Software Design, Modelling and Analysis in UML

Lecture 20: Inheritance III

2012-02-14

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- Inheritance and Sub-Typing
- Early vs. late binding of behavioural features
- Meta-Modelling

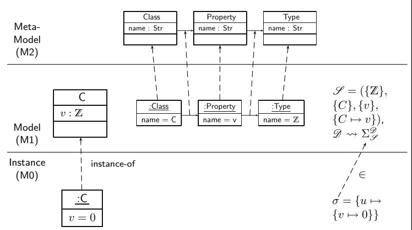
This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is the subset, what the uplink semantics of inheritance?
  - What's the effect of inheritance on LSCs, State Machines, System States?
  - What are anticipated benefits of Meta-Modelling?
  - What is MOF?
- **Content:**
  - MOF
  - Two approaches to obtain desired semantics: domain inclusion semantics and uplink semantics

Recall: Meta-Modelling Principle

Recall:

Modelling vs. Meta-Modelling



Meta Object Facility (MOF)

Open Questions...

- Now you've been "**tricked**" again. Twice.
  - We didn't tell what the **modelling language** for meta-modelling is.
  - We didn't tell what the **is-instance-of** relation of this language is.
- **Idea**: have a **minimal object-oriented core** comprising the notions of **class**, **association**, **inheritance**, **etc.** with "self-explaining" semantics.
- This is **Meta Object Facility** (MOF), which (more or less) coincides with UML Infrastructure [OMG, 2007a].
- So: things on meta level
  - M0 are object diagrams/system states (instances of classes in a UML model)
  - M1 are **words of the language UML** (OD over instances of classes in the meta-model of UML, or word of lang. MOF)
  - M2 are **words of the language MOF** (instances of MOF meta-model)
  - M3 are **words of the language** MOF

  ⋮

MOF Semantics

- One approach:
  - Treat it with **our signature-based theory**
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn't have a notion of Signal, our signature has.)
- Other approach:
  - Define a **generic**, **graph based** "is-instance-of" relation.
  - Object diagrams (that **are** graphs) then **are** the system states — not **only graphical representations** of system states.
  - If this works out, good: We can easily experiment with different language designs, e.g. different flavours of UML that immediately have a semantics.
  - Most interesting: also do generic definition of behaviour within a closed modelling setting, but this is clearly still research, e.g. [Buschermöhle and Oelerink, 2008]

Meta-Modelling: (Anticipated) Benefits

Benefits: Overview

- We'll (superficially) look at three aspects:
  - Benefits for **Modelling Tools**.
  - Benefits for **Language Design**.
  - Benefits for **Code Generation and MDA**.

Benefits for Modelling Tools

- The meta-model $\mathcal{M}_U$ of UML **immediately** provides a **data-structure** representation for the abstract syntax ($\sim$ for our signatures).

  If we have code generation for UML models, e.g. into Java, then we can immediately represent UML models **in memory** for Java.

  (Because each MOF model is in particular a UML model.)
- There exist tools and libraries called **MOF-repositories**, which can generically represent instances of MOF instances (in particular UML models).

  And which can often generate specific code to manipulate instances of MOF instances in terms of the MOF instance.

Benefits for Modelling Tools Cont'd

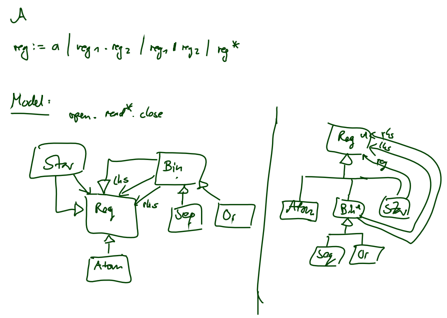
- And not only **in memory**, if we can represent MOF instances in files, we obtain a canonical representation of UML models **in files**, e.g. in XML.

  $\rightarrow$ XML Metadata Interchange (XMI)
- **Note**: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) $\rightarrow$ OMG Diagram Interchange.
- **Note**: There are slight ambiguities in the XMI standard.

  And different tools by different vendors often seem to lie at opposite ends on the scale of interpretation. Which is surely a coincidence.

  In some cases, it's possible to fix things with, e.g., XSLT scripts, but full vendor independence is today not given.

  Plus XMI compatibility doesn't necessarily refer to Diagram Interchange.
- **To re-iterate**: this is **generic for all** MOF-based modelling languages such as UML, CWM, etc.
  And also for **Domain Specific Languages** which don't even exit yet.



Benefits: Overview

- We'll (superficially) look at three aspects:
  - Benefits for **Modelling Tools**. ✔
  - Benefits for **Language Design**.
  - Benefits for **Code Generation and MDA**.

Benefits for Language Design

- Recall: we said that code-generators are possible "readers" of stereotypes.
- For example, (heavily simplifying) we could
  - introduce the stereotypes **Button**, **Toolbar**, ...
  - for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
  - instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. **corresponding** to the stereotype.

  Et voilà: we can model Gtk-GUIs and generate code for them.
- Another view:
  - UML with these stereotypes **is a new modelling language**: Gtk-UML.
  - Which lives on the same meta-level as UML (M2).
  - It's a **Domain Specific** Modelling **Language** (DSL).

One mechanism to define DSLs (based on UML, and "within" UML): **Profiles**.

Benefits for Language Design Cont'd

- For each DSL defined by a Profile, we immediately have
  - in memory representations,
  - modelling tools,
  - file representations.
- **Note**: here, the **semantics** of the stereotypes (and thus the language of Gtk-UML) **lies in the code-generator**.

  That's the first "reader" that understands these special stereotypes. (And that's what's meant in the standard when they're talking about giving stereotypes semantics).
- One can also impose additional well-formedness rules, for instance that certain components shall all implement a certain interface (and thus have certain methods available). (Cf. [Stahl and Völter, 2005].)

Benefits for Language Design Cont'd

- One step further:
  - Nobody hinders us to obtain a model of UML (written in MOF),
  - throw out parts unnecessary for our purposes,
  - add (= integrate into the existing hierarchy) more adequat new constructs, for instance, **contracts** or something more close to hardware as **interrupt** or **sensor** or **driver**,
  - and maybe also stereotypes.

  → a new language standing next to UML, CWM, etc.
- Drawback: the resulting language is not necessarily UML any more, so we **can't use** proven UML modelling tools.
- But we can use all tools for MOF (or MOF-like things).

  For instance, Eclipse EMF/GMF/GEF.

Benefits: Overview

- We'll (superficially) look at three aspects:
  - Benefits for **Modelling Tools**. ✔
  - Benefits for **Language Design**. ✔
  - Benefits for **Code Generation and MDA**.

Benefits for Model (to Model) Transformation

- There are manifold applications for model-to-model transformations:
  - For instance, tool support for **re-factorings**, like moving common attributes upwards the inheritance hierarchy.

    This can now be defined as **graph-rewriting** rules on the level of MOF.

    The graph to be rewritten is the UML model
  - Similarly, one could transform a **Gtk-UML** model into a **UML model**, where the inheritance from classes like Gtk::Button is made explicit:

    The transformation would add this class Gtk::Button and the inheritance relation and remove the stereotype.
  - Similarly, one could have a **GUI-UML** model transformed into a **Gtk-UML** model, or a Qt-UML model.

    The former a PIM (Platform Independent Model), the latter a PSM (Platform Specific Model) — cf. MDA.

Special Case: Code Generation

- Recall that we said that, e.g. Java code, can also be seen as a model.

  So code-generation is a **special case** of model-to-model transformation; only the destination looks quite different.

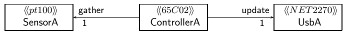
- **Note**: Code generation needn't be as expensive as buying a modelling tool with full fledged code generation.
  - If we have the UML model (or the DSL model) given as an XML file, code generation can be **as simple as** an XSLT script.

    "Can be" in the sense of

    "There may be situation where a graphical and abstract representation of something is desired which has a clear and direct mapping to some textual representation."

  In general, code generation can (in colloquial terms) become **arbitrarily difficult**.

Example: Model and XMI




```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Mon Feb 02 18:23:12 CET 2009'>
  <XMI.content>
    <UML:Model xmi.id = '...'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = '...' name = 'SensorA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name = 'pt100'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id = '...' name = 'ControllerA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name = '65C02'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id = '...' name = 'UsbA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name = 'NET2270'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Association xmi.id = '...' name = 'in' >...</UML:Association>
        <UML:Association xmi.id = '...' name = 'out' >...</UML:Association>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```


Domain Inclusion Semantics

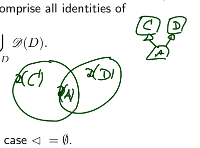
Domain Inclusion Structure

Let $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, F, mth, \lhd)$ be a signature.

Now a **structure** $\mathscr{D}$

- [**as before**] maps types, classes, associations to domains,
- [**for completeness**] methods to transformers,
- [**as before**] indentities of instances of classes not (transitively) related by generalisation are disjoint,
- [**changed**] the indentities of a super-class comprise all identities of sub-classes, i.e.

$$\forall C \in \mathscr{C} : \mathscr{D}(C) \supsetneq \bigcup_{C \lhd D} \mathscr{D}(D).$$



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

Domain Inclusion System States

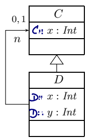
Now: a **system state** of $\mathscr{S}$ wrt. $\mathscr{D}$ is a **type-consistent** mapping

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

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

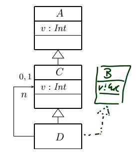
- [**as before**] $\sigma(u)(v) \in \mathscr{D}(\tau)$ if $v : \tau$, $\tau \in \mathscr{T}$ or $\tau \in \{C_*, C_{0,1}\}$.
- [**changed**] $\mathrm{dom}(\sigma(u)) = \bigcup_{C_0 \preceq C} atr(C_0)$,

Example:



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

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 $\lhd$.

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 occcurs 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 $C::v \in atr(C)$.

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

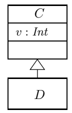
OCL Syntax and Typing

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

$$
\begin{aligned}
expr ::= \;& v(expr_1) && : \tau_C \to \tau(v), && \text{if } v : \tau \in \mathscr{T} \\
\mid\; & r(expr_1) && : \tau_C \to \tau_D, && \text{if } r : D_{0,1} \\
\mid\; & r(expr_1) && : \tau_C \to Set(\tau_D), && \text{if } r : D_*
\end{aligned}
$$

The definition of the semantics remains (textually) **the same**.

More Interesting: Well-Typed-ness



- We want

  context D inv : $v < 0$

  context self : D inv : self.v < 0

  v(self)

  to be well-typed.

  Currently it isn't because

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

  : τ_D

  but $A \vdash self : \tau_D$.

  (Because τ_D and τ_C are still **different types**, although $\mathrm{dom}(\tau_D) \subset \mathrm{dom}(\tau_C)$.)

- So, add a (first) new typing rule

$$\frac{A \vdash expr : \tau_D}{A \vdash expr : \tau_C}, \text{ if } C \preceq D. \qquad \text{(Inh)}$$

Which is correct in the sense that, if '$expr$' is of type τ_D, then we can use it everywhere, where a τ_C is allowed.

The system state is prepared for that.

Well-Typed-ness with Visibility Cont'd

$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = + \qquad \text{(Public)}$$

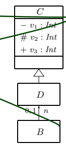
$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = \#,\ C \preceq D \qquad \text{(Protected)}$$

$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = -,\ C = D \qquad \text{(Private)}$$

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

Example:

context/ inv	$(n.)v_1 < 0$	$(n.)v_2 < 0$	$(n.)v_3 < 0$
C			
D			
B			



Satisfying OCL Constraints (Domain Inclusion)

- Let $\mathcal{M} = (\mathscr{CD}, \mathscr{OD}, \mathscr{SM}, \mathscr{I})$ be a UML model, and $\mathscr{D}$ a structure.
- We (**continue to**) say $\mathcal{M} \models expr$ for $\underbrace{\text{context } C \text{ inv} : expr_0}_{=expr} \in Inv(\mathcal{M})$ iff

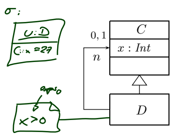
$$\forall\, \pi = (\sigma_i, \varepsilon_i)_{i \in \mathbb{N}} \in [\![\mathcal{M}]\!] \quad \forall\, i \in \mathbb{N} \quad \forall\, u \in \mathrm{dom}(\sigma_i) \cap \mathscr{D}(C) :$$
$$I[\![expr_0]\!](\sigma_i, \{self \mapsto u\}) = 1.$$

comprises all objects more special than C!

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

- **Example**:

  I[[self.x > 0]](σ, {self ↦ u})
  normalization
  I[[self. C::x > 0]](σ, {self ↦ u})
  =
  σ(β(self))(C::x) > 0
  D(D) ∩ D(C)   ∈ dom(σ(u))



Transformers (Domain Inclusion)

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

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

after normalisation e.g. C::v

with

$$\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma)]]$$

C::v

where $u = I[\![expr_1]\!](\sigma)$.

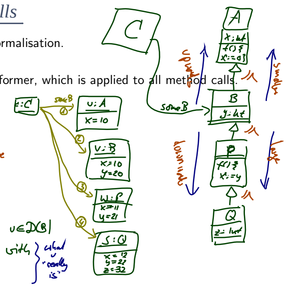
Semantics of Method Calls

- **Non late-binding**: clear, by normalisation.
- **Late-binding**:
  Construct a **method call** transformer, which is applied to all method calls.

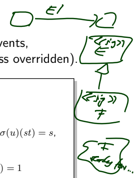
SM_C: /someB → f() (*)

someB	what σ(c)(someB) "really is"	implementation called
① u	A	-NOTHING- doesn't type
② v	B	A::f()
③ w	P	P::f()
④ s	Q	P::f()

- if we evaluate (*) for some u ∈ D(B)
- let D be the largest class with (what u "really is")
  - B ≤ D
  - u ∈ D(D)
- from D find the largest class C' with
  - C' ≤ D
  - C'::f ∈ mth(C')
    - call C'::f



Inheritance and State Machines: Triggers



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

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

- $\exists u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C) \exists u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$
- 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.

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

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.

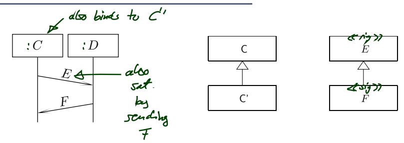
$$(\sigma'', \varepsilon') = t_{act}(\tilde{\sigma}, \varepsilon \ominus u_E),$$

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

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 (σ', ε').
  - Otherwise $b = 0$.
- Consumption of u_E and the side effects of the action are observed, i.e.

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

Domain Inclusion and Interactions



- 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.

$$\sigma, cons, Snd \models_\beta E^!_{x,y}$$

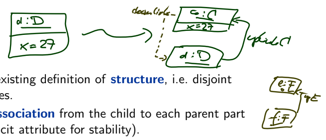
if and only if

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

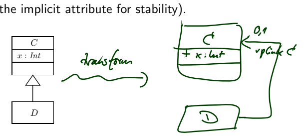
- **Note**: C-instance line also binds to C'-objects.

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).



- Apply (a different) pre-processing to make appropriate use of that association, e.g. rewrite (C++)

$$\mathtt{x} = 0;$$

in D to

$$\mathtt{uplink}_C \texttt{ -> } \mathtt{x} = 0;$$

Pre-Processing for the Uplink Semantics

- For each pair $C \lhd D$, extend D by a (fresh) association

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

(**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
    - $C::v \in atr(D)$
  - then there exists (by definition) $C \lhd C_1 \lhd \ldots \lhd C_n \lhd D$,
  - **normalise** v to (= replace by)

$$uplink_{C_n} \texttt{ -> } \cdots \texttt{ -> } uplink_{C_1}.C::v$$

- 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 $\mathcal{M} = (\mathscr{CD}, \mathscr{OD}, \mathscr{SM}, \mathscr{I})$ be a UML model, and $\mathscr{D}$ a structure.
- We (**continue to**) say
$$\mathcal{M} \models expr$$
for
$$\underbrace{\text{context } C \text{ inv} : expr_0}_{=expr} \in \mathit{Inv}(\mathcal{M})$$
if and only if
$$\forall\, \pi = (\sigma_i)_{i \in \mathbb{N}} \in [\![\mathcal{M}]\!]$$
$$\forall\, i \in \mathbb{N}$$
$$\forall\, u \in \mathrm{dom}(\sigma_i) \cap \mathscr{D}(C) :$$
$$I[\![expr_0]\!](\sigma_i, \{self \mapsto u\}) = 1.$$
- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $\mathit{Inv}(\mathcal{M})$.

Transformers (Uplink)

- What **has to change** is the **create** transformer:
$$create(C, expr, v)$$
- Assume, C's inheritance relations are as follows.
$$C_{1,1} \lhd \ldots \lhd C_{1,n_1} \lhd C,$$
$$\ldots$$
$$C_{m,1} \lhd \ldots \lhd C_{m,n_m} \lhd C.$$
- 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.

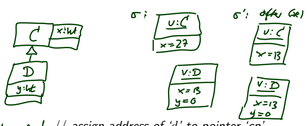
Late Binding (Uplink)

- 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;` ← address/id of 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

	Domain Inclusion	**Uplink**
`C* cp = &d;`	**easy**: immediately compatible (in underlying system state) because `&d` yields an identity from $\mathscr{D}(D) \subset \mathscr{D}(C)$.	**difficult**: we need the identity of the D whose C-slice is denoted by cp. (See next slide.)
`D* dp = (D*)cp;`	**easy**: the value of cp is in $\mathscr{D}(D) \cap \mathscr{D}(C)$ because the pointed-to object is a D. Otherwise, error condition.	**easy**: By pre-processing, `C* cp = d -> uplink`$_C$;
`c = d;`	**bit difficult**: set (for all $C \preceq D$) $(C)(\cdot,\cdot) : \tau_D \times \Sigma \to \Sigma\|_{atr(C)}$ $(u, \sigma) \mapsto \sigma(u)\|_{atr(C)}$ Note: $\sigma' = \sigma[u_C \mapsto \sigma(u_D)]$ is not type-compatible!	**easy**: By pre-processing, `c = *(d -> uplink`$_C$`);`

swap

Identity Downcast with Uplink Semantics

- **Recall** (C++): `D d;` `C* cp = &d;` `D* dp = (D*)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

$$\texttt{all} \in \mathscr{T}, \qquad \mathscr{D}(\texttt{all}) = \bigcup_{C \in \mathscr{C}} \mathscr{D}(C)$$

  - 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.

```
switch(mostspec_type){
  case C :
    dp = cp -> mostspec -> uplink_{D_n} -> ... -> uplink_{D_1} -> uplink_D;
...
}
```


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...

Domain Inclusion vs. Uplink Semantics: Motives

- **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

[Buschermöhle and Oelerink, 2008] Buschermöhle, R. and Oelerink, J. (2008). Rich meta object facility. In Proc. 1st IEEE Int'l workshop UML and Formal Methods.

[OMG, 2007a] OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

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

[Stahl and Völter, 2005] Stahl, T. and Völter, M. (2005). Modellgetriebene Softwareentwicklung. dpunkt.verlag, Heidelberg.