The Object Constraint Language (OCL) Object Constraint Language OCL Object Constraint Language
## COLLABORATORS

<table>
<thead>
<tr>
<th>ACTION</th>
<th>NAME</th>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITTEN BY</td>
<td></td>
<td>April 4, 2008</td>
<td></td>
</tr>
</tbody>
</table>

## REVISION HISTORY

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>DATE</th>
<th>DESCRIPTION</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contents

1 The Object Constraint Language (OCL) 7
  1.1 Introduction to the OCL 7
    1.1.1 What is the OCL? 7
    1.1.2 History of the OCL 7
    1.1.3 Core OCL features 7
    1.1.4 Applicability to the OCL 7
      1.1.4.1 Specifying services contracts 8
      1.1.4.2 Automatic test generation 8
        1.1.4.2.1 Contract tests 8
        1.1.4.2.2 Integrity tests 8
  1.2 The context of an OCL expression 9
    1.2.1 Self 9
  1.3 OCL comments 9
  1.4 Model navigation 10
    1.4.1 Navigating to and from association classes 10
  1.5 The OCL if expression 11
  1.6 Let sub-expressions 12
  1.7 Def expressions 12
    1.7.1 Defining variables via def expressions 12
    1.7.2 Defining re-usable functions 12
  1.8 Data types available in OCL 13
    1.8.1 Primitive OCL types and operations 13
    1.8.2 Data types from the UML model 13
      1.8.2.1 Enumeration types 13
    1.8.3 OCL collections 14
      1.8.3.1 Collections types 14
      1.8.3.2 Collections literals 15
      1.8.3.3 Collection iterators 15
      1.8.3.4 Select and reject operations 16
      1.8.3.5 The collect operation 16
List of Figures

1.1 For each policy we have an associated policyAccount. .................................................. 10
1.2 Jobs and ClientAccount as association classes .............................................................. 11
1.3 An example of a UML enumeration type ......................................................................... 14
1.4 OCL collection types & services they provide ................................................................. 15
List of Tables

1.1 Basic OCL data types and operations ...................................................... 13
1.2 Services provides by OclMessage ......................................................... 21
Chapter 1

The Object Constraint Language (OCL)

1.1 Introduction to the OCL

1.1.1 What is the OCL?

The OCL is a constraint specification language which enables one to specify constraints on an object model, usually a UML model.

In particular, OCL provides a formal specification language for defining design by contract constraints, including:

- pre- and post-conditions on services, and
- invariance constraints on objects.

1.1.2 History of the OCL

The OCL was developed in 1995 by a team led by Jos Warmer and Steve Cook from IBM. This development was part of a business modeling initiative done by IBM. This is prior to UML, but IBM submitted a request to include the OCL as the official constraint specification language for UML and this was adopted by the OMG with UML version 1.1.

Since then the UML team has used the OCL extensively to rigorize UML’s meta-model by formalizing UML rules and constraints. OCL is also a core component of the MDA technologies which support the OMG’s vision of model driven development (MDD).

1.1.3 Core OCL features

- **OCL is a pure specification language** The statements are purely declarative and do not affect the subject, i.e. the state and behaviour of an object model are unaffected by the presence of OCL constraints. The constraints effectively provide additional information about the constraints which the model should adhere to.

- **Supports querying across an object graph** The OCL provides a syntax for querying across an object graph.

- **OCL is typed** You cannot mix types like comparing a string to a floating point number.

1.1.4 Applicability to the OCL

- **Specify pre- and post-conditions on services** Both the pre-conditions for a service (i.e. the conditions under which the service provider may refuse the service without breaking the contract) and the post-conditions for a service (i.e. the conditions with must hold after the service has been provided -- thus for the success scenarios of the service) can be formally specified in OCL. For example, there may be a pre-condition for the debit service of an account that the amount must be less or equal to the balance on the account and a post-conditions specifying that the state of the balance after the transaction as well as the fact that the transaction must have been entered into the transaction history.
The Object Constraint Language

Specify general invariance constraints/business rules Invariance constraints are symmetry rules which must always apply. If, for a subject, any of the invariance rules are violated, then the subject is in an inconsistent state, i.e in a state of failure. For example, there may be an invariance rule that for an account, the sum of all credits minus the sum of all debits must always be equal to the balance of the account.

Specify guard conditions Guard conditions are commonly used to specify the conditions under which alternative paths in a process are followed. OCL enables one to formally specify these guard conditions in a machine processable way.

Specify derivation rules for attributes and return values The attributes for a service request must be derived from the information currently available. OCL provides a formal mechanism to do this. In addition, OCL can be used to specify constraints on the return value across externally visible information.

Specify type invariants for stereotypes Stereotypes are UML’s extension mechanism. They enable one to refine basic UML concepts into specialized concepts which can then be used within a UML model. As such there must be additional constraints which apply to the specialized concept. It is these constraints which distinguish it from the more generic concepts. OCL can be used to formally specify these constraints.

Specify target or target sets for messages and actions OCL enables one to specify constraints around messages which would have to be sent in the context of realizing a service.

Specify which messages would have to be sent in the context of realizing a service OCL enables one to specify the messages which would have to be sent upon realizing a service as well as constraints on the message contents itself.

Specify initial state of objects OCL constraints can be used to specify initial values for attributes.

Specify derivation rules for derived attributes UML supports derived attributes whose value is fully specified (constrained) by the values of other attributes. OCL enables one to formally specify the derivation rules used to derive the value of derived attributes.

Specify initial state of objects OCL constraints can be used to specify initial values for attributes.

Specify initial state of objects OCL constraints can be used to specify initial values for attributes.

1.1.4.1 Specifying services contracts

OCL is commonly used to specify services contracts by adding pre- and post-conditions for each service.

1.1.4.2 Automatic test generation

Defining OCL constraints for a model enables one to automatically generate both, contract tests and integrity tests.

1.1.4.2.1 Contract tests

Specifying formal services contracts in OCL enables one to automatically create contract tests for contracts. A class would typically be tested against all tests across all contracts it claims to realize.

1.1.4.2.2 Integrity tests

Invariance constraints can be used to generate integrity (e.g. system integrity) tests. These could be used at any stage to validate the system integrity and hence to identify possible security breaches or system errors.
1.2 The context of an OCL expression

The context definition of an OCL expression defines the model entity for which the OCL expression is defined. Usually the context of an OCL expression is either

- an interface,
- a class, or
- a service/operation.

Note that the context has a type, the *contextual type*.

Any OCL expression starts with a context declaration. For example

```
context Account ...
```

or

```
context Account::debit(amount:Money) ...
```

1.2.1 Self

Even though an OCL expression is defined for a type, it is ultimately evaluated for an instance of the model entity, i.e.

- an instance of any class realizing an interface/contract,
- an instance of an implementation class, or
- an instance of a service realization.

The keyword *self* is used to refer to a model element instance.

For example, if the model type is an account, then the contextual instance, *self*, will be an instance of that Account class.

```
context Account
    inv: self.accountNo > 0
```

**Note**

The *inv:* specifies an invariance constraint, i.e. that the constraint must always hold for any instance of the context.

As is the case for many programming languages, the *self* is by default implied. The above OCL is thus equivalent to

```
context Account
    inv: accountNo > 0
```

1.3 OCL comments

Comments are specified in OCL by starting some text with two dashed lines. Everything from the start of the double-dash is ignored by any OCL interpreter and treated purely as a comment.

For example context Account -- This is not a bank account but an account a client has at the local grocer *inv:* -- the start of an invariance constraint expression which applied to the context.
1.4 Model navigation

One of the features of OCL is that it enables one to navigate an object model. The navigation operator is a dot.

Consider, for example, the simple object graph depicted in the UML class diagram of Figure 1.1. If we would like to specify that the balance of any account which plays the role of a policy account must always be greater or equal to zero, then we can do this with the following OCL constraint:

```ocl
class Policy

context Policy
inv: self.policyAccount.balance > 0
```

Note
OCL also supports navigation across one-to-many and many-to-many associations using OCL collections.

1.4.1 Navigating to and from association classes

The name of the association class converted to lower case is used to navigate, in OCL, from an associated object to the association class. Consider, for example, the UML class diagram shown in Figure 1.1: For each policy we have an associated policyAccount.
If we wanted to navigate from a person to his or her salary we could simply use the following OCL query

```
person.job.salary
```

If the association is a one-to-many or many-to-many association, then the result of the query may be a collection of objects. For example, the following query will yield a collection of client accounts

```
person.clientAccount
```

### 1.5 The OCL if expression

OCL supports standard if-then-else expressions for defining conditional constraints. For example, below we specify an invariance constraint for an account which states that the interest rate should be zero if the balance is less than 100 and that it should be 5% otherwise:

```
context Account
inv:
   if self.balance < 1000
      self.interestRate = 0
   else
      self.interestRate = 0.05
```
1.6 Let sub-expressions

The let sub-expression enables one to assign a value to a variable which can be repetitively used within the expression. For example, if we wanted to use use the power output of an engine of a car repetitively within an if-then-else expression, we could assign the result of the appropriate query to a variable

```ocl
let
  power : Real = self.engine.powerOutput
in
  if self.engine.energySource = EnergySource::battery
    power < 200
  else
    power > 30 and power < 1000
```

Note

Variables declared via let sub-expressions can only be used within the outer expression. If one would like to define variables which can be used across expressions, one needs to use definition, i.e. def expressions.

1.7 Def expressions

Variables declared via let sub-expressions are only available within the expression containing the let declaration. If one would like to define variables which is available across multiples expressions, one needs to use def expressions.

Definition expressions can not only be used to define variable, but also to define functions. Both, variables defined for a context and functions defined for the context are globally available as if the context had the corresponding attributes and services. They are accessed like any other attribute or operation.

1.7.1 Defining variables via def expressions

To define a variable via a definition expression, one uses the following syntax:

```ocl
def: <variableName> : <VariableType> = <InitializationExpression>
```

For example, if we would like to define a variable which represents the total available funds for an account, we could use the following OCL definition expression:

```ocl
context Account
  def: availableFunds : Real = self.balance - self.minimumBalance
```

The availableFunds attribute is now available to the UML model like any other attribute of the class and can be used across various OCL expressions.

1.7.2 Defining re-usable functions

OCL definitions can also be used to define reusable functions for classes. For example, if we wanted to define a reusable function which returns true if an organization has a particular employee who has a specified identity number, we could use the following OCL definition expression:

```ocl
context Organization
  def: hasEmployee(idNumber: String): Boolean = self.employees.exists(idNo=idNumber)
```
1.8 Data types available in OCL

Within OCL expressions one can make use of

- a set of built-in primitive types,
- any datatypes available within the object model, and
- OCL collections.

the equals (==) and not-equals (<>) operators are available across all datatypes. In addition

1.8.1 Primitive OCL types and operations

OCL defines 4 primitive types and a set of basic operators for each. These are shown in Table ??.

<table>
<thead>
<tr>
<th>Type</th>
<th>Domain of values</th>
<th>Available operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>true or false</td>
<td>and, or, xor, not, implies, if-then-else</td>
</tr>
<tr>
<td>Integer</td>
<td>any positive or negative whole (integral) number.</td>
<td>*, -, +, /, abs(), div(i), mod(i), max(i), min(i)</td>
</tr>
<tr>
<td>Real</td>
<td>any positive or negative floating point number</td>
<td>*, -, +, /, &lt;, &gt;, &lt;=, &gt;=, floor(), round(), min(r), max(r)</td>
</tr>
<tr>
<td>String</td>
<td>Any character sequence</td>
<td>concat(), size(), substring(), toInteger(), toReal()</td>
</tr>
</tbody>
</table>

Table 1.1: Basic OCL data types and operations

1.8.2 Data types from the UML model

Any data type/classifier defined in the associated UML model can be used when defining OCL constraints on that model. We can thus directly use user defined types like Account or Person within OCL expressions.

1.8.2.1 Enumeration types

One can also directly access UML types defined in the UML model and their enumeration values. For example, if we wanted to specify that a poerson who plays the role of a clown in a circus must always be in a happy state, we can use the following OCL expression

```ocl
class Circus
context Circus inv: circus.circus.emotionalState = EmotionalState::happy
```
1.8.3 OCL collections

In order to specify constraints across

- collection attributes,
- one-to-many and many-to-many associations, and
- services which return collections,

OCL defines a set of collection types with collection operators which can be used when specifying constraints.

---

**Note**
The OCL collection has nothing to do with the collection chosen in the implementation. It is purely a virtual collection providing a mechanism to specify constraints.

---

**Note**
From version 2.0 onwards, OCL supports collections of collections, i.e. collections where the elements themselves are collections. These nested collections can be flattened out using the `flatten` operation available for all collections.

---

1.8.3.1 Collections types

The OCL defines the five collection types shown in Figure ??.

- **Collection** is the base type which defines a large number of operations available across all collections.
- **Bag** is an unordered collection which only adds the functionality of querying the intersection of the bag with another collection, i.e. the common elements across the bag and a target collection.
- **Sequence** defines an ordered collection which provides random access to its elements.
- **Set** enforces the uniqueness of elements, i.e. that each element is contained only once in the set. Like Bag, it also provides the facility of querying the intersection between itself and another collection. In addition one can also query the symmetric difference between two collections which are all those elements which are contained in either of the two collections, but not in both.
- **OrderedSet** is a subclass of Set which enforces not only uniqueness of its elements, but also maintains its elements in ordered sequence. They are generally used for the many-end of one-to-many and many-to-many associations which have been assigned an `{ordered}` constrained in the UML model. They can provide a subset view onto a subset of the elements.
1.8.3.2 Collections literals

One can also construct an OCL collection and populate it with elements within an OCL expression. For example, if we wanted to construct a set of names, we could do it via

```ocl
Set('Amie', 'Bernhard', 'Thabo', 'Jay', 'Tandie', 'Thomas', 'Wilma', 'Zandile')
```

or a sequence of numbers

```ocl
Sequence(1, 2, 3, 4, 5)
```

or a sequence declaration based on an interval specification

```ocl
Sequence(1..5)
```

1.8.3.3 Collection iterators

OCL provides an iterate operation which enables one to iterate across a collection of elements, applying an expression to each element and accumulating the expression results within an accumulator.

For example, if we wanted to query the total outstanding debt across a collection of accounts, then we could iterate as follows:

```ocl
accounts->iterate(account : Account; totalDebt : Real = 0 | if account.balance < 0 then
```
Similarly, if we wanted to accumulate all woman employees in a collection, we could do it as follows

```ocl
employees->iterate (employee : Employee; women : Set(Employee) = Set() | if employee.sex = Sex::female then women.including(employee) else women endif)
```

**Note**

Often one can use select, reject, collect or sum operations instead of iterate.

### 1.8.3.4 Select and reject operations

Select and reject operations are used to select a sub-collection of objects which satisfy some constraint, i.e. for which some boolean expression either holds (select) or does not hold (reject). They are convenience operations which can be used instead of iterating directly across OCL collections.

For example,

```ocl
accounts.select(a:Account | a.balance > 10000)
```

returns a collection of all accounts in the accounts collection for which the balance property is larger than 10 000.

For convenience reasons, one need not specify the element type

```ocl
accounts.select(a | a.balance > 10000)
```

or the iterator variable

```ocl
accounts.select(balance>10000)
```

explicitly. All three statements will yield the same result, i.e. can have the element type default to the type of the what is contained in the collection by default, the expression in select and reject is applied to the elements of the collection.

The reject statement is an inverse statement which returns a collection of all those accounts which do not satisfy the expression, i.e. in our example all those accounts which have a balance of 10 000 or less.

```ocl
accounts.reject(balance>10000)
```

### 1.8.3.5 The collect operation

The collect operation can be used to collect the results of an operation which is applied across the elements of some collection. For example, if we wanted to create a collection of all product names which are manufactured by a particular manufacturer, then we could use the following expression

```ocl
manufacturer.products->collect(name)->asSet()
```

We can also explicitly use the iteration variable by reformulating the query as follows

```ocl
manufacturer.products->collect(product:Product | product.name)->asSet()
```

This expression is equivalent to the previous one.
1.8.3.6 forall and exists operations

The **forall** and **exists** operations are two of the convenience operations which can be used instead of iterating directly over an OCL collection.

For example, we could specify that for all savings accounts, the balance must be non-negative:

```ocl
class Bank
inv: self.savingsAccounts.forall(balance >= 0)
```

In a similar way we could specify that there must exist an employee within the organization who has the role of central executive officer:

```ocl
class Organization
inv: self.employees.exists(role = EmployeeRoles::centralExecutiveOfficer)
```

1.8.4 Tuples

Tuples can be used to construct new composite data structures. They are assembled as a collection of name-value pairs. The value can be of any valid OCL type.

Consider, for example, the case that we want to define a query whose result contains all

```ocl
class Bank
def: statistics: Set(TupleType(accountId:String, balance:Real, client:Person, otherAccounts: Set(Account))) =
    self.accounts->select(balance>100000)->collect(acc:Account |
    Tuple {
        accountId = acc.id,
        balance = acc.balance,
        client = acc.accountHolder,
        otherAccounts = acc.accountHolder.accounts->excluding(acc)
    } ->

Note
Tuples enable one to obtain a table-based query-result view which is in some ways similar as what is often created with SQL database queries.

1.8.5 Accessing overridden properties

In exceptional cases, one may need to specify constraints on a property of the superclass which is overridden in the subclass. In order to access such an overridden property of the superclass, one uses the **oclAsType** operator.

For example, assume we have a **Mercedes_C200** class which has as specialization an **Alpine_Merc_C200** class which is a modification of the latter. Mercedes may be the manufacturer of the vanilla Mercedes C200 class, but the Alpine Mercedes C200 is manufactured by Alpine. We may want to constrain that the manufacturer of the the superclass, **Mercedes_C200**, is Mercedes:

```ocl
class Alpine_Merc_C200
inv: self.oclAsType(Mercedes_C200).manufacturer = 'Mercedes'
```
1.9 OCL object properties

The OCL language defines a number of properties which are available for all objects. These enable one to query

- the type of the object,
- whether an object is in a particular state,
- in the context of a post-condition for an operation, whether a particular object has been created within that operation,
- and whether two types are equal.

1.9.1 The oclIsTypeOf and oclIsKindOf properties

Using the oclIsTypeOf property, we can query whether an object is a direct instance of a particular class. Thus

```ocl
Context ChequeAccount
inv: self.oclIsTypeOf(ChequeAccount) -- evaluates to true
inv: self.oclIsTypeOf(Account) -- evaluates to false
inv: self.oclIsTypeOf(Apple) -- evaluates to false
```

In many cases one would like to check type compatibility polymorphically. In general an instance of a cheque account is indeed also an instance of an account. To perform this more conventional type checking, one uses the oclIsKindOf property:

```ocl
Context ChequeAccount
inv: self.oclIsKindOf(ChequeAccount) -- evaluates to true
inv: self.oclIsKindOf(Account) -- evaluates to true
inv: self.oclIsKindOf(Apple) -- evaluates to false
```

1.9.2 The oclIsInState property

One can also query whether an object is in a particular state. This often is useful when specifying the post-conditions for a service, i.e. when specifying a constraint which specifies that, after having completed a particular service or operation, the object needs to be in a particular state.

For example, in order to query whether an account is in an overdrawn state, one can use the expression

```ocl
account.oclIsInState(AccountStates::overdrawn)
```

1.9.2.1 Accessing nested states

If the state machine for an object specifies nested states, then one may want to query whether an object is in a particular nested state. For example, one can use the following query to assess whether a mobile phone is in a receivingCall state, which is a sub-state of the connected state:

```ocl
account.oclIsInState(CommStates::receivingCall)
```

1.9.3 The oclIsNew property

At times, one needs to assess whether a service has created a new instance of a particular class. In such a case, one performs a oclIsNew query in the post-condition of that service.

For example, the rentVehicle service of a vehicle rentals company may return a Rental object which contains all the information about the rental, like the vehicle details, the rental period, the insurance cover and so on. If we would like to ascertain that the returned rental is indeed a new rental which was created in the rentVehicle service, we would query the oclIsNew property of the returned rental context VehicleRentals::rentVehicle(:VehicleRentalRequest):Rental post: result.oclIsNew
1.10 Accessing class members

Class members are accessed directly form the class name. For example, if we wanted to constrain the `numInstances` class member of a `Thread` class, to below 500, then we could use the expression

```
context Thread
    inv: Thread.numInstances < 500
```

1.10.1 Pre-defined class members

OCL supports a number of pre-defined class members. One of these is the `allInstances()` service which is available for any OCL or model class. We could use this service to specify the uniqueness of account numbers as follows:

```
context Account
    inv: Account.allInstances()-&gt;forAll(acc1, acc2 | acc1 &lt;&gt; acc2 implies acc1.accountNo &lt;&gt; acc2.accountNo
```

1.11 Constraint types

OCL supports the three types of constraints introduced with design by contract

- **invariance constraints** which represent symmetry rules which define constraints on the valid state of objects,
- **pre-conditions** which are used to specify under which conditions a service provider may refuse to honour a service request without breaking a services contract, and
- **post-conditions** which are the conditions which must hold after having provided the service.

1.11.1 Invariance constraints

Invariance constraint are used to specify symmetry rules which constrain the valid states of objects. If any of the invariance constraints for an object are violated, then the object is said to be in an invalid or *inconsistent* state.

**Note**

The consistency requirement of the ACID criteria of a transaction specifies that if a system was in a consistent state prior to the start of the transaction, then it must, at the end of a transaction, be again in a consistent state, i.e. that none of the invariance constraints of the objects is violated.

An invariance constraint is specified by a `inv:` prefix on the constraint expression. The context of an invariance constraint is usually a class.

For example, we may have a constraint that the sum of all transaction amounts of a transaction history of an account must always add up to the current balance of the account. This could be specified as follows:

```
context Account
    inv: self.transactions-&gt;collect(amount)-&gt;sum() = self.balance
```

We can assign a name to the invariance constraint. For example, we could name the consistency constraint on the account balance `consistentBalance`:

```
context Account
    inv consistentBalance: self.transactions-&gt;collect(amount)-&gt;sum() = self.balance
```

Invariance constraints are often used to specify *business rules*. like that the balance of a savings account may never be negative

```
context SavingsAccount
    inv: self.balance &gt;= 0
```
1.11.2 Preconditions

Preconditions are assigned to services. They specify those conditions under which the service provider may refuse the service without violating the services contract.

Preconditions are specified in OCL by a `pre:` prefix on the expression. The context of the constraint must be a service which is specified either for a class or for an interface.

For example, we may want to specify that an account may refuse the debit service if the debit amount is such that the resultant balance would fall below the minimum balance for the account. This precondition could be specified as follows:

```ocl
class Account{
  operation debit(amount: Real): TransactionConfirmation
    pre: amount <= self.balance - self.minimumBalance
}
```

**Note**
Each precondition is usually mapped onto an exception class. The exception does not signal an error in the service provider -- it only specifies that the service provider chose or was not able to provide the requested service because a particular precondition was not met.

We can give the precondition a name. For example, we could name the above precondition `sufficientFunds`:

```ocl
class Account{
  operation debit(amount: Real): TransactionConfirmation
    pre sufficientFunds: amount <= self.balance - self.minimumBalance
}
```

1.11.3 Postconditions

Postconditions are assigned to services. They specify those conditions which must hold after the service has been provided, i.e. after any success scenario for the service.

A post condition is specified using a `post:` prefix in front of the constraint expression. Postconditions are assigned to services of classes and interfaces.

1.11.3.1 Accessing the return value of a service

Consider a `RootSolver` which can be used to find a root of a function, i.e. a value from its domain where the function value is zero. Assume you want to specify a postcondition which states that the return value of a `rootSolver` service should be such that if the function value is evaluated at the returned approximation to the root, it is zero to within some `eps`. This can be achieved with the following expression:

```ocl
class RootSolver{
  operation findRoot(f: Function, initialGuess: Real, eps: Real): Real
    post: f.value(result).abs() < eps
}
```

1.11.3.2 The `@pre` postfix

It is common to have to specify postconditions of a service which require the value of certain properties prior to the service having been requested. For example, if one would want to specify that the balance of an account after debiting should be equal to the balance prior to debiting minus the withdrawal amount (i.e. that any transaction fees must be raised in the context of a separate transaction), and that the number of transactions in the transaction history must have increased by one, then one could use the following postcondition:

```ocl
class Account{
  operation debit(amount: Real): TransactionConfirmation
    post: balance = balance@pre - amount
    post: self.transactions->size() = self.transactions@pre->size() + 1
}
```
1.11.3.3 Specifying that a communication must have taken place

One may want to specify, as postcondition, that a particular message (e.g. a synchronous or asynchronous service request message) must have been sent within the context of realizing a service request.

To this end, OCL supports a hasSent operator, \(^\wedge\). For example, if one wants to specify that within the context of processing a claim, Claims must have requested the claim coverage from policies, we could add the following postcondition:

```ocl
context Claims::processClaim(claim : Claim): ClaimSettlementReport
  post: self.policies^determineClaimCoverage(claim : Claim)
```

This example assumes that claims has an association to policies and that the claim will contain the information about the policy against which the claim is made. It also specifies that the claim as received from the user is passed to policies when requesting the claim coverage.

At times we may want to just specify a constraint that a particular type of message was sent without constraining the message parameters. In such cases one can use question marks for the free parameter values. For example, if we want to specify that, in the context of processing a claim, a claim valuation request must have been sent to an assessor without constraining the actual request parameter, then we could use the following OCL expression:

```ocl
context Claims::processClaim(claim : Claim): ClaimSettlementReport
  post: self.assessor^assessLosses( ? : AssessmentRequest)
```

1.11.3.4 OCL messages

In order to specify constraints on OCL messages, OCL introduces the OclMessage type with a number of properties and services. An OCL message encapsulates a specific message sent in the context of an interaction.

1.11.3.4.1 Services offered by OclMessage

OclMessage provides the following operations:

<table>
<thead>
<tr>
<th>Service</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>isSignalSent()</td>
<td>true if the message has been sent, false otherwise</td>
</tr>
<tr>
<td>isOperationCall()</td>
<td>true if the message is a synchronous request which has a return, false otherwise</td>
</tr>
<tr>
<td>hasReturned()</td>
<td>true if a return for an operational call has been received, false otherwise</td>
</tr>
<tr>
<td>result()</td>
<td>provides a handle to the returned object</td>
</tr>
</tbody>
</table>

Table 1.2: Services provided by OclMessage

1.11.3.4.2 Obtaining access to exchanged messages

In order to obtain access to messages exchanged within an iteration, OCL introduces the message operator, \(^\wedge\). In the case of referring to a specific interaction, the return value of this operator is either a single OclMessage. In the case where multiple messages satisfying the message operator may have been exchanged, the operator returns a sequence of messages.

We can, for example, obtain access to the claim coverage request message and its associated return via:

```ocl
context Claims::processClaim(claim : Claim): ClaimSettlementReport
  post: let
    message : OclMessage = self.policies^determineClaimCoverage(claim : Claim)
  in
    message.hasReturned()
    and
    message.result().claimId = claim.id
```
or, in the case where multiple messages may have been sent

```oclnl
context Claims::processClaim(claim : Claim): ClaimSettlementReport
  post:
    let
      messages : Sequence(OclMessage) = self.assessor^assessLosses( ? : ← AssessmentRequest)
    in
      messengered.forAll
```

1.11.3.4.3 Accessing message parameters

The message parameters are directly accessed by their names in the formal service declaration.

-->
Chapter 2

Index

A
  allInstances
    OCL class service, 19

B
  business
    rule, 19

C
  constraint, see invariance constraint pre-condition post-condition

D
  design by contract, 7

I
  invariance constraint, 7, 19

M
  MDA, see Model Driven Architecture
  MDD, see Model Driven Development
  Model Driven Architecture, 7
  Model Driven Development, 7

N
  navigation operator, see Object Constraint Language navigation operator

O
  Object Constraint Language, 7
    accessing class members, 19
    collect operation, 16
    collection iterators, 15
    collection literals, 15
    collection types, 14
    collections, 14
    comments, 9
    constraint types, 19
    context, 9
    context declaration, 9
    contextual instance, 9
    contextual type, 9
    data types available in, 13
    data types from UML model, 13
    def expression, 12
    definition of, 7
    enumeration types, 13
    exists operation, 17
    forAll operation, 17
    if expression, 11
    invariance constraint, 19
    let sub-expression, 12
    model navigation, 10
    navigation operator, 10
    OCL object properties, 18
    postconditions, 20
    preconditions, 20
    primitive types, 13
    reject operation, 16
    select operation, 16
    self, 9
    tuples, 17
  Object Management Group, 7
  OCL, see Object Constraint Language
  oclAsType, 17
  oclIsInState, 18
  oclIsKindOf, 18
  oclIsNew, 18
  oclIsTypeOf, 18
  OMG, see Object Management Group

P
  post-condition, 19
  postcondition, 7
  pre-condition, 19
  precondition, 7
  preconditions
    in OCL, 20
  postconditions
    in OCL, 20

U
  UML
    meta-model, 7