Contents & Goals

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

This Lecture:

Educational Objectives:
- 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

Inheritance: Syntax

Abstract Syntax

A signature with inheritance is a tuple

\[ \mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{V}, \mathcal{a}, \mathcal{E}, \mathcal{F}, \mathcal{m}, \triangleright) \]

where
- \((\mathcal{T}, \mathcal{C}, \mathcal{V}, \mathcal{a}, \mathcal{E})\) is a signature with signals and behavioural features (\(\mathcal{F}/\mathcal{m}\) are methods, analogous to \(\mathcal{V}/\mathcal{a}\) attributes), and
- \(\triangleright \subseteq \mathcal{C} \times \mathcal{C} \cup \mathcal{E} \times \mathcal{E}\) is an acyclic generalisation relation, i.e. \(\mathcal{C} \triangleright \mathcal{D} \triangleright \mathcal{C}\) for no \(\mathcal{C} \in \mathcal{C}\).

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

Read \(\mathcal{C} \triangleright \mathcal{D}\) as:
- \(\mathcal{D}\) inherits from \(\mathcal{C}\),
- \(\mathcal{C}\) is a generalisation of \(\mathcal{D}\),
- \(\mathcal{D}\) is a specialisation of \(\mathcal{C}\),
- \(\mathcal{C}\) is a super-class of \(\mathcal{D}\),
- \(\mathcal{D}\) is a sub-class of \(\mathcal{C}\),
- . . .

Helper Notions

Definition.
- (i) For classes \(\mathcal{C}_0, \mathcal{C}_1, \mathcal{D} \in \mathcal{C}\), we say \(\mathcal{D}\) inherits from \(\mathcal{C}_0\) via \(\mathcal{C}_1\) if and only if there are \(\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{C}_1, \ldots, \mathcal{C}_m \in \mathcal{C}\), \(n, m \geq 0\), s.t. \(\mathcal{C}_0 \triangleright \mathcal{C}_1 \triangleright \ldots \triangleright \mathcal{C}_n \triangleright \mathcal{C}_1 \triangleright \ldots \triangleright \mathcal{C}_m \triangleright \mathcal{D}\).
- (ii) We use \(\triangleright^\ast\) to denote the reflexive, transitive closure of \(\triangleright\).

Inheritance: Concrete Syntax

Common graphical representations (of \(\triangleright = \{(\mathcal{C}, \mathcal{D}_1), (\mathcal{C}, \mathcal{D}_2)\}\)):

\[
\begin{align*}
\mathcal{C} & \triangleright \mathcal{D}_1 \\
\mathcal{C} & \triangleright \mathcal{D}_2
\end{align*}
\]

Mapping Concrete to Abstract Syntax by Example:

\[
\begin{align*}
\mathcal{C}_0 & \triangleright \mathcal{X} \\
\mathcal{C}_1 & \triangleright \mathcal{X}\end{align*}
\]

Note: we can have multiple inheritance.
Note: \( \emptyset = \triangleright \) implies \( C \not\triangleright D \) when \( C \) is disjoint, i.e., all attributes of \( C \) or all attributes of \( C \) and identities of instances of classes not (transitively) related by generalisation are changed without a client being able to tell the difference whenever an instance of the supertype was expected. 

In other words: the old setting coincides with the special case of \( \emptyset = \triangleright \) \( C \).
Example: Let’s build an interesting feature.

Expression Normalisation

- Case I: $\sigma : x < D$ context (where $\sigma$ is public)

or

- Case II: $\sigma : x < D$ context (where $\sigma$ is not public)

In Case I, we require a unique biggest superclass $C$ with $\sigma$, i.e., $\tau$.

Recall (part of the) OCL syntax and typing (OCL Syntax and Typing).
Late Binding – 20 – 2016-02-04 – Slatebind –

19/30

What transformer applies in what situation?

(Early (compile time) binding.)

```c
f not overridden in D
C
f() : Int
D
C
0
someC
someD
```

What one could want is something different:

(Late binding.)

```c
f overridden in D
C
f() : Int
```

```c
D
f() : Int
```

```c
valueof
someC/someD
```

```c
someC -> f()
C
:: f()
```

```c
C
:: f()
u1
```

```c
someD -> f()
D
:: f()
```

```c
D
:: f()
u2
```

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.)

Behaviour (Inclusion Semantics)

• 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]

  update(expr1, v, expr2) : (σ, ε) ↦→ (σ', ε)

  where
  σ' = σ[u↦→ σ(u)] [v↦→ I_{D}(\llbracket expr2 \rrbracket(σ))]

  — after normalisation, e.g. assume
  v qualified.

Inheritance and State-Machines: Example

⟨ ⟨signal, env⟩ ⟩

E

⟨ ⟨signal, env⟩ ⟩
F

s1
s2

•Es/SM

A

:s1
s2

•E/SM

D

C

A

D
n
0, 1

u1:
A

st = s1
stable = 0

u2:
D

st = s1
stable = 1

n
(ii) Dispatch

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

if \( u \in \text{dom}(\sigma) \cap D(C) \land \exists u_E \in D(E) : u_E \in \text{ready}(\epsilon, u) \)

- \( u \) is stable and in state machine state \( s \), i.e.
  \[ \sigma(u)(\text{stable}) = 1 \land \sigma(u)(\text{st}) = s \],

- a transition is enabled, i.e.
  \[ \exists(s, F, \text{expr}, \text{act}, s') \in \rightarrow(SM_C) : F = E \land \llbracket \text{expr} \rrbracket(\tilde{\sigma}, u) = 1 \]

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

and

- \( (\sigma', \epsilon') \) results from applying \( \text{tact} \) to \( (\sigma, \epsilon) \) and removing \( u_E \) from the ether, i.e.
  \[ (\sigma'', \epsilon') \in \text{tact}[u](\tilde{\sigma}, \epsilon \ominus u_E) \],

\[ \sigma' = (\sigma''[u.\text{st} \mapsto s', u.\text{stable} \mapsto b, u.\text{params} \mapsto \emptyset] | D(C)\{u_E}\) \]

where \( b \) depends (see (i))

- Consumption of \( u_E \) and the side effects of the action are observed, i.e.
  \[ \text{cons} = \{u_E\}, \text{Snd} = \text{Obs}_{\text{tact}}[u](\tilde{\sigma}, \epsilon \ominus u_E) \].

Inheritance and Interactions

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

\[ a : A \rightarrow C \]

\[ n_0, 1 \]

Domain Inclusion vs. Uplink Semantics

Wanted: a formal representation of "if \( C \triangleright \ast D \) 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


