Software Design, Modelling and Analysis in UML

Lecture 07: A Type System for Visibility

2013-11-18

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

Contents & Goals

Last Lecture:
- Representing class diagrams as (extended) signatures — for the moment without associations (see Lecture 08).
- And: in Lecture 03, implicit assumption of well-typedness of OCL expressions.

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - Is this OCL expression well-typed or not? Why?
  - How/in what form did we define well-definedness?
  - What is visibility good for?

- Content:
  - Recall: type theory/static type systems.
  - Well-typedness for OCL expression.
  - Visibility as a matter of well-typedness.
**Extended Classes**

From now on, we assume that each class \( C \in \mathcal{C} \) has:

- a finite (possibly empty) set \( S_C \) of **stereotypes**,  
- a boolean flag \( a \in B \) indicating whether \( C \) is **abstract**,  
- a boolean flag \( t \in B \) indicating whether \( C \) is **active**.

We use \( S_\mathcal{C} \) to denote the set \( \bigcup_{C \in \mathcal{C}} S_C \) of stereotypes in \( \mathcal{S} \).

(Alternatively, we could add a set \( St \) as 5-th component to \( \mathcal{S} \) to provides the stereotypes (names of stereotypes) to choose from. But: too unimportant to care.)

**Convention:**

- We write \( \langle C, S_C, a, t \rangle \in \mathcal{C} \)
  when we want to refer to all aspects of \( C \).
- If the new aspects are irrelevant (for a given context),  
  we simply write \( C \in \mathcal{C} \) i.e. old definitions are still valid.
Extended Attributes

- From now on, we assume that each attribute \( v \in V \) has (in addition to the type):
  - a visibility
    
    \[ \xi \in \{ \text{public, private, protected, package} \} \]

  - an initial value \( expr_0 \) given as a word from language for initial values, e.g. OCL expressions.
    (If using Java as action language (later) Java expressions would be fine.)
  - a finite (possibly empty) set of properties \( P_v \).
    We define \( P_v \) analogously to stereotypes.

Convention:
- We write \( \langle v : \tau, \xi, expr_0, P_v \rangle \in V \) when we want to refer to all aspects of \( v \).
- Write only \( v : \tau \) or \( v \) if details are irrelevant.

From Class Boxes to Extended Signatures

A class box \( n \) induces an (extended) signature class as follows:

\[
V(n) := \{(v_1 : \tau_1, \xi_1, v_{0,1}, \{P_{1,1}, \ldots, P_{1,m_1}\}), \ldots, (v_\ell : \tau_\ell, \xi_\ell, v_{0,\ell}, \{P_{\ell,1}, \ldots, P_{\ell,m_\ell}\})\}
\]

where
- "abstract" is determined by the font:
  \( a(n) = \begin{cases} 
  \text{true} & \text{if } n = \text{C} \text{ or } n = \text{C (..)} \\
  \text{false} & \text{otherwise}
  \end{cases} \)
- "active" is determined by the frame:
  \( t(n) = \begin{cases} 
  \text{true} & \text{if } n = \text{C} \text{ or } n = \text{C} \\
  \text{false} & \text{otherwise}
  \end{cases} \)

\( \text{attr}(n) := \{ C \mapsto \{ v_1, \ldots, v_\ell \} \} \)
Excursus: Type Theory (cf. Thiemann, 2008)

---

Type Theory

**Recall:** In lecture 03, we introduced OCL expressions with types, for instance:

\[
expr ::= \begin{array}{ll}
w & : \tau \quad \ldots \text{logical variable } w \\
\text{true} | \text{false} & : \text{Bool} \quad \ldots \text{constants} \\
0 | -1 | 1 | \ldots & : \text{Int} \quad \ldots \text{constants} \\
expr_1 + expr_2 & : \text{Int} \times \text{Int} \rightarrow \text{Int} \quad \ldots \text{operation} \\
\text{size}(expr) & : \text{Set}(\tau) \rightarrow \text{Int} \\
\text{not } expr & : \text{Bool} \rightarrow \text{Bool}
\end{array}
\]

We then say \( expr \) is well-typed if and only if we can derive

\[ A, C \vdash expr : \tau \quad \text{(read: "expression } expr \text{ has type } \tau") \]

for some OCL type \( \tau \), i.e. \( \tau \in T_B \cup T_{\mathcal{E}} \cup \{ \text{Set}(\tau_0) | \tau_0 \in T_B \cup T_{\mathcal{E}} \} \), \( C \in \mathcal{E} \).
We will give a finite set of type rules (a type system) of the form

\[ (\text{"name"}) \frac{\text{"premises"}}{\text{"conclusion"}} \text{"side condition"} \]

These rules will establish well-typedness statements (type sentences) of three different “qualities”:

(i) Universal well-typedness:
\[ \vdash expr : \tau \]
\[ \vdash 1 + 2 : \text{Int} \]

(ii) Well-typedness in a type environment \( A \): (for logical variables)

\[ A \vdash expr : \tau \]
\[ self : \tau_C \vdash self . v : \text{Int} \]

(iii) Well-typedness in type environment \( A \) and context \( B \): (for visibility)

\[ A, B \vdash expr : \tau \]
\[ self : \tau_C, C \vdash self . r . v : \text{Int} \]
### Constants and Operations

- If \( \text{expr} \) is a **boolean constant**, then \( \text{expr} \) is of type \( \text{Bool} \):
  \[
  \text{(BOOL)} \quad \vdash B : \text{Bool}, \quad B \in \{\text{true}, \text{false}\}
  \]

- If \( \text{expr} \) is an **integer constant**, then \( \text{expr} \) is of type \( \text{Int} \):
  \[
  \text{(INT)} \quad \vdash N : \text{Int}, \quad N \in \{0, 1, -1, \ldots\}
  \]

- If \( \text{expr} \) is the application of **operation** \( \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \) to expressions \( \text{expr}_1, \ldots, \text{expr}_n \) which are of type \( \tau_1, \ldots, \tau_n \), then \( \text{expr} \) is of type \( \tau \):
  \[
  \text{(Fun}_0 \text{)} \quad \vdash \text{expr}_1 : \tau_1 \quad \vdash \cdots \quad \vdash \text{expr}_n : \tau_n \quad \vdash \omega(\text{expr}_1, \ldots, \text{expr}_n) : \tau, \quad \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin \text{atr}(\mathcal{E})
  \]

(Note: this rule also covers ‘=’, ‘is empty’, and ‘size’.)

### Constants and Operations Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BOOL)</td>
<td>( \vdash B : \text{Bool}, ) ( B \in {\text{true}, \text{false}} )</td>
<td><strong>Boolean</strong></td>
</tr>
<tr>
<td>(INT)</td>
<td>( \vdash N : \text{Int}, ) ( N \in {0, 1, -1, \ldots} )</td>
<td><strong>Integer</strong></td>
</tr>
<tr>
<td>(Fun(_0))</td>
<td>( \vdash \text{expr}_1 : \tau_1 \quad \vdash \cdots \quad \vdash \text{expr}_n : \tau_n \quad \vdash \omega(\text{expr}_1, \ldots, \text{expr}_n) : \tau, ) ( \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, ) ( n \geq 1, \omega \notin \text{atr}(\mathcal{E}) )</td>
<td><strong>Function</strong></td>
</tr>
</tbody>
</table>

**Example:**

- **not** \( \text{true} \)  
  \[
  \text{(Def)} \quad \vdash \text{true} : \text{Bool} \quad \vdash \text{not} : \text{Bool} \rightarrow \text{Bool} \]

- **true + 3**  
  got stuck — we cannot  
  \( \text{false} \)  
  \( \text{false} \)  
  \( \text{false} \)  
  \( \text{false} \)  
  \( \text{false} \)  
  \( \text{false} \)  

  \[
  \text{(Def)} \quad \vdash \text{true} : \text{Bool} \quad \vdash \text{3} : \text{Int} \quad \vdash + : \text{Int} \times \text{Int} \rightarrow \text{Int} \]

  \( \text{(Fun)} \quad \vdash \text{true + 3} : \text{Int} \]

\( \Box \) \( \text{true + 3} \) **is not well-typed**
**Type Environment**

- **Problem**: Whether
  \[ w + 3 \]
  is well-typed or not depends on the type of logical variable \( w \in W \).

- **Approach**: **Type Environments**

  **Definition.** A type environment is a (possibly empty) finite sequence of type declarations. The set of type environments for a given set \( W \) of logical variables and types \( T \) is defined by the grammar
  
  \[ A ::= ∅ | A, w : τ \]
  
  where \( w \in W \), \( τ \in T \).

  **Clear**: We use this definition for the set of OCL logical variables \( W \) and the types \( T = T_B \cup T_E \cup \{ \text{Set}(τ_0) \mid τ_0 \in T_B \cup T_E \} \).

**Environment Introduction and Logical Variables**

- If \( expr \) is of type \( τ \), then it is of type \( τ \) in any type environment:
  
  \[
  \frac{}{A ⊢ expr : τ} \quad \text{(EnvIntro)}
  \]

- Care for logical variables in **sub-expressions** of operator application:
  
  \[
  \frac{A ⊢ expr_1 : τ_1 \ldots A ⊢ expr_n : τ_n}{A ⊢ ω(expr_1, \ldots, expr_n) : τ}, \quad ω : τ_1 \times \cdots \times τ_n \rightarrow τ, \quad n ≥ 1, \quad ω \notin atr(ℰ)
  \]

- If \( expr \) is a logical variable such that \( w : τ \) occurs in \( A \), then we say \( w \) is of type \( τ \),
  
  \[
  \frac{}{A ⊢ w : τ} \quad \text{(Var)}
  \]
Type Environment Example

Example:
• \( w + 3, A = w : \text{Int} \)

All Instances and Attributes in Type Environment

• If \( \text{expr} \) refers to all instances of class \( C \), then it is of type \( \text{Set}(\tau_C) \),

\[
(\text{AllInst}) \quad \vdash \text{allInstances}_C : \text{Set}(\tau_C)
\]

• If \( \text{expr} \) is an attribute access of an attribute of type \( \tau \) for an object of \( C \) as denoted by \( \text{expr}_1 \), then the premise is that \( \text{expr}_1 \) is of type \( \tau_C \):

\[
(\text{Attr}_0) \quad \vdash \text{expr}_1 : \tau_C \quad \vdash \omega(\text{expr}_1) : \tau, \quad \omega \in \text{atr}(C), \tau \in \mathcal{F}
\]

\[
(\text{Attr}^{0.1}_0) \quad \vdash \text{expr}_1 : \tau_C \quad \vdash r_1(\text{expr}_1) : \tau_D, \quad r_1 : D^{0.1} \in \text{atr}(C)
\]

\[
(\text{Attr}^*_0) \quad \vdash \text{expr}_1 : \tau_C \quad \vdash r_2(\text{expr}_1) : \text{Set}(\tau_D), \quad r_2 : D^* \in \text{atr}(C)
\]
Attributes in Type Environment Example

(\text{Attr}_{0}) \quad \Gamma \vdash \text{expr}_1 : \tau_C \\
\quad \Gamma \vdash v(\text{expr}_1) : \tau \\
\quad v : \tau \in \text{atr}(C), \tau \in \mathcal{T}

(\text{Attr}_{0,1}) \quad \Gamma \vdash \text{expr}_1 : \tau_C \\
\quad \Gamma \vdash r_1(\text{expr}_2) : \tau_D \\
\quad r_1 : D_{0,1} \in \text{atr}(C)

(\text{Attr}_{0}) \quad \Gamma \vdash \text{expr}_1 : \tau_D \\
\quad \Gamma \vdash r_2(\text{expr}_1) : \text{Set}(\tau_D) \\
\quad r_2 : D \in \text{atr}(C)

- \text{self} : \tau_C \vdash \text{self}.y : \text{Int}
- \text{self} : \tau_C \vdash \text{self}.x : \text{Int} \quad \text{well-typed by (}\text{Attr}_{0}, (\text{Attr}_{0,0})\text{)}
- \text{self} : \tau_C \vdash \text{self}.r : \tau_D \quad \text{well-typed by (}\text{Attr}_{0,0}^\ast\text{), (}\text{Attr}_{0,0}\text{), (}\text{Attr}_{0}\text{)}
- \text{self} : \tau_C \vdash \text{self}.r.x : \text{Int} \quad \text{well-typed by (}\text{Attr}_{0,0}^\ast\text{), (}\text{Attr}_{0,0}\text{), (}\text{Attr}_{0}\text{)}

\text{Iterate}

- If \text{expr} is an iterate expression, then
  - the iterator variable has to be type consistent with the base set, and
  - initial and update expressions have to be consistent with the result variable:

(\text{Iter}) \quad A \vdash \text{expr}_1 : \text{Set}(\tau_1) \\
\quad A \vdash \text{expr}_2 : \tau_2 \\
\quad A \vdash \text{expr}_3 : \tau_3 \\
\quad A \vdash \text{expr}_1 \rightarrow \text{iterate}(w_1 : \tau_1 ; w_2 : \tau_2 = \text{expr}_2 | \text{expr}_3) : \tau_2

where \( A' = A \oplus (w_1 : \tau_1) \oplus (w_2 : \tau_2). \)
**Iterate Example**

\[
\begin{align*}
\text{(AllInst)} & \quad A \vdash \text{allInstances}_{C} : \text{Set}(\tau_{C}) \\
\text{(Attr)} & \quad A \vdash \text{expr}_{1} : \tau_{C} \\
\text{(Iter)} & \quad A \vdash \text{expr}_{1} : \tau_{1} \\
& \quad A \vdash \text{expr}_{2} : \tau_{2} \\
& \quad A' \vdash \text{expr}_{3} : \tau_{2}
\end{align*}
\]

where \( A' = A \oplus (w_{1} : \tau_{1}) \oplus (w_{2} : \tau_{2}) \).

**Example:** \( \forall x \in \{\text{Int}\}, \forall C, \{x : \text{Int}\}, \{C \mapsto \{x\}\} \)

---

**First Recapitulation**

- *I only* defined for well-typed expressions.
- *What can hinder* something, which looks like a well-typed OCL expression, from being a well-typed OCL expression...

\( \forall x \in \{\text{Int}\}, \forall C, \{x : \text{Int}\}, \{C \mapsto \{n\}, \{D \mapsto \{x\}\} \)

- Plain syntax error:
  
  context \( C : \text{false} \)

- Subtle syntax error (depends on signature) \( \text{not in } \forall \)
  
  context \( C \text{ inv} : y = 0 \)

- Type error:
  
  context \( \text{self} : C \text{ inv} : \text{self} \cdot n = \text{self} \cdot n \cdot x \)
One Possible Extension: Implicit Casts

- We may wish to have
  \[ \vdash 1 \text{ and } \text{false} : \text{Bool} \]  
  \[(*)\]

  **In other words:** We may wish that the type system allows to use 0, 1 : Int instead of true and false without breaking well-typedness.

- Then just have a rule:
  \[
  \text{(Cast)} \quad \frac{A \vdash \text{expr} : \text{Int}}{A \vdash \text{expr} : \text{Bool}}
  \]

- With (Cast) (and (Int), and (Bool), and (Fun₀)), we can derive the sentence \((*)\), thus conclude well-typedness.

- **But:** that’s only half of the story — the definition of the interpretation function \(I\) that we have is not prepared, it doesn’t tell us what \((*)\) means...
### Implicit Casts Cont’d

So, why isn’t there an interpretation for (1 and false)?

- First of all, we have (syntax)

\[
\text{expr}_1 \text{ and } \text{expr}_2 : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}
\]

- Thus,

\[
I(\text{and}) : I(\text{Bool}) \times I(\text{Bool}) \rightarrow I(\text{Bool})
\]

where \(I(\text{Bool}) = \{\text{true}, \text{false}\} \cup \{\perp_{\text{Bool}}\}\).

- By definition,

\[
I[1 \text{ and false}]_{(\sigma, \beta)}(\quad I[1](\sigma, \beta), \quad I[\text{false}]_{(\sigma, \beta)}(\quad ),
\]

and there we’re stuck.

---

### Implicit Casts: Quickfix

- Explicitly define

\[
I[\text{and}(\text{expr}_1, \text{expr}_2)](\sigma, \beta) := \begin{cases} 
  b_1 \land b_2 & \text{if } b_1 \neq \perp_{\text{Bool}} \neq b_2 \\
  \perp_{\text{Bool}} & \text{otherwise}
\end{cases}
\]

where

- \(b_1 := \text{toBool}(I[\text{expr}_1](\sigma, \beta))\),
- \(b_2 := \text{toBool}(I[\text{expr}_2](\sigma, \beta))\),

and where

\[
\text{toBool} : I(\text{Int}) \cup I(\text{Bool}) \rightarrow I(\text{Bool})
\]

\[
\begin{align*}
  x & \mapsto \begin{cases} 
    \text{true} & \text{if } x \in \{\text{true}\} \cup I(\text{Int}) \setminus \{0, \perp_{\text{Int}}\} \\
    \text{false} & \text{if } x \in \{\text{false}, 0\} \\
    \perp_{\text{Bool}} & \text{otherwise}
  \end{cases}
\end{align*}
\]
Bottomline

- There are wishes for the type-system which require changes in both, the definition of \( I \) and the type system. In most cases not difficult, but tedious.

- Note: the extension is still a basic type system.

- Note: OCL has a far more elaborate type system which in particular addresses the relation between \( \text{Bool} \) and \( \text{Int} \) (cf. [OMG, 2006]).
Visibility — The Intuition

Let’s study an Example:

\[ \mathcal{S} = \{\text{Int}, \{C, D\}, \{n : D_{0,1}, m : D_{0,1}, (x : \text{Int}, \xi, \text{expr}_0, 0)\}, 
\{C \mapsto \{n\}, D \mapsto \{x, m\}\} \]

and

\[ C \xrightarrow{n} D \xrightarrow{\xi x : \text{Int} = \text{expr}_0} m \]

Assume \(w_1 : \tau_C\) and \(w_2 : \tau_D\) are logical variables. Which of the following syntactically correct (?) OCL expressions shall we consider to be well-typed?

<table>
<thead>
<tr>
<th>(\xi) of (x):</th>
<th>public</th>
<th>private</th>
<th>protected</th>
<th>package</th>
</tr>
</thead>
<tbody>
<tr>
<td>(w_1 \cdot n \cdot x = 0)</td>
<td>✔</td>
<td>✔</td>
<td>later</td>
<td>not</td>
</tr>
<tr>
<td>(?)</td>
<td>✘</td>
<td>later</td>
<td>not</td>
<td></td>
</tr>
<tr>
<td>(w_2 \cdot m \cdot x = 0)</td>
<td>✔</td>
<td>✔</td>
<td>later</td>
<td>not</td>
</tr>
<tr>
<td>(x (w_1 (w_2)) = 0)</td>
<td>✔</td>
<td>✘</td>
<td>later</td>
<td>not</td>
</tr>
</tbody>
</table>

Context

• Example: A problem?

\[ Y = (\{\text{self}, C, D\}, \{\text{expr}, \text{self}, \text{r}, v\} : \text{expr}_0, \text{left, right, D} : (\text{r}, \text{v}, \text{s})) \]

\[ C \xrightarrow{r} D \xrightarrow{r} \]

\(\text{self} : \tau_D \vdash \text{self} \cdot r \cdot v > 0\) ✔

\(\text{self} : \tau_C \vdash \text{self} \cdot r \cdot v > 0\) ✘

• That is, whether an expression involving attributes with visibility is well-typed depends on the class of objects for which it is evaluated.

• Therefore: well-typedness in type environment \(A\) and context \(B \in \mathcal{C}\):

\[ A, B \vdash \text{expr} : \tau \]

• In particular: prepare to treat “protected” later (when doing inheritance).
**Attribute Access in Context**

- If `expr` is of type `τ` in a type environment, then it is in any context:
  \[
  \frac{A \vdash \text{expr} : \tau}{A, B \vdash \text{expr} : \tau}
  \]

- **Accessing attribute** `v` of a `C`-object via logical variable `w` is well-typed if
  - `w` is of type `τ_B`
  \[
  \frac{A \vdash w : \tau_B}{A, B \vdash v(w) : \tau, \langle v : \tau, \xi, \text{expr}_0, P_\phi \rangle \in \text{atr}(B)}
  \]

- **Accessing attribute** `v` of a `C`-object of via expression `expr_1` is well-typed in context `B` if
  - `v` is public, or `expr_1` denotes an object of class `B`:
  \[
  \frac{A, B \vdash \text{expr}_1 : \tau_C}{A, B \vdash v(\text{expr}_1) : \tau, \langle v : \tau, \xi, \text{expr}_0, P_\phi \rangle \in \text{atr}(C), \xi = +, \text{or } C = B}
  \]

- Accessing `C_{0,1}`- or `C_\ast`-typed attributes: similar.

**Context in Operator Application**

- **Operator Application**:
  \[
  \frac{A, B \vdash \text{expr}_1 : \tau_1 \ldots A, B \vdash \text{expr}_n : \tau_n}{A, B \vdash \omega(\text{expr}_1, \ldots, \text{expr}_n) : \tau, \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \ n \geq 1, \omega \notin \text{atr}(\phi)}
  \]

- **Iterate**:
  \[
  \frac{A, B \vdash \text{expr}_1 : \text{Set}(\tau_1) \quad A', B \vdash \text{expr}_2 : \tau_2 \quad A', B \vdash \text{expr}_3 : \tau_2}{A, B \vdash \text{expr}_1 \rightarrow \text{iterate}(\text{w}_1 : \tau_1 ; \text{w}_2 : \tau_2 = \text{expr}_2 | \text{expr}_3) : \tau_2}
  \]

  where `A' = A \oplus (w_1 : \tau_1) \oplus (w_2 : \tau_2)`.
Attribute Access in Context Example

\[
\begin{align*}
A, B \vdash \text{expr} : \tau & \quad (\text{Context Intro}) \\
A \vdash \text{expr} : \tau & \quad (\text{Exp}) \\
A, B \vdash \text{expr} : \tau & \quad \varepsilon_l, \varepsilon_r, P \in \text{str}(C), \\
A, B \vdash v(\text{expr}_1) : \tau & \quad \xi = +, \text{ or } \xi = - \text{ and } C = B
\end{align*}
\]

Example:

\[
\text{self} : \tau_C \quad \vdash \text{self}.r.v > 0
\]

The Semantics of Visibility

- **Observation:**
  - Whether an expression does or does not respect visibility is a matter of well-typedness only.
  - We only evaluate (= apply \( I \)) well-typed expressions.
  
  \( \rightarrow \) We need not adjust the interpretation function \( I \) to support visibility.
What is Visibility Good For?

- Visibility is a property of attributes — is it useful to consider it in OCL?
- In other words: given the picture above, is it useful to state the following invariant (even though \( x \) is private in \( D \))

\[
\text{context } C \text{ inv } n \cdot x > 0
\]

- It depends. (cf. [OMG, 2006], Sect. 12 and 9.2.2)
  
  - Constraints and pre/post conditions:
    - Visibility is sometimes not taken into account. To state “global” requirements, it may be adequate to have a “global view”, be able to look into all objects.
    - But: visibility supports “narrow interfaces”, “information hiding”, and similar good design practices. To be more robust against changes, try to state requirements only in the terms which are visible to a class.
    
    Rule-of-thumb: if attributes are important to state requirements on design models, leave them public or provide get-methods (later).
  
  - Guards and operation bodies:
    If in doubt, yes (= do take visibility into account).
    Any so-called action language typically takes visibility into account.

Recapitulation
Recapitulation

Class Diagrams $\mathcal{C} \mathcal{D}$

\[
\begin{align*}
\text{induces} & \quad \text{extended (!) signature } \mathcal{P}(\mathcal{C} \mathcal{D}) \\
\text{gives rise to} & \quad \text{Basic Type System}
\end{align*}
\]

- We extended the type system for
  - casts (requires change of $I$) and see slides
  - visibility (no change of $I$).
- Later: navigability of associations.

**Good:** well-typedness is decidable for these type-systems. That is, we can have automatic tools that check, whether OCL expressions in a model are well-typed.

References
References

