Software Design, Modelling and Analysis in UML

Lecture 20: Inheritance

2016-02-04

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

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

Inheritance: Syntax

Abstract Syntax

A **signature with inheritance** is a tuple

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

where

F, mth

- $(\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ is a signature with signals and behavioural features (F/mth are methods, analogous to V/atr attributes), and
- $\lhd \subseteq (\mathscr{C} \times \mathscr{C}) \cup (\mathscr{E} \times \mathscr{E})$
  is an **acyclic generalisation** relation, i.e. $C \lhd^+ C$ for **no** $C \in \mathscr{C}$.

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

Read $C \lhd D$ 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.

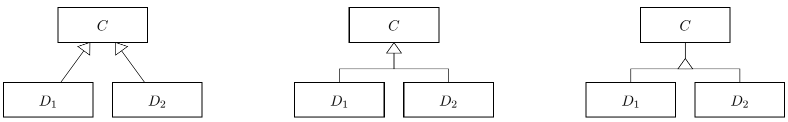
(i) For classes $C_0, C_1, D \in \mathscr{C}$, we say D **inherits from** C_0 **via** C_1 if and only if there are $C_0^1, \dots C_0^n, C_1^1, \dots C_1^m \in \mathscr{C}$, $n, m \geq 0$, s.t.

$$C_0 \lhd C_0^1 \lhd \dots C_0^n \lhd C_1 \lhd C_1^1 \lhd \dots C_1^m \lhd D.$$

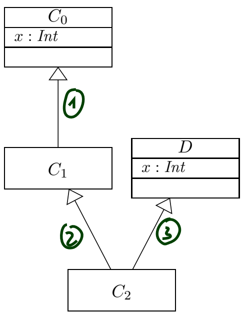
(ii) We use $\lhd^*$ to denote the reflexive, transitive closure of $\lhd$.

Inheritance: Concrete Syntax

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



Mapping Concrete to Abstract Syntax by Example:



1

$\lhd = \{\ (C_0, C_1),$
2 $(C_1, C_2),$
$(D, C_2)\ \}$
3

Note: we can have **multiple inheritance**.

Inheritance: Desired Semantics

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:

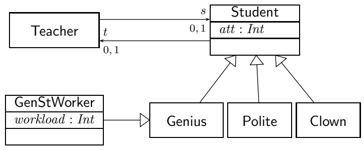
The principle of type substitutability Liskov (1988); Liskov and Wing (1994) (**Liskov Substitution Principle** (LSP)).

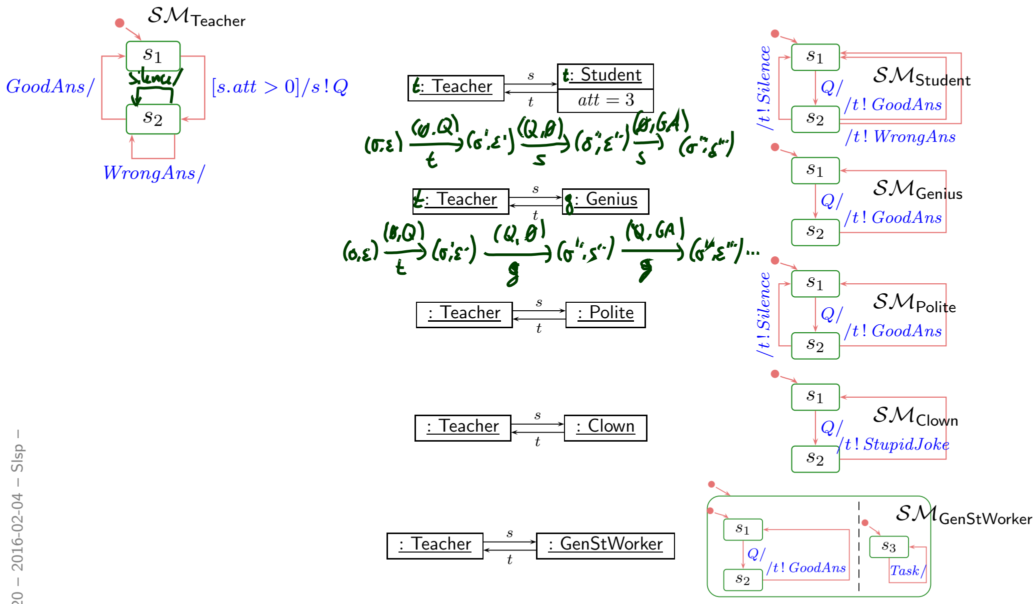
> "If for each object o_1 of type S
> there is an object o_2 of type T
> such that for all programs P defined in terms of T
> **the behavior of P is unchanged** when o_1 is substituted for o_2
> then S is a **subtype** of T."

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

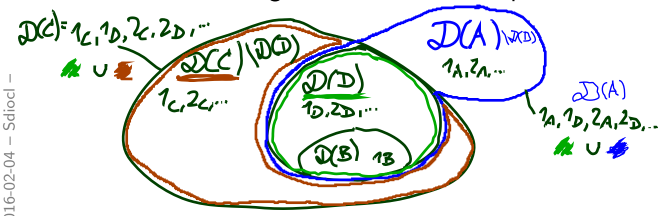
A **domain inclusion structure** $\mathscr{D}$ for signature $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}, F, mth, \lhd)$

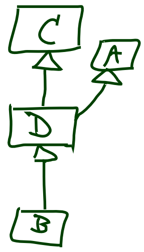
- [**as before**] maps types, classes, associations to domains,
- [**for completeness**] maps methods to transformers,
- [**as before**] has infinitely many object identities per class in $\mathscr{D}(D)$, $D \in \mathscr{C}$,
- [**changed**] the indentities of a super-class comprise all identities of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

$$\forall\, C \lhd D \in \mathscr{C} : \mathscr{D}(D) \subsetneq \mathscr{D}(C)$$

  and indentities of instances of classes not (transitively) related by generalisation are disjoint, i.e. $C \not\lhd^+ D$ and $D \not\lhd^+ C$ implies $\underbrace{\mathscr{D}(C) \cap \mathscr{D}(D) = \emptyset}_{\text{broken}}$.

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





Domain Inclusion System States

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

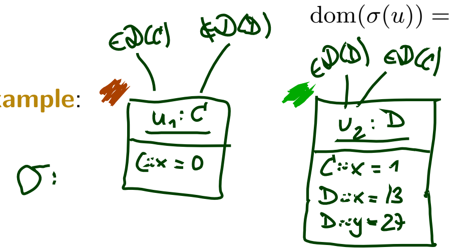
$$\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow (\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_{0,1}) \cup \mathscr{D}(\mathscr{C}_*)))$$

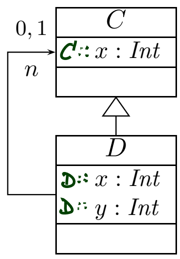
that is, for all $u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$,

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

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

Example:





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

OCL Syntax and Typing

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

$$
\begin{array}{llll}
expr ::= & v(expr_1) & : \tau_C \to T(v), & \text{if } v : T \in atr(C), \quad T \in \mathscr{T} \\
& \mid r(expr_1) & : \tau_C \to \tau_D, & \text{if } r : D_{0,1} \in atr(C) \\
& \mid r(expr_1) & : \tau_C \to Set(\tau_D), & \text{if } r : D_* \in atr(C)
\end{array}
$$

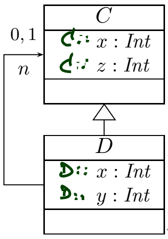
The syntax **basically** stays the same:

$$
\begin{array}{llll}
expr ::= & C::v(expr_1) & : \tau_C \to T(v), & \text{if } C::v : T \in atr(C), \quad T \in \mathscr{T} \\
& \mid \dots & & \\
& \mid v(expr_1) & : \tau_C \to T(v), & \\
& \mid r(expr_1) & : \tau_C \to \tau_D, & \\
& \mid r(expr_1) & : \tau_C \to Set(\tau_D), &
\end{array}
$$

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

Example:

context C inv C::x > 0
context D inv C::x > 0
context D inv x > 0

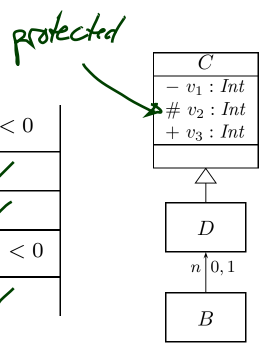


Note: the old setting still coincides with the special case $\lhd \ = \emptyset$.

Visibility and Inheritance

Example:

	$v_1 < 0$	$v_2 < 0$	$v_3 < 0$
context C inv :	✓	✓	✓
context D inv :	✗	✓	✓
	$n.v_1 < 0$	$n.v_2 < 0$	$n.v_3 < 0$
context B inv :	✗	✗	✓



E.g. $v(\dots(self)\dots)$ is well-typed

- if v is public, or
- if v is private, and $self : \tau_C$ and $v \in atr(C)$, or
- if v is protected, and $self : \tau_C$ and $D \lhd^* C$ (unique, biggest) and $v \in atr(D)$.

Satisfying OCL Constraints (Domain Inclusion)

$$I_{DI}[\![expr]\!](\sigma) := I[\![Normalise(expr)]\!](\sigma)$$

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

Expression Normalisation

Normalise:

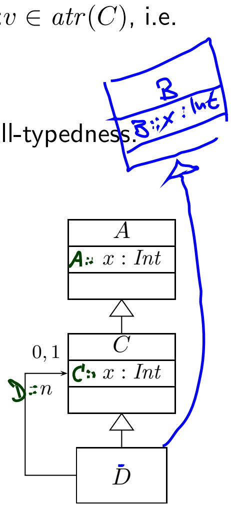
- Given expression $v(\dots(w)\dots)$ with $w : \tau_D$,
- **normalise** v to (= replace by) $C::v$,
- where C is the **unique most special more general** class with $C::v \in atr(C)$, i.e.

$$\forall C \lhd^* C_0 \lhd^* D \bullet C_0 = C.$$

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

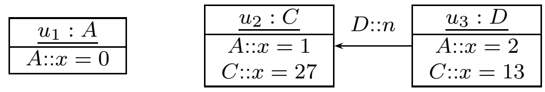
Example:

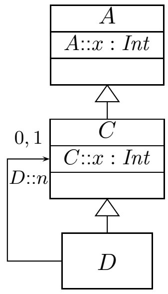
- context D inv : $x < 0$ $\rightsquigarrow$ context D inv : C::x > 0 ↯
- context C inv : $x < 0$ $\rightsquigarrow$ … : C::x > 0 ✓
- context A inv : $x < 0$ $\rightsquigarrow$ … : A::x > 0 ✓
- context D inv : $n.x < 0$ $\rightsquigarrow$ … : D::n. C::x < 0 ✓
- context C inv : $n.x < 0$ ↯
- context D inv : $A::x < 0$ $\rightsquigarrow$ … : A::x < 0 ✓



OCL Example

σ:





- $I_{DI}[\![\text{context } D \text{ inv} : A::x < 0]\!](\sigma, \{self \mapsto u_3\})$

  $= <(\sigma(u_3)(A::x), 0) = <(2,0) = false$

- $I_{DI}[\![\text{context } D \text{ inv} : x < 0]\!](\sigma, \overbrace{\{self \mapsto u_3\}}^{=:\beta})$

  $= I[\![\text{context } D \text{ inv} : C::x < 0]\!](\sigma, \beta)$

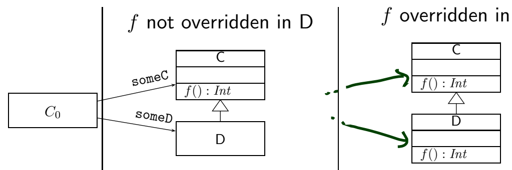
  $= <(\sigma(u_3)(C::x), 0) = <(13,0) = false$

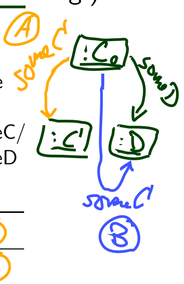
$$I[\![v(expr_1)]\!](\sigma, \beta) := \begin{cases} \sigma(u_1)(v) & \text{, if } u_1 \in \mathrm{dom}(\sigma) \\ \bot & \text{, otherwise} \end{cases}$$

Excursus: Late Binding of Behavioural Features

Late Binding

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





type of the link determines which impl. is used

	f not overridden in D	f overridden in D	value of someC/ someD
`someC -> f()`	C::f()	C::f()	A
`someD -> f()`	C::f()	D::f()	A
`someC -> f()`	C::f()	C::f()	B

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

type of the object ...

`someC -> f()`	C::f()	C::f()	A
`someD -> f()`	D::f()	D::f()	A
`someC -> f()`	C::f()	D::f()	B

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)

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)

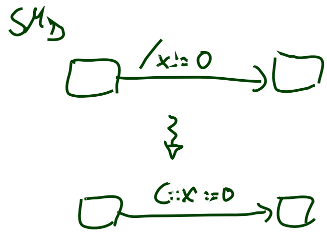
- Transformers also basically remain **the same**, e.g. [VL 12, p. 18]

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

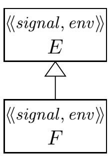
with

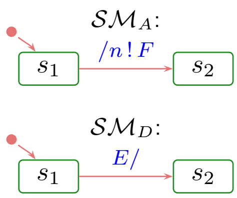
$$\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I_{DI}[\![expr_2]\!](\sigma)]]$$

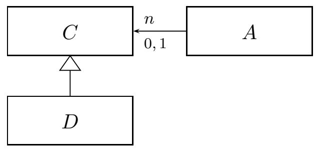
where $u = I_{DI}[\![expr_1]\!](\sigma)$ — after normalisation, e.g. assume v qualified.

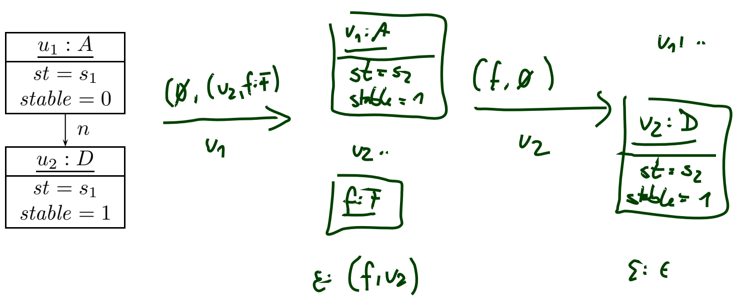


Inheritance and State-Machines: Example









(ii) Dispatch

add: and C is most-specialised class of u

$$(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon')$$

if

- $u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C) \wedge \exists\, u_E \in \mathscr{D}(E) : u_E \in ready(\varepsilon, u)$
- u is stable and in state machine state s, i.e. $\sigma(u)(stable) = 1$ and $\sigma(u)(st) = s$,
- a transition is **enabled**, i.e.

$$\exists\, (s, F, expr, act, s') \in \to (\mathcal{SM}_C) : F = E \wedge I[\![expr]\!](\tilde{\sigma}, u) = 1$$

where $\tilde{\sigma} = \sigma[u.params_E \mapsto u_E]$.

and

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

$$(\sigma'', \varepsilon') \in t_{act}[u](\tilde{\sigma}, \varepsilon \ominus u_E),$$

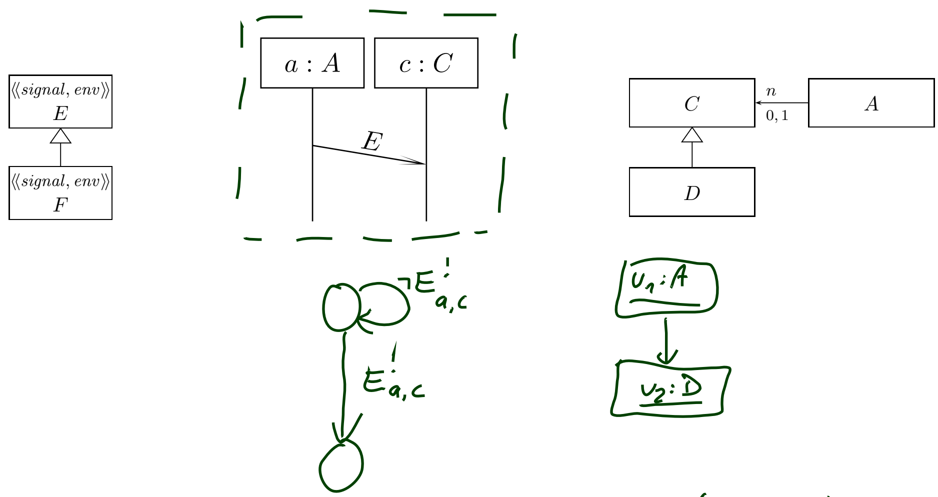
$$\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|_{\overbrace{\mathscr{D}(\mathscr{C})\setminus\{u_E\}}^{\text{remove } u_E}}$$

where b **depends** (see (i))

- Consumption of u_E and the side effects of the action are observed, i.e.

$$cons = \{u_E\}, \quad Snd = Obs_{t_{act}}[u](\tilde{\sigma}, \varepsilon \ominus u_E).$$

Inheritance and Interactions



$\ldots \exists\, \beta, \ \underset{u_1}{\beta(a)} \in \mathscr{D}(A), \ \underset{u_2}{\beta(c)} \in C' \bullet \ \ldots$

(σ, u, cns, Snd)

$F_\beta E^!_{a,c}$ ✓

Domain Inclusion vs. Uplink Semantics

System States with Inheritance

Wanted: a formal representation of "if $C \lhd^* D$ then D '**is a**' C", that is,

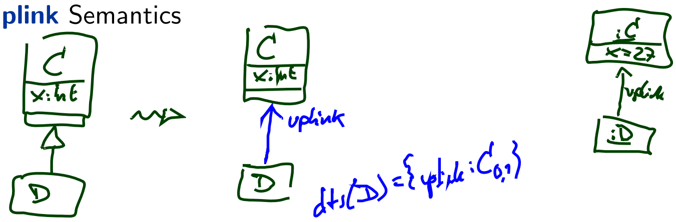
(i) D has the same attributes and behavioural features as C, and

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

Two approaches to semantics:

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

- **Uplink** Semantics (more **technical**)



$dts(D) = \{uplink : C_{0,1}\}$

context D inv : x > 0

⇓

context D inv : uplink.x > 0

References

References

Fischer, C. and Wehrheim, H. (2000). Behavioural subtyping relations for object-oriented formalisms. In Rus, T., editor, AMAST, number 1816 in Lecture Notes in Computer Science. Springer-Verlag.

Liskov, B. (1988). Data abstraction and hierarchy. SIGPLAN Not., 23(5):17–34.

Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6):1811–1841.

OMG (2007). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

OMG (2011a). Unified modeling language: Infrastructure, version 2.4.1. Technical Report formal/2011-08-05.

OMG (2011b). Unified modeling language: Superstructure, version 2.4.1. Technical Report formal/2011-08-06.