Contents & Goals

Last Lecture:

- Live Sequence Charts Semantics

This Lecture:

- **Educational Objectives**: Capabilities for following tasks/questions.
  - What’s the Liskov Substitution Principle?
  - What is late/early binding?
  - What is the subset, what the uplink semantics of inheritance?
  - What’s the effect of inheritance on LSCs, State Machines, System States?
  - What’s the idea of Meta-Modelling?

- **Content**:
  - Quickly: Behavioural Features, Active vs. Passive
  - Inheritance in UML: concrete syntax
  - Liskov Substitution Principle — desired semantics
  - Two approaches to obtain desired semantics
  - The UML Meta Model
Active and Passive Objects [Harel and Gery, 1997]
What about non-Active Objects?

Recall:

- We’re still working under the assumption that all classes in the class diagram (and thus all objects) are active.
- That is, each object has its own thread of control and is (if stable) at any time ready to process an event from the ether.

But the world doesn’t consist of only active objects.
For instance, in the crossing controller from the exercises we could wish to have the whole system live in one thread of control.

So we have to address questions like:

- Can we send events to a non-active object?
- And if so, when are these events processed?
- etc.
Active and Passive Objects: Nomenclature

[Harel and Gery, 1997] propose the following (orthogonal!) notions:

- A class (and thus the instances of this class) is either active or passive as declared in the class diagram.
  - An active object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
  - A passive object doesn’t.

- A class is either reactive or non-reactive.
  - A reactive class has a (non-trivial) state machine.
  - A non-reactive one hasn’t.

Which combinations do we understand?

<table>
<thead>
<tr>
<th></th>
<th>active</th>
<th>passive</th>
</tr>
</thead>
<tbody>
<tr>
<td>reactive</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>non-reactive</td>
<td>(✓)</td>
<td>(✓)</td>
</tr>
</tbody>
</table>
Passive and Reactive

- So why don’t we understand passive/reactive?
- Assume passive objects \( u_1 \) and \( u_2 \), and active object \( u \), and that there are events in the ether for all three.

  Which of them (can) start a run-to-completion step...?
  Do run-to-completion steps still interleave...?

Reasonable Approaches:

- **Avoid** — for instance, by
  - require that reactive implies active for model well-formedness.
  - requiring for model well-formedness that events are never sent to instances of non-reactive classes.

- **Explain** — here: (following [Harel and Gery, 1997])
  - Delegate all dispatching of events to the active objects.
Firstly, establish that each object $u$ knows, via (implicit) link $\text{itsAct}$, the active object $u_{act}$ which is responsible for dispatching events to $u$.

If $u$ is an instance of an active class, then $u_{a} = u$. 

With inheritance (how it really looks in Rhapsody)
Passive Reactive Classes

- Firstly, establish that each object \( u \) knows, via (implicit) link \( \text{itsAct} \), the **active object** \( u_{\text{act}} \) which is responsible for dispatching events to \( u \).

- If \( u \) is an instance of an active class, then \( u_{\text{a}} = u \).

**Sending an event:**

- Establish that of each signal we have a version \( E_C \) with an association \( \text{dest} : C_{0,1}, C \in \mathcal{C} \).
- Then \( n!E \) in \( u_1 : C_1 \) becomes:
- Create an instance \( u_e \) of \( E_{C_2} \) and set \( u_e \)’s \( \text{dest} \) to \( u_d := \sigma(u_1)(n) \).
- Send to \( u_a := \sigma(\sigma(u_1)(n))(\text{itsAct}) \), i.e., \( \varepsilon' = \varepsilon \oplus (u_a, u_e) \).

**Dispatching an event:**

- Observation: the ether only has events for active objects.
- Say \( u_e \) is ready in the ether for \( u_{\text{a}} \).
- Then \( u_{\text{a}} \) asks \( \sigma(u_e)(\text{dest}) = u_d \) to process \( u_e \) — and waits until completion of corresponding RTC.
- \( u_d \) may in particular discard event.
And What About Methods?
And What About Methods?

- In the current setting, the (local) state of objects is only modified by actions of transitions, which we abstract to transformers.

- In general, there are also methods.

- UML follows an approach to separate
  - the **interface declaration** from
  - the **implementation**.

  In C++ lingo: distinguish **declaration** and **definition** of method.

- In UML, the former is called **behavioural feature**
  and can (roughly) be
  - a **call interface** \( f(\tau_1, \ldots, \tau_{n_1}) : \tau_1 \)
  - a **signal name** \( E \)

  \[
  \begin{array}{ccc}
  \text{\( C \)} & \\
  \xi_1 & f(\tau_{1,1}, \ldots, \tau_{1,n_1}) : \tau_1 & P_1 \\
  \xi_2 & F(\tau_{2,1}, \ldots, \tau_{2,n_2}) : \tau_2 & P_2 \\
  \langle signal \rangle & E \\
  \end{array}
  \]

  Note: The signal list is redundant as it can be looked up in the state machine of the class. But: certainly useful for documentation.
Behavioural Features

<table>
<thead>
<tr>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\xi_1 ; f(\tau_{1,1}, \ldots, \tau_{1,n_1}) : \tau_1 ; P_1$</td>
</tr>
<tr>
<td>$\xi_2 ; F(\tau_{2,1}, \ldots, \tau_{2,n_2}) : \tau_2 ; P_2$</td>
</tr>
<tr>
<td>$\langle \langle \text{signal} \rangle \rangle ; E$</td>
</tr>
</tbody>
</table>

Semantics:

- The **implementation** of a behavioural feature can be provided by:
  - An **operation**.
    - In our setting, we simply assume a transformer like $T_f$.
    - It is then, e.g. clear how to admit method calls as actions on transitions: function composition of transformers (clear but tedious: non-termination).
    - In a setting with Java as action language: operation is a method body.
  - The class' **state-machine** ("triggered operation").
    - Calling $F$ with $n_2$ parameters for a stable instance of $C$ creates an auxiliary event $F$ and dispatches it (bypassing the ether).
    - Transition actions may fill in the return value.
    - On completion of the RTC step, the call returns.
    - For a non-stable instance, the caller blocks until stability is reached again.
**Behavioural Features: Visibility and Properties**

<table>
<thead>
<tr>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\xi_1 f(\tau_{1,1}, \ldots, \tau_{1,n_1}) : \tau_1 P_1$</td>
</tr>
<tr>
<td>$\xi_2 F(\tau_{2,1}, \ldots, \tau_{2,n_2}) : \tau_2 P_2$</td>
</tr>
<tr>
<td>$\langle signal \rangle E$</td>
</tr>
</tbody>
</table>

- **Visibility**: Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.

- **Useful properties**:
  - **concurrency**
    - **concurrent** — is thread safe
    - **guarded** — some mechanism ensures/should ensure mutual exclusion
  - **sequential** — is not thread safe, users have to ensure mutual exclusion
  - **isQuery** — doesn’t modify the state space (thus thread safe)

- For simplicity, we leave the notion of steps untouched, we construct our semantics around state machines. Yet we could explain pre/post in OCL (if we wanted to).
State Machines: Discussion.
Semantic Variation Points

Pessimistic view: They are legion...

- For instance,
  - allow absence of initial pseudo-states
    can then “be” in enclosing state without being in any substate; or assume one of the children states non-deterministically
  - (implicitly) enforce determinism, e.g.
    by considering the order in which things have been added to the CASE tool’s repository, or graphical order
  - allow true concurrency

Exercise: Search the standard for “semantical variation point”.

- [Crane and Dingel, 2007], e.g., provide an in-depth comparison of Statemate, UML, and Rhapsody state machines — the bottom line is:
  - the intersection is not empty
    (i.e. there are pictures that mean the same thing to all three communities)
  - none is the subset of another
    (i.e. for each pair of communities exist pictures meaning different things)

Optimistic view: tools exist with complete and consistent code generation.
\[ \mathcal{I} = (\mathcal{I}, C, V, \text{attr}), \ \text{SM} \]

\[ M = (\Sigma_{\mathcal{I}}, A_{\mathcal{I}}, \rightarrow_{\text{SM}}) \]

\[ \varphi \in \text{OCL} \]

\[ \pi = (\sigma_0, \varepsilon_0) \xrightarrow{(\text{cons}_0, \text{Snd}_0)} (\sigma_1, \varepsilon_1) \cdots \]

\[ w_\pi = ((\sigma_i, \text{cons}_i, \text{Snd}_i))_{i \in \mathbb{N}} \]

\[ G = (N, E, f) \]

\[ \mathcal{O} \]

\[ \mathcal{O} \]

\[ \mathcal{O} \]

\[ \mathcal{O} \]
Inheritance: Syntax
Recall: a signature (with signals) is a tuple $\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E})$.

Now (finally): extend to $\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E}, F, mth, \triangleleft)$

where $F/mth$ are methods, analogously to attributes and

$$\triangleleft \subseteq (\mathcal{C} \times \mathcal{C}) \cup (\mathcal{E} \times \mathcal{E}) \cup (\mathcal{C} \setminus \mathcal{E} \times \mathcal{C} \setminus \mathcal{E})$$

is a generalisation relation such that $C \triangleleft^+ C$ for no $C \in \mathcal{C}$ (“acyclic”).

$C \triangleleft D$ reads as

- $C$ is a generalisation of $D$,
- $D$ is a specialisation of $C$,
- $D$ inherits from $C$,
- $D$ is a sub-class of $C$,
- $C$ is a super-class of $D$,
- ...
$\Delta = \{ C_0 \triangleleft C_1, C_1 \triangleright C_2, D \triangleright C_2 \}$

"$C_2$ inherits from $C_0$ via $C_1"
Recall: Reflexive, Transitive Closure of Generalisation

**Definition.** Given classes $C_0, C_1, D \in \mathcal{C}$, we say $D$ inherits from $C_0$ via $C_1$ if and only if there are $C_0^1, \ldots C_0^m, C_1^1, \ldots C_1^m \in \mathcal{C}$ such that

\[ C_0 \prec C_0^1 \prec \ldots C_0^m \prec C_1 \prec C_1^1 \prec \ldots C_1^m \prec D. \]

We use ‘$\preceq$’ to denote the reflexive, transitive closure of ‘$\prec$’.

In the following, we assume

- that all attribute (method) names are of the form
  
  $C::v, \quad C \in \mathcal{C} \cup \mathcal{E} \quad (C::f, \quad C \in \mathcal{C}),$

- that we have $C::v \in \text{atr}(C)$ resp. $C::f \in \text{mth}(C)$ if and only if $v$ ($f$) appears in an attribute (method) compartment of $C$ in a class diagram.

We still want to accept “context $C$ inv : $v < 0$”, which $v$ is meant? Later!
Inheritance: Desired Semantics
Desired Semantics of Specialisation: Subtyping

There is a classical description of what one expects from sub-types, which in the OO domain is closely related to inheritance:

The principle of type substitutability \cite{Liskov, 1988, Liskov and Wing, 1994}. (Liskov Substitution Principle (LSP).)

"If for each object \(o_1\) of type \(S\) there is an object \(o_2\) of type \(T\) such that for all programs \(P\) defined in terms of \(T\), the behavior of \(P\) is unchanged when \(o_1\) is substituted for \(o_2\) then \(S\) is a subtype of \(T\)."

\[
S \text{ sub-type of } T \iff \forall o_1 \in S \exists o_2 \in T \forall P_T : [P_T(o_1) = [P_T(o_2)]
\]
**Desired Semantics of Specialisation: Subtyping**

There is a classical description of what one expects from sub-types, which in the OO domain is closely related to inheritance:

The principle of type substitutability \[\text{[Liskov, 1988, Liskov and Wing, 1994]}\].

(\textbf{Liskov Substitution Principle (LSP).})

“If for each object \(o_1\) of type \(S\) there is an object \(o_2\) of type \(T\) such that for all programs \(P\) defined in terms of \(T\),

\textbf{the behavior of} \(P\) \textbf{is unchanged} when \(o_1\) is substituted for \(o_2\)

then \(S\) is a \textbf{subtype} of \(T\).”

In other words: \[\text{[Fischer and Wehrheim, 2000]}\]

“An instance of the \textbf{sub-type} shall be \textbf{usable} whenever an instance of the supertype was expected,

\textbf{without a client being able to tell the difference}.”

So, what’s “\textbf{usable}”? Who’s a “\textbf{client}”? And what’s a “\textbf{difference}”?
“...shall be usable...”?

OCL:
- context C inv : x > 0

Actions:
- itsC.x = 0
- itsC.f(0)
- itsC!F

Triggers:
- E[...]/...

Sequence Diagrams:

\[
\text{context } C \quad \text{inv : } x > 0
\]

\[
\begin{align*}
\text{actions:} \\
\text{itsC.x = 0} \quad \text{must be defined} \\
\text{itsC.f(0)} \\
\text{itsC!F}
\end{align*}
\]

\[
\text{triggers:} \\
E[...]/...
\]

\[
\text{should bind to } u_1 \text{ as well as } u_2
\]

\[
\text{should bind to } F \text{ instances as well}
\]
“...a client...”?

“An instance of the **sub-type** shall be **usable** whenever an instance of the supertype was expected, without a **client** being able to tell the **difference**.”

- **Narrow** interpretation: another object in the model.

- **Wide** interpretation: another modeler.
“...can’t tell difference...”?

![Diagram]

- **OCL:**
  - $I[[\text{context } C \text{ inv : } x > 0]](\sigma_1, \emptyset)$ vs. $I[[\text{context } C \text{ inv : } x > 0]](\sigma_2, \emptyset)$
“...can’t tell difference...”?

- **Triggers, Actions**: if

\[
(\sigma_0, [u_2/u_1], \varepsilon_0) \xrightarrow{u_2} \sigma_1 \xrightarrow{[u_2/u_1]} \varepsilon_1
\]

is possible, then

\[
(\sigma_0, \varepsilon_0) \xrightarrow{u_1} (\sigma_1, \varepsilon_1)
\]

should be possible – sub-type does less on inputs of super-type.

for some \( u_1 \in \text{D}(C) \) and a proper definition of \([\cdot/\cdot]\)
“...can’t tell difference...”?

- **Sequence Diagram**: $w[u_1/u_2] \in \mathcal{L}(B_L)$ implies $w \in \mathcal{L}(B_L)$.

\[ u_1 : C \quad u_2 : D \]

\[ C \]
\[ x : \text{Int} \]
\[ f(\text{Int}) : \text{Int} \]

\[ \langle \langle \text{signal} \rangle \rangle \]

\[ E \]

\[ D \]

\[ \langle \langle \text{signal} \rangle \rangle \]

\[ F \]
Motivations for Generalisation

- Re-use,
- Sharing,
- Avoiding Redundancy,
- Modularisation,
- Separation of Concerns,
- Abstraction,
- Extensibility,
- ...

→ See textbooks on object-oriented analysis, development, programming.
What Does [Fischer and Wehrheim, 2000] Mean for UML?

"An instance of the sub-type shall be usable whenever an instance of the supertype was expected, without a client being able to tell the difference."

- Wanted: sub-typing for UML.
- With

```
C

D_1
```

we don’t even have usability.

- It would be nice, if the well-formedness rules and semantics of

```
C

D_1

D_2
```

would ensure $D_1$ is a sub-type of $C$:

- that $D_1$ objects can be used interchangeably by everyone who is using $C$’s,
- is not able to tell the difference (i.e. see unexpected behaviour).
“...shall be usable...” for UML
Easy: Static Typing

Given:

Wanted:

- \( x > 0 \) also **well-typed** for \( D_1 \)
- assignment \( \text{its}C1 := \text{its}D1 \) being **well-typed**
- \( \text{its}C1.x = 0, \text{its}C1.f(0), \text{its}C1 ! F \)
  being well-typed (and doing the right thing).

Approach:

- Simply define it as being well-typed, adjust system state definition to do the right thing.
Static Typing Cont’d

Notions (from category theory):

- invariance,
- covariance,
- contravariance.

We could call, e.g. a method, **sub-type preserving**, if and only if it

- accepts **more general** types as input (contravariant),
- provides a **more specialised** type as output (covariant).

This is a notion used by many programming languages — and easily type-checked.
Excursus: Late Binding of Behavioural Features
Late Binding

What transformer applies in what situation? (Early (compile time) binding.)

<table>
<thead>
<tr>
<th></th>
<th>f not overridden in D</th>
<th>f overridden in D</th>
<th>value of someC/someD</th>
</tr>
</thead>
<tbody>
<tr>
<td>someC</td>
<td>C::f()</td>
<td>C::f()</td>
<td>u₁</td>
</tr>
<tr>
<td>someD</td>
<td>C::f()</td>
<td>D::f()</td>
<td>u₂</td>
</tr>
<tr>
<td>someC</td>
<td>C::f()</td>
<td>D::f()</td>
<td>u₂</td>
</tr>
</tbody>
</table>

What one could want is something different: (Late binding.)

<table>
<thead>
<tr>
<th></th>
<th>C::f()</th>
<th>C::f()</th>
<th>u₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>someC</td>
<td>C::f()</td>
<td>D::f()</td>
<td>u₂</td>
</tr>
<tr>
<td>someD</td>
<td>D::f()</td>
<td>D::f()</td>
<td>u₂</td>
</tr>
<tr>
<td>someC</td>
<td>C::f()</td>
<td>C::f()</td>
<td>u₂</td>
</tr>
</tbody>
</table>
Late Binding in the Standard and Programming Lang.

- In the standard, Section 11.3.10, “CallOperationAction”:
  “Semantic Variation Points
  The mechanism for determining the method to be invoked as a result of a call operation is unspecified.” [OMG, 2007b, 247]

- In C++,
  - methods are by default “(early) compile time binding”,
  - can be declared to be “late binding” by keyword “virtual”,
  - the declaration applies to all inheriting classes.

- In Java,
  - methods are “late binding”;
  - there are patterns to imitate the effect of “early binding”

Exercise: What could have driven the designers of C++ to take that approach?
Late Binding in the Standard and Programming Lang.

- In the standard, Section 11.3.10, “CallOperationAction”:
  “Semantic Variation Points
  The mechanism for determining the method to be invoked as a result of a call operation is unspecified.” [OMG, 2007b, 247]

- In C++,
  - methods are by default “(early) compile time binding”,
  - can be declared to be “late binding” by keyword “virtual”,
  - the declaration applies to all inheriting classes.

- In Java,
  - methods are “late binding”;
  - there are patterns to imitate the effect of “early binding”

Exercise: What could have driven the designers of C++ to take that approach?

Note: late binding typically applies only to methods, not to attributes.
(But: getter/setter methods have been invented recently.)
Back to the Main Track: “...tell the difference...” for UML
...we’re done (if we realise it correctly in the framework).

Then

• if we’re calling method $f$ of an object $u$,
• which is an instance of $D$ with $C \preceq D$
• via a $C$-link,
• then we (by definition) only see and change the $C$-part.
• We cannot tell whether $u$ is a $C$ or an $D$ instance.

So we immediately also have behavioural/dynamic subtyping.
Difficult: Dynamic Subtyping

- $C::f$ and $D::f$ are type compatible, but $D$ is not necessarily a sub-type of $C$.

- Examples: (C++)

```cpp
typedef int C::f(int) {
    return 0;
};

vs.

typedef int D::f(int) {
    return 1;
};
```
**Difficult: Dynamic Subtyping**

- $C::f$ and $D::f$ are **type compatible**, but $D$ is not necessarily a **sub-type** of $C$.

- **Examples**: (C++)

```cpp
int C::f(int) {
    return 0;
}
```
vs.

```cpp
int D::f(int) {
    return 1;
}
```

```cpp
int C::f(int) {
    return (rand() % 2);
}
```
vs.

```cpp
int D::f(int x) {
    return (x % 2);
}
```
Sub-Typing Principles Cont’d

- In the standard, Section 7.3.36, “Operation”:

  “Semantic Variation Points
  […] When operations are redefined in a specialization, rules regarding invariance, covariance, or contravariance of types and preconditions determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations.” [OMG, 2007a, 106]
Sub-Typing Principles Cont’d

- In the standard, Section 7.3.36, “Operation”:
  “Semantic Variation Points
  [...] When operations are redefined in a specialization, rules regarding
  invariance, covariance, or contravariance of types and preconditions
determine whether the specialized classifier is substitutable for its more
general parent. Such rules constitute semantic variation points with
respect to redefinition of operations.” [OMG, 2007a, 106]

- So, better: call a method sub-type preserving, if and only if it
  (i) accepts more input values (contravariant),
  (ii) on the old values, has fewer behaviour (covariant).

   Note: This (ii) is no longer a matter of simple type-checking!
In the standard, Section 7.3.36, “Operation”:

“Semantic Variation Points

[...] When operations are redefined in a specialization, rules regarding invariance, covariance, or contravariance of types and preconditions determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations.” [OMG, 2007a, 106]

So, better: call a method sub-type preserving, if and only if it

(i) accepts more input values (contravariant),

(ii) on the old values, has fewer behaviour (covariant).

Note: This (ii) is no longer a matter of simple type-checking!

And not necessarily the end of the story:

- One could, e.g. want to consider execution time.
- Or, like [Fischer and Wehrheim, 2000], relax to “fewer observable behaviour”, thus admitting the sub-type to do more work on inputs.

Note: “testing” differences depends on the granularity of the semantics.
**Sub-Typing Principles Cont’d**

- In the standard, Section 7.3.36, “*Operation*”:
  
  “*Semantic Variation Points*
  
  [...] When operations are redefined in a specialization, rules regarding *invariance*, *covariance*, or *contravariance* of types and preconditions determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations.”  
  
  [OMG, 2007a, 106]

- So, better: call a method **sub-type preserving**, if and only if it
  
  (i) accepts **more input values** *(contravariant)*,
  
  (ii) on the **old values**, has **fewer behaviour** *(covariant)*.

  **Note**: This (ii) is no longer a matter of simple type-checking!

- And not necessarily the end of the story:
  
  - One could, e.g. want to consider execution time.
  
  - Or, like [Fischer and Wehrheim, 2000], relax to “fewer observable behaviour”, thus admitting the sub-type to do more work on inputs.

  **Note**: “testing” differences depends on the **granularity** of the semantics.

- **Related**: “has a weaker pre-condition,” *(contravariant)*,
  
  “has a stronger post-condition.” *(covariant)*.
Ensuring Sub-Typing for State Machines

- In the CASE tool we consider, multiple classes in an inheritance hierarchy can have state machines.
Ensuring Sub-Typing for State Machines

• In the CASE tool we consider, multiple classes in an inheritance hierarchy can have state machines.

• But the state machine of a sub-class cannot be drawn from scratch.

• Instead, the state machine of a sub-class can only be obtained by applying actions from a restricted set to a copy of the original one.

Roughly (cf. User Guide, p. 760, for details),

• add things into (hierarchical) states,
• add more states,
• attach a transition to a different target (limited).
Ensuring Sub-Typing for State Machines

- In the CASE tool we consider, multiple classes in an inheritance hierarchy can have state machines.

- But the state machine of a sub-class cannot be drawn from scratch.

- Instead, the state machine of a sub-class can only be obtained by applying actions from a restricted set to a copy of the original one. Roughly (cf. User Guide, p. 760, for details),
  - add things into (hierarchical) states,
  - add more states,
  - attach a transition to a different target (limited).

- They ensure, that the sub-class is a behavioural sub-type of the super class. (But method implementations can still destroy that property.)

- Technically, the idea is that (by late binding) only the state machine of the most specialised classes are running.

By knowledge of the framework, the (code for) state machines of super-classes is still accessible — but using it is hardly a good idea...
Towards System States

Wanted: a formal representation of “if $C \preceq D$ then $D$ ‘is a’ $C$”, that is,

(i) $D$ has the same attributes and behavioural features as $C$, and
(ii) $D$ objects (identities) can replace $C$ objects.
Towards System States

**Wanted**: a formal representation of “if $C \preceq D$ then $D$ ‘is a’ $C$’”, that is,

(i) $D$ has the same attributes and behavioural features as $C$, and

(ii) $D$ objects (identities) can replace $C$ objects.

We’ll discuss **two approaches** to semantics:

- **Domain-inclusion** Semantics  
  (more **theoretical**)

- **Uplink** Semantics  
  (more **technical**)
Domain Inclusion Semantics
Domain Inclusion Structure

Let $\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, \text{atr}, \mathcal{E}, F, \text{mth}, \sqsubseteq)$ be a signature.

Now a structure $\mathcal{D}$

- [as before] maps types, classes, associations to domains,
- [for completeness] methods to transformers,
- [as before] identities of instances of classes not (transitively) related by generalisation are disjoint,

- [changed] the identities of a super-class comprise all identities of sub-classes, i.e.

  $$\forall C \in \mathcal{C} : \mathcal{D}(C) \supseteq \bigcup_{C \sqsubset D} \mathcal{D}(D).$$

**Note:** the old setting coincides with the special case $\sqsubseteq = \emptyset$. 
**Domain Inclusion System States**

**Now:** a **system state** of \(\mathcal{I}\) wrt. \(\mathcal{D}\) is a **type-consistent** mapping

\[
\sigma : \mathcal{D}(C) \rightarrow (V \rightarrow (\mathcal{D}(\mathcal{T}) \cup \mathcal{D}(C_{0,1}) \cup \mathcal{D}(C_*)))
\]

that is, for all \(u \in \text{dom}(\sigma) \cap \mathcal{D}(C)\),

- **[as before]** \(\sigma(u)(v) \in \mathcal{D}(\tau)\) if \(v : \tau, \tau \in \mathcal{T}\) or \(\tau \in \{C_*, C_{0,1}\}\).
- **[changed]** \(\text{dom}(\sigma(u)) = \bigcup_{C_0 \preceq C} \text{atr}(C_0)\).

**Example:**

![Diagram](image)

**Note:** the old setting still coincides with the special case \(\preceq = \emptyset\).
Recall:

- we want to allow, e.g., “context $D\ inv : v < 0$”.
- we assume **fully qualified names**, e.g. $C::v$.

Intuitively, $v$ shall denote the **“most special more general”** $C::v$ according to $\triangleleft$. 
**Preliminaries: Expression Normalisation**

**Recall:**
- we want to allow, e.g., “context $D$ inv : $v < 0$”.
- we assume **fully qualified names**, e.g. $C::v$.

Intuitively, $v$ shall denote the “**most special more general**” $C::v$ according to $\triangleleft$.

To keep this out of typing rules, we assume that the following **normalisation** has been applied to all OCL expressions and all actions.

- Given expression $v$ (or $f$) in **context** of class $D$, as determined by, e.g.
  - by the (type of the) navigation expression prefix, or
  - by the class, the state-machine where the action occurs belongs to,
  - similar for method bodies,
- **normalise** $v$ to ($=$ replace by) $C::v$,
- where $C$ is the **greatest** class wrt. “$\preceq$” such that
  - $C \preceq D$ and $C::v \in \text{atr}(C)$. 
**Preliminaries: Expression Normalisation**

**Recall:**
- we want to allow, e.g., “context $D$ inv : $v < 0$”.
- we assume **fully qualified names**, e.g. $C::v$.

Intuitively, $v$ shall denote the “most special more general” $C::v$ according to $\triangleleft$.

To keep this out of typing rules, we assume that the following **normalisation** has been applied to all OCL expressions and all actions.

- Given expression $v$ (or $f$) in **context** of class $D$, as determined by, e.g.
  - by the (type of the) navigation expression prefix, or
  - by the class, the state-machine where the action occurs belongs to,
  - similar for method bodies,
- **normalise** $v$ to ($= replace by$) $C::v$,
- where $C$ is the **greatest** class wrt. “$\preceq$” such that
  - $C \preceq D$ and $C::v \in atr(C)$.

If no (unique) such class exists, the model is considered **not well-formed**; the expression is ambiguous. Then: explicitly provide the **qualified name**.

![Diagram of class hierarchy]
Recall (part of the) OCL syntax and typing: \( v, r \in V; C, D \in \mathcal{C} \)

\[
\text{expr ::= } v(\text{expr}_1) : \tau_C \rightarrow \tau(v), \quad \text{if } v : \tau \in \mathcal{T} \\
| r(\text{expr}_1) : \tau_C \rightarrow \tau_D, \quad \text{if } r : D_{0,1} \\
| r(\text{expr}_1) : \tau_C \rightarrow \text{Set}(\tau_D), \quad \text{if } r : D_*$
\]

The definition of the semantics remains (textually) the same.
More Interesting: Well-Typed-ness

- We want

  \[
  \text{context } D \quad \text{inv : } v < 0
  \]

to be well-typed.

Currently it isn’t because

\[
v(expr_1) : \tau_C \rightarrow \tau(v)
\]

but \( A \vdash \text{self} : \tau_D \).

(Because \( \tau_D \) and \( \tau_C \) are still different types, although \( \text{dom}(\tau_D) \subset \text{dom}(\tau_C) \).)

- So, add a (first) new typing rule

\[
\begin{align*}
A & \vdash expr : \tau_D, \text{ if } C \preceq D. \\
A & \vdash expr : \tau_C
\end{align*}
\]  \quad \text{(Inh)}

Which is correct in the sense that, if ‘expr’ is of type \( \tau_D \), then we can use it everywhere, where a \( \tau_C \) is allowed.

The system state is prepared for that.
\[
A, D \vdash \text{expr} : \tau_C \\
A, D \vdash C :: v(\text{expr}) : \tau, \quad \xi = +
\] (Pub)

\[
A, D \vdash \text{expr} : \tau_C \\
A, D \vdash C :: v(\text{expr}) : \tau, \quad \xi = \#, \ C \preceq D
\] (Prot)

\[
A, D \vdash \text{expr} : \tau_C \\
A, D \vdash C :: v(\text{expr}) : \tau, \quad \xi = -, \ C = D
\] (Priv)

\[
\langle C :: v : \tau, \xi, v_0, P \rangle \in \text{atr}(C).
\]

**Example:**

<table>
<thead>
<tr>
<th>context/inv</th>
<th>(n.)(v_1 &lt; 0)</th>
<th>(n.)(v_2 &lt; 0)</th>
<th>(n.)(v_3 &lt; 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(D)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Satisfying OCL Constraints (Domain Inclusion)

- Let $\mathcal{M} = (\mathcal{C} \mathcal{D}, \mathcal{O} \mathcal{D}, \mathcal{S} \mathcal{M}, \mathcal{I})$ be a UML model, and $\mathcal{D}$ a structure.

- We (continue to) say $\mathcal{M} \models expr$ for context $C$ inv : $expr_0 \in Inv(\mathcal{M})$ iff

\[
\forall \pi = (\sigma_i, \varepsilon_i)_{i \in \mathbb{N}} \in [\mathcal{M}] \quad \forall i \in \mathbb{N} \quad \forall u \in \text{dom}(\sigma_i) \cap \mathcal{D}(C) : I[(expr_0)](\sigma_i, \{\text{self} \mapsto u\}) = 1.
\]

- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $Inv(\mathcal{M})$.

- Example:

```
  0, 1
  n
  x : Int

  \node (n1) at (0, 0) {$D$};
  \node (n2) at (0, 1) {$x$};
  \draw[->] (n1) -- (n2);
  \node (n3) at (0, 2) {$C$};
  \draw[->] (n2) -- (n3);
```
Transformers (Domain Inclusion)

- Transformers also remain the same, e.g. [VL 12, p. 18]

\[ \text{update}(expr_1, v, expr_2) : (\sigma, \varepsilon) \mapsto (\sigma', \varepsilon) \]

with

\[ \sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[expr_2]\sigma]] \]

where \( u = I[expr_1]\sigma \).
Semantics of Method Calls

- **Non late-binding**: clear, by normalisation.
- **Late-binding**: Construct a *method call* transformer, which is applied to all method calls.
**Wanted**: triggers shall also be sensitive for inherited events, sub-class shall execute super-class’ state-machine (unless overridden).

\[(\sigma, \varepsilon) \xrightarrow{\text{cons, Snd}} (\sigma', \varepsilon') \text{ if}\]

- \(\exists u \in \text{dom}(\sigma) \cap \mathcal{D}(C) \exists u_E \in \mathcal{D}(\varepsilon) : u_E \in \text{ready}(\varepsilon, u)\)
- \(u\) is stable and in state machine state \(s\), i.e. \(\sigma(u)(\text{stable}) = 1\) and \(\sigma(u)(\text{st}) = s\)
- a transition is enabled, i.e.
  \[
  \exists (s, F, expr, act, s') \in \leftrightarrow (\mathcal{SM}_C) : F = E \land I[expr](\tilde{\sigma}) = 1
  \]
  where \(\tilde{\sigma} = \sigma[u.params_E \mapsto u_e]\).

and

- \((\sigma', \varepsilon')\) results from applying \(t_{\text{act}}\) to \((\sigma, \varepsilon)\) and removing \(u_E\) from the ether, i.e.
  \[
  (\sigma'', \varepsilon') = t_{\text{act}}(\tilde{\sigma}, \varepsilon \oplus u_E),
  \]
  \[
  \sigma' = (\sigma''[u.\text{st} \mapsto s', u.\text{stable} \mapsto b, u.params_E \mapsto \emptyset])|_{\mathcal{D}(\varepsilon) \setminus \{u_E\}}
  \]
  where \(b\) depends:
  - If \(u\) becomes stable in \(s'\), then \(b = 1\). It **does** become stable if and only if there is no transition without trigger enabled for \(u\) in \((\sigma', \varepsilon')\).
  - Otherwise \(b = 0\).
  - Consumption of \(u_E\) and the side effects of the action are observed, i.e.
    \[
    \text{cons} = \{(u, (E, \sigma(u_E)))\}, \text{Snd} = \text{Obs}_{t_{\text{act}}}(\tilde{\sigma}, \varepsilon \oplus u_E).
    \]
• Similar to satisfaction of OCL expressions above:
  • An instance line stands for all instances of $C$ (exact or inheriting).
  • Satisfaction of event observation has to take inheritance into account, too, so we have to fix, e.g.

$$\sigma, \text{cons}, \text{Snd} \models_{\beta} E_{x,y}^!$$

if and only if

$$\beta(x)$$ sends an $F$-event to $\beta y$ where $E \preceq F$.

• **Note**: $C$-instance line also binds to $C'$-objects.
Uplink Semantics
• **Idea:**
  
  - Continue with the existing definition of **structure**, i.e. disjoint domains for identities.
  - Have an **implicit association** from the child to each parent part (similar to the implicit attribute for stability).

![Diagram](image)

- Apply (a different) pre-processing to make appropriate use of that association, e.g. rewrite (C++)

\[ x = 0; \]

in \( D \) to

\[ \text{uplink}_C \rightarrow x = 0; \]
Pre-Processing for the Uplink Semantics

• For each pair \( C \prec D \), extend \( D \) by a (fresh) association

\[
uplink_C : C \text{ with } \mu = [1, 1], \ \xi = +
\]

(Exercise: public necessary?)

• Given expression \( v \) (or \( f \)) in the context of class \( D \),
  • let \( C \) be the smallest class wrt. “\( \preceq \)” such that
    • \( C \preceq D \), and
    • \( C::v \in \text{atr}(D) \)
  • then there exists (by definition) \( C \prec C_1 \prec \ldots \prec C_n \prec D \),
  • normalise \( v \) to (= replace by)

\[
uplink_{C_n} \rightarrow \ldots \rightarrow uplink_{C_1}.C::v
\]

• Again: if no (unique) smallest class exists, the model is considered not well-formed; the expression is ambiguous.
Uplink Structure, System State, Typing

- Definition of structure remains unchanged.
- Definition of system state remains unchanged.
- Typing and transformers remain unchanged — the preprocessing has put everything in shape.
Satisfying OCL Constraints (Uplink)

- Let \( \mathcal{M} = (\mathcal{C} \mathcal{D}, \mathcal{O} \mathcal{D}, \mathcal{I} \mathcal{M}, \mathcal{I}) \) be a UML model, and \( \mathcal{D} \) a structure.

- We (continue to) say

  \[
  \mathcal{M} \models expr
  \]

  for

  \[
  \underbrace{\text{context } C \ \ \text{inv} : expr_0 \in Inv(\mathcal{M})}_{=expr}
  \]

  if and only if

  \[
  \forall \pi = (\sigma_i)_{i \in \mathbb{N}} \in \lbrack \mathcal{M} \rbrack
  \]

  \[
  \forall i \in \mathbb{N}
  \]

  \[
  \forall u \in \text{dom}(\sigma_i) \cap \mathcal{D}(C) :
  \]

  \[
  I[expr_0](\sigma_i, \{\text{self} \mapsto u\}) = 1.
  \]

- \( \mathcal{M} \) is (still) consistent if and only if it satisfies all constraints in \( Inv(\mathcal{M}) \).
Transformers (Uplink)

- What **has to change** is the **create** transformer:

\[ create(C, expr, v) \]

- Assume, \( C \)'s inheritance relations are as follows.

\[
\begin{align*}
C_{1,1} \triangleleft \ldots \triangleleft C_{1,n_1} \triangleleft C, \\
\ldots \\
C_{m,1} \triangleleft \ldots \triangleleft C_{m,n_m} \triangleleft C.
\end{align*}
\]

- Then, we have to
  - create one fresh object for each part, e.g.

\[
\begin{aligned}
u_{1,1}, \ldots, u_{1,n_1}, \ldots, u_{m,1}, \ldots, u_{m,n_m},
\end{aligned}
\]
  - set up the uplinks recursively, e.g.

\[
\sigma(u_{1,2})(\text{uplink}_{C_{1,1}}) = u_{1,1}.
\]
  - And, if we had constructors, be careful with their order.
Late Binding (Uplink)

• Employ something similar to the “mostspec” trick (in a minute!). But the result is typically far from concise.

(Related to OCL’s isKindOf() function, and RTTI in C++.)
Domain Inclusion vs. Uplink Semantics
Cast-Transformers

- C c;
- D d;

**Identity upcast** (C++):
- C* cp = &d;  \(\text{// assign address of 'd' to pointer 'cp'}\)

**Identity downcast** (C++):
- D* dp = (D*)cp;  \(\text{// assign address of 'd' to pointer 'dp'}\)

**Value upcast** (C++):
- *c = *d;  \(\text{// copy attribute values of 'd' into 'c', or,}\)
  \(\text{// more precise, the values of the C-part of 'd'}\)
# Casts in Domain Inclusion and Uplink Semantics

<table>
<thead>
<tr>
<th>Domain Inclusion</th>
<th>Uplink</th>
</tr>
</thead>
</table>
| \( C \ast \text{cp} = \& d \); | \textbf{easy}: immediately compatible (in underlying system state) because \( \& d \) yields an identity from \( \mathcal{D}(D) \subset \mathcal{D}(C) \).
| \( D \ast \text{dp} = (D \ast \text{cp}) \); | \textbf{easy}: the value of \( \text{cp} \) is in \( \mathcal{D}(D) \cap \mathcal{D}(C) \) because the pointed-to object is a \( D \).
| \( c = d \); | \textbf{bit difficult}: set (for all \( C \preceq D \)) \( (C)(\cdot, \cdot): \tau_D \times \Sigma \rightarrow \Sigma|_{\text{atr}(C)} \)
| | \( (u, \sigma) \mapsto \sigma(u)|_{\text{atr}(C)} \)
| | Note: \( \sigma' = \sigma[u_C \mapsto \sigma(u_D)] \) is not type-compatible! |
Identity Downcast with Uplink Semantics

- **Recall** (C++): \( D \ d; \ C^* \ cp = &d; \ D^* \ dp = (D^*)cp; \)
- **Problem**: we need the identity of the \( D \) whose \( C \)-slice is denoted by \( cp \).
- **One technical solution**:
  - Give up disjointness of domains for one additional type comprising all identities, i.e. have
    \[
    \text{all} \in \mathcal{T}, \quad \mathcal{D}(\text{all}) = \bigcup_{C \in \mathcal{C}} \mathcal{D}(C)
    \]
  - In each \( \preceq \)-minimal class have associations “mostspec” pointing to most specialised slices, plus information of which type that slice is.
  - Then downcast means, depending on the mostspec type (only finitely many possibilities), going down and then up as necessary, e.g.

    ```
    switch(mostspec_type){
      case C:
        dp = cp -> mostspec -> uplink_{D_n} -> ... -> uplink_{D_1} -> uplink_{D};
      ...
    }
    ```
Domain Inclusion vs. Uplink Semantics: Differences

- **Note:** The uplink semantics views inheritance as an abbreviation:
  - We only need to touch transformers (create) — and if we had constructors, we didn’t even needed that (we could encode the recursive construction of the upper slices by a transformation of the existing constructors.)

- **So:**
  - Inheritance **doesn’t add** expressive power.
  - And it also **doesn’t improve** conciseness **soo dramatically.**

As long as we’re “**early binding**”, that is...
Domain Inclusion vs. Uplink Semantics: Motives

- Exercise:

What's the point of

- having the tedious adjustments of the theory if it can be approached technically?
- having the tedious technical pre-processing if it can be approached cleanly in the theory?
Meta-Modelling: Idea and Example
**Meta-Modelling: Why and What**

- **Meta-Modelling** is one major prerequisite for understanding
  - the standard documents [OMG, 2007a, OMG, 2007b], and
  - the MDA ideas of the OMG.

- The idea is **simple**:  
  - if a **modelling language** is about modelling **things**,  
  - and if UML models are and comprise **things**,  
  - then why not **model** those in a modelling language?
Meta-Modelling: Why and What

- **Meta-Modelling** is one major prerequisite for understanding
  - the standard documents [OMG, 2007a, OMG, 2007b], and
  - the MDA ideas of the OMG.

- The idea is **simple**:
  - if a **modelling language** is about modelling **things**, 
  - and if UML models are and comprise **things**, 
  - then why not **model** those in a modelling language?

- In other words:
  Why not have a model $\mathcal{M}_U$ such that
  - the set of legal instances of $\mathcal{M}_U$ 
    is
  - the set of well-formed (!) UML models.
Meta-Modelling: Example

- For example, let’s consider a class.

- A **class** has (on a superficial level)
  - a **name**,  
  - any number of **attributes**,  
  - any number of **behavioural features**.

Each of the latter two has
- a **name** and  
- a **visibility**.

Behavioural features in addition have
- a boolean attribute **isQuery**,  
- any number of parameters,  
- a return type.

- Can we model this (in UML, for a start)?
Figure 7.12 - Classes diagram of the Kernel package
Figure 7.11 - Operations diagram of the Kernel package
Figure 7.10 - Features diagram of the Kernel package
Figure 7.9 - Classifiers diagram of the Kernel package
Figure 7.4 - Namespaces diagram of the Kernel package
Figure 7.3 - Root diagram of the Kernel package
Figure 13.6 - Common Behavior
Meta-modelling has already been used for UML 1.x.

For UML 2.0, the request for proposals (RFP) asked for a separation of concerns: **Infrastructure** and **Superstructure**.

One reason: sharing with MOF (see later) and, e.g., CWM.

Figure 0-1 *Overview of architecture*
Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure
Meta-Modelling: Principle
Modelling vs. Meta-Modelling

\[ \mathcal{I} = (\{Z\}, \{C\}, \{v\}, \{C \mapsto v\}) \]

Model (M1)

\[
\begin{array}{c|c}
C & \mathcal{I} \\
\hline
v : Z & \\
\end{array}
\]

\[ \mathcal{D} \leadsto \Sigma_{\mathcal{I}} \]
Modelling vs. Meta-Modelling

\[ \mathcal{J} = (\{\mathbb{Z}\}, \{C\}, \{v\}, \{C \leftrightarrow v\}), \mathcal{D} \sim \Sigma_{\mathcal{J}} \]

\[ \sigma = \{u \mapsto \{v \mapsto 0\}\} \]
Modelling vs. Meta-Modelling

\[ C \]
\[ v : \mathbb{Z} \]

\[ \text{Class} \]
\[ \text{name} = C \]

\[ \text{Property} \]
\[ \text{name} = v \]

\[ \text{Type} \]
\[ \text{name} = \mathbb{Z} \]

\[ I = (\{\mathbb{Z}\}, \{C\}, \{v\}, \{C \mapsto v\}), \mathcal{I} \sim \Sigma_{\mathcal{I}} \]

\[ \sigma = \{u \mapsto \{v \mapsto 0\}\} \]
Modelling vs. Meta-Modelling

Meta-Model (M2)

Model (M1)

Instance (M0)

\[ \mathcal{I} = (\{Z\}, \{C\}, \{v\}, \{C \mapsto v\}), \mathcal{D} \sim \Sigma \mathcal{I} \]

\[ \sigma = \{u \mapsto \{v \mapsto 0\}\} \]
Modelling vs. Meta-Modelling

- So, if we have a meta model $\mathcal{M}_U$ of UML, then the set of UML models is the set of instances of $\mathcal{M}_U$.

- A UML model $\mathcal{M}$ can be represented as an object diagram (or system state) wrt. the meta-model $\mathcal{M}_U$.

- Other view: An object diagram wrt. meta-model $\mathcal{M}_U$ can (alternatively) be rendered as the UML model $\mathcal{M}$. 

\[ \mathcal{I} = \{ \{ Z \}, \{ C \}, \{ v \}, \{ C \mapsto v \} \}, \quad \mathcal{D} \sim \sum_\mathcal{I} \]

\[ \sigma = \{ u \mapsto \{ v \mapsto 0 \} \} \]
The set of well-formed UML models can be defined as the set of object diagrams satisfying all constraints of the meta-model.

For example,

```
"[2] Generalization hierarchies must be directed and acyclic. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.

not self . allParents() -> includes(self)"  [OMG, 2007b, 53]
```
The set of **well-formed UML models** can be defined as the set of object diagrams satisfying all constraints of the **meta-model**.

For example,

“[2] Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.

\[
\text{not self . allParents()} \rightarrow \text{includes(self)}
\]  

[OMG, 2007b, 53]

The other way round:

Given a **UML model** \( \mathcal{M} \), unfold it into an object diagram \( O_1 \) wrt. \( \mathcal{M}_U \).

If \( O_1 \) is a **valid** object diagram of \( \mathcal{M}_U \) (i.e. satisfies all invariants from \( \text{Inv}(\mathcal{M}_U) \)), then \( \mathcal{M} \) is a well-formed UML model.
The set of well-formed UML models can be defined as the set of object diagrams satisfying all constraints of the meta-model.

For example,

“[2] Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.

\[
\text{not self . allParents()} \rightarrow \text{includes(self)}
\]  

[OMG, 2007b, 53]

The other way round:

Given a UML model \( M \), unfold it into an object diagram \( O_1 \) wrt. \( M_U \). If \( O_1 \) is a valid object diagram of \( M_U \) (i.e. satisfies all invariants from \( \text{Inv}(M_U) \)), then \( M \) is a well-formed UML model.

That is, if we have an object diagram validity checker for of the meta-modelling language, then we have a well-formedness checker for UML models.
Table of Contents

1. Scope ................................................................. 1

2. Conformance ....................................................... 1
   2.1 Language Units ............................................... 2
   2.2 Compliance Levels .......................................... 2
   2.3 Meaning and Types of Compliance ...................... 6
   2.4 Compliance Level Contents .................................. 8

3. Normative References ........................................... 10

4. Terms and Definitions .......................................... 10

5. Symbols ............................................................. 10

6. Additional Information .......................................... 10
   6.1 Changes to Adopted OMG Specifications ................ 10
   6.2 Architectural Alignment and MDA Support .............. 10
   6.3 On the Run-Time Semantics of UML ....................... 11
      6.3.1 The Basic Premises ................................... 11
      6.3.2 The Semantics Architecture ......................... 11
      6.3.3 The Basic Causality Model .......................... 12
      6.3.4 Semantics Descriptions in the Specification ....... 13
   6.4 The UML Metamodel ......................................... 13
      6.4.1 Models and What They Model ....................... 13
      6.4.2 Semantic Levels and Naming ....................... 14
   6.5 How to Read this Specification ............................ 15
      6.5.1 Specification format ................................... 15
      6.5.2 Diagram format ....................................... 18
   6.6 Acknowledgements ............................................ 19

Part I - Structure .................................................. 21

7. Classes ............................................................... 23
7.3.47 Relationship (from Kernel) ................................................................. 132
7.3.48 Slot (from Kernel) ................................................................................ 132
7.3.49 StructuralFeature (from Kernel) .......................................................... 133
7.3.50 Substitution (from Dependencies) ......................................................... 134
7.3.51 Type (from Kernel) ............................................................................... 135
7.3.52 TypedElement (from Kernel) ................................................................. 136
7.3.53 Usage (from Dependencies) ................................................................. 137
7.3.54 ValueSpecification (from Kernel) ......................................................... 137
7.3.55 Visibility (from Kernel) ......................................................................... 139
8. Components ......................................................................................... 143
8.1 Overview .............................................................................................. 143
8.2 Abstract syntax ..................................................................................... 144
8.3 Class Descriptions ................................................................................ 146
8.3.1 Component (from BasicComponents, PackagingComponents) .......... 146
8.3.2 Connector (from BasicComponents) ...................................................... 154
8.3.3 ConnectorKind (from BasicComponents) ............................................. 157
8.3.4 ComponentRealization (from BasicComponents) ................................. 157
8.4 Diagrams ............................................................................................. 159
9. Composite Structures .......................................................................... 161
9.1 Overview .............................................................................................. 161
9.2 Abstract syntax ..................................................................................... 161
9.3 Class Descriptions ................................................................................ 166
9.3.1 Class (from StructuredClasses) ........................................................... 166
9.3.2 Classifier (from Collaborations) ......................................................... 167
9.3.3 Collaboration (from Collaborations) ..................................................... 168
9.3.4 CollaborationUse (from Collaborations) ............................................. 171
9.3.5 ConnectableElement (from InternalStructures) .................................... 174
9.3.6 Connector (from InternalStructures) ................................................... 174
9.3.7 ConnectorEnd (from InternalStructures, Ports) ................................. 176
9.3.8 EncapsulatedClassifier (from Ports) ................................................... 178
9.3.9 InvocationAction (from InvocationActions) ....................................... 178
9.3.10 Parameter (from Collaborations) ....................................................... 179
9.3.11 Port (from Ports) ................................................................................. 179
9.3.12 Property (from InternalStructures) ................................................... 183
9.3.13 StructuredClassifier (from InternalStructures) ................................. 186
9.3.14 Trigger (from InvocationActions) ...................................................... 190
9.3.15 Variable (from StructuredActivities) ................................................ 191
9.4 Diagrams ............................................................................................. 191
10. Deployments ....................................................................................... 193
7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)

A classifier is a classification of instances, it describes a set of instances that have features in common.

Generalizations

- “Namespace (from Kernel)” on page 99
- “RedefinableElement (from Kernel)” on page 130
- “Type (from Kernel)” on page 135

Description

A classifier is a namespace whose members can include features. Classifier is an abstract metaclass.

A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to other classifiers. A classifier can specify a generalization hierarchy by referencing its general classifiers.

A classifier is a redefinable element, meaning that it is possible to redefine nested classifiers.

Attributes

- isAbstract: Boolean
  - If true, the Classifier does not provide a complete declaration and can typically not be instantiated. An abstract classifier is intended to be used by other classifiers (e.g., as the target of general metarelationships or generalization relationships). Default value is false.

Associations

- /attribute: Property [*]
  - Refers to all of the Properties that are direct (i.e., not inherited or imported) attributes of the classifier. Subsets Classifier::feature and is a derived union.
- / feature : Feature [*]
  - Specifies each feature defined in the classifier. Subsets Namespace::member. This is a derived union.
- / general : Classifier[*]
  - Specifies the general Classifiers for this Classifier. This is derived.
7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)

A classifier is a classification of instances, it describes a set of instances that have features in common.

Generalizations

- "Namespacem (from Kernel)"
- "RedefinableElement (from Kernel)"
- "Type (from Kernel)"

Description

A classifier is a namespace whose members can include features. Classifier is an abstract meta-class.

A classifier is a redefinable element, meaning that it is possible to redefine nested classifiers, thereby making it possible to define generalization relationships to other classifiers.

A classifier can specify a generalization hierarchy by referencing its general classifiers.

Attributes

- isAbstract: Boolean
  If true, this classification is not complete and cannot be instantiated. An abstract classifier is intended to be used by other classifiers (e.g., as the target of generalization or generalization relationships).
  Default value is false.

Associations

- /attribute: Property[*]
  Refers to all of the Properties that are direct (i.e., not inherited or imported) attributes of the classifier.

- /feature: Feature[*]
  Specifies each feature defined in the classifier. This is a derived union.

- /general: Classifier[*]
  Specifies the general classifiers for this classifier. This is derived.

- /inheritedMember: NamedElement[*]
  Specifies all elements inherited by this classifier from the general classifiers.

- /redefinedClassifier: Classifier[*]
  References the classifiers that are redefined by this classifier.

Package Dependencies

- substitution: Substitution
  References the substitutions that are owned by this classifier.

Package PowerTypes

- powertypeExtent: GeneralizationSet
  Designates the GeneralizationSet of which the associated Classifier is a power type.

Constraints

1. The general classifiers are the classifiers referenced by the generalization relationships.
   
   general = self.parents()

2. Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.
   
   not self.allParents()->includes(self)

3. A classifier may only specialize classifiers of a valid type.
   
   self.parents()->forall(c | self.maySpecializeType(c))

4. The inheritedMember association is derived by inheriting the inheritable members of the parents.
   
   self.inheritedMember->includesAll(self.inherit(self.parents()->collect(p | p.inheritableMembers(self))))

Package PowerTypes

5. The Classifier that maps to a GeneralizationSet may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that GeneralizationSet. In other words, a power type may not be an instance of itself nor may its instances also be its subclasses.

Additional Operations

1. The query allFeatures() gives all of the features in the namespace of the classifier. In general, through mechanisms such as inheritance, this will be a larger set than feature.
   
   Classifier::allFeatures(): Set(Feature);

2. The query parents() gives all of the immediate ancestors of a generalized Classifier.
   
   Classifier::parents(): Set(Classifier);
   
   parents = generalization.general
The query `allParents()` gives all of the direct and indirect ancestors of a generalized Classifier.

```
Classiifie r::allParents(): Set(Classifie r);
allParents = self.parents()->union(self.parents())->collect(p | p.allParents())
```

4. The query `inheritableMembers()` gives all of the members of a classifier that may be inherited in one of its descendants, subject to whatever visibility restrictions apply.

```
Classiifie r::inheritableMembers(c: Classifie r): Set(NamedElement);
pre: c.allParents()->includes(self)
inheritableMembers = member->select(m | c.hasVisibilityOf(m))
```

5. The query `hasVisibilityOf()` determines whether a named element is visible in the classifier. By default all are visible. It is only called when the argument is something owned by a parent.

```
Classiifie r::hasVisibilityOf(n: NamedElement): Boolean;
pre: self.allParents()->includes(n)
if (self.inheritableMember->includes(n)) then
  hasVisibilityOf = (n.visibility <> #private)
else
  hasVisibilityOf = true
```

6. The query `conformsTo()` gives true for a classifier that defines a type that conforms to another. This is used, for example, in the specification of signature conformance for operations.

```
Classiifie r::conformsTo(other: Classifie r): Boolean;
conformsTo = (self=other) or (self.allParents()->includes(other))
```

7. The query `inherit()` defines how to inherit a set of elements. Here the operation is defined to inherit them all. It is intended to be redefined in circumstances where inheritance is affected by redefinition.

```
Classiifie r::inherit(inhs: Set(NamedElement));
inherit = inhs
```

8. The query `maySpecializeType()` determines whether this classifier may have a generalization relationship to classifiers of the specified type. By default a classifier may specialize classifiers of the same or a more general type. It is intended to be redefined by classifiers that have different specialization constraints.

```
Classiifie r::maySpecializeType(c: Classifier): Boolean;
maySpecializeType = self.oclIsKindOf(c.oclType)
```

A classifier is a classification of instances according to their features.

A classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is also an (indirect) instance of each of the general Classifiers. Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier.

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a Classifier have values corresponding to the Classifier’s attributes.

A Classifier defines a type. Type conformance between generalizable Classifiers is defined so that a Classifier conforms to itself and to all of its ancestors in the generalization hierarchy.
Reading the Standard Cont’d

The query allParents () gives all of the direct and indirect ancestors of a generalized Classifier.

In essence, then, a power type is a class whose instances are subclasses. The powertypeExtent association relates a Classifier with a set of generalizations that a) have a common specific Classifier, and b) represent a collection of subsets for that class.

The query allFeatures () gives all of the features in the namespace of the Classifier. In general, through mechanisms such as

The precise lifecycle semantics of aggregation is a semantic variation point.

The classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier.

A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is to itself and to all of its ancestors in the generalization hierarchy.

Any compartment may be suppressed. A separator line is not drawn for a suppressed compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it. Compartment names can be used to remove ambiguity, if necessary.

An abstract Classifier can be shown using the keyword [abstract] after or before the name of the Classifier.

The type, visibility, default, multiplicity, property string may be suppressed from being displayed, even if there are values in the model.

The individual properties of an attribute can be shown in columns rather than as a continuous string.

Style Guidelines

- Attribute names typically begin with a lowercase letter. Multi-word names are often formed by concatenating the words and using lowercase for all letters except for upcasing the first letter of each word but the first.
- Center the name of the Classifier in boldface.
- Center keyword (including stereotype names) in plain face within guillemets above the classifier name.
- For those languages that distinguish between uppercase and lowercase characters, capitalize names (i.e., begin them with an uppercase character).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Show full attributes and operations when needed and suppress them in other contexts or references.

UML Superstructure Specification, v2.1.2
The query allParents () gives all of the direct and indirect ancestors of a generalized Classifier.

• generalization: Generalization[*] a Classifier with a set of generalizations that a) have a common specific Classifier, and b) represent a collection of subsets for that class.

ClassA::area is a derived attribute with type Integer. It is marked as read-only.

ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle.

An attribute can be shown in guillemets above the name. Some specializations of Classifier have their own distinct notations. The name of an attribute can be defined in one place a default notation available for any concrete subclass of Classifier for which this notation is suitable. The precise lifecycle semantics of aggregation is a semantic variation point.

Figure 7.29 - Classifier notation: attributes and operations grouped according to visibility

Figure 7.30 - Examples of attributes

Figure 7.31 - Association-like notation for attribute
Reading the Specification

Package PowerTypes

For example, a Bank Account Type classifier could have a powertype association with a GeneralizationSet. This GeneralizationSet could then associate with two Generalizations where the class (i.e., general Classifier) Bank Account has two specific subclasses (i.e., Classifiers): Checking Account and Savings Account. Checking Account and Savings Account, then, are instances of the power type: Bank Account Type. In other words, Checking Account and Savings Account are both: instances of Bank Account Type, as well as subclasses of Bank Account. (For more explanation and examples, see Examples in the GeneralizationSet subclause, below.)

7.3.9 Comment (from Kernel)

A comment is a textual annotation that can be attached to a set of elements.

Generalizations

- “Element (from Kernel)” on page 64.

Description

A comment gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

A comment can be owned by any element.

Attributes

- multiplicity:body: String [0..1]
  Specifies a string that is the comment.

Associations

- annotatedElement: Element[*]
  References the Element(s) being commented.

Constraints

No additional constraints

Semantics

A Comment adds no semantics to the annotated elements, but may represent information useful to the reader of the model.

Notation

A Comment is shown as a rectangle with the upper right corner bent (this is also known as a “note symbol”). The rectangle contains the body of the Comment. The connection to each annotated element is shown by a separate dashed line.

Presentation Options

The dashed line connecting the note to the annotated element(s) may be suppressed if it is clear from the context, or not important in this diagram.

UML Superstructure Specification, v2.1.2

57
Meta Object Facility (MOF)
Open Questions...

- Now you’ve been “tricked” again. Twice.
  - We didn’t tell what the modelling language for meta-modelling is.
  - We didn’t tell what the is-instance-of relation of this language is.

- Idea: have a minimal object-oriented core comprising the notions of class, association, inheritance, etc. with “self-explaining” semantics.

- This is Meta Object Facility (MOF), which (more or less) coincides with UML Infrastructure [OMG, 2007a].

- So: things on meta level
  - M0 are object diagrams/system states
  - M1 are words of the language UML
  - M2 are words of the language MOF
  - M3 are words of the language ...
MOF Semantics

- One approach:
  - Treat it with our signature-based theory
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn’t have a notion of Signal, our signature has.)
MOF Semantics

- One approach:
  - Treat it with our signature-based theory
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn’t have a notion of Signal, our signature has.)

- Other approach:
  - Define a generic, graph based “is-instance-of” relation.
  - Object diagrams (that are graphs) then are the system states — not only graphical representations of system states.
**MOF Semantics**

- One approach:
  - Treat it with *our signature-based theory*
  
  This is (in effect) the right direction, but may require new (or extended) signatures for each level.  
  (For instance, MOF doesn’t have a notion of Signal, our signature has.)

- Other approach:
  - Define a *generic, graph based* “is-instance-of” relation.
  
  Object diagrams (that *are* graphs) then *are* the system states — not *only graphical representations* of system states.

  - If this works out, good: We can easily experiment with different language designs, e.g. different flavours of UML that immediately have a semantics.
MOF Semantics

- One approach:
  - Treat it with **our signature-based theory**
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn’t have a notion of Signal, our signature has.)

- Other approach:
  - Define a **generic, graph based** “is-instance-of” relation.
  - Object diagrams (that **are** graphs) then **are** the system states — not **only graphical representations** of system states.

  - If this works out, good: We can easily experiment with different language designs, e.g. different flavours of UML that immediately have a semantics.

  - Most interesting: also do generic definition of behaviour within a closed modelling setting, but this is clearly still research, e.g.
    [Buschermöhle and Oelerink, 2008]
Meta-Modelling: (Anticipated) Benefits
Benefits: Overview

- We’ll (superficially) look at three aspects:
  - Benefits for **Modelling Tools**.
  - Benefits for **Language Design**.
  - Benefits for **Code Generation and MDA**.
Benefits for Modelling Tools

- The meta-model $\mathcal{M}_U$ of UML immediately provides a data-structure representation for the abstract syntax ($\sim$ for our signatures).

If we have code generation for UML models, e.g. into Java, then we can immediately represent UML models in memory for Java.

(Because each MOF model is in particular a UML model.)

- There exist tools and libraries called MOF-repositories, which can generically represent instances of MOF instances (in particular UML models).

And which can often generate specific code to manipulate instances of MOF instances in terms of the MOF instance.
• And not only **in memory**, if we can represent MOF instances in files, we obtain a canonical representation of UML models **in files**, e.g. in XML.
→ XML Metadata Interchange (XMI)
• And not only in memory, if we can represent MOF instances in files, we obtain a canonical representation of UML models in files, e.g. in XML.

→ XML Metadata Interchange (XMI)

• Note: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) → OMG Diagram Interchange.
Benefits for Modelling Tools Cont’d

- And not only in memory, if we can represent MOF instances in files, we obtain a canonical representation of UML models in files, e.g. in XML.
  → XML Metadata Interchange (XMI)

- Note: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) → OMG Diagram Interchange.

- Note: There are slight ambiguities in the XMI standard.
  And different tools by different vendors often seem to lie at opposite ends on the scale of interpretation. Which is surely a coincidence.
  In some cases, it’s possible to fix things with, e.g., XSLT scripts, but full vendor independence is today not given.
  Plus XMI compatibility doesn’t necessarily refer to Diagram Interchange.
• And not only *in memory*, if we can represent MOF instances in files, we obtain a canonical representation of UML models *in files*, e.g. in XML.

→ XML Metadata Interchange (XMI)

• **Note**: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) → OMG Diagram Interchange.

• **Note**: There are slight ambiguities in the XMI standard.

And different tools by different vendors often seem to lie at opposite ends on the scale of interpretation. Which is surely a coincidence.

In some cases, it’s possible to fix things with, e.g., XSLT scripts, but full vendor independence is today not given.

Plus XMI compatibility doesn’t necessarily refer to Diagram Interchange.

• **To re-iterate**: this is *generic for all* MOF-based modelling languages such as UML, CWM, etc.

And also for *Domain Specific Languages* which don’t even exit yet.
Benefits: Overview

We’ll (superficially) look at three aspects:

- Benefits for **Modelling Tools**.
- Benefits for **Language Design**.
- Benefits for **Code Generation and MDA**.
Benefits for Language Design

- Recall: we said that code-generators are possible “readers” of stereotypes.

- For example, (heavily simplifying) we could
  - introduce the stereotypes **Button**, **Toolbar**, ...
  - for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
  - instruct the code-generator to automatically add inheritance from `Gtk::Button`, `Gtk::Toolbar`, etc. **corresponding** to the stereotype.

One mechanism to define DSLs (based on UML, and “within” UML): **Profiles**.
Benefits for Language Design

- Recall: we said that code-generators are possible “readers” of stereotypes.

- For example, (heavily simplifying) we could
  - introduce the stereotypes **Button**, **Toolbar**, ...
  - for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
  - instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. **corresponding** to the stereotype.

  Et voilà: we can model Gtk-GUIs and generate code for them.

One mechanism to define DSLs (based on UML, and “within” UML): **Profiles**.
Recall: we said that code-generators are possible “readers” of stereotypes.

For example, (heavily simplifying) we could

- introduce the stereotypes Button, Toolbar, ...
- for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
- instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. corresponding to the stereotype.

\textit{Et voilà}: we can model Gtk-GUIs and generate code for them.

Another view:

- UML with these stereotypes is a new modelling language: Gtk-UML.
- Which lives on the same meta-level as UML (M2).
- It’s a Domain Specific Modelling Language (DSL).

One mechanism to define DSLs (based on UML, and “within” UML): Profiles.
For each DSL defined by a Profile, we immediately have
- in memory representations,
- modelling tools,
- file representations.

Note: here, the semantics of the stereotypes (and thus the language of Gtk-UML) lies in the code-generator.

That’s the first “reader” that understands these special stereotypes. (And that’s what’s meant in the standard when they’re talking about giving stereotypes semantics).

One can also impose additional well-formedness rules, for instance that certain components shall all implement a certain interface (and thus have certain methods available). (Cf. [Stahl and Völter, 2005].)
Benefits for Language Design Cont’d

- One step further:
  - Nobody hinders us to obtain a model of UML (written in MOF),
  - throw out parts unnecessary for our purposes,
  - add (= integrate into the existing hierarchy) more adequate new constructs, for instance, contracts or something more close to hardware as interrupt or sensor or driver,
  - and maybe also stereotypes.

→ a new language standing next to UML, CWM, etc.

- Drawback: the resulting language is not necessarily UML any more, so we can’t use proven UML modelling tools.

- But we can use all tools for MOF (or MOF-like things).
  For instance, Eclipse EMF/GMF/GEF.
Benefits: Overview

- We’ll (superficially) look at three aspects:
  - Benefits for **Modelling Tools**.
  - Benefits for **Language Design**.
  - Benefits for **Code Generation and MDA**.
There are manifold applications for model-to-model transformations:

For instance, tool support for re-factorings, like moving common attributes upwards the inheritance hierarchy.

This can now be defined as graph-rewriting rules on the level of MOF.

The graph to be rewritten is the UML model.
Benefits for Model (to Model) Transformation

- There are manifold applications for model-to-model transformations:
  - For instance, tool support for re-factorings, like moving common attributes upwards the inheritance hierarchy.
    This can now be defined as graph-rewriting rules on the level of MOF.
    The graph to be rewritten is the UML model
  - Similarly, one could transform a Gtk-UML model into a UML model, where the inheritance from classes like Gtk::Button is made explicit:
    The transformation would add this class Gtk::Button and the inheritance relation and remove the stereotype.
There are manifold applications for model-to-model transformations:

- For instance, tool support for **re-factorings**, like moving common attributes upwards the inheritance hierarchy.

  This can now be defined as **graph-rewriting** rules on the level of MOF.

  The graph to be rewritten is the UML model.

- Similarly, one could transform a **Gtk-UML** model into a **UML model**, where the inheritance from classes like Gtk::Button is made explicit:

  The transformation would add this class Gtk::Button and the inheritance relation and remove the stereotype.

- Similarly, one could have a **GUI-UML** model transformed into a **Gtk-UML** model, or a Qt-UML model.

  The former a PIM (Platform Independent Model), the latter a PSM (Platform Specific Model) — cf. MDA.
Special Case: Code Generation

- Recall that we said that, e.g. Java code, can also be seen as a model.
  So code-generation is a **special case** of model-to-model transformation; only the destination looks quite different.
Special Case: Code Generation

- Recall that we said that, e.g. Java code, can also be seen as a model. So code-generation is a special case of model-to-model transformation; only the destination looks quite different.

- **Note**: Code generation needn’t be as expensive as buying a modelling tool with full fledged code generation.

  - If we have the UML model (or the DSL model) given as an XML file, code generation can be as simple as an XSLT script.

    “Can be” in the sense of

    “There may be situation where a graphical and abstract representation of something is desired which has a clear and direct mapping to some textual representation.”

In general, code generation can (in colloquial terms) become arbitrarily difficult.
Example: Model and XMI

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<XMI xmi.version='1.2' xmlns:UML='org.omg.xmi.namespace.UML' timestamp='Mon Feb 02 18:23:12 CET 2009'>
<XMI.content>
<UML:Model xmi.id='...'>
  <UML:Namespace.ownedElement>
    <UML:Class xmi.id='...' name='SensorA'>
      <UML:ModelElement.stereotype>
        <UML:Stereotype name='pt100'/>
      </UML:ModelElement.stereotype>
    </UML:Class>
    <UML:Class xmi.id='...' name='ControllerA'>
      <UML:ModelElement.stereotype>
        <UML:Stereotype name='65C02'/>
      </UML:ModelElement.stereotype>
    </UML:Class>
    <UML:Class xmi.id='...' name='UsbA'>
      <UML:ModelElement.stereotype>
        <UML:Stereotype name='NET2270'/>
      </UML:ModelElement.stereotype>
    </UML:Class>
  </UML:Namespace.ownedElement>
  <UML:Association xmi.id='...' name='in'/>...
  <UML:Association xmi.id='...' name='out'/>...
</UML:Model>
</XMI.content>
</XMI>
```
References
References


