Software Design, Modelling and Analysis in UML
Lecture 21: Inheritance

2015-02-05
Prof. Dr. Andreas Podelski,
Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:
• Live Sequence Charts Semantics

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?

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

Motivations for Generalisation

• Re-use
• Sharing
• Avoiding Redundancy
• Modularisation
• Separation of Concerns
• Abstraction
• Extensibility
→ See textbooks on object-oriented analysis, development, programming.

Inheritance: Syntax

Abstract Syntax

Recall: a signature (with signals) is a tuple $S = (T, C, V, atr, E)$. Now (finally): extend to $S = (T, C, V, atr, E, F, mth, \sqsubseteq)$ where $F/mth$ are methods, analogously to attributes and $\sqsubseteq \subseteq (C \setminus E) \times (C \setminus E) \cup (E \times E)$ is a generalisation relation such that $C \sqsubseteq + C$ for no $C \in C$ ("acyclic").

$C \sqsubseteq D$ 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 $D$,
• . . .
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 $C_0 \triangleright C_1_0 \triangleright \ldots \triangleright C_n_0 \triangleright C_1 \triangleright C_1_1 \triangleright \ldots \triangleright C_m_1 \triangleright D$.

We use '⪯' to denote the reflexive, transitive closure of '⊳'.

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 $C :: v \in atr(C)$ resp. $C :: f \in mth(C)$ if and only if $v(f)$ appears in an attribute (method) compartment of $C$ in a class diagram.
Example 1

\(\forall x (x \in T \land x \in S)\)

\(T = \{0, 1\}\) and \(S = \{1\}\)

\(\models \) Domain Inclusion System States

\(\models \) Domain Inclusion Semantics: Idea
• 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, so we have to fix, e.g. $\sigma$, $\text{cons}$, $Snd \mid = \beta E ! x, y$ if and only if $\beta(x)$ sends an $F$-event to $\beta y$ where $E \sqsubseteq F$.
  • $C$-instance line also binds to $C'$-objects.

### 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).
- 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 $C \triangleright D$, extend $D$ by a (fresh) association $\text{uplink } C$: $\mu = [1, 1], \xi = +$ (Exercise: public necessary?)

Given expression $v$ (or $f$) in the context of class $D$,
- let $C$ be the smallest class wrt. "\sqsubseteq" such that
  - $C \sqsubseteq D$, and
  - $C :: v \in \text{atr}(D)$
- then there exists (by definition) $C \triangleright C_1 \triangleright \ldots \triangleright C_n \triangleright D$,
- normalise $v$ to ($= replace by$) $\text{uplink } C_n \rightarrow \ldots \rightarrow \text{uplink } C_1$. $C :: v$
- 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.
Casts in Domain Inclusion and Uplink Semantics

Let's consider the scenario where we have a UML model, and we want to ensure that our model satisfies certain constraints. We can do this by using transformers, which are essentially functions that transform one model into another while preserving certain properties.

### Domain Inclusion vs. Uplink Semantics

In the context of domain inclusion, we are interested in the relationships between classes and their attributes, ensuring that the relationships are maintained across different domains. On the other hand, Uplink semantics deals with the transformation of models, ensuring that the relationships are preserved during the transformation process.

### Cast Transformations

When dealing with casts, we need to ensure that the source model is transformed into the target model in a way that preserves the relationships. This involves creating transformers that can handle the cast operations correctly.

### Satisfying OCL Constraints (Uplink)

To satisfy OCL (Object Constraint Language) constraints in the context of Uplink semantics, we need to ensure that the casts are handled in a way that maintains the integrity of the constraints. This involves creating cast transformers that can handle the casts correctly, ensuring that the constraints are satisfied.

### Transformers (Uplink)

Transformers are crucial in handling casts, as they allow us to transform one domain into another while preserving the relationships. By defining the transformers correctly, we can ensure that the casts are handled in a way that satisfies the constraints.

### Identity Downcast with Uplink Semantics

Identity downcasts are used to change the type of an object to a more specific one. In the context of Uplink semantics, we need to ensure that the downcast operation is handled correctly, preserving the relationships between classes.

### Identity Upcast with Uplink Semantics

Identity upcasts are used to change the type of an object to a more general one. In the context of Uplink semantics, we need to ensure that the upcast operation is handled correctly, preserving the relationships between classes.

### Value Upcast with Uplink Semantics

Value upcasts are used to change the value of an object's attribute. In the context of Uplink semantics, we need to ensure that the upcast operation is handled correctly, preserving the relationships between classes.

### Value Downcast with Uplink Semantics

Value downcasts are used to change the value of an object's attribute. In the context of Uplink semantics, we need to ensure that the downcast operation is handled correctly, preserving the relationships between classes.

### Casts in Domain Inclusion and Uplink Semantics

Casts are a critical aspect of both domain inclusion and Uplink semantics. They allow us to transform models from one domain to another while preserving the relationships. By understanding how to handle casts correctly, we can ensure that our models are transformed in a way that satisfies the constraints.

### Constructing A Transformers (Uplink)

Constructing cast transformers requires a deep understanding of the relationships between classes and their attributes. By carefully designing the transformers, we can ensure that the casts are handled correctly, preserving the integrity of the relationships.

### Transformers (Uplink) - Continued

Transformers (Uplink) are an essential part of handling casts in the context of Uplink semantics. They allow us to transform models from one domain to another while preserving the relationships. By understanding how to construct these transformers, we can ensure that our models are transformed in a way that satisfies the constraints.

### Transformations (Uplink)

Transformations (Uplink) are an essential part of handling casts in the context of Uplink semantics. They allow us to transform models from one domain to another while preserving the relationships. By understanding how to perform these transformations, we can ensure that our models are transformed in a way that satisfies the constraints.

### Satisfying Casts (Uplink)

Satisfying casts (Uplink) is a critical aspect of handling models in the context of Uplink semantics. It involves ensuring that the casts are handled correctly, preserving the relationships between classes. By understanding how to satisfy these casts, we can ensure that our models are transformed in a way that satisfies the constraints.

### Constructing A Transformers (Uplink) - Continued

Constructing cast transformers requires a deep understanding of the relationships between classes and their attributes. By carefully designing the transformers, we can ensure that the casts are handled correctly, preserving the integrity of the relationships.

### Transformers (Uplink) - Continued

Transformers (Uplink) are an essential part of handling casts in the context of Uplink semantics. They allow us to transform models from one domain to another while preserving the relationships. By understanding how to construct these transformers, we can ensure that our models are transformed in a way that satisfies the constraints.

### Transformations (Uplink) - Continued

Transformations (Uplink) are an essential part of handling casts in the context of Uplink semantics. They allow us to transform models from one domain to another while preserving the relationships. By understanding how to perform these transformations, we can ensure that our models are transformed in a way that satisfies the constraints.

### Satisfying Casts (Uplink) - Continued

Satisfying casts (Uplink) is a critical aspect of handling models in the context of Uplink semantics. It involves ensuring that the casts are handled correctly, preserving the relationships between classes. By understanding how to satisfy these casts, we can ensure that our models are transformed in a way that satisfies the constraints.
Domain Inclusion vs. Uplink: 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 so dramatically.
As long as we're "early binding", that is...

Domain Inclusion vs. Uplink: Motivations

Exercise:
• What's the point of having the tedious adjustments of the theory if it can be approached technically?
• What's the point of having the tedious technical pre-processing if it can be approached cleanly in the theory?

More Interesting: Behaviour

Example: Behaviour of Kinds of Students

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one expects from sub-types,
which is the OO domain is closely related to inheritance:
The principle of type substitutability by [Liskov, 1988, Liskov and Wing, 1994].
"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..."

...shall be usable... "for UML

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

---

"...shall be usable... " for UML

---

More Interesting: Behaviour

---

---
Given:

\[ C_1 \times : \text{Int} f(\text{Int}) : \text{Int} \]

\[ C_2 \times : \text{Int} f(\text{Int}) : \text{Int} \]

\[ x : \text{Int} \]

\[ f(\text{Float}) : \text{Int} \]

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

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

Wanted:

- \( x > 0 \) also well-typed for \( D_1 \)
- Assignment \( \text{itsC}_1 := \text{itsD}_1 \) being well-typed
- \( \text{itsC}_1.x = 0 \), \( \text{itsC}_1.f(0) \), \( \text{itsC}_1!_F \) being well-typed (and doing the right thing).

Approach:

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

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.

Excursus: Late Binding of Behavioural Features

What transformer applies in what situation?

(Early (compile time) binding.)

\[ f \text{ not overridden in } D \]

\[ C f() : \text{Int} \]

\[ D f() : \text{Int} \]

\[ \text{valueofsomeC/someD} \]

\[ \text{someC} \rightarrow f() \]

\[ \text{someD} \rightarrow f() \]

(Late binding.)

\[ \text{someC} \rightarrow f() \]

\[ C f() \]

\[ \text{someD} \rightarrow f() \]

\[ D f() \]

\[ \text{someC} \rightarrow f() \]

\[ C f() \]

\[ \text{someD} \rightarrow f() \]

Late Binding in the Standard and in 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) 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 be the rationale of the designers of C++?

Note: late binding typically applies only to methods, not to attributes.

(But: getter/setter methods have been invented recently.)
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 a \( D \) instance. So we immediately also have behavioural/dynamic subtyping.

### Difficult: Dynamic Subtyping

\[
C \; f \left( \text{Int} \right) : \text{Int} \\
D \; f \left( \text{Int} \right) : \text{Int}
\]

\( C :: f \) 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);
}
```

### Subtyping Principles Cont'd

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).

### Ensuring Subtyping for State Machines

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...

### References


