The Object Constraint Language (OCL)

- OCL is used for adding important information to UML models
- OCL is used
  - To add constraints, restrictions on values that formed part of an object-oriented model
  - To state conditions on execution of a system
  - To define business rules
  - To define queries on the model – so questions about a model can be answered automatically by executing an OCL query
- OCL2.0 is a new improved version.

Modelling with OCL

- Used for queries and constraints over UML2 models
- OCL syntax covers the following aspects of models:
  - Initial values
  - Contracts
  - Constraints
- If you read the reference (the Warmer and Kleppe book from http://www.klasse.nl/ocl/), these aspects are detailed for class diagrams, not component diagrams; however the approach is similar
- To understand how OCL can be used to provide a formal semantics of components

Context of OCL Expressions

- An OCL expression annotates a particular model entity (class, interface, datatype, component or entire application) – this is called the context of the expression
- Annotation takes the following form:
  
  ```
  Context ModelEntity
  [init:/pre:/post:/inv:]
  ```

  OCL expression about ModelEntity
  - The context acts as a namespace for the OCL expressions
  - The OCL expression can refer to aspects of the context:
    - the attributes and methods of the class
    - the provided interfaces of the component
    - the required interfaces of the component
Interface Semantics

- Recall that we should always include *a behavioural semantics* before using an interface within an architecture.
- So far we have settled with using English to define these behaviours.

<table>
<thead>
<tr>
<th>Interface Semantics</th>
<th>Interface Specifications – Initial Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall that we should always include <em>a behavioural semantics</em> before using an interface within an architecture.</td>
<td>Initial value rules are associated with an interface context.</td>
</tr>
<tr>
<td>So far we have settled with using English to define these behaviours.</td>
<td>Initial value rules define how a component interface implementation must behave at initialization.</td>
</tr>
<tr>
<td>Can make reference to getter (side-effect-free) methods of the interface context.</td>
<td>Can make reference to getter (side-effect-free) methods of the interface context.</td>
</tr>
</tbody>
</table>

**Interface name**

**List of operations** (all public)

**Specification of behaviour**

```
<<interface>>
ITransportable

+sendMessage(msg:Message):void
+getSmtpHost():String
+getSmtpPort():int
+setSmtpHost(smtpHost:String)
+setSmtpPort(smtpPort:int)
```

Interface Specifications – Initial Values

Initial value rules are associated with an interface context.

Initial value rules define how a component interface implementation must behave at initialization.

Can make reference to getter (side-effect-free) methods of the interface context.

**Context:**

**Init:** OCL Expression about side-effect-free interface methods

```
<<interface>>
ITransportable

+sendMessage(msg:Message):void
+getSmtpHost():String
+getSmtpPort():int
+setSmtpHost(smtpHost:String)
+setSmtpPort(smtpPort:int)
```

Interface Specifications - Contracts

Contracts are used to define how operations of an interface should execute.

Contracts add a lot of value to a component’s interface specification.

- Components are black-box, so we cannot see how methods work through inspection of code.
- The architect relies solely on documentation of component interface.
- OCL contracts provide a precise definition of how a component must behave if it provides the interface, and what a component needs if it requires the interface.

A contract consists of two parts:

- **Precondition** -- a boolean expression that must be true at the moment that the method starts its execution.
- **Postcondition** -- a boolean expression that must be true at the moment that the operation ends its execution.

Contracts in OCL

Contract for an interface operation has the following form:

```
context interfaceName::methodName
pre: OCL precondition expression
post: OCL postcondition expression
```

Context consists of the interface name followed by the operation name (demarcated by scope :: symbol).

Pre- and post-condition can make use of any getter (side-effect-free) operations of the interface.

Let `isSMTP(s)` returns true when `s` is an SMTP compliant message, and `getSentMail()` returns a folder of mail that has been sent.

```
context: ITransportable::

sendMessage(msg:Message)
pre: isSMTP(msg)
post: getSentMail() =
    getSentMail()@pre>union(Set{msg})
```

Example: interface sends SMTP compliant messages and adds them to a sent mail folder.

```
context: ITransportable

sendMessage(msg:Message)
pre: isSMTP(msg)
post: getSentMail() =
    getSentMail()@pre>union(Set{msg})
```

Example: interface sends SMTP compliant messages and adds them to a sent mail folder.

```
context: ITransportable

sendMessage(msg:Message)
pre: isSMTP(msg)
post: getSentMail() =
    getSentMail()@pre>union(Set{msg})
```

Example: interface sends SMTP compliant messages and adds them to a sent mail folder.

```
context: ITransportable

sendMessage(msg:Message)
pre: isSMTP(msg)
post: getSentMail() =
    getSentMail()@pre>union(Set{msg})
```
The @pre Symbol

- The @pre symbol permits a postcondition to refer to the value a getter method returned at the beginning of the method call.
- So, in:
  
  \[
  \text{post: } \text{getSentMail()} = \text{getSentMail()}@\text{pre}\rightarrow\text{union(Set\{msg\})}
  \]

  the LHS getSentMail() refers the value of getSentMail() after executing the method, while the RHS getSentMail()@pre refers to getSentMail prior to executing the method.

  Initial value of getSentMail() is referred to in postcondition as getSentMail()@pre.

Messaging in Postconditions

- OCL contracts are used to define side-effects of interface operations.
- OCL expressions are not permitted side-effects – and therefore generally only refer to getter methods of interfaces.
- It is possible to say that a particular side-effect-producing method can be called (e.g., assert that a message has been passed), using the ^ operator: b^m() asserts that the method m has been called on element b.
- If an interface has a sendMsg method that should result in a subsequent call to a passMsg method, this can be specified as:
  
  \[
  \text{context: } \text{ITransportable::sendMsg(msg:Message)} \\
  \text{post: } \text{self}^\text{passMsg()}
  \]

  Note: self is like this in Java.

Components as Substitutable Units

- **Substitutability**: Replace a component with an alternative or updated version without breaking the systems in which the component is used.

New system = old system with Hotel reservation component replaced.

Laws of Substitutability

- A component B can be substituted for component A if, and only if:
  - A’s provided interfaces are preserved by B (that is, signature and expected behaviour preserved)
  - B’s required interfaces are the same as or fewer than A’s (that is, B cannot require more)

- These constraints must be satisfied in order to avoid system breakage.
Substitutability

- That definition was vague about “expected behaviour” being preserved.
- If the interfaces have OCL contracts, then we will define the formal laws of substitutability.
- Component B can be substituted for component A if, and only if,
  - A’s provided interfaces are preserved by B
  - B’s required interfaces are the same as or fewer than A’s (that is, B cannot require more)
  - Preconditions of B’s provided interfaces are weaker than (are entailed by) preconditions of A’s interfaces
  - Postconditions of B’s provided interfaces are stronger than (entail) postconditions of A

Substitutability Constraints

- Constraints Over Models
  - OCL is also extremely useful to define constraints on the overall structure of a model
  - These are given as contracts and invariants for a context, with the following syntax
    - context modelElement
      - inv: OCL constraint over the model element
    - The OCL constraint can involve any aspect of the model element and its associations
  - Navigation through associations permits powerful constraints to be specified

Navigation Mechanism

- An OCL expression can navigate from its context element to associated elements using the “dot” notation, just as in Java
- Notation: To navigate over connections between components in architecture diagrams, we need to label the beginnings and ends of connections, giving names to the component instances referenced by the connections
  - HolidaySys accesses CarRes component via ICarBooking interface, referring to component instance as myCarRes
  - If c.pinterfaces returns the set of all provided interfaces for a component, then
    - Context HotelRes
    - inv: myHolidaySys.myCarRes. pinterfaces->size() = 1
    - uses navigation to make a statement about CarRes relative to the HotelRes context
Assumed sets

- We assume the following sets are available for UML2 architectures:
  - `App.components` provides the set of all components used within `App`
  - `c.provides` gives the set of all the provided interfaces of a component `c`
  - `c.requires` gives the set of all the required interfaces of a component `c`
  - `c.connections` gives the set of all connected components to a component `c`
- Full OCL expressions can be defined to obtain these sets

Pre- and Post- conditions over Architectures

- **Pre and post conditions can express important interaction across an architecture** (not just local interaction with respect to interface implementations)
- Example: we can express that a call to `HolidaySys`'s `makeRes(person, date)` operation should result in calls to `CarRes` and `HotelRes` in the following way:

```
Context HolidaySys::
makeRes(person, date)
Post: self.myCarRes^makeRes(person, date) and self.myHolidaySys^makeRes(person, date)
```

Invariants over Architectures

- **Invariants can be used to specify roles, relationships and dependencies that do not change over the execution of an architecture**
- Example: assume `c->down()` means a component `c` is unavailable – the invariant that the `HolidaySys` component is dependent on `HotelRes` and `CarRes` is defined as follows

```
Context HolidaySys
Inv: self.myCarRes->down() and self.myHoliday->down() implies self->down()
```

Expressing Styles Formally

- When we discussed styles, we defined constraints over target architectures informally, using English.
- Example:
  - Pipes-and-Filters has the constraint that “every component used in the style provides and requires the same interface, and each component is connected to the other in a chain”
- These constraints are usually very general, involving whole sets of components and connectors used in an architecture
- **Using sets and set operations, it is possible to formalize these constraints**
Pipes-and-Filters in OCL

- The pipes-and-filters style can be formally expressed as follows:
  
  Let App be the part of an architecture that implements the style. Then:
  
  App.components->forall(c | c.provides = c.requires and c.connections->size() > 0 and c.connections->size() < 3) and App.components->forall(c | App.components->forall(d | c.provides = d.requires))
  
  Alternatively, expressed using context/invariant syntax:
  
  Context App
  Inv: self.components->forall(c | c.provides = c.requires and c.connections->size() > 0 and c.connections->size() < 3) and self.components->forall(d | c.provides = d.requires)