Software Design, Modelling and Analysis in UML

Lecture 21: Inheritance II

2013-02-05

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
Contents & Goals

Last Lecture:
• State Machine semantics completed
• Inheritance in UML: concrete syntax

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:
  • Liskov Substitution Principle — desired semantics
  • Two approaches to obtain desired semantics
Inheritance: Syntax
Recall: Abstract Syntax

Recall: a signature (with signals) is a tuple $\mathcal{S} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E})$.

Now (finally): extend to

$$\mathcal{S} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E}, F, mth, \triangleleft)$$

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

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

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 $D$,
- …
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
Recall
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 \[\text{[Liskov, 1988, Liskov and Wing, 1994].}\]

(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 \(T\).”

\[
\forall o_1 \in S \exists o_2 \in T \forall p \in \mathcal{P} \left( \pi \left( \left[ P \right](o_1) \equiv \left[ P \right](o_1/o_2) \right) \right)
\]

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

“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.”

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

- **OCL**: context $C$ inv : $x > 0$

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

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

**Sequence Diagrams**:
- Wanted: bind $v_0$ as well as $v_2$ to its instance line
- Use $F$ instances here
“...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.
- **Wider** interpretation: another modeler.

```
C
x : Int
f(Int) : Int

D
```
“...can’t tell difference...”?

\[ \sigma : \]

\[
\begin{array}{c}
\begin{array}{c}
C \vdash x = 1 \\
D \vdash x = 27 \\
\end{array} \\
\begin{array}{c}
C \vdash y = 1 \\
D \vdash x = 13 \\
\end{array}
\end{array}
\]

\[ OCL: \]

- \( I[\text{context } C \text{ inv: } x > 0](\sigma_1, \emptyset) \) vs. \( I[\text{context } C \text{ inv: } x > 0](\sigma_2, \emptyset) \)

\[ LSP: \]

- \( I[\Psi](\sigma, \{x \leftarrow \text{false}\}) = \text{true} \)
- \( \forall \sigma', v \in D(c) \cdot I[\Psi](\sigma', \{x \leftarrow v\}) = \text{true} \)

\[ \emptyset \]

\[ \{ \text{sel}(u_1, u_1) \} \]

\[ \{ \text{sel}(u_2, u_2) \} \]

\[ \text{usage} \]
“...can’t tell difference...”?

- **Triggers, Actions**: if

\[
(\sigma_0; \varepsilon_0) \xrightarrow{(\text{cons}_0, \text{Snd}_0)} u_2 \xrightarrow{\sigma_1; \varepsilon_1}
\]

is possible, then

\[
(\sigma_0; \varepsilon_0) \xrightarrow{(\text{cons}_0, \text{Snd}_0)} u_1 \xrightarrow{[v_2/v_1]} (\sigma_1; \varepsilon_1)
\]

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

for some \( u_1 \) and

a proper definition of \([v_1/v_2]\).
“...can’t tell difference...”? 

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

\[
\begin{array}{c}
\text{C} \\
\text{D_1} \\
\text{f():Int} \\
\end{array}
\]

we don’t even have usability.

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

\[
\begin{array}{c}
\text{C} \\
\text{D_1} \\
\text{D_2} \\
\end{array}
\]

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:

\[
\begin{align*}
C_1 & : x : \text{Int} \quad f(\text{Int}) : \text{Int} \\
D_1 & : x : \text{Int} \quad f(\text{Int}) : \text{Int} \\
C_2 & : x : \text{Int} \quad f(\text{Int}) : \text{Int} \\
D_2 & : x : \text{Bool} \quad f(\text{Float}) : \text{Int}
\end{align*}
\]

Wanted:

- \( x > 0 \) also well-typed for \( D_1 \)
- assignment \( \text{its}C_1 := \text{its}D_1 \) being well-typed
- \( \text{its}C_1.x = 0, \text{its}C_1.f(0), \text{its}C_1 ! F \)
  being well-typed (and doing the right thing).

Approach:

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

\[
\begin{array}{|c|c|}
\hline
C_1 & x : \text{Int} \\
\hline
& f(D_2) : \text{Int} \\
\hline
D_1 & f(C_2) : \text{IntE} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
C_2 & x : \text{Int} \\
\hline
& f(\text{Int}) : \text{Int} \\
\hline
D_2 & x : \text{Bool} \\
\hline
& f(\text{Float}) : \text{Bool} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\langle \langle \text{signal} \rangle \rangle E & \\text{Notions (from category theory):} \\
\hline
\langle \langle \text{signal} \rangle \rangle F & \cdot \text{invariance}, \\
\hline
& \cdot \text{covariance}, \\
\hline
& \cdot \text{contravariance}. \\
\end{array}
\]

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

\begin{itemize}
  \item accepts more general types as input \hspace{1cm} \text{(contravariant)},
  \item provides a more specialised type as output \hspace{1cm} \text{(covariant)}.
\end{itemize}

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>f not overridden in D</th>
<th>f overridden in D</th>
<th>value of someC/someD</th>
</tr>
</thead>
<tbody>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td></td>
</tr>
<tr>
<td>someD -&gt; f()</td>
<td>C::f()</td>
<td></td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>D::f()</td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>D::f()</td>
</tr>
</tbody>
</table>

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

| someC -> f()          | C::f()            |                      |
| someD -> f()          | C::f()            |                      |
| someC -> f()          | C::f()            | D::f()               |
| someC -> f()          | C::f()            | D::f()               |

The type of the link determines which implementation is used (not caring for what an object really is).
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
With Only Early Binding...

- ...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
  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);
  }
  ```
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).
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.

We’ll discuss two approaches to semantics:

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

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

Let $\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E}, F, mth, \triangleleft)$ 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.

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

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

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

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

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 \( \tau \in \{C_{\ast}, C_{0,1}\} \).

- **[changed]** \( \text{dom}(\sigma(u)) = \bigcup_{C_{0} \preceq C} \text{attr}(C_{0}) \),

**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 $	riangleleft$. 
**Preliminaries: Expression Normalisation**

**Recall:**
- we want to allow, e.g., “context $D \text{ inv} : v < 0$”.
- 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 ($= \text{ replace by}$) $C::v$,
- 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 $C::v \in \text{attr}(C)$.

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

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

\[
expr ::= \begin{align*}
v(expr_1) & : \tau_C \rightarrow \tau(v), \quad \text{if } v : \tau \in \mathcal{T} \\
r(expr_1) & : \tau_C \rightarrow \tau_D, \quad \text{if } r : D_{0,1} \\
r(expr_1) & : \tau_C \rightarrow \text{Set}(\tau_D), \quad \text{if } r : D_*
\end{align*}
\]

The definition of the semantics remains (textually) the same.
We want

class $D$ inv : $v < 0$
to be well-typed.

Currently it isn’t because

$$v(expr_1) : \tau_C \rightarrow \tau(v)$$

but $A \vdash self : \tau_D$.

(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

$$A \vdash expr : \tau_D \quad \frac{C \preceq D}{A \vdash expr : \tau_C} \quad \text{(Inh)}$$

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.
Well-Typed-ness with Visibility Cont’d

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

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

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

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

**Example:**

<table>
<thead>
<tr>
<th>context/inv</th>
<th>(n.)v₁ &lt; 0</th>
<th>(n.)v₂ &lt; 0</th>
<th>(n.)v₃ &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 $\mathcal{M} = (\mathcal{C} \mathcal{D}, \mathcal{C} \mathcal{D}, \mathcal{I} \mathcal{M}, \mathcal{I})$ be a UML model, and $\mathcal{D}$ a structure.

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

  $$\forall \pi = (\sigma_i, \varepsilon_i)_{i \in \mathbb{N}} \in \llbracket \mathcal{M} \rrbracket \quad \forall i \in \mathbb{N} \quad \forall u \in \text{dom} (\sigma_i) \cap \mathcal{D}(C) :$$

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

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

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

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

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.
Inheritance and State Machines: Triggers

• **Wanted**: triggers shall also be sensitive for inherited events, sub-class shall execute super-class’ state-machine (unless overridden).

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

- \(\exists u \in \text{dom}(\sigma) \cap \mathcal{D}(C) \exists u_E \in \mathcal{D}(\varepsilon) : u_E \in \text{ready}(\varepsilon, u)\)
- \(u\) 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.

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

where \(\tilde{\sigma} = \sigma[u.params_E \mapsto ue]\).

and

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

\[
(\sigma'', \varepsilon') = t_{act}(\tilde{\sigma}, \varepsilon \Theta u_E),
\]

\[
\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset]) | \mathcal{D}(\varepsilon) \setminus \{u_E\}
\]

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.

\[
\text{cons} = \{(u, (E, \sigma(u_E)))\}, Snd = \text{Obs}_{t_{act}}(\tilde{\sigma}, \varepsilon \Theta u_E).
\]
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.

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

if and only if

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

- **Note**: $C$-instance line also binds to $C'$-objects.
Uplink Semantics
**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 $C \triangleleft D$, extend $D$ by a (fresh) association

$$\text{uplink}_{C} : C \text{ with } \mu = [1, 1], \xi = +$$

(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) $C \triangleleft C_1 \triangleleft \ldots \triangleleft C_n \triangleleft D$,
  - normalise $v$ to (≡ replace by)

$$\text{uplink}_{C_n} \rightarrow \cdots \rightarrow \text{uplink}_{C_1}.C::v$$

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

- We (continue to) say

  $$
  \mathcal{M} \models expr
  $$

  for

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

  if and only if

  $$
  \forall \pi = (\sigma_i)_{i \in \mathbb{N}} \in \llbracket \mathcal{M} \rrbracket
  $$

  $$
  \forall i \in \mathbb{N}
  $$

  $$
  \forall u \in \text{dom}(\sigma_i) \cap D(C) : I[expr_0]\mathcal{M}(\sigma_i, \{ \text{self } \mapsto u \}) = 1.
  $$

- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $\text{Inv}(\mathcal{M})$. 
What has to change is the create transformer:

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

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

\[
C_{1,1} \triangleleft \ldots \triangleleft C_{1,n_1} \triangleleft C, \\
\ldots \\
C_{m,1} \triangleleft \ldots \triangleleft C_{m,n_m} \triangleleft C.
\]

Then, we have to

- create one fresh object for each part, e.g.
  \[ u_{1,1}, \ldots, u_{1,n_1}, \ldots, u_{m,1}, \ldots, u_{m,n_m}, \]

- set up the uplinks recursively, e.g.
  \[ \sigma(u_{1,2})(uplink_{C_{1,1}}) = u_{1,1}. \]

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; // assign address of ‘d’ to pointer ‘cp’

- **Identity downcast** (C++):
  - D* dp = (D*)cp; // assign address of ‘d’ to pointer ‘dp’

- **Value upcast** (C++):
  - *c = *d; // copy attribute values of ‘d’ into ‘c’, or,
  // 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>
<tbody>
<tr>
<td><em><em>C</em> cp = &amp;d;</em>*</td>
<td><strong>easy</strong>: immediately compatible (in underlying system state) because &amp;d yields an identity from ( \mathcal{D}(D) \subset \mathcal{D}(C) ).</td>
</tr>
<tr>
<td><em><em>D</em> dp = ((D</em>)cp;**</td>
<td><strong>easy</strong>: the value of cp is in ( \mathcal{D}(D) \cap \mathcal{D}(C) ) because the pointed-to object is a ( D ). Otherwise, error condition.</td>
</tr>
<tr>
<td><strong>c = d;</strong></td>
<td><strong>bit difficult</strong>: set (for all ( C \preceq D )) ( (C)(\cdot , \cdot ) : \tau_D \times \Sigma \to \Sigma</td>
</tr>
</tbody>
</table>
**Identity Downcast with Uplink Semantics**

- **Recall** (C++): $D \mathcal{d}; \quad C^* \mathcal{cp} = \& \mathcal{d}; \quad D^* \mathcal{dp} = (D^*) \mathcal{cp};$

- **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.

    ```
    \text{switch(mostspec\_type)}\
    \quad \text{case } C :\
    \quad \quad \mathcal{dp} = \mathcal{cp} \rightarrow \text{mostspec} \rightarrow \text{uplink}_{D_n} \rightarrow \ldots \rightarrow \text{uplink}_{D_1} \rightarrow \text{uplink}_D;\
    \quad \ldots\
    ```
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...
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*?
References
References


