

# Operation Schemas and OCL

Version SEVEN, October 2001

Alfred Strohmeier and Shane Sendall

*Swiss Federal Institute of Technology in Lausanne  
Software Engineering Laboratory  
1015 Lausanne EPFL, Switzerland*

*email: {Alfred.Strohmeier, Shane.Sendall}@epfl.ch*

The goal of this document is to describe the syntax and usage of Operation Schemas and also to show how we use UML's Object Constraint Language (OCL) in the schemas.

This document is composed of two parts. Part 1 introduces the syntax and usage of the standard concepts of Operation Schemas and relates them to UML in section 1; section 2 provides a summary of UML's Object Constraint Language (OCL) and highlights some enhancements that were made to it for the purposes of writing Operation Schemas; and section 3 provides examples of Operation Schemas and OCL. Part 2 covers some of the more sophisticated concepts of Operation Schemas, in particularly those for specifying concurrent operations, blocking calls with return values, and exception handling.

## Part 1. Standard Concepts

### 1. Operation Schemas

An Operation Schema declaratively describes the effect of a system operation on a conceptual state representation of the system and by events sent to the outside world. It describes the assumed initial state by a precondition, and the required change in system state after the execution of the operation by a postcondition, written in UML's OCL formalism. The change of state resulting from an operation's execution is described in terms of objects, attributes and association links, which conform to the constraints imposed by the analysis class model of the respective system. The postcondition of the system operation can assert that objects are created, attribute values are changed, association links are added or removed, and certain events are sent to outside actors. The association links between objects act like a network, guaranteeing that one can navigate to any state information that is used by an operation. Note that objects manipulated by Operation Schemas do not have behavior, they are purely domain concepts that have more similarities to entities in an Entity-Relationship (ER) diagram than to a "Design" Class Diagram.

#### 1.1 Naming Conventions

Note: It is not quite clear if OCL is case sensitive or not, because the specification uses a naming convention, but the OCL grammar does not enforce any conventions, the rules being the same for typename and name.

The following conventions should be used for better readability, and even enforced because otherwise ambiguities might result, e.g. between the class Account, and the

implicitly defined rolename account in the composition association between the Bank and its Account(s).

- Capital first letter for datatypes, e.g. String, classes, e.g., Person, and associations, e.g. Owns;

- Lowercase first letter for data values and data constants, e.g. true, name: String; objects, e.g., john: Person; roles, e.g. wife; collections, e.g. allEmployees, company.employee; attributes, e.g. john.birthdate; and functions (methods), e.g. isUnique ().

A comment is written following two successive dashes (minus signs), which signify that the rest of the line is a comment (this comes directly from OCL):

-- this is a comment

## 1.2 Declarations

The following subsection provides an BNF-like description of declarations in Operation Schemas with examples.

### 1.2.1 Values, Objects, Classes and Associations

TypeExpression ::=

```
ClassName
| "Collection" "("ClassName ")"
| "Set" "("ClassName ")"
| "Bag" "("ClassName ")"
| "Sequence" "("ClassName ")"
| DatatypeName
| "Collection" "("DatatypeName ")"
| "Set" "("DatatypeName ")"
| "Bag" "("DatatypeName ")"
| "Sequence" "("DatatypeName ")"
```

Examples:

```
String
Person
Set (Person)
```

ObjectDeclaration ::=

```
Name ("," Name)* ":" ClassName
```

ObjectCollectionDeclaration ::=

```
Name ("," Name)* ":" "Collection" "("ClassName ")"
| "Set" "("ClassName ")"
| "Bag" "("ClassName ")"
| "Sequence" "("ClassName ")"
```

DataDeclaration ::=

```
Name ("," Name)* ":" DatatypeName
```

DataCollectionDeclaration ::=

```
Name ("," Name)* ":" Collection "(" DatatypeName ")"
| "Set" "("DatatypeName ")"
| "Bag" "("DatatypeName ")"
```

| "Sequence" "("DatatypeName ")"

EntityDeclaration ::=

Name ("," Name)\* ":" TypeExpression

-- Is any of the previous

Examples

john: Person

p: Person -- an object of the class Person

amount: Money -- an amount of the type Money

participants: Collection (Person) -- a collection of objects of the class Person.

AssociationName ::=

Name | -- in the case of a named association or an association class

[Name] "(" [RoleName ":" ] ClassName ("," [RoleName ":" ] ClassName)+ ")"

-- in alphabetical order of RoleNames; if there is no RoleName in the

-- definition of the association, ClassName is used for sorting the list.

Examples

Owns

(owner: Person, property: Car)

(Car, Person)

Owns (owner: Person, property: Car)

## 1.3 Events

According to UML, events are model elements. They have parameters. Events are specifications of observable occurrences. An event is quite similar to a class, and its occurrences are similar to objects of the class; they have unique identity and by-reference semantics. Also, the parameters of an event are similar to the attributes of a class (in graphical representation and meaning).

When there is no possible ambiguity, we will sometimes say event with the meaning of event occurrence.

UML uses events in statecharts. Following UML: "An event is received when it is placed on the event queue of its target (the system or an actor in our case). An event is dispatched when it is dequeued from the event queue and delivered to the state machine for processing (e.g. the SIP of the system in our approach). At this point, it is referred to as the current event. Finally, it is consumed when event processing is completed. A consumed event is no longer available for processing." Our concept of an event is in agreement with this definition.

Operation schemas specify not only the changes to the system state, but also the *system events* that are output by the operation. Communications between the system and actors are through event occurrence delivery. In our approach, we distinguish input from output events. Input events are incoming to the system and trigger system operations. Usually, their names are the same. The parameters of the input event are the parameters of the system operation. Output event occurrences are outgoing from the system and are delivered to a destination actor.

We propose to interpret a system event occurrence as:

- having by-reference semantics;

- having unique identity;
- having an implicit reference to its sender, referred to by sender;
- being reliably and instantaneously delivered (no latency).

There are several kinds of system events, which can be thought of as either a specialization of SignalEvent or CallEvent in UML, depending on whether the event is asynchronous or synchronous. We will distinguish three kinds of system event types that we call Event, Exception, CallWithReturn, respectively stereotyped <<event>>, <<exception>>, and <<callwithreturn>>. They all have a compartment containing the parameters and another compartment that contains the sender, an actor, of the event. A CallWithReturn event has in addition to the sender (i.e., in the same compartment) the result, which references the event carrying the returned result.

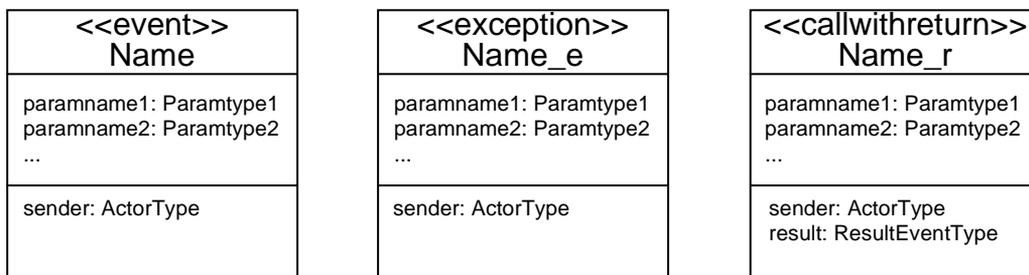
An Event occurrence instigates an asynchronous communication; it usually triggers the execution of an operation. An Exception occurrence signals an unusual outcome to the receiver, e.g., an overdraft of an account. A CallWithReturn occurrence triggers the synchronous execution of an operation that returns a result to the sender. The result is modelled by an Event occurrence.

Often we use the term event with the meaning of any of the above kinds or even occurrences.

We use a naming convention to differentiate the different kinds of events: suffix “\_e” for an Exception, and suffix “\_r” for CallWithReturn. The reason for this naming convention is to help specifiers visually differentiate between different kinds of events.

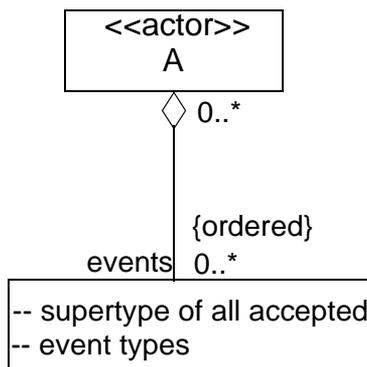
All parameters of an event sent by an operation must be defined.

Figure 1 shows a general example of each kind of event.



**Figure 1: Events defined in graphical form**

Each actor has an event queue—just as the system has an event queue. If the actor is able to deal with occurrences of a given event (type), then it is possible to state that an event was placed in the actor’s (input) event queue as a result of an operation. We do not make mention of output queues, because we suppose that delivery is reliable, etc.



**Figure 2: All actors have an event queue denoted by the association end role name “events”**

The textual syntax for declaring event types is as follows (an alternative to the graphical form):

```

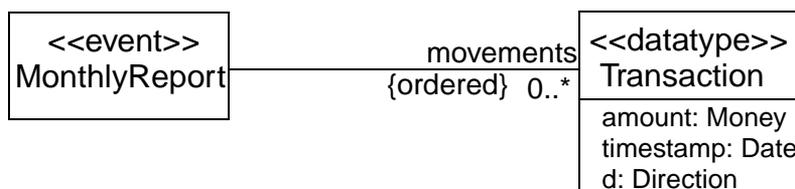
EventDeclaration ::=
  EventName (“ ParameterList “) [: ResultEventType]
ParameterList ::=
  Parameter (“Parameter”)*
Parameter ::=
  EntityDeclaration
ResultEventType ::= EventName
  
```

Note that the “sender” is not explicitly declared. Indeed, any actor is allowed as an actual, and it cannot be constrained to a subtype. It is set implicitly when an event is delivered to the receiver.

Examples of event declarations:

```

InsufficientFunds_e ();
DispenseCash (amount: Money);
DebitReport (amount: Money, timestamp: Date);
CreditReport (amount: Money, timestamp: Date);
  type Direction is enum {debit, credit};
  type Transaction is record
    amount: Money; timestamp: Date; d: Direction;
  end record;
Report (t: Transaction); -- a debit or a credit
MonthlyReport (movements: Sequence (Transaction));
-- its graphical representation is shown in figure 3
  
```



**Figure 3: A graphical example of an “event declaration”**

### 1.3.1 Event Occurrences and Event Collections

It is possible to declare event occurrences and collections of event occurrences:

```

EventOccurrenceDeclaration ::=
    Name ("," Name)*: EventName
EventCollectionDeclaration ::=
    Name ("," Name)* ":" ( Collection "("EventName")"
                        | Set "("EventName")"
                        | Bag "("EventName")"
                        | Sequence "("EventName")" )

```

All event occurrences have to be created within the execution of the operation. Therefore we will not state explicitly that they were created (to the contrary of newly created objects).

A Name declared in an EventOccurrenceDeclaration has by-reference semantics.

Note: Otherwise, bags of events would not make sense.

Events in a Sequence are ordered.

Events in a Set or Bag are not ordered.

If a Collection is specified, ordering is not dealt with during analysis, but deferred to design.

Examples:

```

denied: InsufficientFunds_e
reports: Collection (MonthlyReport)

```

### 1.3.2 Connecting Classes and Actors

In order to send an event to an actor, it is often necessary to identify the actor from its representation in the system. We propose to define an association stereotype <<id>> that can be used, and only used, to connect classes belonging to the system with external actors. The implication for later development activities is that some mechanism for identifying the actor(s) must be realized (e.g., a name server)



Figure 4: An <<id>> association links a class belonging to the system to an actor type

## 1.4 Reserved Words and Predefined Identifiers

We list below the reserved words and predefined identifiers of Operation Schemas and OCL.

### Reserved Words in the OCL

**context, inv, let, in, pre, post, def, package, endpackage** -- unused in Operation Schemas  
**if, then, else, endif**  
**and, or, xor, implies, not**

### Reserved Words added to the OCL

**elsif**  
**all**

### Reserved Words specific to Schema, Predicate and Function Syntax

**Operation, Description, Notes, Use Cases, Scope,**

**Declares, Sends, Pre, Post, Exceptions** --clauses  
**Type, Occurrence, Order** -- subclauses  
**Is, Throws, HandledBy**  
**Predicate, Function, Body, Predicate, Aliases**

### **Predefined Identifiers of the OCL**

false, true  
self, result

Collection, Set, Bag, and Sequence

### **Added Predefined Identifiers**

sender is mainly used in the **Post** clause.

## **1.5 Operation Schema**

In this subsection we provide a description of the different clauses of an Operation Schema.

### **1.5.1 Schema**

**Operation:** The entity that services the operation (aka the name of the system), followed by the name of the operation and parameter list, and the type of the returned event, if any.

**Description:** A concise natural language description of the purpose and effects of the operation.

**Notes:** This clause provides additional comments.

**Use Cases:** This clause provides cross-references to related use case(s).

**Scope:** All classes and associations from the *class model* of the system defining the *name space* of the operation. (Note that it would be possible to have a tool generate this clause automatically from the contents of the other clauses.)

**Declares:** This clause provides two kinds of declarations: aliasing, and naming.

Aliases are name substitutions that override precedence rules, i.e., treated as an atom, and not just as a macro expansion.

A name declaration designates an object to be “created” by the operation, i.e. the post-condition will state `oclIsNew()` for it. Each name declares a distinct object.

**Sends:** This clause contains three subclauses: **Type**, **Occurrence**, and **Order**. **Type** declares all the events that are output by the operation together with their destinations, i.e. the receiving actor classes. **Occurrence** declares event occurrences and collections of event occurrences. **Order** defines the constraints on the order of events output by the operation.

**Pre:** The condition that must be met for the postcondition to be guaranteed. It is a boolean expression written in OCL, standing for a predicate.

**Post:** The condition that will be met after the execution of the operation. It is a boolean expression written in OCL, standing for a predicate.

Schema ::=

“**Operation**” “:”

SystemClassName “:” OperationName (“ [ParameterList] ”) [“:” EventType] “;”

```

[ "Description" ":" Text "," ]
[ "Notes" ":" Text "," ]
[ "Use Cases" ":" UseCaseList ]
[ "Scope" ":" NameList ]
[ "Declares" ":" ItemList ]
[ "Sends" ":" [ "Type" ":" ActorWithEventsList [ "Occurrence:" EventOccurrenceList ]
                [ "Order:" OrderingConstraint ] ] ]
[ "Pre" ":" Condition "," ]
[ "Post" ":" Condition "," ]

```

## Operation Clause

**"Operation"** ":"

```
SystemClassName "::" OperationName (" [ParameterList] ") [ ":" EventType ] ":"
```

The SystemClassName defines the context of the schema, i.e., self always refers to a system object, otherwise stated, an instance of this class.

The OperationName together with the ParameterList follows, more or less, the syntax of an event declaration, since it corresponds to an input event sent to the system.

All parameters in ParameterList are of mode *in*. This does not mean that the state of an object which is a parameter cannot be changed.

The EventType defines the type of the returned event.

Examples:

```

Bank :: withdrawCash (acc: Account, amount: Money);
-- acc is an object and amount is a data value.
GolfProShop::getNumGolfClubs (): NumOfItems;
-- getNumGolfClubs returns an event NumOfItems, that has as a parameter
-- the number of golf clubs that the GolfProShop has in stock.

```

## Use Cases Clause

**"Use Cases"** ":" UseCaseList

UseCaseList ::= (Name ",")\*

## Scope Clause

**"Scope"** ":" NameList

NameList ::= (NameListElement ",")\*

NameListElement ::= ClassName  
| AssociationName

NameList is a list of class names, and association names.

Examples of NameListElements:

```

Person
Owns
(owner: Person, property: Car)

```

## Declares Clause

**"Declares"** ":" ItemList

ItemList ::= (Item ",")\*

Item ::= ObjectDeclaration

| Alias

Alias ::= EntityDeclaration **Is** Expression

Everything declared in the **Declares** clause is local to the schema.

An Item can be an alias or a name declaration.

Aliases are name substitutions that override precedence rules, i.e., treated as an atom, and not just as a macro expansion.

A name declaration designates an object to be “created” by the operation, i.e. the post-condition will state `ocllsNew()` for it. Each name declares a distinct object.

Examples

```
acc1, acc2: Account;
p: Person Is self.person -> any (p | p.firstName = “arthur”);
john: Person;
x1: Integer Is p.account.balance;
x2: Integer Is acc1.balance;
```

If a `ClassName` is used in an `ObjectDeclaration` or `EntityDeclaration`, it must be in the scope of the schema, i.e. declared in the **Scope** clause. Similarly, if a `ClassName` is used in an `Expression`, it must be in the scope of the schema. Also, if a property, e.g. a role-name, is used in an `Expression`, the “owner” of the property must be in scope, e.g. the association with the rolename.

Note that when writing in the postcondition `acc1.ocllsNew` and `acc2.ocllsNew`, the meaning is that two different objects were created.

## Sends Clause

**“Sends”** ::=“

```
[“Type” ::= ActorWithEventsList
 [ “Occurrence:” EventOccurrenceList ]
 [ “Order:” OrderingConstraint ] ]
```

## Type Subclause

ActorWithEventsList ::= (ActorWithEvents “,;”)\*

ActorWithEvents ::= ActorClassName “::” “{” (EventName [ **“Throws”** ExceptionEvents ] “,;”)\* “}”

ExceptionEvents ::= EventName (“, ” EventName)\*

ActorWithEvents shows which kinds of events are sent to a given actor class. This liaison between an actor class and a set of event classes is specific to an operation. In another operation it may be different.

ExceptionEvents defines the exceptions that are thrown by the operation triggered by the event preceding the **Throws** keyword. The **Throws** keyword and its usage is described in section 4 (part 2).

Example

**Type:**

```
ATM :: {DispenseCash; InsufficientFunds_e; Report;};
Bank::{Withdraw_r Throws InsufficientFunds_e;};
Clerk :: {AccountNumber;};
```

## Occurrence Subclause

It is possible to declare event occurrences and collections of event occurrences. All output events are created as an effect of executing the operation, i.e., they did not exist before the execution of the operation.

EventOccurrenceList := (EventOccurrenceItem “,”)\*

EventOccurrenceItem := EventOccurrenceDeclaration | EventCollectionDeclaration

Examples

```
denied: InsufficientFunds_e;  
receipt: Report; dispense: DispenseCash;  
reports: Collection (MonthlyReport);
```

## Order Subclause

Constraints on the temporal delivery order of (output) event occurrences is defined by this subclause. Note that ordering of input events is defined by the System Interface Protocol.

OrderingConstraint := “<” (EventOccurrenceName (“,” EventOccurrenceName)\* “>” “,”

EventOccurrenceName := Name

Examples

```
<receipt, dispense>;  
-- The receipt is delivered before the cash (so the client does not forget it!)
```

## Pre and Post Clauses

“**Pre**” “:” Condition “;”

“**Post**” “:” Condition “;”

Condition ::= BooleanExpression (“&” BooleanExpression)\*

Condition is a boolean expression, the meaning of the “commercial and” sign & being that of a logical **and**. Expressions are written in OCL. Even though it is also possible to write expressions in natural language, if natural language is used, the expressive power should be limited to OCL one’s.

Note that the **Pre** and **Post** clauses refer only to entities declared in the **Declares** clause, to parameters of the operation, to self, to sender, to result, or to entities navigated to from any of the previous ones.

Only an alias declared in a **Declares** clause or a Condition in a **Post** clause (or a parameterized predicate -- described later) can use the @pre suffix. Only a Condition in a **Post** clause can use the result keyword (used to denote the reply to a synchronous call).

Note that the Post clause may make use of parameterized predicates and functions (described below) but it is not possible to refer to another Operation Schema within the postcondition of a schema. When such situations are deemed necessary, parameterized predicates should be used to describe the commonality and then “called on” in the respective schemas.

An empty precondition can be expressed by the constant condition true.

If the precondition is true, the operation terminates and the postcondition is true after the execution of the operation.

The pre- and postconditions assertions constitute the contract model of the operation. If the precondition is met, then the operation will meet the postcondition, but if the precondition is not met, then nothing is guaranteed, i.e., the effect of the operation is undefined.

## 1.6 Parameterized Predicates

A parameterized predicate is used in **Pre** and **Post** clauses to better support readability of schemas and to allow one to reuse commonly recurring predicates. They are inspired from the Catalysis approach. They are used to encapsulate a ‘piece’ of the pre- or postcondition and therefore they can use the suffix ‘@pre’ (in the case that it is used in the postcondition); they evaluate to true or false.

At definition, the scope of a parameterized predicate is the schema (i.e. the names declared in the **Scope**, **Declares** and **Sends** clauses) where it is supposed to be used; it can then be used in all schemas having this scope or a wider one.

Parameterized predicates can have a declaration clause for aliases, which are local to the predicate; this clause is called **Aliases**. When a predicate is referred to in a postcondition, it must be possible to resolve all references within the current context.

Parameterized predicates are declared e.g. in a UML package for constraints.

```
"Predicate" ":" PredicateName "(" [ParameterList] ")" ";"  
    [ "Aliases" ":" (Alias ";")* ]  
    "Body" ":" Condition ";"
```

## 1.7 Functions

A function may be used to encapsulate a computation. They do not have any side effects, i.e. they are pure mathematical functions, and to the contrary of a system operation, they do not change the system state. Functions may be used as a reuse mechanism for commonly recurring calculations.

We separate the function declaration (its signature) from the function definition. In that way, they can be used as a placeholder when the need for the function is known, but its realization is deferred to a later stage of development, i.e. design or implementation. For example, we might know that we have to determine the best suited lift to service a particular request, which can be expressed by a function, but the choice of the algorithm is deferred until design:

**Function:** bestSuitedCabin (options: Set (Cabin), requestedFlr: Floor): Cabin;

-- A function that hides the algorithm for choosing the best suited cabin to service a request

Functions can also be used when OCL is not suitable for expressing the algorithm, e.g. in the case of numeric computations. Functions are therefore a way to escape the limited expressive power of OCL when necessary. However, we admit that such a facility can be misused.

Functions can be referred to anywhere, in contrast to parameterized predicates, whose use is limited to pre- and postconditions. They can refer to the model elements of the analysis class model. If a function does not refer to any model elements, then it is a universal function, e.g. the sine function, and it is possible to refer to it “anywhere”. Functions can have a declaration clause for aliases, which are local to the function; this

clause is called **Aliases**. When referring to a function, it must be possible to resolve all references within the current context.

```

Function ":" FunctionName "(" ([ParameterList] ")" [":" TypeName] ","
Function Body ":" FunctionName "(" ("["ParameterList"])" [":" TypeName] ","
    [ Aliases ":" (Alias ";")* ]
    Post ":" Condition ","

```

Note that Condition must define result.

Also, note recursive function definitions are possible: according to the OCL specification "The right-hand side of this definition may refer to the function being defined (i.e., the definition may be recursive) as long as the recursion is not infinite."

## 1.8 Aliases

An alias consists of three parts: name, type, and substitution expression. The type part is provided simply as a specifier check, i.e., helps early detection of specifier errors (type mismatches). Substitution expressions are those that are on the right-hand side of "Is" in a declaration (**Declares** or **Aliases** clause); they define what is substituted for the name given on the left-hand side.

Given figure 5, we show some declarations that make use of aliases.

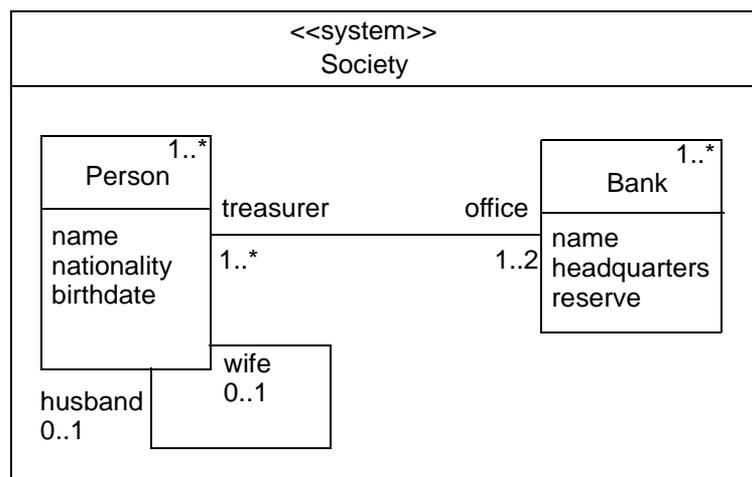


Figure 5: A class model of a society that is used to show substitution expressions

### Example 1

```
john: Person Is self.person -> any (p: Person | p.name = "John");
```

The above alias means that whenever john is used in the pre- and postcondition it is substituted for an object of the class Person corresponding to the person named "John". Note that the substitution expression defines a set with the any collection operation applied to it; if the set has one or more elements then it results in any element of the set; otherwise, if the set is empty, the expression is undefined (according to OCL). Thus, care needs to be taken that there is always a resulting element.

### Example 2

```
ubs_treasurers: Set (Person) Is ubs.treasurer;
```

In this case, the set is always defined, but it might be empty.

### Example 3

allTreasurers: Set (Person) **Is** self.bank.treasurer -> asSet ();

In OCL, the rule is that when we navigate through more than one association with multiplicity greater than 1, we end up with a Bag. In order to eliminate the duplicates, we convert the bag to a set.

### Example 4

myHusband: Person **Is** anne.husband;

The navigation is over an association that is 0..1 at the husband role end. In this case, the alias has the meaning of a Person object. When evaluating the alias, if anne does not have a husband then the expression is undefined.

## 1.8.1 Type Matching Rules

If the type of the left-hand side of an alias or relational expression is a collection, the right-hand side must be a collection of the same subtype (Set, Bag, or Sequence), and the types of the elements belonging to the collection must be conforming types (same type, or a subtype).

If the type of the left-hand side of an alias or relational expression is an object reference, the right-hand side must be a conforming type (same type, or a subtype).

If the type matching rules are not met, the schema is incorrect, and its meaning undefined.

## 1.8.2 Undefined Expressions

If the type matching rules are satisfied, the schema is erroneous in any of the following cases:

- There is a contradiction; e.g. the postcondition states both:  $x = 2$  **and**  $x = 3$ .
- There is an expression that operates on a set, the state of the set should not be empty, but it is empty. This case also includes dereferencing after navigation to an association end of multiplicity 0..1 when there is no object having the role. Note that it is allowed to apply an operation to an empty set as long as it is defined for empty sets, e.g. size or union.

## 1.9 Additional Features of Operation Schemas

In this subsection we discuss additional features that can be used in Operation Schemas to make writing assertions less laborious, etc.

### 1.9.1 Aggregates

We propose to use an Ada-like aggregate notation for denoting the attribute values of an object and the values of a composite datatype, i.e. a record type. Aggregates can also be used for denoting the actual parameters of an event.

Aggregate ::= (“ AggregatItem ( “,” AggregatItem)\* “)”

AggregatItem ::= name “=>” value

The name is the name of an object attribute, datatype field or event parameter.

An aggregate must always be complete, i.e. values must be defined for all attributes, fields, or parameters.

To avoid any confusion, or in order to resolve ambiguities, it is possible to qualify an aggregate by its type:

```
QualifiedAggregate ::= Name "" Aggregate
-- Name must be the name of a datatype, a class, or an event.
```

Examples

```
Bank'(name => "ANZ", headquarters => "Auckland", reserve => 50E12)
-- the attributes of a bank object (the composite value of the object should not be
-- confused with the object itself)
DispenseCash'(amount => request)
-- the list of actual parameters of an event occurrence DispenseCash (not to be confused
-- with the event occurrence itself).
Person'(name => "Josh Kronfield",
        birthdate => Date'(year => 1971, month => 6, day => 20),
        nationality => "New Zealander")
-- aggregate notation for a person object
-- it also uses an aggregate for the birthdate (of datatype Date)
```

## 1.9.2 Denotation of an Object or an Event by an Aggregate

We define a special shorthand that makes it possible to match objects and events directly with aggregates. The shorthand is defined for each object/event type. It uses the name of the type and takes a composite value as parameter, resulting in a reference to the corresponding object/event in the system that has the matching composite value:

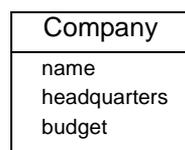
```
ShorthandForObjectOrEvent ::= ClassName Aggregate
```

For example, given figure 6:

```
Company ((name => "Microsoft", headquarters => "Richmond", budget => 50.0E9))
```

is a shorthand for the expression:

```
Company.allInstances->any (c | c.all =
        (name => "Microsoft", headquarters => "Richmond", budget => 50.0E9))
```



**Figure 6: Company class in UML class notation**

The precondition of the any collection operator states that the supplied collection, i.e., the expression on the left-hand side, must have at least one element satisfying the expression. This means that if there are no objects matched, then the shorthand is undefined. Thus, the specifier should ensure that the corresponding object exists for all valid system states. The shorthand notation is particularly useful for denoting event sending.

The shorthand allows one to write concise and, we believe, intuitive expressions in postconditions, e.g.:

```
aPerson = Person ((name => "Josh Kronfield",
        birthdate => (year => 1971, month => 6, day => 20),
        nationality => "New Zealander"))
```

-- aPerson references the object that represents Josh.

The next expression results in true if Microsoft is a member of the local companies in the region, region:

```
region.localCompanies->includes (  
  Company ((name => "Microsoft", headquarters => "Richmond", budget => 50.0E9)))
```

Due to the “by-reference” semantics of objects and events, we denote their composite value by the property all. Thus we can write expressions like the following:

```
aPerson.all = (name => "Josh Kronfield",  
  birthdate => (year => 1971, month => 6, day => 20),  
  nationality => "New Zealander")
```

which evaluates to true if the object referenced to by aPerson has the corresponding value attributes. The above expression is equivalent to:

```
aPerson.name = "Josh Kronfield" and  
aPerson.nationality = "New Zealander" and  
aPerson.birthdate = (year => 1971, month => 6, day => 20)
```

### 1.9.3 Consistency of Associations

An association link can only link existing objects; it is therefore a well known consistency constraint for class models that when an object is removed from the system state all association links connected to it have to be removed too. Although it would be possible to explicitly state all association links that must be destroyed, this is quite cumbersome in the presence of numerous associations. Therefore we propose the association consistency assumption.

Assumption 1: Removal of an object from the system implies implicitly that all association links in the system that included the destroyed object are destroyed, in addition.

### 1.9.4 Frame Assumption

The frame of a specification is the list of all variables that can be changed by the operation, which in our model is always a subset of all objects and all association links that are part of the system state. The postcondition of a specification describes all the changes to the frame variables, and since the specification is declarative, the postcondition must also state all the frame variables that stay unchanged. The reason is simple: if the unchanged frame variables are left unmentioned, they are free to be given any value and the result will still conform to the specification.

Formal approaches such as Z, VDM, Larch, etc. explicitly state what happens to each one of these frame variables—even for those variables that stay unchanged. This approach soon becomes cumbersome to write and error-prone, particularly for specifications that have complex case distinctions (where the complete frame is the combination of all the variables read/changed in each different case). One approach that avoids this extra work is to imply a “... and nothing else changes” rule when dealing with these types of declarative specifications. This means that the specification implies that the frame variables are changed according to the postcondition with the unmentioned frame variables being left unchanged. This approach reduces the size of the specification, thus increases its readability, and makes the activity of writing specifications less error prone. We therefore adhere to this convention.

However, there is a slight problem with this assumption in the case of implicit removal—a consequence of the association consistency assumption. For an example, let us reconsider an extract of the postcondition for an operation of an elevator control system.

```
self.request->excludesAll (reqsToStopFor)
```

If we strictly apply the frame assumption “... and nothing else changes”, as a result the associations (not shown) `HasIntRequest`, `HasExtRequest`, `HasCurrentRequest`, and `HasTargetFloor` would stay unchanged which would lead to an inconsistent system state. At least three of the associations have to be changed, and will be changed following our implicit consistency of associations convention stated in section 1.9.3.

Also, we need to cover two more cases: what happens to attributes of frame objects that are not mentioned by the postcondition, and what happens to attributes of newly “created” objects that are not mentioned in the postcondition.

We propose the following amended frame assumption.

Assumption 2: No frame variables (including, if a variable denotes an object, the object attributes) are changed with the execution of the operation other than those that are explicitly mentioned to be changed by the postcondition, the associations that are implicitly modified as defined by the association consistency assumption, and the objects, and their attributes, that are new to the system state as a consequence of the operation.

This assumption forces all attributes of objects that are not mentioned to keep the same value with the exception of new objects added to the system state; in this case, we provide three possible interpretations: 1) attributes of new objects that are not mentioned in the postcondition can take any value, 2) the unmentioned attributes get predefined default values, or 3) the specification is incorrect if values are not defined for all attributes. The last interpretation gives more of a prescriptive flavor and one could probably expand this to also prohibit specifications where attribute values are constrained to a range rather than a precise value, e.g., `acc.num > 0` would not be allowed in the description of an effect.

## 1.9.5 Minimum Set Principle

The minimum set principle together with the frame assumption (section 1.9.4) ensure that there are no unwanted additions to the post-state of the system after executing the operation.

In addition, it makes the task of writing Operation Schemas less laborious. Indeed, the minimum set principle allows one to state the change of contents of sets—the observed change to the set when comparing the set before the execution of the operation with the set after the execution—in an incremental fashion. Moreover, the post-state of the set is defined in terms of the collective combination of the `includes` and `excludes` operations applied to the pre-state set. Thus, we make the assumption that no changes to the pre-state of the set are made unless explicitly stated via `includes` and `excludes` operations in the postcondition.

For example, in the case of inclusion the following postcondition

```
Post: setX -> includes (x1);
```

is equivalent to:

```
Post: setX = setX@pre -> including (x1);
```

and

**Post:** setX -> includes (x1) &  
setX -> includes (x2);

is equivalent to:

**Post:** setX = setX@pre -> union ( Set {x1, x2});

Similarly for excludes.

The combination of includes and excludes in a post clause proves to be even more straightforward with this approach. For example, the following postcondition,

**Post:** setX -> includes (x1) &  
setX -> excludes (x2);

is more convenient than the traditional approach of stating explicitly the contents of the post-state set:

**Post:** setX = setX@pre -> including(x1) -> excluding (x2);

## Minimum Set Principle Applied to Collections

The minimum set principle can also be applied to collections in general.

However, when it is applied to a bag, duplicates are not accounted for, e.g.,

**Pre:** bagX = Bag {};  
**Post:** bagX->includes (x1) **and**  
bagX->includes (x1);

is equivalent to:

**Post:** bagX->includes (x1);

An additional constraint is therefore required for the bag to contain two x1 elements, e.g., bagX->count(x1) = 2.

We take the opportunity, even though this has nothing to do with the minimum set principle, to insist that ordering of conditions does not suffice to order elements in a sequence, e.g.,

**Pre:** seqX = Sequence {};  
**Post:** seqX->includes (x1) **and**  
seqX->includes (x2);

does not mean that x1 precedes x2 in seqX. The correct postcondition would be

seqX = seqX@pre->union (Seq{x1, x2})

which, using the minimum set principle, can be simplified to:

seqX->union (Seq{x1, x2})

## 1.9.6 Incremental Plus and Minus

We can use an idea similar to the minimum set principle for numeric types. We use the operators, “+=” and “-”. Thus, the value of a numeric entity in the post-state is equivalent to the value in the pre-state plus all the right-hand sides of all += operators used in the postcondition that refer to the numeric entity, and minus all the right-hand sides of all -= operators that refer to the numeric entity. For example:

aPerson.salary += 5 **and**  
aPerson.salary -= 4

is equivalent to:

aPerson.salary = aPerson.salary@pre + 1

However, care needs to be taken when the incremental style is mixed with the other styles.

```
Post: ...
obj.x += 5 and           -- line one
obj.x -= 4 and           -- line two
obj.x = obj.x@pre + 2 and -- line three
obj.x = 2                 -- line four
```

The above example is an erroneous specification: line three is in contradiction with the result defined by the incremental plus and minus, and line four would require that obj.x@pre be either 0 or 1 depending on whether line three was brought into agreement with line one and two or vice versa.

Unfortunately, the facility cannot be extended to more complex expressions (e.g. multiplication) because it relies on the commutativity of additions and subtractions.

### 1.9.7 Dealing with Events in Postconditions

Events that are output by the system during the execution of an operation are specified in the respective schema by stating:

- the type of the event and the destination actor type;
- the condition(s) under which the event occurrence is sent;
- the actual parameters of the event occurrence;
- the destination actor instance(s);
- any ordering constraints that the event occurrence may have relative to other events output by the same operation.

To assert that an event was sent to some actor, the event is stated to be an element of the actor's event queue.

#### Examples

The examples could be part of a withdrawal operation performed on an ATM of a bank.

Event declarations:

```
InsufficientFunds_e (); DispenseCash (amount: Money);
Report (t: Transaction) -- a debit or a credit
  type Direction is enum {debit, credit};
  type Transaction is record
    amount: Money; timestamp: Date; d: Direction;
  end record;
```

Possible contents of the Sends clause:

```
Type: ATM :: {InsufficientFunds_e, DispenseCash, Report};
Occurrence: denied: InsufficientFunds_e, dispense: DispenseCash, receipt: Report;
```

We can make assertions about parameters:

```
receipt.t.amount = 1000 &
receipt.t.timestamp = (year => 2000, month => 2, day =>14) &
```

```
receipt.t.d = Direction::debit
```

Instead of making assertions about all the individual actual parameters, it is better to use an aggregate:

```
receipt.all = ( t => (amount => 1000,  
                    timestamp => (year => 2000, month => 2, day =>14),  
                    d => Direction::debit))
```

To assert that an event is sent to an actor, we will state that as an effect of the operation, it becomes part of the destination actor's event queue.

### Examples

```
dispense = DispenseCash ((amount => request))  
-- dispense is an event occurrence of the type DispenseCash  
-- that matches the aggregate (i.e. the formal parameter amount has the value of request)  
caller.events -> includes (dispense)  
-- the event queue of caller includes a reference to the event occurrence dispense  
receipt = Report (( t => Transaction'  
                  (amount => 1000,  
                    timestamp => (year => 2000, month => 2, day =>14),  
                    d => Direction::debit)))  
-- receipt is a new event that matches the aggregate (i.e., the formal parameter  
-- t has the value of the transaction given by the inner aggregate)  
caller.events -> includes ( receipt)  
-- the event queue of the caller includes receipt, an event occurrence of the type Report.
```

### Shorthand for “Sending” Events

In addition to explicitly writing that an event is placed on the target actor's event queue, we propose a shorthand that we have found in practice to be more intuitive to users and writers. It has the following form, where actorX denotes any identifiable actor and eventOccurrenceX denotes any appropriate event occurrence:

```
actorX.sent (eventOccurrenceX)
```

and is equivalent to or syntactic sugar for:

```
actorX.events->includes (eventOccurrenceX)
```

We emphasize that *sent* is just a shorthand and should not be confused with a property of the actor.

### Second Set of Examples

The following examples refer to the class diagram shown in figure 7.

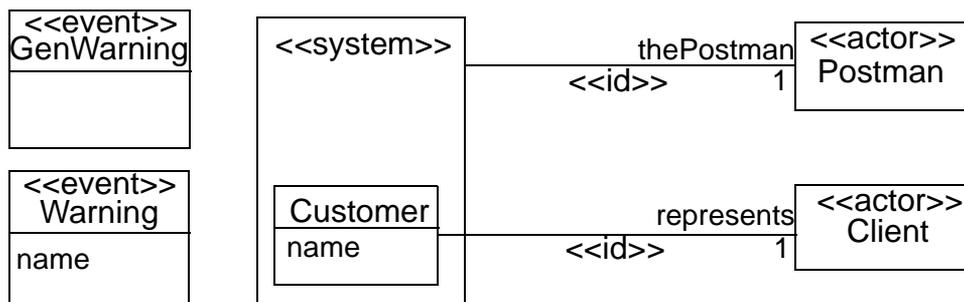


Figure 7: Sending warning events

Extracts of Operation Schemas follow, hence self refers to the system instance.

### **Example: Sending one event to a group of actors (multicast)**

**Declares:** allClients : Set (Client) **Is** self.customer.represents;

**Sends:**

**Type:** Client :: {GenWarning};

**Occurrence:** warn: GenWarning;

**Post:**

```
warn = GenWarning (()) & -- second pair of brackets refer to empty aggregate (no
                        -- event parameters)
allClients -> forall (c | c.sent (warn));
```

### **Example: Sending many events to a single (postman) actor**

**Declares:** allCusts : Set (Customer) **Is** self.customer;  
postman: Postman **Is** self.thePostman;

**Sends:**

**Type:** Client {Warning};

**Occurrence:** warningMessages: Collection (Warning);

**Post:**

```
allCusts -> forall (c | warningMessages -> includes (Warning ((name => c.name)))) &
-- assumes minimum set principle and collection is constructed from empty
postman.events -> includesAll (warningMessages);
```

### **Example: Sending many events to many actors (one-to-one correspondence)**

**Declares:**

allCusts : Set (Customer) **Is** self.customer;

**Sends:**

**Type:** Client {Warning};

**Occurrence:** warningMessages: Collection (Warning);

**Post:**

```
allCusts -> forall (c | warningMessages->includes (Warning ((name => c.name)))) &
allCusts -> forall (c | c.represents.sent->includes (warningMessages -> any(name = c.name)));
```

### **Interpretation:**

Remember that an actor's event queue holds references to events. Note therefore that each actor dequeues the reference to the event rather than the event itself. An event only ceases to exist once it is no longer referenced by any queue.

## **2. Extracts from the Object Constraint Language Specification (v1.4) and its Use in Operation Schemas**

### **2.1 Connection with the UML Metamodel (6.3)**

Each expression is written in the context of an instance of a specific type. The reserved word self refers to this instance.

In an Operation Schema, self can be viewed as the instance of the type which owns the operation as a feature.

Examples of context declarations and constraints

**context** Company

**inv:** self.numberOfEmployees > 50

**context** c: Company

**inv:** c.numberOfEmployees > 50

**context** Person :: income (d: Date): Integer

**post:** result = 500

## 2.2 Types (6.4)

Boolean, Integer, Real, String are predefined types.

Collection (abstract type), Set, Bag, Sequence are basic types as well.

All types from the UML model can be used in OCL.

## 2.3 OCL Operators

The precedence order for the operations, starting with highest precedence, in OCL is:

- @pre
- dot and arrow operations: ‘.’ and ‘->’
- unary **not** and unary minus ‘-’
- ‘\*’ and ‘/’
- ‘+’ and binary ‘-’
- **if ... then ... else ... endif**
- ‘<’, ‘>’, ‘<=’, ‘>=’
- ‘=’, ‘<>’
- **and**, **or** and **xor**
- **implies**

Parentheses ‘(’ and ‘)’ can be used to change precedence.

## 2.4 Properties of Objects (6.5)

OCL expressions can refer to types, classes, interfaces, and datatypes, and to all properties of objects.

A property is one of the following:

- an Attribute
- an AssociationEnd
- an Operation or a Method with isQuery being true (the rule guarantees that OCL expressions have no side-effects)

Properties can be accessed via the dot operator.

Examples:

- a.balance accesses the attribute balance of the account a,
- a.isVIP () accesses the isVIP method of account a,
- if the context is Person, self.age accesses the age of a Person instance.

Starting from a specific object, we can navigate an association to refer to other objects. To do so, we use the opposite association-end:

```
object.rolename
```

The value of this expression is the set of objects on the other side of the association. When a rolename is missing, the name of the type, starting with a lowercase character is used as the rolename.

Example:

- `a.owner` accesses the association role, and results in the collection of objects on the other end of the association.

## 2.5 Preconditions and Postconditions

In a postcondition, an expression can refer to two values for each property of an object:

- the value of the property at the start of the operation,
- the value of the property upon completion of the operation.

The value of a property in a precondition (and all other clauses except the postcondition) is the one at the start of the operation.

The value of a property in a postcondition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, one has to postfix the property name with the suffix `@pre`.

Example:

```
self.age = self.age@pre + 1
```

## 2.6 Collections

Collection is the abstract supertype of all collection types in OCL: Set, Bag and Sequence. A Set is the mathematical set. It does not contain duplicate elements. A Bag is like a set, but may contain duplicates (i.e., the same element may be in a bag twice or more). A Sequence is like a bag in which the elements are ordered. Both bags and sets have no order defined on them.

A collection can be specified by literals.

Examples

```
Set {1, 2, 5, 88}
Set {'apple', 'orange', 'strawberry'}
Sequence {1, 3, 45, 2, 3}
Sequence {1..10}
Bag {1, 3, 45, 2, 3}
Bag {1, 2, 3, 3, 45}
-- two identical bags.
```

It is possible to get a collection by navigation.

Example: from a company object, navigate the employee role, which will yield a collection of Person instances:

```
company.employee
```

Single navigation results in a Set, combined navigations in a Bag, and navigation over associations adorned with `{ordered}` results in a Sequence. Therefore, the collection

types play an important role in OCL expressions. The type `Collection` is a predefined abstract type in OCL. The `Collection` type defines a large number of predefined operations to enable the OCL expression author (the modeler) to manipulate collections. Consistent with the definition of OCL as an expression language, collection operations never change collections; `isQuery` is always true. They may result in a collection, but rather than changing the original collection they project the result into a new one. `Collection` is an abstract type, with the concrete collection types as its subtypes.

Operations on collections may result in new collections.

Example:

```
set1 -> union (set2) -- the union of set1 and set2
```

A property of the collection is accessed by using an arrow `->` followed by the name of the property.

Example: the number of employers of a person:

```
self.employer -> size ()
```

If the multiplicity of the association-end is 0..1 then the expression results in an object. However, such an expression can be treated like it results in a set as well. For example

```
person.wife -> notEmpty () implies person.wife.sex = Sex::female
```

## 2.7 Expressions

Properties can be combined to make more complicated expressions. An important rule is that an OCL expression always evaluates to an object or value of a specific type, or to a collection of objects of a specific type. After obtaining a result, one can always apply another property to get a new result value. Therefore, each OCL expression can be read and evaluated left-to-right.

Examples

Married people are of age  $\geq 18$ :

```
context Person inv:
```

```
self.wife -> notEmpty () implies self.wife.age  $\geq 18$  and  
self.husband -> notEmpty () implies self.husband.age  $\geq 18$ 
```

A company has at most 50 employees:

```
context Company inv:
```

```
self.employee -> size ()  $\leq 50$ 
```

Navigation from an object to an association class link:

```
person.job
```

The above expression evaluates to all the jobs a person has with the companies that are his/her employer (see figure 8 in section 3.1). Note that the name of the association class, in lowercase, is used to show the role for navigation.

Navigation from an association class link to an object

```
context Job
```

```
self.employer.numberOfEmployees  
self.employee.age
```

## 2.8 Predefined Properties on All Objects

`oclIsTypeOf (t: OclType): Boolean -- direct type`

Examples:

```
person.oclIsTypeOf (Person) -- true
person.oclIsTypeOf (Company) -- false
```

`oclIsKindOf (t: OclType): Boolean -- t is a direct type or one of the supertypes`

Example:

```
checking.oclIsKindOf (Account) -- true
```

## 2.9 Features of Classes

`Person.allInstances`

Denotes all instances of the class `Person` when the expression is evaluated. It is of type `Set (Person)`.

The use of `allInstances` is discouraged. It is considered better to navigate from some context object.

Example

```
context Person inv:
  Person.allInstances ->
    forAll (p1, p2 | p1 <> p2 implies p1.name <> p2.name)
```

`oclIsNew ()`: Boolean

```
john.oclIsNew () -- e.g.
```

The operation can be applied to any object; OCL says it is a property of any object. It results in true if the object is newly created. It only makes sense in a postcondition.

### 2.9.1 Enhancing the `oclIsNew` property

We propose to overload the operation with a version having a single parameter providing the attribute values for the object:

`oclIsNew (value)`: Boolean

Results in true if the object is newly created with the attributes having the values indicated by *value*.

For example, asserting that a new object has the same value as another can be described as simply as:

```
anotherCompany.oclIsNew (company.all)
```

It is also possible to use an aggregate, which denotes the actual attribute values. For example, a postcondition could state:

```
newTreasurer.oclIsNew ((name => "Josh Kronfield",
                        birthdate => (year => 1971, month => 6, day => 20),
                        nationality => "New Zealander"))
```

which means that the object, `newTreasurer`, became a new element of the system state with the execution of the operation, and all its value attributes, i.e., `name`, `birthdate`, and

nationality, were given the values, "Josh Kronfield", (1971, 6, 20), and "New Zealander", respectively.

The above expression is directly equivalent to the following one:

```
newTreasurer.ocllsNew and  
newTreasurer.name = "Josh Kronfield" and  
newTreasurer.nationality = "New Zealander" and  
newTreasurer.birthdate = (year => 1971, month => 6, day => 20)
```

The proposed notation ensures that all attributes of a newly created object were constrained to the given values, and none of them were forgotten.

In addition, we propose to overload the operation for collections with a single parameter, having the meaning that the specified number of objects are newly created in the collection:

```
ocllsNew (size: Integer): Boolean
```

Results in true if the collection contains size number of newly created objects (and only size number of objects). It only makes sense in a postcondition. Its meaning is as follows:

```
context c: Collection (T)  
pre:    c -> isEmpty ();  
post:  c -> forall (e | e.ocllsNew () ) and  
        c -> size () = size;
```

Example:

```
self.family -> occllNew (4) -- e.g. the new family Adam, Eve, Abel, and Cain.
```

## 2.10 Collection Operations (6.6, 6.8), extracts only

Examples are given according to figure 8.

Selecting in a collection: select

Example:

```
company.employee -> select (p | p.age > 50)  
-- p iterates through the employees of the company  
-- selecting all those who are older than 50.
```

Excluding from a collection: reject

Example:

```
company.employee -> reject (p | p.isMarried)  
-- the collection of not married employees.
```

Collecting the values yielded by an expression: collect

Example

```
company.employee -> collect (p | p.birthDate)  
-- birthdates are collected, not persons. The result is a bag.
```

Projection on an attribute can be used as a shorthand for collect:

Example

```
company.employee.birthdate
```

Conversion of a bag to a set, by eliminating duplicates: asSet

Example

```
company.employee -> collect (p | p.birthDate) -> asSet ()
```

All objects meet the expression (universal quantification): forAll

Example

```
company.employee -> forAll (p | p.salary > 100000)
-- true if all employees have a great salary,
-- assuming salary is an attribute of person.
```

Existence of an object that meets the expression (existential quantification): exists

Example

```
company.employee -> exists (p | p.salary > 100000)
-- true if at least one employee has a great salary.
```

Asserting that an element is a member of a collection or not: includes and excludes

Examples

```
company.employee -> includes (john);
company.employee -> excludes (bill);
```

Existence of a single object: one

Example

```
company.employee -> one (p | p.salary > 100000)
-- true if and only if one employee has a great salary.
```

Selecting an object: any

Example

```
company.employee -> any (true)
-- results in any element of the collection that satisfies the expression.
```

### 3. Examples

This section provides some examples of Operation Schemas and OCL.

#### 3.1 Example with an Association Class

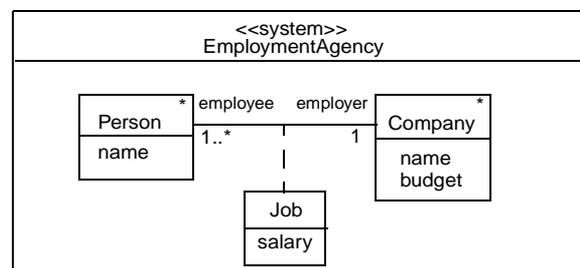


Figure 8: Class model for Employment Agency System

##### 3.1.1 Navigation Expressions

context EmploymentAgency

Get all names of persons in the system (without repeated names):

```
self.person -> collect (name) -> asSet ()
```

-- or more simply:  
-- self.person.name->asSet ()

*Find all companies with a budget greater than 10'000'000 dollars:*

self.company -> select (c: Company | c.budget > 1.0E7)

*Find all the people working for NASA with the name John that have a salary greater than 50'000 dollars:*

self.person -> select (p: Person |  
p.name = "John" **and**  
p.employer.name = "NASA" **and**  
p.job.salary > 5.0E4)

### 3.1.2 Operation Schema

#### Operation:

EmploymentAgency::jobFilled (pName: Name, cName: Name, amount: Money);

**Description:** Creates a job for a given person and company, where company has a budget smaller than equal to 10 million;

**Scope:** Person; Company; Job;

#### Declares:

worker: Person **Is** self.person -> any (p: Person | p.name = pName);  
comp: Company **Is** self.company -> any (c: Company | c.name = cName);  
bigCompanies: Set (Company) **Is**  
self.company -> select (c: Company | c.budget > 1.0E7);  
researchJob: Job;

#### Pre:

bigCompanies -> excludes (comp);

#### Post:

researchJob.oclIsNew ((salary => amount)) &  
researchJob.employee = worker &  
researchJob.employer = comp;

### 3.1.3 Object Creation

#### Declares:

epfl: Company **Is** self.company -> any (c | c.name = "epfl");  
epflEmployees: Set (Employee) **Is** epfl.employee;

#### Post:

(epflEmployees.account).oclIsNew (epflEmployees -> size ()) &  
epflEmployees -> forall (e | epflEmployees.account -> exists (a | a.owner = e));

## 3.2 Example of a Class Model with a Constraint

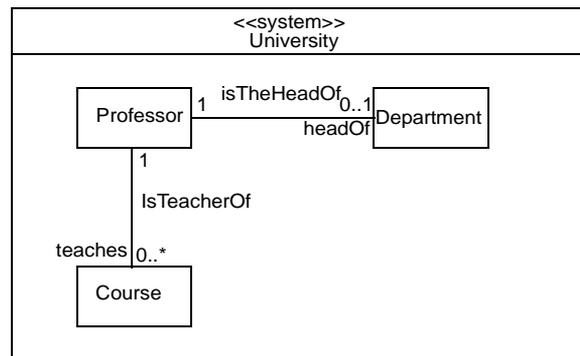


Figure 9: University analysis class model

Constraint on figure 9:

“Department heads have lighter teaching loads than other professors.”

**context** University

**inv:**

```
self.professor -> forall (head, nonHead: Professor |
    (head.headOf -> notEmpty () and nonHead.headOf -> isEmpty ())
    implies head.teaches -> size () < nonHead.teaches -> size () )
```

For further examples of Operation Schemas (as part of the Fondue approach, applied to a number of small case studies) see:

<http://lglwww.epfl.ch/research/fondue/case-studies/>

## Part 2. Additional Concepts

In this second part of the document we introduce and explore some of the newer concepts of Operation Schemas that cover specifying operations that return results and operations that possibly execute in parallel.

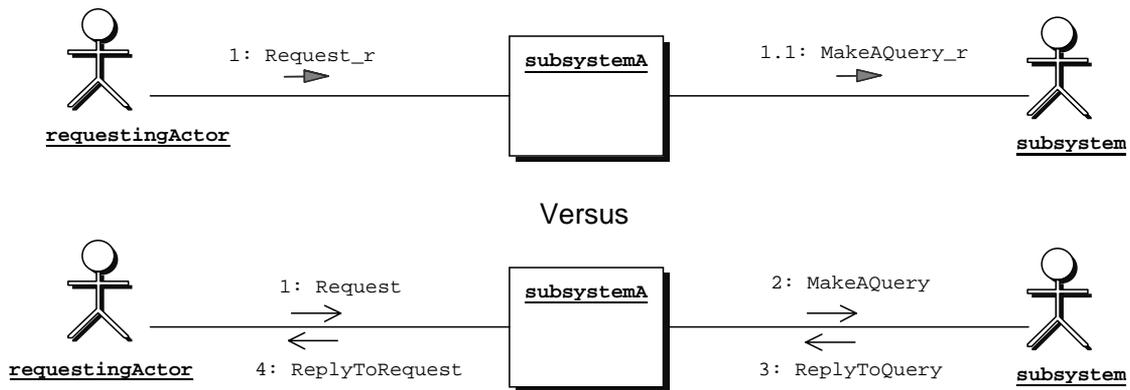
## 4. Modeling Results Returned by Operations

In this subsection, we discuss our ideas on how to use Operation Schemas for modeling results returned by operations to other actors or subsystems.

Figure 10 shows two approaches for servicing a particular request from an actor. The two approaches produce the same result. The first approach (top) shows a blocking call from requestingActor to subsystemA. During the execution of this operation, subsystemA executes a blocking call to subsystemB. Once the call returns, subsystemA returns the result of the request to requestingActor. For modeling this situation, we will use Call-WithReturn occurrences and operations returning results.

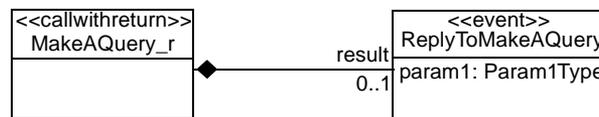
The second approach (bottom) achieves the same result by exchanging asynchronous events. Consequently, two asynchronous calls are made to subsystemA, as opposed to a single synchronous call in the first approach. This second case is handled with sending event occurrences as we have already seen in this paper. It is our preferred approach and we recommend it for systems specified from scratch.

However, both approaches are needed when we are modeling already existing components.



**Figure 10: Alternatives for Returning Results from “Calls”**

A CallWithReturn occurrence has an associated result event (figure 11). It is possible to navigate to this returned result.



**Figure 11: Relationship between a CallWithReturn and its Reply**

With the event declarations shown in figure 11, here is a postcondition fragment that asserts that a CallWithReturn occurrence was delivered to subsystemB and shows how the returned result can be accessed via result.

```

subsystemB.sent (makeAQuery) & -- like for a non-blocking call
...makeAQuery.result.param1... -- note the reply event has possibly many parameters

```

The first line asserts that the event makeAQuery has been delivered to the actor instance subsystemB. The second line asserts that the value attribute objX.addr was given the same value as the first parameter of the result of the call. The assumption is that the results are always available when the postcondition is evaluated.

Finally, we have to show how an operation returning a result can be specified by an operation schema. In the postcondition that describes such an operation, the reply event is referred to by result, and from result one can navigate to the return parameters.

For example,

**Operation:** SubsystemB::makeAQuery (): ReplyToMakeAQuery;

**Post:**

```

result = ReplyToMakeAQuery ((param1 => Color::blue));

```

We could have equally replaced the last line with:

```

result.param1 = Color::blue;

```

The reply event is implicitly sent back to the sender (who made the call), e.g., the following is redundant and may be omitted:

```

sender.sent (result);

```

## 4.1 Exceptions

Despite our assumption for reliable communications, there are often situations where the called actor cannot provide what was requested for. We will use exceptions for handling these situations, rather than returning some “dummy” value. We require that any actor requesting a service that can throw an exception must provide an exception handler. It may choose to pass it on, but this is to be asserted explicitly in a handler. We therefore propose to add an additional clause in the schema format called **Exceptions**. This clause is used to handle all exceptions stated in the **Sends** clause (associated with the **Throws** keyword).

The syntax is the following:

```
“Exceptions” “:” ( ExceptionName (“ [ParameterList] ”) “HandledBy” Condition “;” )*
```

This clause is placed after the **Post** clause in an Operation Schema.

The **Post** clause of the schema should be written in such a way that the functionality associated with exception handling is asserted within the **Exceptions** clause and not in the **Post** clause. In the case that the called operation throws an exception, instead of getting a result via the result rolename of the output event, the caller will receive an exception in its event queue, and the semantics of the Operation Schema’s postcondition will be the conjunction of the **Post** clause and the **Exceptions** clause. It is possible to write a specification that conforms to this rule because in the **Post** clause, the expression, `event.result->isEmpty ()`, is true if an exception occurred.

We demonstrate exception handling on a call to a lift scheduler. The handler deals with the case when the lift scheduler is unable to return a request to be serviced, and instead raises an exception called `noRequests_e`:

```
Post:  
scheduler.sent (gnr) &  
if gnr.result->notEmpty () then  
    self.currentRequest = gnr.result.nextRequest  
endif;  
Exceptions:  
noRequests_e () HandledBy  
    self.mode = Mode::express;
```

The **Post** clause asserts that the scheduler actor, `scheduler`, is delivered `gnr`, an event of type `GetNextRequest_r`, and if there is a reply, then the system’s current request is equivalent to the `nextRequest` parameter of the result. The **Exceptions** clause states that if the exception occurrence of type `NoRequests_e` is thrown as a consequence of a call made by the operation, then the condition after the **HandledBy** keyword is fulfilled.

Like for a signal, an exception can be specified as a subclass of another exception. This indicates that an occurrence of the exception can trigger in addition any transition of its ancestors (given the parameter lists match). If more than one handler is matched, then the most specialized match is taken.

An example of the **Exceptions** clause:

```
Exceptions:  
insufficientNumInStock (i: Item, quantityLeft: Natural) HandledBy  
    stockWarning = OutOfStock_e ((itemId => i.id)) &  
    sender.sent (stockWarning) &
```

```

receipt = OrderItem ((id => i.id, quantity => self.requiredStockAmount - quantityLeft)) &
warehouse.events->includes (receipt);

```

### Example of Call with Return Result and Exception

We show an example of a transfer (makeTransfer) that is requested by an AutoPayer actor to the YellowNet system, which in turn calls the centralBank, as shown in figure 12.

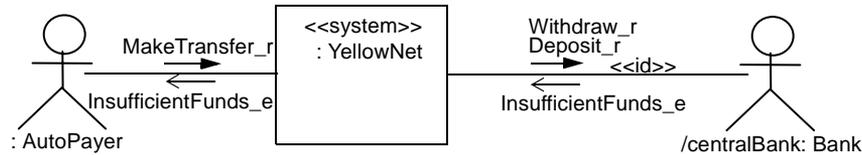


Figure 12: Collaboration Diagram showing interaction for makeTransfer operation

The declaration of events and parameters related to the MakeTransfer\_r event are shown in figure 13.

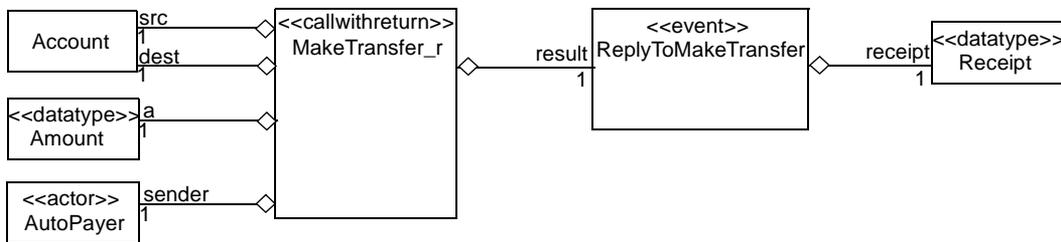


Figure 13: Declaration of Events related to MakeTransfer

The Operation Schema of the makeTransfer operation (triggered by a MakeTransfer\_r event occurrence) follows:

**Operation:** YellowNet::makeTransfer

(src: Account, dest: Account, a: Money) : ReplyToMakeTransfer;

**Sends:**

**Type:** AutoPayer::{TransferFailed\_e};

Bank::{Deposit\_r; Withdraw\_r **Throws** InsufficientFunds\_e};

**Occurrence:** deposit: Deposit\_r; withdraw: Withdraw\_r; transferFailure: TransferFailed\_e;

**Order:** <withdraw, deposit>; -- assert that withdraw is sent before deposit

**Post:**

withdraw = Withdraw\_r ((account => src, amount => a) &

(self.centralBank).sent (withdraw) &

**if** withdraw.result->notEmpty () **then** -- we received a reply

-- this if block represents an additional constraint on the ordering of the

-- two events: the deposit was not sent until the reply of the withdraw was received

deposit = Deposit\_r ((account => dest, amount => a) &

(self.centralBank).sent (deposit) &

transferTrans.ocllsNew (withId => withdraw.result.id, depld => deposit.result.id, ...) &

result.receipt = Receipt' (amount => a, ...)

**endif;**

**Exceptions:**

insufficientFunds\_e () **HandledBy**

transferFailure = TransferFailed\_e ((reason => Reason::insufficientFunds)) &

sender.sent (transferFailure);

-- passes the exception on

## 5. Specifying concurrent operations with Operation Schemas

Most specification languages assume interleaving semantics for operation execution: the reception of an event plus the associated operation is executed instantaneously, which means that there is never two operations executing at the same time even in a concurrent environment.

Our assumption for operation execution semantics up until now has been the above one. However, in this section we discuss our approach for specifying Operation Schemas where operations can possibly execute in parallel.

Exclusive updating of a shared resource is a property that we want all possible solutions to exhibit (interference avoidance). To highlight this constraint on shared resources in Operation Schemas, we add another clause to the Operation Schema format called **Shared**. The syntax for the clause is the following:

```
"Shared" ":" ( SharedItem ";" )*  
SharedItem ::= SharedType ("," SharedType)* ":" AssociationName  
              | ClassName ":" AttributeName ("," AttributeName)*  
SharedType ::= ClassName | "Collection" "("ClassName ")"
```

This clause is placed between the **Declares** and the **Sends** clauses in an Operation Schema. Resources that are listed in the **Shared** clause are constrained to be updated in mutual exclusion by the operation. The **Shared** clause only indicates the types of the shared resources.

Example:

**Shared:**

```
Collection(Transaction): Credits; Collection(Transaction): Debits;  
Collection(Transaction): (Bank, Transaction); Account::balance;
```

Shared resources are either object attributes or association ends. Shared association ends are denoted by the shared type or collection at the association end. For instance, the associations Credits, Dedits, and composition (Bank, Transaction) are shared at the Transaction end of the associations. It is denoted as a collection because the multiplicity is many and the collection of objects is accessed concurrently, rather than the transaction objects themselves.

Shared resources can be found by analyzing the System Interface Protocol (SIP) for operations that can execute in parallel. Moreover, these operations are placed into separate groups, each group containing all those operations that could possibly execute in parallel. For each operation in a group, all resources which it has access to (referred to as the frame of the specification) are placed in a set. The union of all possible intersections between any two sets in the group becomes the contents of the **Shared** clause for each schema of the respective operation, if and only if the variable is updated by at least one of the operations (and of course the shared variable is in the frame of the schema).

### Shared Variables

The @pre and the implicit "@post" suffixes for shared variables are less useful for describing system state changes in postconditions when the variable can be changed by

competing concurrent operations. Instead, the values of shared variables immediately before and after an update under mutual exclusion are more meaningful for describing changes to system state.

For example, if an operation adds 8 to a shared integer variable `self.val`, the effect that one wishes to state is that 8 was added to the value of the variable that was observed immediately before the mutually exclusive update. We, therefore, introduce the suffixes `@preAU` and `@postAU` for shared variables, which signify the state of the prefix variable immediately before and after an atomic update by the operation, respectively. The above example is defined by the following assertion:

```
self.val@postAU = self.val@preAU + 8
```

Furthermore, when we need to read the value of a shared variable, we need to ensure that we are referring to a consistent value of the variable, i.e., the variable was read outside of any period where the variable was updated. Any consistent value of a shared variable that is taken within the period of the operation's execution is denoted by suffixing `@rd` to the variable name.

The possible suffixes for shared and unshared variables are detailed in figure 14.

Between the time the operation starts execution until a shared variable is updated under mutual exclusion by the operation (pre-mutex state), we cannot (normally) make any guarantees about what happens to the shared variable. This makes aliases less useful, and therefore we modify our statement of section 1.8: “aliases always refer to the value of the expression in the operation's pre-state”, to allow one to use the `@preAU` and `@rd` suffixes in the **Declares** clause of the schema—if they make sense in the context they are used (e.g. the postcondition). In this case, aliases take more of a “macro” style.

For example,

**Declares:**

```
oldAccBal : Money Is src.balance@preAU;  
accsToUpdate: Set (Account) Is debtor.account@preAU;  
accsInCredit: Set (Account) Is accsToUpdate->select (a | a.balance@rd >= 0);  
numTrans: Natural Is acc.numTrans@rd;
```

	<b>Shared variables</b>	<b>Unshared variables</b>
exprX@pre	The last possible <i>consistent</i> value of exprX immediately before the start of the operation's execution.	The value of exprX immediately before the execution of the operation.
exprX@post	The first possible <i>consistent</i> value of exprX immediately after the termination of the operation's execution.	the suffix "@post" is normally implicit—see exprX (without suffix).
exprX@preAU	The value of the variable immediately before the operation's (atomic) update of exprX.	– (unused)
exprX@postAU	The value of the variable immediately after the operation's (atomic) update of exprX.	– (unused)
exprX@rd	Any <i>consistent</i> value of exprX inside the bounds of the operation execution (but outside of any updates to exprX).	– (unused)
exprX	only allowed in eventually functions (not discussed in this document).	The value of exprX immediately after the execution of the operation.

**Figure 14: A summary of the possible variable suffixes in postconditions**

The following Operation Schema fragment describes an operation that sets the value of an item. The variable, `self.val`, is placed in the **Shared** clause because it can be updated/read by operations that can be invoked in parallel. The postcondition (**Post** clause) asserts that the value of the item has the value `v` once the atomic update has been completed.

**Operation:** `Item::setValue (v: Integer);`

...

**Shared:** `Item::val;`

...

**Post:** `self.val@postAU = v;`

We could equally imagine the complementary “getter” operation to the “setter” operation from above. In this case, the value of the shared variable, `self.val`, is read by the operation. The postcondition, below, asserts that an event was delivered to the calling actor, which yields the value of the item. Being a shared variable that is read, `self.val` is given the `@rd` suffix.

**Operation:** `Item::getValue ();`

...

**Shared:** `Item::val;`

...

**Post:** `sender.events->includes (Result ((value => self.val@rd)));`

`self.val@rd` can be interpreted as any consistent value of `self.val`.

Note that all event queues of actors (in this example, `sender.events`) are *shared* by default and therefore are implicitly part of the **Shared** clause. Furthermore, the minimum set principle can be applied to shared collections; where it defines the change during the critical section (`@preAU` to `@postAU`).

Note that we could have used the shorthand version, which has exactly the same meaning:

**Post:** `sender.sent (Result ((value => self.val@rd)));`

## if-then-else

Schemas can still be structured with *if-then-else* blocks. The branch conditions are evaluated atomically with respect to the blocks, i.e., there is no possibility for racing between the evaluation of the branch conditions and the evaluation of the effects. Also, *if-then-else* blocks are evaluated immediately, i.e., a condition is either true or false, and there is no waiting for the condition to become true.

For example, the following Operation Schema describes an operation that increments the value of the item by 1 if `self.val` is smaller than `threshold`, otherwise it increments the value by 10; in either case the value is updated. The *if* condition further confirms that racing is not possible due to the `@preAU` suffix on `self.val` (note that the condition could have equivalently been written: `self.val@rd < threshold`).

```
Operation: Item::incrementValue ();
...
Shared: Item::val;
...
Post:    if self.val@preAU < threshold then
           self.val@postAU = self.val@preAU + 1
           else
           self.val@postAU = self.val@preAU + 10
           endif;
```

Below, we show another schema fragment where the variable in the *if* condition is different to the one we are updating.

```
Shared: Item::val, anotherVal;
...
Post:    if self.anotherVal@rd < threshold then
           self.val@postAU = self.val@preAU + 1
           else
           self.val@postAU = self.val@preAU + 10
           endif;
```

The interpretation is the following: the time at which the snapshot for `self.anotherVal@rd` is taken corresponds to the last possible consistent value before the atomic update of the body (the *if* part or *else* part).

Below, we show a *if-then-elsif* block. It takes uses a bank example, where a group in the bank has three accounts. This operation (schema only shown in part) involves enforcing group bonuses and loss of benefits depending on the balances of the three accounts, according to business rules.

```
Shared: Account::bal; Group::generalBonus, xmasBonus, easterBonus;
...
Post:    if acc1.bal@rd > 1000 then
           grp.generalBonus@postAU = grp.generalBonus@preAU + 1
           elsif acc2.bal@rd > 1000 then -- if acc1.bal@rd <= 1000 and acc2.bal@rd > 1000 then
           grp.generalBonus@postAU = grp.generalBonus@preAU - 1 &
           grp.xmasBonus@postAU & -- boolean
           grp.easterBonus@postAU -- boolean
           elsif acc3.bal@rd < 500 then -- if ... and acc2.bal@rd <= 1000 and acc3.bal@rd < 500 then
           grp.generalBonus@postAU = 0 &
           not grp.noBankCharges & -- non-concurrent resource (boolean)
```

```

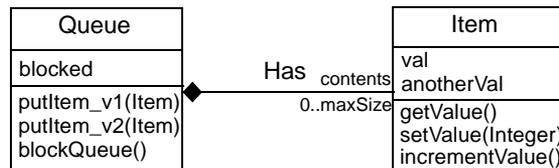
not grp.receiveXmasCard -- non-concurrent resource (boolean)
elsif (not grp.vip@pre and grp.generalBonus@preAU > 0) then
  -- if ... and acc3.bal@rd >= 500 and not grp.vip@pre and grp.generalBonus@preAU > 0 then
  grp.generalBonus@postAU = grp.generalBonus@preAU - 1 &
  not grp.receiveXmasCard
endif; -- else don't change a thing

```

The interpretation of the above postcondition with respect to the *if* conditions is that each condition is evaluated atomically in relation to the effects, and all conditions are evaluated atomically as a unit.

## Rely blocks

A situation that can arise in a concurrent environment is when a group of effects relies on a certain condition to stay true during its whole “execution”. We model such situations with *rely* blocks (based on the concept of rely conditions, first introduced by Cliff Jones). The *rely* block states a condition that must be true immediately before, immediately after, and during the execution of the body of the block for the body to take effect. If the *rely* condition does not stay true throughout execution, then the effect of the fail part of the *rely* block is observed to execute instead. The *rely* block does not impose either immediate or wait semantics on the condition, i.e., an implementation that does a wait until the condition becomes true and then tries to execute the body or one that fails if the condition is not initially true are both valid refinements.



**Figure 15: Design class diagram showing the relationship between Queue and Item**

For example, we could write a postcondition for the operation of a Queue that inserts an item into itself (see figure 15) in the following way:

```

Operation: Queue::putItem_v1 (x: Item);
...
Shared: Collection(Item): Has;
...
Post: rely (self.contents@rd->size () <= maxSize) then
  self.contents@postAU->includes (x)
  -- minimum set principle interpreted on the critical section
fail
  sender.sent (AbortPutItem_e (()))
endre;

```

The interpretation is that the queue must stay non-full during the insertion; if this condition cannot be relied upon, then sender is sent an error message. Notice also that the postcondition says nothing about how long the operation might wait, if at all, for the queue to become no longer full before it proceeds with placing the item in the queue.

Taking a closer look at the *rely* condition, we can see that we could weaken the condition by assuring that the queue is not full immediately before the insert. Thus we could write something like the following:

```

Operation: Queue::putItem_v1 (x: Item);
...
Shared: Collection(Item): Has;
...
Post:  rely (self.contents@preAU->size () < maxSize) then
        self.contents@postAU->includes (x)
        fail
            sender.sent (AbortPutItem_e ({}))
        endre;

```

In this case, we are only concerned with the size of the queue before the insert is made, due to `self.contents@preAU` not changing value after the update started execution. Note that it is possible that the critical section is not entered, in the fail case, thus the suffix `@preAU` does not necessarily imply any change is made to the variable.

The condition of the *rely* block can be a compound condition that involves many different shared variables. For example, we can expand the put item operation to assert that the queue stays unblocked and not full while the operation is modifying the queue.

```

Operation: Queue::putItem_v2 (x: Item);
...
Shared: Collection(Item): Has; Queue::blocked;
...
Post:  rely (self.contents@preAU->size () < maxSize and not self.blocked@rd) then
        self.contents@postAU->includes (x)
        fail
            sender.sent (AbortPutItem_e ({}))
        endre;

```

*Rely* blocks may have compound condition, as can be seen with the example above. In such cases, it can be useful to be able to differentiate parts of the compound condition that fail.

For example,

```

Operation: Queue::putItem_v2 (x: Item);
...
Shared: Collection(Item): Has; Queue::blocked;
...
Post:  rely (self.contents@preAU->size () < maxSize and not self.blocked@rd) then
        self.contents@postAU->includes (x)
        fail (not self.blocked@rd) then
            sender.sent (QueueIsBlocked_e ({}))
        fail (self.contents@preAU->size () < maxSize) then
            sender.sent (QueueIsFull_e ({}))
        endre;

```

The *rely* block can now fail in three ways. Either the queue is blocked, the queue is full, or both. In the first two cases, the body of the respective fail part is observed to have executed. In the last case, both fail parts are observed to have executed, i.e., the result of the block is the output of two exceptions occurrences of type `QueueIsBlocked_e` and `QueueIsFull_e`. Thus, if the *rely* condition fails, all the corresponding fail parts of the *rely* block that are “anded”.

Note that the fail condition can only refer to condition atoms of the rely condition. Thus, three fail conditions would be the most the above *rely* block could have: LHS boolean expression, RHS boolean expression, and both expressions “anded”.

### Example: Transfer money from one account to another account

The following Operation Schema details a transfer operation, where the transfer of money is done between accounts within the system.

**Operation:** Bank::transfer (src: Account, dest: Account, amount: Money);

**Sends:**

**Type:** Client::{TransferNotPossible\_e, AccountIsBlocked\_e};

**Shared:** Account::blocked, balance;

**Pre:** true;

**Post:** **rely** src.balance@preAU >= amount **and not** src.blocked@rd  
**and not** dest.blocked@rd **then**  
src.balance@postAU = src.balance@preAU - amount &  
dest.balance@postAU = dest.balance@preAU + amount  
**fail** (src.balance@preAU >= amount) **then**  
sender.sent (TransferNotPossible\_e (()))  
**fail** (**not** src.blocked@rd) **then**  
sender.sent (AccountIsBlocked\_e ((acc => src)))  
**fail** (**not** dest.blocked@rd) **then**  
sender.sent (AccountIsBlocked\_e ((acc => dest)))  
**endre;**

The idea with the blocked attribute is that a manager actor can change the status of the attribute at anytime with a separate system operation. Therefore, we need to *rely* on the fact that neither account becomes blocked during the transfer. Also, the **Post** clause shows the use of multiple fail parts to the *rely* block.

Also, *rely* blocks should not be nested in *if* blocks due to the immediate evaluation semantics of the *if*. But, *rely* blocks nested in *rely* blocks, and *if* blocks nested in *rely* blocks are possible.

For example, the following would **NOT** be allowed:

**Post:** **if** acc.balance@rd > 1000 **then**  
**rely not** acc.blocked@rd **then**  
acc.balance@postAU = acc.balance@preAU \* 1.15  
**fail**  
sender.sent (AccountIsBlocked\_e ((acc => acc)))  
**endre**  
**endif;**

### Example: Dining Philosophers

The following example is based on the classic concurrency problem: Dining Philosophers. The idea is that a table has a certain number of places (with a chair at each place) and a certain number of chopsticks. The chopsticks are placed on the table in such a way that someone sitting at the table has a left and right chopstick. However, there is only as many chopsticks as there are places. Philosophers can take a seat; while seated, they may switch between two states: thinking (initial state) and eating. Once

they have finished philosophizing, they can leave the table, but only if they are not eating.

The SIP for the Room system is shown in figure 16.

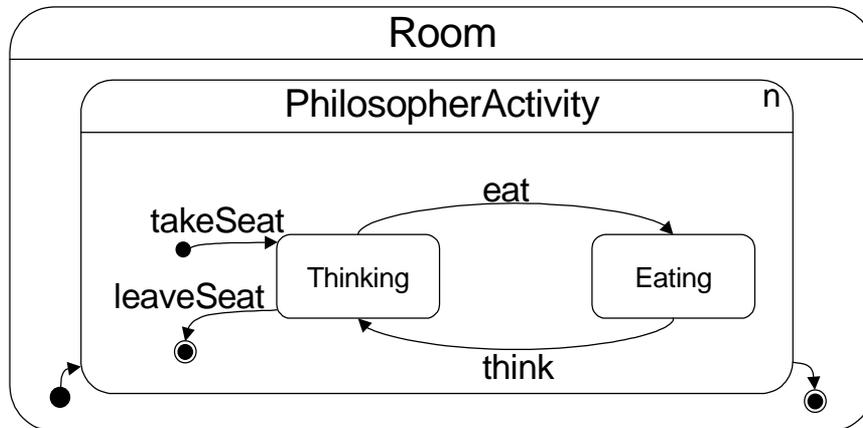


Figure 16: SIP for Room system

The Analysis Class Model (ACM) for the Room system is shown in figure 17.

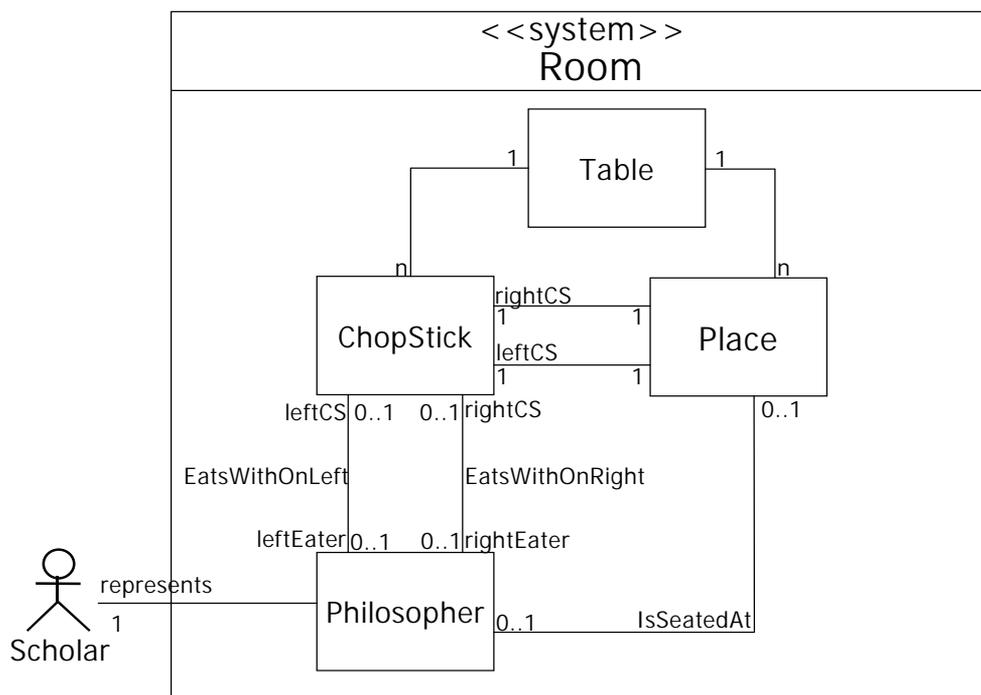


Figure 17: Analysis Class Model of the Room system

The Operation Schemas for each operation of Room are shown below.

**Fact:** A philosopher does not make concurrent requests with respect to him/herself (see SIP)

**Operation:** Room::eat (philo: Philosopher, p: Place);

**Shared:** ChopStick: EatsWithOnLeft; ChopStick: EatsWithOnRight;

**Sends:**

**Type:** Scholar::(NotAllForksCouldBeObtained\_e);

**Pre:** philo.leftCS->isEmpty () **and** philo.rightCS->isEmpty ();

**Post:**

```
rely p.leftCS.leftEater@preAU->isEmpty () and
      p.rightCS.rightEater@preAU->isEmpty () then
  p.leftCS.leftEater@postAU = philo &
  p.rightCS.rightEater@postAU = philo
fail
  (philo.represents).events-> includes (NotAllForksCouldBeObtained_e ((place => p)))
endre;
```

The operation must rely on the chopsticks both being free before they can be taken for a philosopher to start eating.

**Operation:** Room::think (philo: Philosopher);

**Shared:** ChopStick: EatsWithOnLeft; ChopStick: EatsWithOnRight;

**Pre:** true;

**Post:**

```
philo.leftCS@postAU->isEmpty () &
philo.rightCS@postAU->isEmpty ();
```

A philosopher simply drops the chopsticks s/he has.

**Operation:** Room::takeSeat (philo: Philosopher, p: Place);

**Shared:** Place: IsSeatedAt;

**Sends:**

```
Type: Scholar::{CannotObtainSeat_e};
```

**Pre:** philo.place->isEmpty ();

**Post:**

```
rely p.philosopher@preAU->isEmpty () then
  p.philosopher@postAU = philo
fail
  (philo.represents).sent (CannotObtainSeat_e ((place => p)))
endre;
```

A philosopher can only take a seat when it is empty.

**Operation:** Room::leaveSeat (philo: Philosopher, p: Place);

**Shared:** Place: IsSeatedAt;

**Sends:**

```
Type: Scholar::{NeedToStopEating_e};
```

**Pre:** p.philosopher = philo **and** philo.leftCS->isEmpty () **and** philo.rightCS->isEmpty ();

**Post:**

```
p.philosopher@postAU->isEmpty ();
```

The philosopher must not be holding any chopsticks to leave the table.

Note also that Place is a shared resource only for the takeSeat and leaveSeat operations; this is because the philosopher execute actions in sequence, and thus it not possible for a philosopher to leave his/her seat while trying to execute eat or think.