Contents & Goals

Last Lecture:
- Inheritance and Sub-Typing
- Early vs. late binding of behavioral 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

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
  - M1 are words of the language UML
  - M2 are words of the language MOF
  - M3 are words of the language MOF
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.
- Most interesting: also do generic definition of behaviour within a closed modelling setting, but this is clearly still research, e.g. [Buschermohle and Oelerink, 2008]

Benefits for Modelling Tools
- The meta-model \( M_U \) 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.

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 Meta-data 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 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 is possible to fix XMI 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 exist 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-generation is possible "readers" of stereotypes.
- For example, (heavily simplifying) we could:
  - Introduce the stereotypes Button, Toolbox, ...
  - 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 Glk::Button, Glk::Toolbar, etc. corresponding to the stereotype.
  - Et voilà — we can model Glk-GUIs and generate code for them.
- Another view:
  - UML with these stereotypes is a new modelling language: Glk-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 Glk-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 Volter, 2005.)

Benefits for Model (to Model) Transformation

- One step further:
  - Nobody hinders us to obtain a model of UML (written in MOF), throwout 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, 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/GEN/GEF.
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 different.
- Note: Code generation needs’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.

Domain Inclusion Structure

Let \( \mathcal{D} = (\mathcal{F}, \mathcal{V}, \mathcal{O}, \mathcal{P}, \mathcal{C}, \mathcal{S}, \mathcal{W}, \mathcal{X}) \) be a signature.

Now a structure is \( \langle X, Y, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \) a substructure, i.e.

- \( \langle [\text{from}] \rangle \) maps types, classes, associations to domains.
- \( \langle \text{[for completeness]} \rangle \) methods to transformers.
- \( \langle \text{[as before]} \rangle \) domains of instances of classes (not transitively) related by generalization are disjoint,
- \( \langle \text{[changed]} \rangle \) the identities of a sub-class are all identities of sub-classes, i.e.

\[ \forall C \subseteq X : \forall (C') \subseteq (X) \exists C \in \{C(C), C(C') \} \]

Note: the old setting coincides with the special case \( \emptyset = \emptyset \).

Domain Inclusion System States

Now, a system state is \( \langle x \rangle \) in \( \mathcal{D} \) if \( \forall y \in \mathcal{D} \exists x \in \mathcal{D} \).

That is, for all \( v \in \mathcal{D} \) is \( \exists \forall (y \in \mathcal{D}) \).

- \( \langle [\text{as before}] \rangle \) domains are \( \forall (y \in \mathcal{D}) \)
- \( \langle \text{all states } \rangle \) are \( \forall (y \in \mathcal{D}) \)
- \( \langle \text{all actions } \rangle \) are \( \forall (y \in \mathcal{D}) \)

Example:

\[ \langle x \rangle = \langle y \rangle \]

Note: the old setting still coincides with the special case \( \emptyset = \emptyset \).

Preliminaries: Expression Normalisation

Recall
- \( \emptyset = \emptyset \)
- \( \emptyset = \emptyset \)
- \( \emptyset = \emptyset \)
- \( \emptyset = \emptyset \)
- \( \emptyset = \emptyset \)

To keep the list of typing rules, we assume that the following normalisation has been applied to all OCL expressions and their actions:

- Given expression \( e \) in context of class \( C \), 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 \( e \) to \( \langle e \rangle \) replace by \( C \cdot \langle e \rangle \)
- where \( C \) is the greatest class \( \emptyset \) such that
- \( C \subseteq D \) and \( C \cdot \langle e \rangle \in \emptyset \).

If no \( \langle e \rangle \) such class, the model is considered not well-formed; the expression is ambiguous. Then: explicitly provide the qualified name.
Construct a transformer, which is applied to all method calls.

$$\text{update}$$ the same system state is prepared for that.

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

$\text{semanticsofmethodcalls}$

$\text{well-typedness}$
Inheritance and State Machines: Triggers

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

\[
\text{Uplink Structure, System State, Typing}
\]

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

\[
\text{Pre-Processing for the Uplink Semantics}
\]

- For each pair \( C \in D \), assign \( D \) by a (least) association

\[
\text{uplink}_{\text{C}} \to C
\]

(Exercise: public necessity?)

- Given expression \( = (w, f) \) in the context of class \( D \),
  - let \( C \) be the smallest class such that
    - \( C \leq D \), and
    - \( C \in \text{class}(D) \)
  - then there exists (by definition) \( C_1 < \cdots < C_n < D \),

\[
\text{normalisation} = \{ = \text{replace by} \}
\]

\[
\text{uplink}_{\text{C}} \to \text{uplink}_{\text{C}} \cdot C
\]

Again: if no (unique) smallest class exists, the model is considered not well-formed, the expression is ambiguous.

\[
\text{Domain Inclusion and Interactions}
\]

- 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.

\[
\pi, \text{now, find } \pi_{E_{C}}^{L E_{F}}
\]

if and only if

\[
\beta(x) \text{ sends an } E_{x} \text{ event to } I_{y} \text{ where } E \leq F.
\]

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

\[
\text{Uplink Semantics}
\]

- 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 \([C \cup +] \)

\[
\text{uplink}_{\text{C}} \to \text{uplink}_{\text{C}} \cdot C
\]

in \( D \) to

\[
\text{uplink}_{\text{C}} \to x \in \text{B}
\]
Satifying OCL Constraints (Uplink)

- Let $M = (C, \phi)$ be a UML model, and $\phi$ a structure.
- We (continue to) say $M \models \phi$ for
  
  $\text{context } C \models \text{expr} : \text{expr} \in \text{locs}(M)$
  
  if and only if
  
  $\forall x = \{x\}, x \in [M]
  \forall y \in \mathbb{R}
  \forall z \in \text{dom}(\sigma) \cap \text{dom}(\phi(C))
  \forall \alpha \in \mathbb{R}.
  \{x, y, z \rightarrow \alpha\} \subseteq L$.
- $M$ is (still) consistent if and only if it satisfies all constraints in $\text{locs}(M)$.

Transformers (Uplink)

- What has to change is the create transformer:
  
  $\text{create}(C, \phi)$
  
  $\triangleright$ Assume, $C$’s inheritance relations are as follows:
  
  $C_1 \subset \ldots \subset C_n \subset C$
  
  Then, we have to
  
  - create one fresh object for each part, e.g.
    
    $v_1, \ldots, v_n, \ldots, v_m$,
  
  - set up the uplinks recursively, e.g.
    
    $\sigma(v_1)(\text{uplink}(v_2)) = v_3$
  
  - And, if we had constructors, be careful with their order.

Cast-Transformers

- $C$;
- $\phi$;
- Identity upcast ($C \rightarrow$);
- $\phi$;
- Identity downcast ($C \rightarrow$);
- $\phi$;
- Value upcast ($C \rightarrow$);
- $\phi$;

\begin{align*}
\text{Cast-Transformers} & \\
\text{Domain Inclusion vs. Uplink Semantics} & \\
\text{Late Binding (Uplink)} & \\
\text{Casts in Domain Inclusion and Uplink Semantics} &
\end{align*}
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.
  all ∈ T , (p)(all) or \exists \phi 
- In each \phi- minimal class have associations “most spec” pointing to most specialised slice, plus information of which type that slice is.
- Then downcast means, depending on the “most spec” type (only finitely many possibilities), going down and then up as necessary, e.g.
  ```
  case C:
    dp = cp \rightarrow most spec \rightarrow uplink D_n \rightarrow uplink D_1 \rightarrow 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 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 also doesn’t improve conciseness so 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?
  - having the tedious technical pre-processing?

References