Contents & Goals

Last Lecture:
- Firedset, Cut
- Automaton construction
- Transition annotations

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

- Content:
  - Inheritance in UML: concrete syntax
  - Liskov Substitution Principle — desired semantics
  - Two approaches to obtain desired semantics
Abstract Syntax

A signature with inheritance is a tuple

\[ \mathcal{S} = (\mathcal{P}, \mathcal{C}, V, vtr, F, mth, \triangleleft) \]

where

- \((\mathcal{P}, \mathcal{C}, V, vtr, F, mth)\) is a signature with signals and behavioural features (\(F/mth\) are methods, analogous to \(V/vtr\) attributes), and
- \(\triangleleft \subseteq (\mathcal{C} \times \mathcal{C}) \cup (\mathcal{P} \times \mathcal{P})\)
  is an acyclic generalisation relation, i.e. \(C \triangleleft^+ C\) for no \(C \in \mathcal{C}\).

In the following (for simplicity), we assume that all attribute (method) names are of the form \(C::v\) and \(C::f\) for some \(C \in \mathcal{C} \cup \mathcal{P}\) ("fully qualified names").

Read \(C \triangleleft D\) as...

- \(D\) inherits from \(C\),
- \(C\) is a generalisation of \(D\),
- \(D\) is a specialisation of \(C\),
- \(C\) is a super-class of \(D\),
- \(D\) is a sub-class of \(C\),
- ...
**Helper Notions**

**Definition.**

(i) For 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 $C_0^1, \ldots, C_0^n, C_1^1, \ldots, C_1^m \in \mathcal{C}$, $n, m \geq 0$, s.t.

$$C_0 \triangleleft C_0^1 \triangleleft \ldots \triangleleft C_0^n \triangleleft C_1^1 \triangleleft \ldots \triangleleft C_1^m \triangleleft D.$$

(ii) We use $\triangleleft^*$ to denote the reflexive, transitive closure of $\triangleleft$.

---

**Inheritance: Concrete Syntax**

**Common graphical representations** (of $\triangleleft = \{(C, D_1), (C, D_2)\}$):

![Diagram of inheritance relations between classes](image)

**Mapping** Concrete to Abstract Syntax by Example:

$$\mathcal{A} = \{ (C_0, C_1), (C_1, C_2), (D, C_2) \}$$

**Note:** we can have multiple inheritance.
Desired Semantics of Specialisation: Subtyping

There is a classical description of what one expects from sub-types, which is closely related to inheritance in object-oriented approaches:

The principle of type substitutability 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 \( T \)."

In other words: Fischer and Wehrheim (2000)

"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."
Subtyping: Example

Domain Inclusion Semantics
Domain Inclusion Structure

A domain inclusion structure $\mathcal{D}$ for signature $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{V}, \text{atr}, \mathcal{E}, \mathcal{F}, \text{mth}, \preceq)$

- [as before] maps types, classes, associations to domains,
- [for completeness] maps methods to transformers,
- [as before] has infinitely many object identities per class in $\mathcal{D}(D), \mathcal{D} \in \mathcal{C},$
- [changed] the identities of a super-class comprise all identities of sub-classes, i.e.

$\forall C \triangleleft D \in \mathcal{C}: \mathcal{D}(D) \subseteq \mathcal{D}(C)$

and identities of instances of classes not (transitively) related by generalisation are disjoint, i.e. $C \not\triangleleft D$ and $D \not\triangleleft C$ implies $\mathcal{D}(C) \cap \mathcal{D}(D) = \emptyset.$

Note: the old setting coincides with the special case $\triangleleft = \emptyset.$

Domain Inclusion System States

A system state of $\mathcal{S}$ wrt. (domain inclusion structure) $\mathcal{D}$ is a type-consistent mapping

$\sigma: \mathcal{D}(\mathcal{C}) \rightarrow (\mathcal{V} \rightarrow (\mathcal{D}(\mathcal{T}) \cup \mathcal{D}(\mathcal{C}_0,1) \cup \mathcal{D}(\mathcal{C}_*)))$

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

- [as before] $\sigma(u)(v) \in \mathcal{D}(T)$ if $v : T,$
- [changed] $\sigma(u), u \in \mathcal{D}(C),$ has values for all attributes of $C$ and all of its superclasses, i.e.

$\text{dom}(\sigma(u)) = \bigcup_{C_0 \triangleleft^* C} \text{atr}(C_0).$

Example:

$\sigma^C: \mathcal{C} \rightarrow \mathcal{D}(\mathcal{C}), \sigma^D: \mathcal{D} \rightarrow \mathcal{D}(\mathcal{D}),$ etc.

Note: the old setting still coincides with the special case $\triangleleft = \emptyset.$
OCL Syntax and Typing

- Recall (part of the) OCL syntax and typing ($C, D \in \mathcal{C}, v, r \in V$)

\[
\text{expr} ::= v(\text{expr}_1) : \tau_C \rightarrow T(v), \quad \text{if } v : T \in \text{atr}(C), \ T \in \mathcal{F}
\]

\[
| r(\text{expr}_1) : \tau_C \rightarrow \tau_D, \quad \text{if } r : D_{0,1} \in \text{atr}(C)
\]

\[
| r(\text{expr}_1) : \tau_C \rightarrow \text{Set}(\tau_D), \quad \text{if } r : D_\ast \in \text{atr}(C)
\]

The syntax basically stays the same:

\[
\text{expr} ::= C::v(\text{expr}_1) : \tau_C \rightarrow T(v), \quad \text{if } C::v : T \in \text{atr}(C), \ T \in \mathcal{F}
\]

\[
| \ldots
\]

\[
| v(\text{expr}_1) : \tau_C \rightarrow T(v),
\]

\[
| r(\text{expr}_1) : \tau_C \rightarrow \tau_D,
\]

\[
| r(\text{expr}_1) : \tau_C \rightarrow \text{Set}(\tau_D),
\]

but typing rules change: we require a unique biggest superclass $C_0 \triangleleft^* C \in \mathcal{C}$ with, e.g., $v \in \text{atr}(C_0)$ and for this $v$ we have $v : T$.

Example:

\[
\text{context } C \text{ inv : } v \in C > 0
\]

\[
\text{context } D \text{ inv : } v \in C > 0
\]

\[
\text{context } B \text{ inv : } v \in C > 0
\]

Note: the old setting still coincides with the special case $\triangleleft = \emptyset$.

Visibility and Inheritance

Example:

<table>
<thead>
<tr>
<th></th>
<th>$v_1 &lt; 0$</th>
<th>$v_2 &lt; 0$</th>
<th>$v_3 &lt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>context $C$ inv :</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>context $D$ inv :</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>context $B$ inv :</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

E.g. $v(\ldots (\text{self}) \ldots)$ is well-typed

- if $v$ is public, or
- if $v$ is private, and $\text{self : } \tau_C$ and $v \in \text{atr}(C)$, or
- if $v$ is protected, and $\text{self : } \tau_C$ and $D \triangleleft^* C$ (unique, biggest) and $v \in \text{atr}(D)$. 
I_{DI}[expr](\sigma) := I[\text{Normalise}(expr)](\sigma)

using the same textual definition of $I$ that we have.

---

**Expression Normalisation**

*Normalise:*
- Given expression $v(...(w)...)$ with $w : \tau_D$
- normalise $v$ to (= replace by) $C::v$
- where $C$ is the unique most special more general class with $C::v \in \text{atr}(C)$, i.e.

\[ \forall C \triangleleft^* C_0 \triangleleft^* D \bullet C_0 = C. \]

**Note:** existence of such an $C$ is guaranteed by (the new) OCL well-typedness.

**Example:**
- context $D$ inv : $x < 0$  \(\triangleright\) context $D$ inv : $C::x > 0$
- context $C$ inv : $x < 0$  \(\triangleright\) context $C$ inv : $C::x > 0$
- context $A$ inv : $x < 0$  \(\triangleright\) context $A$ inv : $A::x > 0$
- context $D$ inv : $n < 0$  \(\triangleright\) context $D$ inv : $D::n, C::x < 0$
- context $C$ inv : $n < 0$  \(\triangleright\) context $C$ inv : $A::n < 0$
- context $D$ inv : $A::x < 0$  \(\triangleright\) context $D$ inv : $A::x < 0$
**OCL Example**

\[ \sigma: \]

- \( \mathbb{u}_1 : A \)
  - \( A :: x = 0 \)

- \( \mathbb{u}_2 : C \)
  - \( A :: x = 1 \)
  - \( C :: x = 27 \)

- \( \mathbb{u}_3 : D \)
  - \( A :: x = 2 \)
  - \( C :: x = 13 \)

\[ D : n \]

\[ 0, 1 \]

\[ A :: x : \text{Int} \]

\[ C :: x : \text{Int} \]

\[ D :: n \]

- \( \llbracket \text{context } D \hspace{1mm} \text{inv}: A :: x < 0 \rrbracket (\sigma, \{ \text{self} \mapsto \mathbb{u}_3 \}) \)
  - \( \llbracket (\sigma(\mathbb{u}_3)(A :: x), 0) \rrbracket(2, 0) = \text{false} \)

- \( \llbracket \text{context } D \hspace{1mm} \text{inv}: x < 0 \rrbracket (\sigma, \{ \text{self} \mapsto \mathbb{u}_3 \}) \)
  - \( \llbracket \text{context } D \hspace{1mm} \text{inv}: C :: x < 0 \rrbracket (\sigma, \beta) \)
  - \( \llbracket (\sigma(\mathbb{u}_3)(C :: x), 0) \rrbracket(13, 0) = \text{false} \)

**Excursus: Late Binding of Behavioural Features**
Late Binding

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

<table>
<thead>
<tr>
<th>Type of the link determine which impl is used</th>
<th>f not overridden in D</th>
<th>f overridden in D</th>
</tr>
</thead>
<tbody>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>C::f()</td>
</tr>
<tr>
<td>someD -&gt; f()</td>
<td>D::f()</td>
<td>D::f()</td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>C::f()</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type of the object</th>
<th>someC -&gt; f()</th>
<th>someD -&gt; f()</th>
</tr>
</thead>
<tbody>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>D::f()</td>
</tr>
<tr>
<td>someD -&gt; f()</td>
<td>D::f()</td>
<td>D::f()</td>
</tr>
</tbody>
</table>

Late Binding in the Standard and Programming Languages

- 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, 2007, 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”

Note: late binding typically applies only to methods, not to attributes. (But: getter/setter methods have been invented recently.)
Semantics of Method Calls

- **Non late-binding**: by normalisation.

- **Late-binding**:
  Construct a *method call* transformer, which looks up the method transformer corresponding to the class we are an instance of.
Transformers (Domain Inclusion)

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

\[ \text{update}(\text{expr}_1, v, \text{expr}_2) : (\sigma, \varepsilon) \mapsto (\sigma', \varepsilon) \]

with

\[ \sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I_{DI}[[\text{expr}_2]](\sigma)]] \]

where \( u = I_{DI}[[\text{expr}_1]](\sigma) \) — after normalisation, e.g. assume \( v \) qualified.

\[ \text{Inheritance and State-Machines: Example} \]

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

\[ \text{SM}_A: \quad E^n \quad s_1 \quad s_2 \]

\[ \text{SM}_D: \quad E^n \quad s_1 \quad s_2 \]

\[ C \quad n \quad A \]

\[ D \]

\[ x_1 : A \]
\[ s_1 = s_1 \]
\[ \text{stable} = 0 \]

\[ u_2 : D \]
\[ s_1 = s_1 \]
\[ \text{stable} = 1 \]

\[ (f_1, D) \]

\[ (f_1, \varepsilon) \]

\[ \varepsilon : (f_1, \varepsilon) \]

\[ f : \varepsilon \]

\[ \varepsilon_1 : \varepsilon \]
(ii) Dispatch

\[(\sigma, \epsilon) \xrightarrow{(\text{cons}, \text{Snd})} u \xrightarrow{a} (\sigma', \epsilon')\]

if

- \(u \in \text{dom}(\sigma) \cap \mathcal{P}(C) \land \exists u_E \in \mathcal{P}(E) : u_E \in \text{ready}(\epsilon, u)\)
- \(u\) 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.

\[\exists (s, F, \text{expr}, \text{act}, s') \in \rightarrow (SM_C) : F = E \land I[\text{expr}](\tilde{\sigma}, u) = 1\]

where \(\tilde{\sigma} = \sigma[u, \text{params}_E \mapsto u_E]\).

and

- \((\sigma', \epsilon')\) results from applying \(t_{\text{act}}\) to \((\sigma, \epsilon)\) and removing \(u_E\) from the ether, i.e.

\[\sigma'' = (\sigma''[u, \text{st} \mapsto s', u, \text{stable} \mapsto b, u, \text{params}_E \mapsto \emptyset])|_{\sigma(E) \setminus \{u\}}\]

where \(b\) depends (see (i))
- Consumption of \(u_E\) and the side effects of the action are observed, i.e.

\[\text{cons} = \{u_E\}, \quad \text{Snd} = \text{Observe}[u](\tilde{\sigma}, \epsilon \oplus u_E)\].

---

**Inheritance and Interactions**

\[
\begin{array}{c}
\langle \text{signal}, \text{env} \rangle \quad E \\
\text{a : A} \\
\langle \text{signal}, \text{env} \rangle \\
\text{F} \\
\end{array}
\]

\[
\begin{array}{c}
\text{c : C} \\
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
E' \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a : A} \\
\text{c : C} \\
\end{array}
\]

\[
\begin{array}{c}
\text{C} \\
\begin{array}{c}
\text{a} \\
\text{n} \\
\text{D} \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{c}
\text{u_1 : A} \\
\end{array}
\]

\[
\begin{array}{c}
\text{u_2 : D} \\
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{E_{a,c}} \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{c}
\text{E_{d,c}} \\
\end{array}
\]

\[\exists \beta, \beta(a \in \text{A}), \beta(c \in \text{C}) : \ldots (\sigma, u, \text{cons}_E, \text{Snd}) \quad \text{E}_{a,c} \]
Domain Inclusion vs. Uplink Semantics

Wanted: a formal representation of “if $C \sqsubseteq D$ 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.

Two approaches to semantics:

- **Domain-inclusion** Semantics (more theoretical)
- **Uplink** Semantics (more technical)
References


