

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-spreadness 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-spreadness.

2/44

Recall: From Class Boxes to Extended Signatures

3/44

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 $a \in \{0, 1\}$ indicating whether C is **abstract**,
 - a boolean flag $t \in \{0, 1\}$ indicating whether C is **active**.
- We use S_C to denote the set $\bigcup_{C \in \mathcal{C}} S_C$ of stereotypes in \mathcal{C} . (Alternatively, we could add a set S as 5-th component to \mathcal{C} to provide 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.

25/44

Extended Attributes

- From now on, we assume that each attribute $v \in V$ has (in addition to the type):
- a **visibility** $\epsilon \in \{\text{public, private, protected, package}\}$
 - an **initial value expr**, 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 $(v : \tau, \epsilon, \text{expr}, P_v) \in V$ when we want to refer to all aspects of v .
 - Write only $v : \tau$ or v if details are irrelevant.

26/44

From Class Boxes to Extended Signatures

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

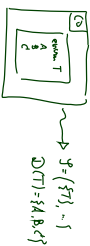
where

- “shorter” is determined by the font: \boxed{C} or $\boxed{C(a)}$ if $a = \text{true}$, \boxed{C} otherwise
- “active” is determined by the frame: \boxed{C} or \boxed{C} if $t = \text{true}$, \boxed{C} otherwise

29/44

Class Diagram Semantics

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



7/14

Semantics

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

8/14

Semantics

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

8/14

What About The Rest?

- **Classes:**
- **Active:** not represented in σ .
- **Later:** relevant for behaviour, i.e., how system states evolve over time
- **Stereotypes:** in a minute
- **Attributes:**
- **Initial value:** not represented in σ .
- **Later:** provides an initial value as effect of "creation action".
- **Visibility:** not represented in σ .
- **Later:** viewed as additional typing information for well-formedness of system transformers; and with inheritance.
- **Properties:** such as readability, ordered, composite
- **(Deprecated)** in the standard
- **readably** — later treated similar to visibility
- **ordered** — too fine for our representation
- **composite** — cf. lecture on associations.

9/14

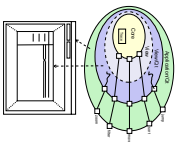
Stereotypes

10/14

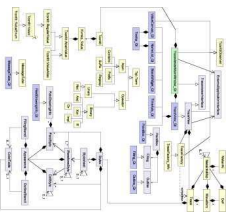
Stereotypes as Labels or Tags

- 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)
- [Osterweich, 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.

11/14

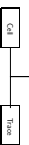


- Example: Timing Diagram Viewer [Schürr et al., 2009]
- Architecture of four layers:
 - core, data layer
 - abstract view layer
 - toolbar-specific view layer/widget
 - application using widget
- Stereotype "view" layer "colour"



12.44

- 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.



13.44

Type Theory

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

```

expr ::= w
      | True | False          : Bool
      | 0 | - | 1 | ...       : Int
      | expr1 + expr2        : Int × Int → Int ... operation
      | set(expr1)           : Set(T) → Int
  
```

Wanted: A procedure to tell **well-typed**, such as (w : *Bool*)
 from **not well-typed**, such as,
 $size(w)$.

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 τ ")

for some OCL type τ , i.e. $\tau \in T_D \cup T_C \cup \{Set(\tau_0) \mid \tau_0 \in T_D \cup T_C\}$, $C \in \mathcal{C}$. 15.44

A Type System for OCL

A Type System for OCL

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

(“name”) $\frac{\text{“premise”}}{\text{“conclusion”}}$ “side condition”

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

- (i) Universal well-typedness:

$$\vdash expr : \tau$$

$$\vdash 1 + 2 : Int$$
- (ii) Well-typedness in a **type environment** A : (for logical variables)

$$A \vdash expr : \tau$$

$$self : TC \vdash self.v : Int$$
- (iii) Well-typedness in type environment A and **context** B : (for visibility)

$$A, B \vdash expr : \tau$$

$$self : TC, C \vdash self.v : Int$$

17.44

Constants and Operations

- If $expr$ is a **boolean constant**, then $expr$ is of type $Bool$:

$$(BOOL) \quad \frac{}{A \vdash Bool} \quad B \in \{true, false\}$$
- If $expr$ is an **integer constant**, then $expr$ is of type Int :

$$(INT) \quad \frac{}{A \vdash Int} \quad N \in \{0, 1, -1, \dots\}$$

If $expr$ is the application of operation $w : \tau_1 \times \dots \times \tau_n \rightarrow \tau$ to expressions $expr_1, \dots, expr_n$ which are of type τ_1, \dots, τ_n , then $expr$ is of type τ :

$$(Funs) \quad \frac{A \vdash expr_1 : \tau_1 \dots A \vdash expr_n : \tau_n \quad w : \tau_1 \times \dots \times \tau_n \rightarrow \tau}{A \vdash w(expr_1, \dots, expr_n) : \tau} \quad n \geq 1, w \notin \text{attr}(C)$$

(Note: this rule also covers '=', 'isEmpty', and 'and')

Constants and Operations Example

(BOOL)	$A \vdash Bool$	$B \in \{true, false\}$
(INT)	$A \vdash Int$	$N \in \{0, 1, -1, \dots\}$
(Funs)	$A \vdash w(expr_1, \dots, expr_n) : \tau$	$w : \tau_1 \times \dots \times \tau_n \rightarrow \tau$ $n \geq 1, w \notin \text{attr}(C)$

Example:

- not true
- $w = +3$
- $w = \text{isEmpty}$
- $w = \text{and}$

Handwritten notes:

- for $w = +3$: $A \vdash +3 : Int$ (true), $A \vdash +3 : Int$ (false)
- for $w = \text{isEmpty}$: $A \vdash \text{isEmpty} : Bool$ (true), $A \vdash \text{isEmpty} : Bool$ (false)
- for $w = \text{and}$: $A \vdash \text{and} : Bool$ (true), $A \vdash \text{and} : Bool$ (false)

Environment Introduction and Logical Variables

- If $expr$ is of type τ , then it is of type τ in any type environment:

$$(Environ) \quad \frac{}{A \vdash expr : \tau}$$
- Care for logical variables in sub-expressions of operator application:

$$(Funs) \quad \frac{A \vdash expr_1 : \tau_1 \dots A \vdash expr_n : \tau_n \quad w : \tau_1 \times \dots \times \tau_n \rightarrow \tau}{A \vdash w(expr_1, \dots, expr_n) : \tau} \quad n \geq 1, w \notin \text{attr}(C)$$

If $expr$ is a **logical variable** such that $w : \tau$ occurs in A , then we say w is of type τ .

$$(Var) \quad \frac{w : \tau \in A}{A \vdash w : \tau}$$

Type Environment Example

(Environ)	$A \vdash expr : \tau$
(Funs)	$A \vdash w(expr_1, \dots, expr_n) : \tau$
(Var)	$w : \tau \in A$

Example:

- $w = +3, A = w : Int$
- (a) $A \vdash +3 : Int$
- (b) $A \vdash +3 : Int$
- (c) $A \vdash +3 : Int$

Handwritten notes:

- for (a): $A \vdash +3 : Int$ (true), $A \vdash +3 : Int$ (false)
- for (b): $A \vdash +3 : Int$ (true), $A \vdash +3 : Int$ (false)
- for (c): $A \vdash +3 : Int$ (true), $A \vdash +3 : Int$ (false)

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

$$A ::= () \mid A, w : \tau$$

where $w \in W, \tau \in T$.

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

All Instances and Attributes in Type Environment

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

$$(AllInst) \quad \frac{}{\text{allInstances} : \text{Set}(TC)}$$
- If $expr$ is an **attribute access** of an attribute of type τ for an object of C as denoted by $expr_1$, then the premise is that $expr_1$ is of type TC :

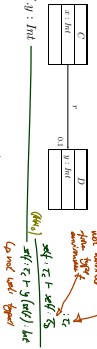
$$(Attr_0) \quad \frac{A \vdash expr_1 : TC \quad v : \theta \in \text{attr}(C), \tau \in \mathcal{S}}{A \vdash v(expr_1) : \tau}$$

Handwritten notes for (Attr₀):

- $A \vdash expr_1 : TC$ (true), $A \vdash expr_1 : TC$ (false)
- $v : \theta \in \text{attr}(C), \tau \in \mathcal{S}$ (true), $v : \theta \in \text{attr}(C), \tau \in \mathcal{S}$ (false)
- $A \vdash v(expr_1) : \tau$ (true), $A \vdash v(expr_1) : \tau$ (false)
- $A \vdash v_2(expr_1) : \text{Set}(TC)$ (true), $A \vdash v_2(expr_1) : \text{Set}(TC)$ (false)

Attributes in Type Environment Example

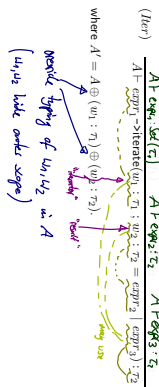
$$\begin{array}{l}
 (Attr_0) \quad \frac{A \vdash \text{expr}_1 : \tau_1}{A \vdash \text{expr}_1 : \tau} \quad v : \tau \in \text{attr}(C), \tau \in \mathcal{T} \\
 (Attr_0^1) \quad \frac{A \vdash \text{expr}_1 : \tau_1 \quad A \vdash \text{expr}_2 : \tau_2 \quad \tau_1, D_{A_1} \in \text{attr}(C)}{A \vdash \text{expr}_1 : \tau_1 \quad A \vdash \text{expr}_2 : \tau_2} \\
 (Attr_1) \quad \frac{A \vdash \text{expr}_1 : \tau_1 \quad A \vdash \text{expr}_2 : \tau_2 \quad \tau_1, D_A \in \text{attr}(C)}{A \vdash \text{expr}_1 : \tau_1 \quad A \vdash \text{expr}_2 : \tau_2}
 \end{array}$$



- $\text{self} : \tau_C \vdash \text{self } y : \text{Int}$ — $\text{self} : \tau_C \vdash \text{self } y : \text{Int}$ (by $(Attr_0)$, $(Attr_1)$)
- $\text{self} : \tau_C \vdash \text{self } x : \tau_D$ — $\text{self} : \tau_C \vdash \text{self } x : \tau_D$ (by $(Attr_1)$, $(Attr_0)$)
- $\text{self} : \tau_C \vdash \text{self } x : \tau_D$ — $\text{self} : \tau_C \vdash \text{self } x : \tau_D$ (by $(Attr_1)$, $(Attr_0)$)
- $\text{self} : \tau_C \vdash \text{self } x : \tau_D$ — $\text{self} : \tau_C \vdash \text{self } x : \tau_D$ (by $(Attr_1)$, $(Attr_0)$)
- $\text{self} : \tau_C \vdash \text{self } x : \tau_D$ — $\text{self} : \tau_C \vdash \text{self } x : \tau_D$ (by $(Attr_1)$, $(Attr_0)$)

Iterate

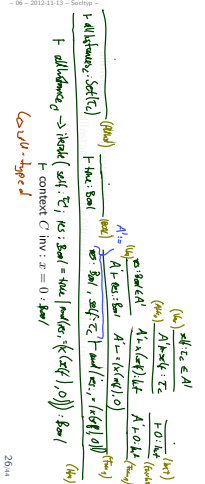
- If expr is an **iterate** expression, then
- the reator variable has to be type consistent with the base set, and
- initial and update expressions have to be consistent with the result variable.



Iterate Example

$$\begin{array}{l}
 (Attr_{Iter}) \quad \frac{A \vdash \text{base} : \tau \quad A \vdash \text{reter} : \tau}{A \vdash \text{iterate}(\text{base}, \text{reter}) : \tau} \\
 (Iter) \quad \frac{A \vdash \text{expr}_1 : \tau_1 \quad A \vdash \text{expr}_2 : \tau_2 \quad \tau_1, D_A \in \text{attr}(C)}{A \vdash \text{iterate}(\text{expr}_1, \text{expr}_2) : \tau_2}
 \end{array}$$

where $A' = A @ (u_1 : \tau_1) @ (u_2 : \tau_2)$



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...?
- $\mathcal{S} = (\{In\}, \{C, D\}, \{\tau : \text{Int}, n : \{a\}, \{D\} \vdash \{a\}\})$
- τ is a **type** key: context $C \vdash \tau : \text{Int}$
- n is a **value** key: context $C \vdash n : \{a\}$
- D is a **domain** key: context $C \vdash D : \{a\}$

References

[Oesterreich, 2006] Oesterreich, B. (2006). *Analyse und Design mit UML 2.1. 8. Auflage*. Oldenbourg, 8. edition.

[OMG, 2006] OMG (2006). *Object Constraint Language, version 2.0*. Technical Report formal/06-05-01.

[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.

[Schumann et al., 2008] Schumann, M., Steinke, J., Deck, A., and Westphal, B. (2008). *Tracereview technical documentation, version 1.0*. Technical report, Carl von Ossietzky Universität Oldenburg und OFHS.