Inheritance: Desired Semantics

Recall: Abstract Syntax

Recall: Reflexive, Transitive Closure of Generalisation

Definition.

Given classes \( C_0, C_1, D \in C \), we say \( D \) inherits from \( C_0 \) via \( C_1 \) if and only if there are \( C_1^0, \ldots, C_n^0, C_1^1, \ldots, C_m^1 \in C \) such that

\[ C_0 \triangleright C_1^0 \triangleright \cdots \triangleright C_n^0 \triangleright C_1 \triangleright C_1^1 \triangleright \cdots \triangleright C_m^1 \triangleright D. \]

We use \( ' \triangleright ' \) to denote the reflexive, transitive closure of \( ' \triangleright ' \).

In the following, we assume

- that all attribute (method) names are of the form \( C :: v, C \in C \cup E \) (\( C :: f, C \in C \))
- 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 \( C \) in \( v < 0 \)", which \( v \) is meant? Later!
Desired Semanticsof 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 [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."

So, what's "usable"? Who's a "client"? And what's a "difference"?...

...shall be usable...

\[ C \xrightarrow{u_1} D \]

\[ C : \text{Int} \]

\[ f : \text{Int} \to \text{Int} \]

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

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

• OCL:

  • context \( C \)

  • actions:

    • its \( C.x = 0 \)

    • its \( C.f(0) \)

  • triggers:

    • \( E[...]/. ... \)

• Sequence Diagrams:

  \[ C : D \]

  \[ expr \]

  \[ E \]

  \[ F \]
Motivations for Generalisation

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

→ See textbook 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 $C_1$, we don't even have usability.
• It would be nice, if the well-formedness rules and semantics of $C_1$ $D_1$ would ensure $D_1$ is a sub-type of $C_1$:
  • that $D_1$ objects can be used interchangeably by everyone who is using $C_1$'s,
  • is not able to tell the difference (i.e. see unexpected behaviour).

Easy: Static Typing

Given:

$C_1 x : \text{Int} f : \text{Int} D_1$

its $C_1$
its $D_1$

$C_2 x : \text{Int} f : \text{Int} D_2$

x : Bool $f : \text{Int}$

$\langle \langle \text{signal} \rangle \rangle E \langle \langle \text{signal} \rangle \rangle F$

Wanted:

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

Approach:

• Simply define it as being well-typed, adjust systems state definition to the right thing.

Static Typing Cont'd

$C_1 x : \text{Int} f : \text{Int} D_1$

its $C_1$
its $D_1$

$C_2 x : \text{Int} f : \text{Int} D_2$

x : Bool $f : \text{Int}$

$\langle \langle \text{signal} \rangle \rangle E \langle \langle \text{signal} \rangle \rangle F$

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.
Late Binding – 21 – 2013-02-05 – main –

19/87

Late Binding

What transformer applies in what situation?

(Early compiletime binding.)

f not overridden in D

c

f(): Int

f(): Int

Value of someC/someD

c::f()
c::f()

u1

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

d::f()

u2

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

d::f()

SomeC -> f()
c::f()

d::f()

SomeD -> f()
c::f()

d::f()

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, 2007b, 247]

• In C++,

• Methods are by default "(early) compiletime 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 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 realize it correctly in the framework).

• Then if we're calling method f of an object u, which is an instance of D with C ⪯ 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 (Int): Int

D f (Int): Int

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

• Examples: (C++)

int C::f(int)
{
    return 0;
}

vs.

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

int C::f(int)
{
    return (rand() % 2);
}

vs.

int D::f(int x)
{
    return (x % 2);
}
The system state is prepared for that.

"I若是"

\[ v_{n'} < v_n \]

\[ \tau, \xi, v \] 

\[ \tau \text{dom} \Rightarrow \tau \text{context/Int} \] 

\[ \tau \text{inv} \] 

\[ \forall v : C \text{ is the greatest } \]

\[ \text{the state machine where the action occurs belongs to, } \]

\[ = \text{to (the class, the state machine where the action occurs belongs to, ) in f} \] 

\[ \text{of class } A \]

\[ \rightarrow \text{the class, the state machine where the action occurs belongs to, } \]

\[ = \text{to (the class, the state machine where the action occurs belongs to, ) in f} \] 

\[ \text{shall denote the } \]

\[ \vdash \] 

\[ \text{most special, e.g. } \]

\[ 
\begin{array}{c}
A,D \ni v, r \\
A,D \ni v, r \\
A,D \ni v, r
\end{array} 
\]

\[ \text{Recall (part of the) OCL syntax and typing: } \]

\[ \text{• Int } \]

\[ \text{• C } \]

\[ \text{• } \]

\[ \text{• } \]

\[ \text{• } \]

\[ \text{• } \]

\[ \text{• } \]

\[ \text{• } \]

\[ \text{• } \]

\[ \text{• } \]
Let\( E \) be a UML model and\( \theta\) be a context expression.

We (\( \theta\)) results from applying an instance of\( E\). DomainInclusion and Interactions

DomainInclusion and Interactions

Example
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).

\[ C_x \in \mathbb{D} \]

Apply a different pre-processing to make appropriate use of that association, e.g., rewrite \( C_{\text{++}} x = 0; \) in \( D \) to uplink
\( C \rightarrow x = 0; \)

Pre-Processing for the Uplink Semantics

For each pair \( C \owns D \), extend \( D \) by a (fresh) association uplink \( C \) 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
  - \( C :: v \in \text{attr}(D) \)

- then there exists (by definition) \( C \owns C_1 \owns \ldots \owns C_n \owns D \),

  - 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.

Satisfying OCL Constraints (Uplink)

Let \( M = (CD, OD, SM, I) \) be a UML model, and \( D \) a structure.

We (continue to) say \( M|_{inv} = expr \) for context \( C \) if

\[ 0 \leq \llbracket inv \rrbracket M \leq \llbracket expr \rrbracket \in \text{Inv}(M) \]

if and only if \( \forall \pi = (\sigma_i)_{i \in N} \in /llbracket M /rrbracket \forall i \in N \forall u \in \text{dom}(\sigma_i) \cap D(C) : \llbracket I /llbracket expr \rrbracket (\sigma_i, \{\text{self} \mapsto u\}) = 1 \).

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

Transformers (Uplink)

What has to change is the create transformer:

\[ \text{create}(C, expr, v) \]

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

\[ C_1, 1 \owns \ldots \owns C_1, n_1 \owns \ldots \owns C_m, 1 \owns \ldots \owns C_m, n_m \owns C \]

Then, we have to
- create one fresh object for each part, e.g., \( u_1, 1, \ldots, u_1, n_1, \ldots, u_m, 1, \ldots, u_m, n_m \),
- setup the uplinks recursively, e.g.,
  \[ \sigma(u_1, 2)(\text{uplink } C_1, 1) = u_1, 1. \]

Late Binding (Uplink)

Employs something similar to the "most spec" trick (in a minute!). But the result is typically far from concise. (Related to OCL's \( \text{isKindOf}() \) function, and RTTI in C++.)


---

Differ from in Domain Inclusion and Uplink Semantics

---


---

Differ from in Domain Inclusion and Uplink Semantics

---

---

---

---

---

---
References


