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 late/early binding?
  - What is the subset, what the uplink semantics of inheritance?
  - What’s the effect of inheritance on LSCs, 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 $\mathcal{M}_U$ such that
  - the set of legal instances of $\mathcal{M}_U$
    - 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)?
Classes [OMG, 2007b, 32]

Figure 7.12 - Classes diagram of the Kernel package

Operations [OMG, 2007b, 31]

Figure 7.11 - Operations diagram of the Kernel package
Operations [OMG, 2007b, 30]

Classifiers [OMG, 2007b, 29]
Namespaces [omg, 2007b, 26]

Root Diagram [omg, 2007b, 25]

Figure 7.3 - Root diagram of the Kernel package

Figure 7.4 - Namespaces diagram of the Kernel package
Meta-modelling has already been used for UML 1.x.

For UML 2.0, the request for proposals (RFP) asked for a separation of concerns: **Infrastructure** and **Superstructure**.

One reason: sharing with MOF (see later) and, e.g., CWM.
**UML Superstructure Packages** [OMG, 2007a, 15]

**Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure**

**Meta-Modelling: Principle**
---

**Modelling vs. Meta-Modelling**

<table>
<thead>
<tr>
<th>Meta-Model (M2)</th>
<th>Model (M1)</th>
<th>Instance (M0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class name: Str</td>
<td>Property name: Str</td>
<td>Type name: Str</td>
</tr>
<tr>
<td>( v : \mathbb{Z} )</td>
<td>( \mathcal{C} )</td>
<td>( C )</td>
</tr>
<tr>
<td>name = ( \mathbb{C} )</td>
<td>name = ( v )</td>
<td>name = ( v )</td>
</tr>
</tbody>
</table>

\[
\mathcal{I} = (\{ \mathbb{Z} \}, \{ \mathbb{C} \}, \{ v \}, \{ C \mapsto v \}), \quad \mathcal{D} \rightsquigarrow \Sigma_{\mathcal{I}}
\]

\( C, v = 0 \) instance-of \( \mathcal{I} \in \text{Meta-Model (M2)} \)

---

**Modelling vs. Meta-Modelling**

<table>
<thead>
<tr>
<th>Meta-Model (M2)</th>
<th>Model (M1)</th>
<th>Instance (M0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class name: Str</td>
<td>Property name: Str</td>
<td>Type name: Str</td>
</tr>
<tr>
<td>( v : \mathbb{Z} )</td>
<td>( \mathcal{C} )</td>
<td>( C )</td>
</tr>
<tr>
<td>name = ( \mathbb{C} )</td>
<td>name = ( v )</td>
<td>name = ( v )</td>
</tr>
</tbody>
</table>

\[
\mathcal{I} = (\{ \mathbb{Z} \}, \{ \mathbb{C} \}, \{ v \}, \{ C \mapsto v \}), \quad \mathcal{D} \rightsquigarrow \Sigma_{\mathcal{I}}
\]

- So, if we have a meta model \( \mathcal{M}_U \) of UML, then the set of UML models is the set of instances of \( \mathcal{M}_U \).
- A UML model \( \mathcal{M} \) can be represented as an object diagram (or system state) wrt. the meta-model \( \mathcal{M}_U \).
- **Other view**: An object diagram wrt. meta-model \( \mathcal{M}_U \) can (alternatively) be rendered as the UML model \( \mathcal{M} \).
Well-Formedness as Constraints in the Meta-Model

- The set of well-formed UML models can be defined as the set of object diagrams satisfying all constraints of the meta-model.

For example,

“[2] Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.

not self . allParents() \rightarrow \textsf{includes}(\textsf{self})” [OMG, 2007b, 53]

- The other way round:

Given a UML model $\mathcal{M}$, unfold it into an object diagram $O_1$ wrt. $\mathcal{M}_U$.

If $O_1$ is a valid object diagram of $\mathcal{M}_U$ (i.e. satisfies all invariants from $\text{Inv}(\mathcal{M}_U)$), then $\mathcal{M}$ is a well-formed UML model.

That is, if we have an object diagram validity checker for the meta-modelling language, then we have a well-formedness checker for UML models.
## 7. Classes

### 7.1 Overview

- T.47 Relationship (from Kernel)
- T.44 Net (Kernel)
- T.49 StructuralFeature (from Kernel)
- T.50 Generalization (from Kernel)
- T.51 Type (from Kernel)
- T.52 Specification (from Kernel)
- T.53 Usage (from Dependencies)
- T.54 ValueSpecification (from Kernel)
- T.55 ValueSpecification (from Kernel)

### 7.2 Abstract Syntax

### 7.3 Class Descriptions

#### 7.3.53 Usage (from Dependencies)

### 8. Components

### 8.1 Overview

### 8.2 Abstract Syntax

### 8.3 Class Descriptions

#### 8.3.1 Component (from BasicComponents, PackagingComponents)

#### 8.3.2 Connector (from BasicComponents)

#### 8.3.3 Operation (from Kernel, Interfaces)

#### 8.3.4 ComponentRealization (from BasicComponents)

### 9. Composite Structures

### 9.1 Overview

### 9.2 Abstract Syntax

### 9.3 Class Descriptions

#### 9.3.1 Composite (from BasicComponents, PackagingComponents)

#### 9.3.2 Connector (from BasicComponents)

#### 9.3.3 Operation (from Kernel, Interfaces)

#### 9.3.4 ComponentRealization (from BasicComponents)

### 10. Deployments

### 10.1 Overview

### 10.2 Abstract Syntax

### 10.3 Class Descriptions

#### 10.3.1 Deployment (from Kernel, Interfaces, SpecificationComponents)

#### 10.3.2 Connector (from Kernel, Interfaces, SpecificationComponents)

#### 10.3.3 Operation (from Kernel, Interfaces, SpecificationComponents)

#### 10.3.4 ComponentRealization (from BasicComponents)

### Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>23</td>
</tr>
<tr>
<td>Abstract Syntax</td>
<td>24</td>
</tr>
<tr>
<td>Class Descriptions</td>
<td>38</td>
</tr>
<tr>
<td>Components</td>
<td>143</td>
</tr>
<tr>
<td>Composite Structures</td>
<td>161</td>
</tr>
<tr>
<td>Deployments</td>
<td>193</td>
</tr>
</tbody>
</table>

### Reading the Standard

- **Part I - Structure**
- **7. Classes**
- **8. Components**
- **9. Composite Structures**
- **10. Deployments**

---

### Additional Information

- **6.1 Changes to Adopted**
- **6.2 Architectural Aligned**
- **6.3 On the Run-Time Semantics of UML**
- **6.4 The UML Metamodel**
- **6.5 How to Read this Specification**
- **6.6 Additional Information**

---

### Normative References

- **2.1 Language Units**
- **2.2 Compliance Level C**
- **2.3 Meaning and Types**
- **2.4 Compliance Level C**
- **3. Normative References**
- **4. Terms and Definitions**
- **5. Symbols**

---

### Symbols

- **6.1 Changes to Adopted**
- **6.2 Architectural Aligned**
- **6.3 On the Run-Time Semantics of UML**
- **6.4 The UML Metamodel**
- **6.5 How to Read this Specification**
- **6.6 Additional Information**
7.3.4 Classifier (from Kernel, Dependencies, PowerTypes)

A classifier is a classification of instances; it describes a set of instances that have features in common.

**Generalizations**
- "Namespace (from Kernel)" on page 96
- "RoleKind (from Kernel)" on page 103
- "Type (from Kernel)" on page 173

**Description**
A classifier is a namespace whose members can include features. Classifier is an abstract meta-class.

**Attributes**
- /name: Property[*]
  - Defines the classifier's name.
  - It is intended to be used by other classifiers (e.g., as the target of generalization hierarchies or generalization relationships).
- /general: Classifier[*]
  - Specifies the general classifiers for this classifier. This is derived.
- /isAbstract: Boolean
  - Determines whether the classifier is abstract.
  - An abstract classifier does not provide a complete declaration and can typically not be instantiated. An abstract classifier is intended to be used by other classifiers (e.g., as the target of generalization hierarchies or generalization relationships).
  - Defaults (false).

**Associations**
- /packageDepenencies: PackageDependency[*]
  - References classifiers that are included in the package. This is derived.
- /substitutedClassifier: Classifier[*]
  - Specifies the classifier that is substituted for the classifier. This is derived.
- /redefinedClassifier: Classifier[*]
  - Specifies the classifiers that are redefined by this classifier. These classifiers navigate to more general classifiers. This is derived.
- /generalization: Generalization[*]
  - Specifies the generalization relationships for this classifier. These relationships navigate to more general classifiers in the generalization hierarchy. Subclass classifier -> superclass classifier.
  - Specifies all classifiers that are subclassed by this classifier. These classifiers navigate to more general classifiers. Subclass classifier <- superclass classifier.
  - Specifies all classifiers that are subclassed by this classifier. These classifiers navigate to more general classifiers. Subclass classifier <- superclass classifier.

**Additional Operations**
- /querySuperkind( )
  - Returns all of the superkinds of the classifier. This is derived.
- /queryKind( )
  - Returns the kind of the classifier. This is derived.
- /queryInheritedMember( )
  - Returns all of the inherited members for the classifier. This is derived.
- /queryFeature( )
  - Returns all of the features for the classifier. This is derived.
- /queryParents( )
  - Returns all of the parents for the classifier. This is derived.
- /queryAbstract( )
  - Returns whether the classifier is abstract. This is derived.
- /queryNonAbstract( )
  - Returns whether the classifier is non-abstract. This is derived.
- /query чемber( )
  - Returns whether the classifier is a member. This is derived.
Reading the Standard Cont’d

7.3.3.8 Classifier (from Kernel, Dependencies, PowerTypes)

- Generalization
  - «general» (Classifier[*])
  - «redefinedClassifier» (Classifier[*])
- Description
  - «name» (name)
  - «type» (default notation for a classifier is a solid-outline rectangle containing the classifier’s name, and optionally with default notation for a classifier is a solid-outline rectangle containing the classifier’s name, and optionally with)
- Attributes
  - «name» (name)
  - «type» (default notation for a classifier is a solid-outline rectangle containing the classifier’s name, and optionally with)
- Generalizations
  - «general» (Classifier[*])
  - «redefinedClassifier» (Classifier[*])
- Dependencies
  - «dependency» (dependency)

7.3.8 Classifier

A classifier is a classification of instances according to their features. A classifier is a classification of instances according to their features. It is a container for defining a set of features, constraints, and operations that can be used to manipulate instances of the classifier.

Generalizations

- A classifier may have one or more general classifiers, which are classifiers whose instances are subtypes of instances of the classifier. A classifier may have one or more general classifiers, which are classifiers whose instances are subtypes of instances of the classifier.

Description

- The name and type of a classifier are shown in boldface. The name and type of a classifier are shown in boldface.

Attributes

- Attributes may include:
  - Name: the name of the classifier.
  - Type: the default notation for a classifier is a solid-outline rectangle containing the classifier’s name, and optionally with.
  - Visibility: the visibility of the classifier (public, private, protected).

Generalizations

- A classifier may have one or more general classifiers, which are classifiers whose instances are subtypes of instances of the classifier. A classifier may have one or more general classifiers, which are classifiers whose instances are subtypes of instances of the classifier.

Dependencies

- Dependencies may include:
  - Dependency: a dependency relationship between classifiers.

Additional Operations

- Additional operations may include:
  - an operation with default notation for a classifier is a solid-outline rectangle containing the classifier’s name, and optionally with.

Constraints

- Constraints may include:
  - Constraint: a constraint on the classifier.

Style Guidelines

- Style guidelines may include:
  - Attribute names: typically begin with a lowercase letter. While note names are often formed by concatenating the result, spelling conventions vary across languages.
  - Enumeration literals: typically begin with a lowercase letter. While note names are often formed by concatenating the result, spelling conventions vary across languages.

Operations

- Operations may include:
  - Operation: an operation that can be performed on instances of the classifier.

Examples

- Example usage of a classifier may include:
  - Classifier::feature: a feature defined in the classifier.
  - Classifier::parents(): a set of classifiers that are parents of the classifier.
  - Classifier::maySpecializeType(c : Classifier): a boolean value indicating whether the classifier may specialize to another classifier.

Semantic Variation Points

- Semantic variation points may include:
  - Semantic variation points: points in the model where the behavior of classifiers may differ based on the context.

Notations

- Notations may include:
  - Notation: a notation for a classifier.

Presentation Options

- Presentation options may include:
  - Presentation options: options for displaying the classifier in a diagram.
A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to other classifiers. A classifier can specify a generalization hierarchy by referencing its general classifiers.

### Figure 7.29: Class notation: attributes and operations grouped according to visibility

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size: Area = (100, 100)</td>
<td>Public</td>
</tr>
<tr>
<td>display()</td>
<td>&quot;RedefinableElement (from Kernel)&quot; on page 130</td>
</tr>
</tbody>
</table>

Generalization hierarchies must be directed and acyclic. A classifier cannot be both a transitively general and itself nor may its instances also be its subclasses.

### Figure 7.30: Examples of attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
<td>Weights of the classifier that conform to the weight of its type</td>
</tr>
<tr>
<td>height: Integer</td>
<td>Derived.</td>
</tr>
<tr>
<td>width: Integer [0..1]</td>
<td>Only called when the argument is something owned by a parent.</td>
</tr>
</tbody>
</table>

Classifiers may also be shown using association notation, with adornments at the tail of the arrow as shown in Figure 7.31.

### Figure 7.31: Association notation for attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasVisibilityOf()</td>
<td>Determines whether a named element is visible in the classifier. By default all are visible. It is intended to be overridden by subclasses.</td>
</tr>
</tbody>
</table>

### Figure 8.1: Diagram showing classifier and generalization relations

For example, a Bank Account Type classifier could have a powertype association with a GeneralizationSet. This GeneralizationSet could have associate with it Generalizations where the class, i.e., "general Classifier: Bank Account has two specific subclasses (i.e., a Checking Account and a Savings Account). Checking Account and Savings Account are both instances of Bank Account Type as well as a specialization of Bank Account. For more explanation and examples, see Examples in the GeneralizationRelations subclass, below.

Semantic Variations

#### 7.3.9 Comment (from Kernel)

A comment is a textual annotation that can be attached to a set of elements.

### Figure 7.32: Examples of comments

<table>
<thead>
<tr>
<th>Comment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;This is a comment.&quot;</td>
<td>Made up of a line of text that begins with &quot;/*&quot;</td>
</tr>
</tbody>
</table>

### Figure 7.33: Package notation

For example, a Bank Account Type classifier could have a powertype association with a GeneralizationSet. This GeneralizationSet could have associate with it Generalizations where the class, i.e., "general Classifier: Bank Account has two specific subclasses (i.e., a Checking Account and a Savings Account). Checking Account and Savings Account are both instances of Bank Account Type as well as a specialization of Bank Account. For more explanation and examples, see Examples in the GeneralizationRelations subclass, below.
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
  - ...
  - (objects/instances of classes in a UML model)
  - (objects/instances of classes in the UML model)
  - (instances of class meta-model)
**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. [Buschermöhle and Oelerink, 2008]

---

**Meta-Modelling: (Anticipated) Benefits**
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 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 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 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’s possible to fix things 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-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 Gtk::Button, Gtk::Toolbar, etc. corresponding to the stereotype.

  Et voilà: we can model Gtk-GUIs and generate code for them.

- Another view:
  - UML with these stereotypes is a new modelling language: Gtk-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 Gtk-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, 2005].)
Benefits for Language Design Cont’d

- 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, 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/GMF/GEF.

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 Model (to Model) Transformation

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

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 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 XMI

```
<?xml version='1.0' encoding='UTF-8'?>
<XMI xmi.version='1.2' xmlns:UML='org.omg.xmi.namespace.UML' timestamp='Mon Feb 02 18:23:12 CET 2009'>
  <XMI.content>
    <UML:Model xmi.id='...'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id='...' name='SensorA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='pt100'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id='...' name='ControllerA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='65C02'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id='...' name='UsbA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='NET2270'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```
Domain Inclusion Structure

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

Now a **structure** $\mathcal{S}$

- [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.
  \[
  \forall C \in \mathcal{C} : \mathcal{S}(C) \supseteq \bigcup_{C \triangleleft D} \mathcal{S}(D).
  \]

**Note**: the old setting coincides with the special case $\triangleleft = \emptyset$. 
**Domain Inclusion System States**

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

\[
\sigma : \mathcal{D}(\mathcal{C}) \rightarrow (V \rightarrow (\mathcal{D}(\mathcal{I}) \cup \mathcal{D}(\mathcal{C}_{0,1}) \cup \mathcal{D}(\mathcal{C}_{\ast})))
\]

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

- [as before] \( \sigma(u)(v) \in \mathcal{D}(\tau) \) if \( v : \tau, \tau \in \mathcal{I} \) or \( \tau \in \{C_{\ast}, C_{0,1}\} \).
- [changed] \( \text{dom}(\sigma(u)) = \bigcup_{C_{0} \preceq C} \text{atr}(C_{0}) \),

**Example:**

\[
\forall v \in \mathcal{D}(\mathcal{I}) \quad \text{dom}(\sigma(u)) = \text{atk}(\mathcal{D}) \cup \text{atk}(\mathcal{C})
\]

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

---

**Preliminaries: Expression Normalisation**

**Recall:**

- we want to allow, e.g., “context \( D \ inv v < 0 \)”.
- we assume fully qualified names, e.g. \( C::v \).

Intuitively, \( D \) shall denote the “most special more general” \( C::v \) 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 \( f \)) 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 (= replace by) \( C::v \),
- where \( C \) is the greatest class wrt. \( \preceq \) such that
  - \( C \preceq D \) and \( C::v \in \text{atr}(C) \).

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: 
  \[ v, r \in V; C, D \in \mathcal{C} \]

  \[
  expr ::= \begin{array}{c}
  v(expr_1) : \tau_C \rightarrow \tau(v), \\
  r(expr_1) : \tau_C \rightarrow \tau_D, \\
  r(expr_1) : \tau_C \rightarrow \text{Set}(\tau_D),
  \end{array}
  \]
  
  if \( v : \tau \in \mathcal{P} \) \( r : D_{0,1} \) \( r : D_\ast \)

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

More Interesting: Well-Typed-ness

- We want
  
  \[
  \text{context } D \ \text{ inv : } v < 0
  \]
  
  to be well-typed.

  Currently it isn’t because
  
  \[ v(expr_1) : \tau_C \rightarrow \tau(v) \]

  but \( A \vdash \text{self : } \tau_D \).

  (Because \( \tau_D \) and \( \tau_C \) are still \textbf{different types}, although \( \text{dom}(\tau_D) \subset \text{dom}(\tau_C) \).)

- So, add a (first) new typing rule

  \[
  \frac{A \vdash expr : \tau_D}{A \vdash expr : \tau_C}, \quad \text{if } C \preceq D.
  \]

  (Inh)

  Which is correct in the sense that, if ’expr’ is of type \( \tau_D \), then we can use it everywhere, where a \( \tau_C \) is allowed.

  The system state is prepared for that.
Well-Typed-ness with Visibility Cont’d

\[
\begin{align*}
A, D \vdash \text{expr} : \tau_C &\quad \xi = + \quad \text{(Pub)} \\
A, D \vdash C :: \text{v(expr)} : \tau, \xi = \# &\quad C \preceq D \quad \text{(Prot)} \\
A, D \vdash \text{expr} : \tau_C &\quad \xi = - \quad C = D \quad \text{(Priv)}
\end{align*}
\]

\langle C :: \text{v} : \tau, \xi, v_0, P \rangle \in \text{atr}(C).

Example:

<table>
<thead>
<tr>
<th>Context</th>
<th>(v_1)</th>
<th>(v_2)</th>
<th>(v_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C)</td>
<td>(&lt;0)</td>
<td>(&lt;0)</td>
<td>(&lt;0)</td>
</tr>
<tr>
<td>(D)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Satisfying OCL Constraints (Domain Inclusion)

- Let \(M = (C\mathcal{D}, D\mathcal{D}, \mathcal{M}, \mathcal{S})\) be a UML model, and \(D\) a structure.
- We (continue to) say \(M \models \text{expr} \) for context \(C\) \(\text{inv} : \text{expr}_0 \in \text{Inv}(M)\) iff

\[
\forall \pi = (\sigma_i, \epsilon_i)_{i \in \mathbb{N}} \in \llbracket M \rrbracket \quad \forall i \in \mathbb{N} \quad \forall u \in \text{dom}(\sigma_i) \cap D(C) : \\
I[\text{expr}_0](\sigma_i, \{\text{self} \mapsto u\}) = 1.
\]
- \(M\) is (still) consistent if and only if it satisfies all constraints in \(\text{Inv}(M)\).

Example:

\[
\sigma : \begin{cases} \\
  \mathcal{D} \quad \mathcal{S} \quad \mathcal{C} \\
  v : \mathbb{D} \\
  C :: x = 20 \\
 \end{cases}
\]

\[
\begin{array}{c}
\sigma \in \text{dom}(x) \\
\sigma(x) \in \text{dom}(C) \\
\sigma(C) \subseteq D
\end{array}
\]

\[
\begin{array}{c}
|\sigma(C)\rangle \in \text{dom}(\text{expr}_0) \\
\text{sat} \text{ expr}_0 \text{ and } C \subseteq D
\end{array}
\]
Transformers (Domain Inclusion)

- Transformers also remain the same, e.g. [VL 12, p. 18]

\[
\text{update}(\text{expr}_1, v, \text{expr}_2) : (\sigma, \epsilon) \mapsto (\sigma', \epsilon)
\]

with

\[
\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\text{expr}_2](\sigma)]]
\]

where \( u = I[\text{expr}_1](\sigma) \).

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

• **Wanted**: triggers shall also be sensitive for inherited events, sub-class shall execute super-class’ state-machine (unless overridden).

\[
(\sigma, \varepsilon) \xrightarrow{\text{cons, } \text{Snd}} (\sigma', \varepsilon') \quad \text{if} \\
\exists u \in \text{dom}(\sigma) \cap \mathcal{D}(C) \exists u_E \in \mathcal{D}(\varepsilon') : u_E \in \text{ready}(\varepsilon, u) \\
u \text{ is stable and in state machine state } s, \text{ i.e. } \sigma(u)(\text{stable}) = 1 \text{ and } \sigma(u)(st) = s, \\
a \text{ transition is enabled, i.e.} \\
\exists (s, F, \text{expr}, \text{act}, s') \in \rightarrow (\text{SM}_{C}) : F = E \land I[\text{expr}] = 1 \text{ where } \hat{\sigma} = \sigma[u.\text{params}_E \mapsto u], \\
\text{and} \\
(\sigma', \varepsilon') \text{ results from applying } t_{\text{act}} \text{ to } (\sigma, \varepsilon) \text{ and removing } u_E \text{ from the ether, i.e.} \\
(\sigma'', \varepsilon') = t_{\text{act}}(\hat{\sigma}, \varepsilon \ominus u_E), \\
\sigma' = (\sigma''[u.st \mapsto s', u.\text{stable} \mapsto b, u.\text{params}_E \mapsto \emptyset])|\sigma(w)\backslash\{u_E\} \\
\text{where } b \text{ depends:} \\
\quad \text{If } u \text{ becomes stable in } s', \text{ then } b = 1. \text{ It } \text{does} \text{ become stable if and only if there is no transition without trigger enabled for } u \text{ in } (\sigma', \varepsilon'). \\
\quad \text{Otherwise } b = 0. \\
\quad \text{Consumption of } u_E \text{ and the side effects of the action are observed, i.e.} \\
\text{cons} = \{(u, (E, \sigma(u_E)))\}, \text{Snd} = \text{Obs}_{\text{act}}(\hat{\sigma}, \varepsilon \ominus u_E).
\]

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.
    \[
    \sigma, \text{cons, Snd} \models_\beta E^I_{x,y} \\
    \text{if and only if} \\
    \beta(x) \text{ sends an } F\text{-event to } \beta y \text{ where } E \preceq F.
    \]
  • **Note**: \( C \)-instance line also binds to \( C' \)-objects.
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++)

\[ x = 0; \]

in \( D \) to

\[ \text{uplink}_C \rightarrow x = 0; \]
**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 $f$) in the *context* of class $D$,
  
  - let $C$ be the *smallest* class wrt. “$\preceq$” such that
    - $C \preceq D$, and
    - $C::v \in \text{atr}(D)$
  
  - then there exists (by definition) $C \triangleleft C_1 \triangleleft \ldots \triangleleft C_n \triangleleft D$,
  
  - normalise $v$ to (= replace by)
    
    $\text{uplink}_{C_n} \rightarrow \ldots \rightarrow \text{uplink}_{C_1}.C::v$

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

---

**Uplink Structure, System State, Typing**

- Definition of structure remains *unchanged*.

- Definition of system state remains *unchanged*.

- Typing and transformers remain *unchanged* — the preprocessing has put everything in shape.
Satisfying OCL Constraints (Uplink)

- Let $M = (E, O, M, S)$ be a UML model, and $D$ a structure.
- We (continue to) say $M \models expr$ for
  
  $$
  \text{context } C \ \text{inv} : expr_0 \in \text{inv}(M) = \text{expr}
  $$

  if and only if

  $$
  \forall \pi = (\sigma_i)_{i \in \mathbb{N}} \in [M]
  
  \forall i \in \mathbb{N}
  
  \forall u \in \text{dom}(\sigma_i) \cap D(C):
  
  I[expr_0](\sigma_i, \{ \text{self} \mapsto u \}) = 1.
  $$

- $M$ is (still) consistent if and only if it satisfies all constraints in $\text{inv}(M)$.

Transformers (Uplink)

- What has to change is the create transformer:

  $$
  \text{create}(C, expr, v)
  $$

- Assume, $C$’s inheritance relations are as follows.

  $$
  C_{1,1} < \ldots < C_{1,n_1} < C,
  
  \ldots
  
  C_{m,1} < \ldots < C_{m,n_m} < C.
  $$

- Then, we have to
  
  - create one fresh object for each part, e.g.
    $$
    u_{1,1}, \ldots, u_{1,n_1}, \ldots, u_{m,1}, \ldots, u_{m,n_m},
    $$
  
  - set up the uplinks recursively, e.g.
    $$
    \sigma(u_{1,2})(\text{uplink}_{C_{1,1}}) = u_{1,1}.
    $$
  
- And, if we had constructors, be careful with their order.
Late Binding (Uplink)

- Employ something similar to the “mostspec” trick (in a minute!). But the result is typically far from concise.
  (Related to OCL’s isKindOf() function, and RTTI in C++.)

Domain Inclusion vs. Uplink Semantics
**Cast-Transformers**

- C c;
- D d;

**Identity upcast (C++):**
- C* cp = &d;  // assign address of ‘d’ to pointer ‘cp’

**Identity downcast (C++):**
- D* dp = (D*)cp;  // assign address of ‘d’ to pointer ‘dp’

**Value upcast (C++):**
- *c = *d;  // copy attribute values of ‘d’ into ‘c’, or,  
             // more precise, the values of the C-part of ‘d’

---

**Casts in Domain Inclusion and Uplink Semantics**

<table>
<thead>
<tr>
<th>Domain Inclusion</th>
<th>Uplink</th>
</tr>
</thead>
<tbody>
<tr>
<td>C* cp = &amp;d;  \hspace{1cm} <strong>easy:</strong> immediately compatible \hspace{1cm} (in underlying system state) because &amp;d yields an identity from ( \mathcal{P}(D) \subset \mathcal{P}(C) ).</td>
<td>easy: By pre-processing, C* cp = d.uplink(_C);</td>
</tr>
<tr>
<td>D* dp = (D*)cp;  \hspace{1cm} <strong>easy:</strong> the value of cp is in ( \mathcal{P}(D) \cap \mathcal{P}(C) ) because the pointed-to object is a D. \hspace{1cm} Otherwise, error condition.</td>
<td>difficult: we need the identity of the D whose C-slice is denoted by cp. (See next slide.)</td>
</tr>
<tr>
<td>c = d;  \hspace{1cm} <strong>bit difficult:</strong> set (for all C \subseteq D) ( (C)(\cdot, \cdot) : \tau_D \times \Sigma \rightarrow \Sigma_{ab(C)} ) ( (u, \sigma) \mapsto \sigma(u)</td>
<td>_{ab(C)} ) \hspace{1cm} Note: ( \sigma' = \sigma[u_C \mapsto \sigma(u_D)] ) is not type-compatible!</td>
</tr>
</tbody>
</table>
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. have
    \[
    \text{all} \in T, \quad \mathcal{P}(\text{all}) = \bigcup_{C \in E} \mathcal{P}(C)
    \]
  - In each \(\leq\)-minimal class have associations “mostspec” pointing to most specialised slices, plus information of which type that slice is.
  - Then downcast means, depending on the mostspec type (only finitely many possibilities), going down and then up as necessary, e.g.
    \[
    \text{switch(mostspec_type)}\{
    \text{case } C:\n    \text{dp = cp -> mostspec -> uplink}_{D_n} \rightarrow \ldots \rightarrow \text{uplink}_{D_1} \rightarrow \text{uplink}_{D};
    \text{...}
    \}
    \]

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 needed 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...
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


