. True`OR`-ed with anything is `true`.
2. False`AND`-ed with anything is `false`.
3. False`IMPLIES`*anything* is `TRue`.

The`implies` keyword evaluate the first half of an expression, and, if that first half is `true`, the res
is taken from the second half. For example, the following expression enforces that if a student's
GPA is less than 1.0, their year of graduation is set to 0. If the GPA is higher than 1.0, Rule #3
applies, and the entire expression is evaluated as `true` (meaning the invariant is valid).

```
context Student inv:
self.GPA < 1.0 IMPLIES self.yearOfGraduation = 0000
```

OCL's boolean expressions are valid *regardless of the order of the arguments* . Specifically, if the
first argument of an `AND` operator is undefined, but the second operator is `false`, the entire
expression is `false`. Likewise, even if one of the arguments to an `OR` operator is undefined, if the
other is `true`, the expression is `true`.`If-then-else-endif` constructs are evaluated similarly; if th
chosen branch can be evaluated to `TRue` or `false`, the nonchosen branch is completely disregard
(even if it would be undefined).

B.3.2. Variable Declaration

OCL supports several complex constructs you can use to make your constraints more expressive and easier to write. You can break complex expressions into reusable pieces (within the same expression) by using the `let` and `in` keywords to declare a variable. You declare a variable by giving it a name, followed by a colon (`:`), its type, an expression for its value, and the `in` keywor The following example declares an expression that ensures a teacher of a high-level course has appropriate salary:

```
context Course inv:
let salary : float = self.instructor.salary in
if self.courseLevel > 4000 then
  salary > 80000.00
else
  salary < 80000.00
endif
```

You can define variables that can be used in multiple expressions on a classifier-by-classifier bas using the `def` keyword. For example, instead of declaring`salary` as a variable using the `let` keyword, you can define it using the `def` keyword for the `Course` context. Once you define a variable using the `def` keyword, you may use that variable in any expression that is in the same context. The syntax for the `def` keyword is the same as that for the `let` keyword:

```
context Course
def: salary : float = self.instructor.salary
```

So, now you can write the previous invariant as:

```
context Course inv:
if self.courseLevel > 4000 then
  salary > 80000.00
getHonorsStudentselse
  salary < 80000.00
endif
```

B.3.3. Operator Precedence

OCL operators have the following order of precedence (from highest to lowest):

- `@pre`
- dot (`.`) and arrow (`->`) operations
- `not` and unary minus (`-`)
- `*` and `/`
- `+` and -
- `if-then-else-endif`
- `<`,`>`,`<=`, and `>=`

- `=` and `<>`
- `and`,`or`, and `xor`
- `implies`

You can use parentheses to group expressions, which will be evaluated from the innermost set c
parentheses to the outermost.

B.3.4. Built-in Object Properties

OCL provides a set of properties on all objects in a system. You can invoke these properties in y
expressions as you do any other properties. The built-in properties are:

`oclIsTypeOf (`*`t`*` : `*`Type`*`): Boolean`

Returns`true` if the tested object is exactly the same type as *`t`*.

`oclIsKindOf(`*`t`*` : `*`Type`*`): Boolean`

Returns`TRue` if the tested object is the same type or a subtype of *`t`*.

`oclInState(s : `*`State`*`): Boolean`

Returns`TRue` if the tested object is in state *`s`*. The states you can test must be part of a sta
machine attached to the classifier being tested.

`oclIsNew( ) : Boolean`

Designed to be used in a postcondition for an operation, it returns `true` if the object being
tested was created as a result of executing the operation.

`oclAsType (`*`t`*` : `*`Type`*`): Type`

Returns the owning object casted to *`Type`*. If the object isn't a descendant of *`t`*, the operati
is undefined.

Here are some examples of the built-in properties:

```
-- test that the instructor is an instance of Teacher
context Course
inv: self.instructor.oclIsTypeOf(Teacher)

-- cast a Date class to a java.sql.Date to verify the minutes
-- (it's very unlikely the foundationDate would be a java.sql.Date
--  so this invariant would be undefined, but this is an example
--  of using oclAsType(  ))
context School
inv: self.foundationDate.oclAsType(java.sql.Date).getMinutes(  ) = 0
```

B.3.5. Collections

OCL defines several types of collections that represent several instances of a classifier. The basi type is `Collection`, which is the base class for the other OCL collection classes. Quite a few operations are defined for `Collection`; see the OCL specification for the complete list.

All collections support a way to select or reject elements using the operations `select( )` and `reject( )`, respectively. To invoke an operation on a collection, you use an arrow (`->`) rather tha dot (`.`) (a dot accesses a property). The result of `select` or `reject` is another collection containir the appropriate elements. Remember that since OCL can't modify a system in any way, the orig collection is unchanged. The notation for a `select` is:

```
collection->select(boolean expression)
```

So, to select students with GPAs higher than 3.0, you can use the expression:

```
context Course::getHonorsStudents(  ) : Student
body: self.students->select(GPA > 3.0)
```

Or, to eliminate honor students that haven't paid their tuition:

```
context Course::getHonorsStudents(  ) : Student
body: self.students->select(GPA > 3.0)->reject(tuitionPaid = false)
```

In the previous examples, the context for the `select` and `reject` statements was implied. You ca explicitly name the element you want to use in the boolean expression by prefixing the expressi with a label and a pipe symbol (`|`). So, a rewrite of the GPA example using a label to identify ea student looks like this:

```
context Course::getHonorsStudents(  ) : Student
body: self.students->select(curStudent | curStudent.GPA > 3.0)
```

Finally, you can specify the type of the element you want to evaluate. You indicate the type by placing a colon (`:`) and the classifier type after the label. Each element of the collection you are evaluating must be of the specified type, or else the expression is undefined. You can rewrite th GPA example to be even more specific and require that it evaluate only `Student`s:

```
context Course::getHonorsStudents(  ) : Student
body: self.students->select(curStudent : Student | curStudent.GPA > 3.0)
```

Often you will need to express a constraint across an entire collection of objects, so OCL provide the`forAll` operation that returns `true` if a given Boolean expression evaluates to `true` for all of t elements in a collection. The syntax for `forAll` is the same as that for `select` and `reject`. So, yo can write a constraint that enforces that all students in a `Course` have paid their tuition:

```
context Course
inv: self.students->forAll(tuitionPaid = true)
```

As you can with `select`, you can name and type the variable used in the expression:

```
context Course
inv: self.students->forAll(curStudent : Student | curStudent.tuitionPaid = tr
```

If you need to check to see if there is at least one element in a collection that satisfies a boolear expression, you can use the operation `exists`. The syntax is the same as that for `select`. The following expression ensures that at least one of the students has paid their tuition:

```
context Course
inv: self.students->exists(tuitionPaid = true)
```

Like`select`, you can name and type the variables used in the expression:

```
context Course
inv: self.students->exists(curStudent : Student | curStudent.tuitionPaid = tr
```

You can check to see if a collection is empty using the operations `isEmpty` or `notEmpty`. The following expression ensures that the school has at least one course offering:

```
context School
inv: self.Course->notEmpty(  )
```

PREV

NEXT