Software Design, Modelling and Analysis in UML

Lecture 06: Type Systems and Visibility

2012-11-13

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 Lectures 07 and 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:
  - Class diagram semantics.
  - Stereotypes — for documentation.
  - Recall: type theory/static type systems.
  - Well-typedness for OCL expression.
  - Visibility as a matter of well-typedness.
Recall: From Class Boxes to Extended Signatures

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 $\mathcal{C}$.

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

Convention:

- We write $(C, S_C, a, t) \in \mathcal{C}$

  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 \( \text{expr}_0 \) 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 \( P_E \) analogously to stereotypes.

Convention:
- We write \( \langle v : \tau, \xi, \text{expr}_0, P_v \rangle \in V \) 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:

\[
\begin{align*}
V(n) := & \{ \langle v_1 : \tau_1, \xi_1, v_{0,1}, \{ P_{1,1}, \ldots, P_{1,m_1} \} \rangle, \ldots, \langle v_\ell : \tau_\ell, \xi_\ell, v_{0,\ell}, \{ P_{\ell,1}, \ldots, P_{\ell,m_\ell} \} \rangle \} \\
\text{atr}(n) := & \{ C \mapsto \{ v_1, \ldots, v_\ell \} \}
\end{align*}
\]

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

- The semantics of a set of class diagrams $\mathcal{D}$ first of all is the induced (extended) signature $\mathcal{I}(\mathcal{C}D)$.
- The signature gives rise to a set of system states given a structure $D$.
- Do we need to redefine/extend $D$? **No.**
  (Would be different if we considered the definition of enumeration types in class diagrams. Then the domain of an enumeration type $\tau$, i.e. the set $D(\tau)$, would be determined by the class diagram, and not free for choice.)

![Class Diagram](image-url)
Semantics

- The semantics of a set of class diagrams \( \mathcal{C} \mathcal{D} \) first of all is the induced (extended) signature \( \mathcal{I}(\mathcal{C} \mathcal{D}) \).
- The signature gives rise to a set of system states given a structure \( \mathcal{D} \).
- Do we need to redefine/extend \( \mathcal{D} \)? No.
  (Would be different if we considered the definition of enumeration types in class diagrams. Then the domain of an enumeration type \( \tau \), i.e. the set \( \mathcal{P}(\tau) \), would be determined by the class diagram, and not free for choice.)
- What is the effect on \( \Sigma_\mathcal{D} \)? Little.
  For now, we only remove abstract class instances, i.e.
  \[
  \sigma : \mathcal{P}(\mathcal{C}) \rightarrow (V \rightarrow (\mathcal{P}(\mathcal{D}) \cup \mathcal{P}(\mathcal{C}_a)))
  \]
  is now only called system state if and only if, for all \((C, S_C, 1, t) \in \mathcal{C}\),
  \[
  \text{dom}(\sigma) \cap \mathcal{P}(C) = \emptyset.
  \]
  With \( a = 0 \) as default “abstractness”, the earlier definitions apply directly.
  We’ll revisit this when discussing inheritance.

What About The Rest?

- Classes:
  - **Active**: not represented in \( \sigma \).
    Later: relevant for behaviour, i.e., how system states evolve over time.
  - **Stereotypes**: in a minute.

- Attributes:
  - **Initial value**: not represented in \( \sigma \).
    Later: provides an initial value as effect of “creation action”.
  - **Visibility**: not represented in \( \sigma \).
    Later: viewed as additional **typing information** for well-formedness of system transformers; and with inheritance.
  - **Properties**: such as readOnly, ordered, composite (Deprecated in the standard.)
    - readOnly — later treated similar to visibility.
    - ordered — too fine for our representation.
    - composite — cf. lecture on associations.
Stereotypes

Stereotypes as Labels or Tags

• So, a class is

\langle C, S_C; a, t \rangle

with \( a \) the abstractness flag, \( t \) activeness flag, and \( S_C \) a set of stereotypes.

• What are Stereotypes?
  • Not represented in system states.
  • Not contributing to typing rules.
    (cf. type theory for UML later)
  
• [Oestereich, 2006]:
  View stereotypes as (additional) “labelling” (“tags”) or as “grouping”.
  Useful for documentation and MDA.
  • Documentation: e.g. layers of an architecture.
    Sometimes, packages (cf. the standard) are already sufficient and “right”.
  • Model Driven Architecture (MDA): later.
Example: Stereotypes for Documentation

- Example: Timing Diagram Viewer [Schumann et al., 2008]
- Architecture of four layers:
  - core, data layer
  - abstract view layer
  - toolkit-specific view layer/widget
  - application using widget
- Stereotype “==” layer “==” colour

Stereotypes as Inheritance

- Another view (due to whom?): distinguish
  - **Technical Inheritance**
    If the target platform, such as the programming language for the implementation of the blueprint, is object-oriented, assume a 1-on-1 relation between inheritance in the model and on the target platform.
  - **Conceptual Inheritance**
    Only meaningful with a common idea of what stereotypes stand for. For instance, one could label each class with the team that is responsible for realising it. Or with licensing information (e.g., LGPL and proprietary).
    Or one could have labels understood by code generators (cf. lecture on MDSE).
- **Confusing**:
  - Inheritance is often referred to as the “is a”-relation.
    Sharing a stereotype also expresses “being something”.
  - We can always (ab-)use UML-inheritance for the conceptual case, e.g.
Excursus: Type Theory (cf. Thiemann, 2008)

Type Theory

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

\[
\begin{align*}
expr & ::= w : \tau & \text{...logical variable } w \\
& | \text{true} | \text{false} & : \text{Bool} & \text{...constants} \\
& | 0 | -1 | 1 | \ldots & : \text{Int} & \text{...constants} \\
& | expr_1 + expr_2 & : \text{Int} \times \text{Int} \to \text{Int} & \text{...operation} \\
& | \text{size}(expr_1) & : \text{Set}(\tau) \to \text{Int} \\
\end{align*}
\]

Wanted: A procedure to tell well-typed, such as \((w : \text{Bool})\)

not \(w\)

from not well-typed, such as,

\(\text{size}(w)\).

Approach: Derivation System, that is, a finite set of derivation rules.
We then say \(expr\) is well-typed if and only if we can derive

\[A, C \vdash expr : \tau\]

(read: "expression \(expr\) has type \(\tau\")

for some OCL type \(\tau\), i.e. \(\tau \in T_B \cup T_{\mathcal{C}} \cup \{\text{Set}(\tau_0) \mid \tau_0 \in T_B \cup T_{\mathcal{C}}\}, C \in \mathcal{C}\).
A Type System for OCL

We will give a finite set of type rules (a type system) of the form

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

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

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

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

- If `expr` is the application of **operation** \(\omega : \tau_1 \times \cdots \times \tau_n \to \tau\) to expressions `expr_1, \ldots, expr_n` which are of type \(\tau_1, \ldots, \tau_n\), then `expr` is of type \(\tau\):
  \[(\text{Fun}_0) \quad \vdash expr_1 : \tau_1 \quad \vdash expr_n : \tau_n \quad \vdash \omega(expr_1, \ldots, expr_n) : \tau, \quad \omega : \tau_1 \times \cdots \times \tau_n \to \tau, \quad n \geq 1, \quad \omega \not\in \text{atr}(\mathcal{E})\]

(Note: this rule also covers ‘\(=\)’, ‘isEmpty’, and ‘size’.)

---

### Constants and Operations Example

\[
\begin{align*}
\text{(BOOL)} & \quad \vdash B : \text{Bool}, \quad B \in \{\text{true}, \text{false}\} \\
\text{(INT)} & \quad \vdash N : \text{Int}, \quad N \in \{0,1,-1,\ldots\} \\
\text{(Fun}_0) & \quad \vdash expr_1 : \tau_1 \quad \vdash expr_n : \tau_n \quad \vdash \omega(expr_1, \ldots, expr_n) : \tau, \quad \omega : \tau_1 \times \cdots \times \tau_n \to \tau, \quad n \geq 1, \quad \omega \not\in \text{atr}(\mathcal{E})
\end{align*}
\]

Example:
- **not true**
  \[
  \begin{align*}
  \vdash \text{true} : \text{Bool} \\
  \vdash \text{not not true : Bool}
  \end{align*}
  \[
  \text{Not constant for the rules}
  \]
- **true + 3**
  \[
  \begin{align*}
  \vdash \text{true} : \text{Nat} \\
  \vdash 3 : \text{Nat} \\
  \vdash \text{true + 3 : Nat}
  \end{align*}
  \]
- **isExpr(\{7,2\})**
  \[
  \begin{align*}
  \vdash \text{Expr} \quad \vdash \text{Expr} \quad \vdash \text{Expr}
  \end{align*}
  \]
- **isExpr(\{1\})**
  \[
  \begin{align*}
  \vdash \text{Expr} \quad \vdash \text{Expr} \quad \vdash \text{Expr}
  \end{align*}
  \]
- **isExpr(\text{null})**
  \[
  \begin{align*}
  \vdash \text{Expr} \quad \vdash \text{Expr} \quad \vdash \text{Expr}
  \end{align*}
  \]
- **isExpr(\text{null})**
  \[
  \begin{align*}
  \vdash \text{Expr} \quad \vdash \text{Expr} \quad \vdash \text{Expr}
  \end{align*}
  \]
**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

  \[
  A ::= \emptyset \mid A, w : \tau
  \]

  where \( w \in W \), \( \tau \in T \).

  **Clear:** We use this definition for the set of OCL logical variables \( W \) and the types \( T = T_B \cup T_E \cup \{ \text{Set}(\tau_0) \mid \tau_0 \in T_B \cup T_E \} \).

**Environment Introduction and Logical Variables**

- If \( expr \) is of type \( \tau \), then it is of type \( \tau \) in any type environment:

  \[
  \frac{\vdash expr : \tau}{A \vdash expr : \tau}
  \]

  \( (\text{EnvIntro}) \)

- Care for logical variables in **sub-expressions** of operator application:

  \[
  \frac{A \vdash expr_1 : \tau_1 \ldots A \vdash expr_n : \tau_n, \quad \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin \text{atr}(E) }{A \vdash \omega(expr_1, \ldots, expr_n) : \tau}
  \]

  \( (\text{Fun}) \)

- If \( expr \) is a logical variable such that \( w : \tau \) occurs in \( A \), then we say \( w \) is of type \( \tau \),

  \[
  \frac{w : \tau \in A}{A \vdash w : \tau}
  \]

  \( (\text{Var}) \)
### Type Environment Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>(EnvIntro)</td>
<td>$\vdash expr : \tau$</td>
<td>$A \vdash expr : \tau$</td>
</tr>
<tr>
<td>(Fun)</td>
<td>$A \vdash expr_1 : \tau_1 \ldots A \vdash expr_n : \tau_n$, $\omega : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$, $n \geq 1$, $\omega \notin atr(\mathcal{E})$</td>
<td>$A \vdash \omega(expr_1, \ldots, expr_n) : \tau$</td>
</tr>
<tr>
<td>(Var)</td>
<td>$w : \tau \in A$</td>
<td>$A \vdash w : \tau$</td>
</tr>
</tbody>
</table>

**Example:**  
- $w + 3$, $A = w : \text{Int}$

\[
\begin{array}{c}
\frac{w : \text{Int} \in \nu \cdot \nu \cdot \nu}{\nu : \text{Int}} \quad \frac{\text{Int} \vdash 3 : \text{Int}}{w + 3 : \text{Int}} \quad A \\
\frac{w \cdot \text{Int} \vdash w : \text{Int}}{\nu \cdot \text{Int} \vdash \nu : \text{Int}} \\
\frac{\nu \vdash \text{Int}}{\nu \cdot \text{Int} \vdash w \cdot \text{Int}} \\
\frac{\nu \cdot \text{Int} \vdash \nu \cdot \text{Int}}{A} \\
\end{array}
\]

\[
A \vdash w + 3 : \text{Int}
\]

\[
\vdash \omega + 3 \text{ well-typed in type environment } A
\]

---

### All Instances and Attributes in Type Environment

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

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

- If $expr$ is an attribute access of an attribute of type $\tau$ for an object of $C$ as denoted by $expr_1$, then the premise is that $expr_1$ is of type $\tau_C$:

\[
\begin{align*}
(\text{Attr}) & \quad \frac{\vdash expr_1 : \tau_C} {A \vdash v(expr_1) : \tau}, \quad v : \mathcal{F} \in \text{atr}(C), \tau \in \mathcal{F} \\
(\text{Attr}_0) & \quad \frac{\vdash expr_1 : \tau_C} {A \vdash r_1(expr_1) : \tau_D}, \quad r_1 : D_{0,1} \in \text{atr}(C) \\
\end{align*}
\]

\[
\begin{align*}
\text{Attr}_0 \quad \frac{\vdash expr_1 : \tau_C} {A \vdash r_2(expr_1) : \text{Set}(\tau_D)}, \quad r_2 : D \in \text{atr}(C)
\end{align*}
\]
### Attributes in Type Environment Example

\[
\begin{align*}
(Attr_0) & \quad A \vdash \text{expr}_1 : \tau \\
& \quad A \vdash v(\text{expr}_1) : \tau, \\
& \quad v : \tau \in \text{atr}(C), \tau \in \mathcal{T}
\end{align*}
\]

\[
(Attr_{0,1}) \quad A \vdash \text{expr}_1 : \tau \\
A \vdash r_1(\text{expr}_1) : \tau_D, \\
& \quad r_1 : D_{0,1} \in \text{atr}(C)
\]

\[
(Attr_0) \quad A \vdash \text{expr}_1 : \tau \\
A \vdash r_2(\text{expr}_1) : \text{Set}(\tau_D), \\
& \quad r_2 : D_0 \in \text{atr}(C)
\]

- \( self : \tau_C \vdash self. y : \text{Int} \)
- \( self : \tau_C \vdash self. x : \text{Int} \)
- \( self : \tau_C \vdash self. r : \tau_D \)
- \( self : \tau_C \vdash self. r. x : \text{Int} \)

\[
\begin{array}{c|c}
C & D \\
\hline
x : \text{Int} & 0 \cdot 2 \\
\end{array}
\]

\[
\begin{array}{c|c}
A' & \text{Type Environment} \\
\hline
\end{array}
\]

### 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:

\[
\begin{align*}
(Iter) & \quad A \vdash \text{expr}_1 : \text{Set}(\tau_1) \\
& \quad A \vdash \text{expr}_2 : \tau_2, \\
& \quad A' \vdash \text{expr}_3 : \tau, \\
& \quad \text{iter}(w_1 : \tau_1 ; w_2 : \tau_2 = \text{expr}_2 | \text{expr}_3) : \tau_2
\end{align*}
\]

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

\[
\vdash \text{allInstances}_C : \text{Set}(\tau_C) \quad (\text{Attr}) \quad A \vdash \text{expr}_1 : \tau_C \quad A \vdash v_1(\text{expr}_1) : \tau
\]

\[
\vdash \text{expr}_1 : \text{Set}(\tau_1) \quad A' \vdash \text{expr}_2 : \tau_2 \quad A' \vdash \text{expr}_3 : \tau_2
\]

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

Example: \( \mathcal{S} = (\{\text{Int}\}, \{C\}, \{x : \text{Int}\}, \{C \mapsto \{x\}\}) \)

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...

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

- *Plain syntax error:*
  
  \[ \vdash \text{context } C : \text{false} \]

- *Type error: Max error:
  
  \[ \vdash \text{context } C \text{ inv }: y = 0 \]

- *Type error:*
  
  \[ \vdash \text{context } self : C \text{ inv } : self . n = self . n . x \]
References


