

Contents & Goals

- **Last Lecture:**
 - Inheritance in UML: desired semantics
- **This Lecture:**
 - Educational Objectives: Capabilities for following tasks/questions
 - What's the Liskov Substitution Principle?
 - What is 'leaf/early binding'?
 - What is the subject: what the uplink semantics of inheritance?
 - What's the effect of inheritance on LSCG, State Machines, System States?
 - What's the idea of Meta-Modelling?
- **Content:**
 - Meta-Modelling
 - Two approaches to obtain desired semantics

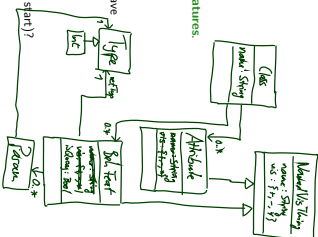
Meta-Modelling: Idea and Example

Meta-Modelling: Why and What

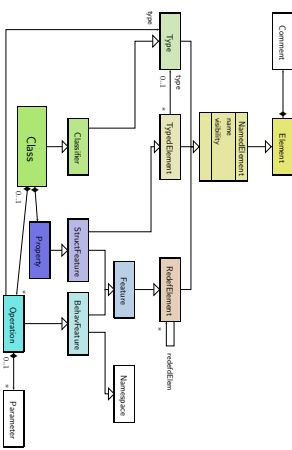
- **Meta-Modelling** is one major prerequisite for understanding
 - the standard documents [OMG, 2007a, OMG, 2007b], and
 - the MDA ideas of the OMG.
- The idea is **simple**:
 - if a **modelling language** is about modelling **things**,
 - and if **UML** models are and comprise **things**,
 - then why not **model** those in a modelling language?
- In other words:
 - Why not have a model M_U such that
 - the set of legal instances of M_U
 - is
 - the set of well-formed (!) UML models.

Meta-Modelling: Example

- For example, let's consider a class.
 - A class has (on a superficial level)
 - a name,
 - any number of **attributes**,
 - any number of **behavioural features**.
 - Each of the latter two has
 - a name and
 - a **visibility**.
 - Behavioural features in addition have
 - a boolean attribute **isQuery**,
 - any number of parameters,
 - a return type.
- Can we model this (in UML, for a start)?



UML Meta-Model: Extract



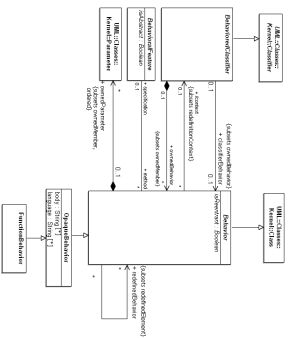


Figure 11.3 Common Behavior

- Meta-modelling has already been used for UML 1.x.
- For UML 2.0, the request for a proposal (RFP) asked for a separation of concerns Infrastructure and Superstructure.
- One reason: sharing with MOF (see later) and, e.g., CWM.

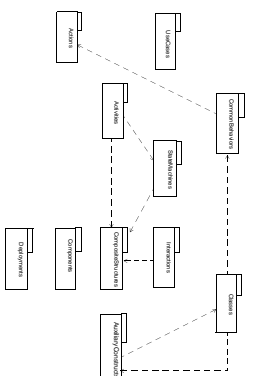
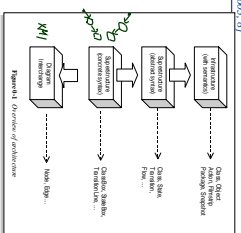
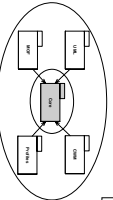
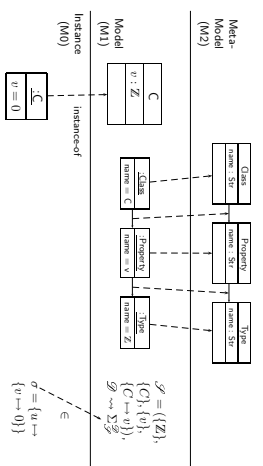


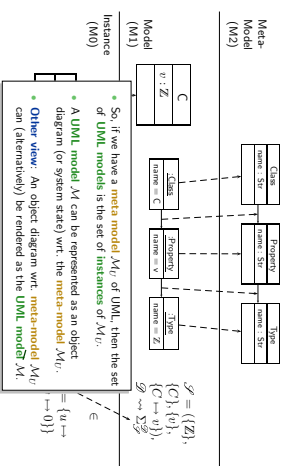
Figure 13.1 The superstructure packages of the UML 2.1 Superstructure

Meta-Modelling: Principle

Modelling vs. Meta-Modelling



Modelling vs. Meta-Modelling



Reading the Standard C++

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

Reading the Standard C++

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

Reading the Standard C++

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

Reading the Standard C++

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

Reading the Standard C++

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

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-referencing semantics
 - This is Meta Object Facility (MOF) which (more or less) coincides with UML Infrastructure [OMG, 2007a]
 - So: things on meta level
 - MO are object diagrams/system states
 - M1 are words of the language UML (Application/Systems of classes in the UML)
 - M2 are words of the language MOF (Instances of Meta-Models)
 - 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. [Bachemhle and Odehnik, 2008]

23/01

Met-Modelling: (Anticipated) Benefits

24/01

24/01

Benefits for Modelling Tools

- The meta-model *M₁* of UML **immediately** provides a **data-structure** representation for the abstract syntax (~ 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.

26/01

Benefits for Modelling Tools Cont'd

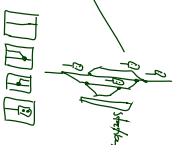
- And not **only in memory**, if we can represent MOF instances in files, we obtain a canonical representation of UML models **in files**, e.g. in XML. — XML Metadata Interchange (XMI)
- **Note:** A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) — OMG Diagram Interchange.
- **Note:** There are slight ambiguities in the XML 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 XML compatibility doesn't necessarily refer to Diagram Interchange.
- **To reiterate:** this is **generic for all MOF-based modelling languages** such as UML, CWM, etc. And also for **Domain Specific Languages** which don't even exist yet.

27/01

27/01

Benefits: Overview

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



28/01

28/01

Benefits: Overview

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

28/01

28/01

- 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 Gk::Button, Gk::Toolbar, etc: **corresponding** to the stereotype.
- **Et voilà**: we can model Gk-GUIs and generate code for them.
- Another view:
 - UML with these stereotypes is a new modelling language: Gk-UML
 - Which lives on the same meta-level as UML (M2).
 - It's a **Domain Specific Modelling Language (DS1)**.

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

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

- 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 Gk-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, 2009].)

- There are manifold applications for model-to-model transformations:
 - For instance, tool support for **refactorings** 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 Gk-UML model into a **UML model**, where the inheritance from classes like Gk::Button is made explicit: The transformation would add this class Gk::Button and the inheritance relation and remove the stereotype.
 - Similarly, one could have a **GUI-UML** model transformed into a **Gk-UML** model, or a **Qt-UML** model.
- The former a PIM (Platform Independent Model), the latter a PSM (Platform Specific Model) — cf. MDA.

- 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 adequate 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, CMM, 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.

- 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 XML



```

<?xml version="1.0" encoding="UTF-8" ?>
<ctrl xmlns:version="1.2" xmlns:ctrl="http://www.omg.org/xml/semantics/ctrl" xmlns:lang="http://www.omg.org/xml/semantics/ctrl" ?>
  <ctrl:Model1 ctrl:id="..." ?>
    <ctrl:Namespace control="http://www.omg.org/xml/semantics/ctrl" ?>
      <ctrl:Class ctrl:id="..." name="ControlA" ?>
        <ctrl:Class ctrl:id="..." name="ControlB" ?>
          <ctrl:Association ctrl:id="..." name="ControlAtoB" ?>
            <ctrl:Association ctrl:id="..." name="ControlBtoA" ?>
          </ctrl:Association>
        </ctrl:Class>
      </ctrl:Namespace>
    </ctrl:Model1>
  </ctrl:Model>
</ctrl:Model>

```

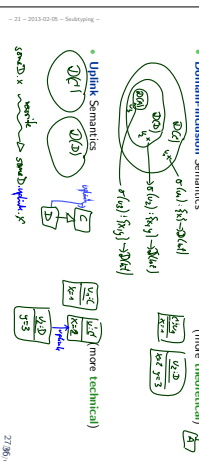
Recall

Towards System States

Wanted: a formal representation of “if $C \leq D$ 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.

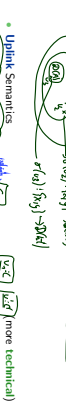
We'll discuss **two approaches** to semantics:



Domain Inclusion Semantics

Domain Inclusion Semantics

(more theoretical)



Domain Inclusion Structure

Let $\mathcal{S} = (\mathcal{C}, \mathcal{E}, V, attr, \mathcal{E}, F, mhd, \triangleleft)$ be a signature.

Now a structure \mathcal{D}

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

$$\forall C \in \mathcal{C} : \mathcal{D}(C) \supseteq \bigcup_{C' \triangleleft C} \mathcal{D}(C')$$

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

Domain Inclusion System States

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

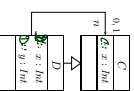
$$\sigma : \mathcal{D}(\mathcal{C}) \rightarrow (V \rightarrow ((\mathcal{D}(\mathcal{C}) \cup \mathcal{D}(R_{0,1})) \cup \mathcal{D}(E_{2,1})))$$

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

- [as before] $\sigma(u)(a) \in \mathcal{D}(r)$ if $r : \tau \in \mathcal{S}$ or $\tau \in \{C_+, C_{0,1}\}$.
- [changed] $\text{dom}(\sigma(u)) = \bigcup_{C' \triangleleft C} \mathcal{D}(C')$.

Example: $u \in \mathcal{D}(D)$

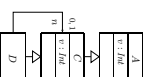
$$\sigma(u)(c_0) = \sigma(u)(c_1) \cup \sigma(u)(c_2) = \{D_0, D_1, D_2, D_3, D_4\}$$



Preliminaries: Expression Normalisation

Recall:

- we want to allow, e.g., “context D inv $\underline{A} < 0'$ ”.
- we assume fully qualified names, e.g. $C::x$.
- Intuitively, \underline{A} shall denote the “most special more general” $C::x$ according to \triangleleft .



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 J) 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 occurs belongs to,
 - similar for method bodies.
 - **normalise** v to (\equiv replace by) $C::v$,
 - where C is the **greatest** class wrt. “ \leq ” such that
 - $C \leq D$ and $C::v \in attr(C)$.
- 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:

$$\begin{aligned} \text{expr} ::= & v(\text{expr}_1) & : \tau_C \rightarrow \tau(v), & \text{if } v : \tau \in \mathcal{V} \\ & f(\text{expr}_1) & : \tau_C \rightarrow \tau_D, & \text{if } f : \tau : D_{0,1} \\ & \text{if } (\text{expr}_1) & : \tau_C \rightarrow \text{Set}(\tau_D), & \text{if } \tau : D_* \\ & \text{if } (\text{expr}_1) & : \tau_C \rightarrow \text{Set}(\tau_D), & \text{if } \tau : D_* \end{aligned}$$

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

More Interesting: Well-Typed-ness

- We want context $D \text{ inv} : v \leq 0$ to be well-typed. Currently it isn't because $v(\text{expr}_1) : \tau_C \rightarrow \tau(v)$ but $A \vdash \text{self} : \tau_D$. (Because τ_D and τ_C are still different types, although $\text{dom}(\tau_D) \subset \text{dom}(\tau_C)$)
- So add a (first) new typing rule

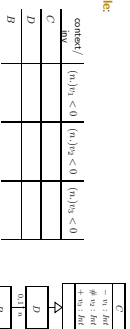
$$\frac{A \vdash \text{expr}_1 : \tau_D \quad \text{if } C \leq D}{A \vdash \text{expr}_1 : \tau_C} \text{ (In)}$$

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

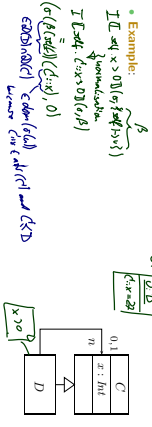
- $A, D \vdash \text{expr}_1 : \tau_C$ (Pub)
- $A, D \vdash C::v(\text{expr}_1) : \tau^+$ $\xi = +$
- $A, D \vdash \text{expr}_1 : \tau_C$ $\xi = \#$, $C \leq D$ (Prot)
- $A, D \vdash C::v(\text{expr}_1) : \tau^+$ $\xi = \#$, $C = D$ (Prot)
- $A, D \vdash \text{expr}_1 : \tau_C$ $\xi = -$, $C = D$ (Prot)
- $(C::v : \tau, \xi, \text{obj}, P) \in \text{atr}(C)$.

Example:



Satisfying OCL Constraints (Domain Inclusion)

- Let $\mathcal{M} = (\mathcal{C}, \mathcal{G}, \mathcal{D}, \mathcal{H}, \mathcal{J})$ be a UML model, and \mathcal{S} a structure.
- We (continue to) say $\mathcal{M} \models \text{expr}$ for context $C \text{ inv} : \text{expr}_0 \in \text{Inv}(\mathcal{M})$ iff $\llbracket \text{expr} \rrbracket_{\llbracket C, \text{self} \rrbracket} = 1$.
- \mathcal{M} is (still) consistent if and only if it satisfies all constraints in $\text{Inv}(\mathcal{M})$.



Transformers (Domain Inclusion)

- Transformers also remain the same, e.g. [VL 12, p. 18] $\text{update}(\text{expr}_1, v, \text{expr}_2) : (v, \varepsilon) \mapsto (v', \varepsilon)$ with $v' = v[v \mapsto v']$ where $u = \llbracket \text{expr}_1 \rrbracket(v)$.

Semantics of Method Calls

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

Inheritance and State Machines: Triggers

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

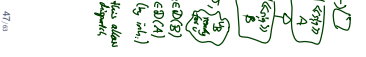
DISPATCH

$$(s, s) \xrightarrow{\text{trans, send}} (s', s') \text{ if } \begin{cases} \exists u \in \text{Ident}(C) \cap \mathcal{D}(C) \exists \text{msg} \in \mathcal{M}(s') : \text{msg} \in \text{recv}(C, u) \\ u \text{ is stable and in state machine state } s, \text{ i.e. } \sigma(u)(\text{stable}) \neq 1 \text{ and } \sigma(u)(s) = s_u \\ \text{a transition is enabled, i.e.} \\ \exists (s, F, \text{expr}, \text{act}, s') \in \text{SM}(C) : F = E \wedge \text{Trans}[s] = 1 \end{cases}$$

where $\delta = \sigma(u, \text{param}_s \rightarrow \text{val}_s)$

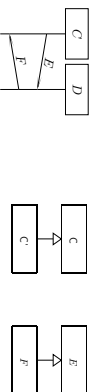
and

- (s', s') results from applying act to (s, s) and removing val from the stack, i.e. $(s', s') = \text{act}(\delta, \sigma \in \text{val}_s)$
- if u becomes stable in s' , then $\delta = 1$. It does become stable if and only if there is no transition without trigger enabled for u in (s', s') .
- Otherwise $\delta = 0$.
- Consumption of msg and the side effects of the action are observed, i.e. $\text{cons} = \{(u, E, \sigma(u))\}$, $\text{Send} = \text{Obs}_{s_u}(s' \in \text{val}_s)$.



47/60

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.

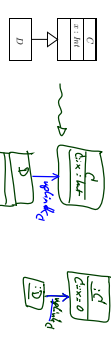
$$\sigma, \text{cons}, \text{Send} \models_{\beta} F_{x, \theta}$$
 if and only if

$$\beta(x) \text{ sends an } F\text{-event to } \beta(y) \text{ where } E \preq F.$$
- Note: C -instance line also binds to C -objects.

48/60

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) preprocessing to make appropriate use of that association, e.g. rewrite $(C++)$

$$x = 0;$$
 in D to

$$\text{uplink}_C \rightarrow x = 0;$$

50/60

Pre-Processing for the Uplink Semantics

- For each pair $C \triangleleft D$, extend D by a (fresh) association $\text{uplink}_C : C$ with $\mu = [1, 1]$, $\xi = +$ (Exercise: public necessary?)
- Given expression v (or J) in the context of class D ,
 - let C be the **smallest** class wrt. " \preceq " such that
 - $C \preceq D$, and
 - $C::x \in \text{attr}(D)$
 - then there exists (by definition) $C \triangleleft C_1 \triangleleft \dots \triangleleft C_n \triangleleft D$,
 - normalise** v to ($=$ replace by)

$$\text{uplink}_{C_n} \rightarrow \dots \rightarrow \text{uplink}_{C_1} \cdot C::v$$
- Again, if no (unique) smallest class exists the model is considered **not well-formed**: the expression is ambiguous.

51/60

Uplink Semantics

- Definition of structure remains **unchanged**.
- Definition of system state remains **unchanged**.
- Typing and transformers remain **unchanged** — the preprocessing has put everything in shape

52/60

52/60

- Let $\mathcal{M} = (\mathcal{G}, \mathcal{D}, \mathcal{R}, \mathcal{I}, \mathcal{J})$ be a UML model, and \mathcal{G} a structure.
- We (continue to) say $\mathcal{M} \models_{\text{expr}}$ for
- context C , $\text{inv} : \text{expr}_0 \in \text{Inv}(\mathcal{M})$
- if and only if
 - $\forall \tau = (a)_{i \in \mathbb{N}} \in \llbracket \mathcal{M} \rrbracket$
 - $\forall i \in \mathbb{N}$
 - $\forall u \in \text{dom}(a_i) \cap \mathcal{D}(C)$:
 - $\llbracket \text{expr} \rrbracket[a_i, [a_i/\tau \mapsto u]] = 1$.
- \mathcal{M} is (still) consistent if and only if it satisfies all constraints in $\text{Inv}(\mathcal{M})$.

Domain Inclusion vs. Uplink Semantics

- What has to change is the create transformer:
 - $\text{creat}_c(C, \text{expr}, v)$
- Assume, C 's inheritance relations are as follows.
 - $C_{1,1} \triangleleft \dots \triangleleft C_{1,n} \triangleleft C_i$
 - \dots
 - $C_{m,1} \triangleleft \dots \triangleleft C_{m,m} \triangleleft C_i$
- Then, we have to
 - create one fresh object for each part, e.g.
 - $m_{1,1}, \dots, m_{1,n}, m_{1,1}, \dots, m_{1,n}, \dots, m_{m,1}, \dots, m_{m,m}$
 - set up the uplinks recursively, e.g.
 - $\sigma^{(v_{1,2})}(\text{uplink}_{C_{1,1}}) = m_{1,1}$
- And, if we had constructors, be careful with their order.

- Employ something similar to the "worsepec" trick (in a minute). But the result is typically far from concise. (Related to OCL's `isKindOf()` function, and RTTI in C++)

Cast-Transformers

- C \Leftarrow d :
- Identity upcast (C++):
 - $C \ast \text{cp} = \text{kd}$;
 - // assign address of 'd' to pointer 'cp'
- Identity downcast (C++):
 - $D \ast \text{dp} = (D \ast) \text{cp}$;
 - // assign address of 'd' to pointer 'dp'
- Value upcast (C++):
 - $\ast \text{q} = \ast \text{p}$;
 - // copy attribute values of 'd' into 'c'; σ ;
 - // more precisely, the values of the C-part of 'd'

Casts in Domain Inclusion and Uplink Semantics

	Domain Inclusion	Uplink
$C \ast \text{cp} = \text{kd}$	easy: immediately compatible (in underlying system state), because <code>kd</code> yields an identity from $\mathcal{D}(D) \subset \mathcal{D}(C)$	easy: By pre-processing $C \ast \text{cp} = \text{dup}(\text{kd}, C)$
$D \ast \text{dp} = (D \ast) \text{cp}$	easy: the value of <code>cp</code> is in $\mathcal{D}(D) \cap \mathcal{D}(C)$ because the pointed-to object is a D . Otherwise, error condition.	difficult: we need the identity of the D whose C-slice is denoted by <code>cp</code> . (See next slide.)
$c = d$	bit difficult: set (for all $C \triangleleft D$) $(C) \ast \cdot : \mathcal{D} \times \Sigma \rightarrow \Sigma_{\text{inv}(C)}$ $(u, \sigma) \mapsto \sigma(u)_{\text{inv}(C)}$. Note: $\sigma' = \sigma _{\text{Inv}(C)}$ is not type-compatible!	easy: By pre-processing $c = \#(d, \text{uplink}_{C_i})$



- **Recall** (C++): $D \cdot d$, $C^* \text{ cp} = \text{kcd}$, $D^* \text{ dp} = (D) \cdot \text{cp}$:
- **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

$$\text{all } \in \mathcal{D}, \quad \mathcal{D}(\text{all}) = \bigcup_{C \in \mathcal{E}} \mathcal{D}(C)$$

- In each \leq -minimal class have associations “**astepic**” pointing to **most specialised** slices, plus information of which type that slice is.
- Then **downcast** means, depending on the **astepic** type (only finitely many possibilities), **going down and then up** as necessary, e.g.

```

struct InAstepicType {
    case C :
        dp = cp -> mostspec -> uplinkC -> ... -> uplinkC1 -> uplinkC2
}

```

- **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 need 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 **doesn't** improve conciseness **so dramatically**.

As long as we're “early binding”, that is...

- **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

[Buschermöhle and Odehrik, 2008] Buschermöhle, R. and Odehrik, J. (2008). Rich meta object facility. In *Proc. 18th IEEE Int'l Workshop UML and Formal Methods*.

[OMG, 2003] OMG (2003). Uml 2.0 proposal of the 30 group, version 0.2. <http://www.omg.org/uml2/index.shtm>.

[OMG, 2009a] OMG (2009a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-02.

[OMG, 2009b] OMG (2009b). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

[Stall and Völter, 2008] Stahl, T. and Völter, M. (2005). Modelldriven Softwareentwicklung. dpunkt-verlag, Heidelberg.