### Meta-Modelling: Idea and Example

**Meta-Modelling** is one major prerequisite for understanding

- the standard documents \[\text{OMG, 2007a, OMG, 2007b}\], 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 $M_U$ such that

- the set of legal instances of $M_U$ 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 $\text{isQuery}$,
- any number of parameters,
- a return type.

Can we model this (in UML, for a start)?

---

**UML Meta-Model: Extract from UML 2.0 Standard**

```
Comment
Element
NamedElement
name
visibility
Type
TypedElement
RedefElement
Feature
Namespace
Classifier
StructFeature
BehavFeature
Class
Property
Operation
Parameter

♦

∗

∗

∗

type

0 .. 1

0 .. 1

0 .. 1

```
Meta-Modelling: Principle

Modelling vs. Meta-Modelling

Class name: Str
Property name: Str
Type name: Str

$C_v: Z: \text{Class name} = C: \text{Property name} = v: \text{Type name} = Z$

$S = (\{Z\}, \{C\}, \{v\}, \{C \mapsto v\})$

$D \Rightarrow \Sigma$

$DS: C_v = 0$

instance-of $\sigma = \{u \mapsto \{v \mapsto 0\}\}$

Meta-Model $(M_2)$
Model $(M_1)$
Instance $(M_0)$

• So, if we have a meta model $M_U$ of UML, then the set of UML models is the set of instances of $M_U$.

• A UML model $M$ can be represented as an object diagram (or system state) wrt. the meta-model $M_U$.

• Other view: An object diagram wrt. meta-model $M_U$ can (alternatively) be rendered as the UML model $M$.

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.

\[ \text{generalization}\rightarrow \text{includes}(\text{self}) \]

[OMG, 2007b, 53]

• The other way round: Given a UML model $M$, unfold it into an object diagram $O_1$ wrt. $M_U$. If $O_1$ is a valid object diagram of $M_U$ (i.e. satisfies all invariants from Inv $(M_U)$), then $M$ is a well-formed UML model.

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

The UML 2.x Standard Revisited

Claim: Extract from UML 2.0 Standard

Comment

Element

NamedElement

name

visibility

Type

TypedElement

RedefElement

Feature

Namespace

Classifier

StructFeature

BehavFeature

Class

Property

Operation

Parameter

♦

∗

redefdElem

∗

type

0..1

♦

♦

0..1

∗

♦

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗

0..1

∗
Figure 7.11 - Operations diagram of the Kernel package

Figure 7.10 - Features diagram of the Kernel package

Figure 7.9 - Classifiers diagram of the Kernel package

Figure 7.4 - Namespaces diagram of the Kernel package

Figure 7.3 - Root diagram of the Kernel package

Figure 13.6 - Common Behavior
A classifier is a namespace whose members can include features. Classifier is an abstract metaclass.

"Type (from Kernel)" on page 135

7.3.42 ParameterDirectionKind (from Kernel) ................................................................ 122

7.3.43 PrimitiveType (from Kernel) ........................................................................... 122

7.3.40 PackageMerge (from Kernel) ........................................................................... 111

7.3.33 NamedElement (from Kernel, Dependencies) ................................................... 97

7.3.29 LiteralSpecification (from Kernel) ................................................................... 92

7.3.27 LiteralInteger (from Kernel) ............................................................................ 90

7.3.26 LiteralBoolean (from Kernel) ........................................................................... 89

9. Composite Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161

8.3.3 ConnectorKind (from BasicComponents) .......................................................... 157

8.3.4 ComponentRealization (from BasicComponents) ............................................. 157

8.3.3 ConnectorKind (from BasicComponents) .......................................................... 157

8.3.4 ComponentRealization (from BasicComponents) ............................................. 157

7.3.16 Enumeration (from Kernel) .............................................................................. 69

7.3.14 Element (from Kernel) ..................................................................................... 64

7.3.15 ElementImport (from Kernel) .......................................................................... 65

7.3.11 DataType (from Kernel) ................................................................................... 62

7.3.13 DirectedRelationship (from Kernel) .................................................................. 63

7.3.9 Comment (from Kernel) .................................................................................... 57

7.3.10 Constraint (from Kernel) .................................................................................. 58

6.6 Acknowledgements  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

6.5.1 Specification format ... Diagram format ....................................................................................................... 18

6.5.1 Specification format ... Diagram format ....................................................................................................... 18

6.3.2 The Semantics Architecture ............................................................................... 11

6.2 The Architecture and Conceptual Semantics .......................................................... 10

4. Terms and Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3. Model Level Semantics ............................................................................................ 10

2. Conformance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1. Scope  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

Table of Contents
Classifier::feature Refers to all of the Properties that are direct (i.e., not inherited or imported) attributes of the classifier. Subsets

- Area
- Window

If inheritance, this will be a larger set than feature.

If the query allFeatures() gives all of the features in the namespace of the classifier. In general, through mechanisms such as itself nor may its instances also be its subclasses.

•

Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and not

Classifiers that have different specialization constraints.

•

Conforms to classifiers in the generalization hierarchy. Subsets

Notation

• Center keyword (including stereotype names) in plain face within guillemets above the classifier name.

• For those languages that distinguish between uppercase and lowercase characters, capitalize names (i.e, begin them

• and using lowercase for all letters except for upcasing the first letter of each word but the first.

Constraints

• The query conformsTo() gives true for a classifier that defines a type that conforms to another. This is used, for example,

• and

• The query maySpecializeType() determines whether this classifier may have a generalization relationship to classifiers of

References

• "Namespace (from Kernel)" on page 99

A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to

All instances of a general classifier also applies to instances of the specific classifier.

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

The individual properties of an attribute can be shown in columns rather than as a continuous string.

self.inheritedMember->includesAll(self.inherit(self.parents()->collect(p | p.inheritableMembers(self)))

The inheritedMember association is derived by inheriting the inheritable members of the parents.

ClassA::height is an attribute of type Integer with a default initial value of 5.

This is

Figure 7.31 - Association-like notation for attribute

an important in this diagram.

true

false

self.allParents()->includes(self)

Package Dependencies

• / feature : Feature 

• / feature : Feature 

A separator line is not drawn for a suppressed compartment. If a compartment is

in one place a default notation available for any concrete subclass of Classifier for which this notation is suitable. The

Classifier is an abstract model element, and so properly speaking has no notation. It is nevertheless convenient to define

Classifier::allParents(): Set(Classifier);allParents = self.parents()->union(self.parents()->collect(p | p.allParents())

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a

The general classifiers are the classifiers referenced by the generalization relationships.

The query allFeatures = member->select(oclIsKindOf(Feature))

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a

The general classifiers are the classifiers referenced by the generalization relationships.

The query inheritableMembers = member->select(m | c.hasVisibilityOf(m))

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a

The general classifiers are the classifiers referenced by the generalization relationships.

The query inheritableMembers = member->select(m | c.hasVisibilityOf(m))

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a

The general classifiers are the classifiers referenced by the generalization relationships.

The query inheritableMembers = member->select(m | c.hasVisibilityOf(m))

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a

The general classifiers are the classifiers referenced by the generalization relationships.

The query inheritableMembers = member->select(m | c.hasVisibilityOf(m))

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a

The general classifiers are the classifiers referenced by the generalization relationships.

The query inheritableMembers = member->select(m | c.hasVisibilityOf(m))

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a

The general classifiers are the classifiers referenced by the generalization relationships.

The query inheritableMembers = member->select(m | c.hasVisibilityOf(m))

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a
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 . . .

• 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. [Buschermühle and Oelerink, 2008].

Meta-Modelling: (Anticipated) Benefits

• We'll (superficially) look at three aspects:

• Benefits for Modelling Tools.

• Benefits for Language Design.

• Benefits for Code Generation and MDA.

The meta-model $M_U$ of UML immediately provides a data-structure representation for the abstract syntax ($\sim$ for our signatures).

If we have code generation for UML models, e.g. into Java, then we can immediately represent UML models in memory (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.

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 exit yet.
• 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.

• 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.
In general, code generation can (in colloquial terms) become an XSLT script.

If we have the UML model (or the DSL model) given as an XML file, code generation can be as simple as a special case of model-to-model transformation;

Example: Model and XMI

Recall that we said that, e.g. Java code, can also be seen as a model.

Note: Code generation needn't be as expensive as buying a modelling tool only the destination looks quite different. So code-generation is a special case of model-to-model transformation;
Lecture 1:

- **Educational Objectives:**
  - You should be able to explain the term **model**.
  - You should know the idea (and hopes and promises) of **model-driven** SW development.
  - You should be able to explain how **UML** fits into this general picture.
  - You should know what we've done in the course, and why.
  - Thus, you should be able to decide whether you want to stay with us...

- How can UML help with software development?
- Where is which sublanguage of UML useful?
- For what purpose? With what drawbacks?

Wrapup: Examining Motivation

- What is a model? For example?
- "A model is an image or a pre-image" of what? Please explain!
- When is a model a good model?
- What is model-based software engineering?
- MDA? MDSE?
- What do people hope to gain from MBSE? Why? Hope Justified?
- What are the fundamental pre-requisites for that?
- What are purposes of modelling guidelines?
- Could you illustrate this with examples?
- How can we establish/enforce them? Can tools or procedures help?
- What's the qualitative difference between the modelling guideline "all association ends have a multiplicity" and "all state-machines are deterministic"?

Wrapup: Examining "The Big Picture"

- What is UML (definitely)? Why?
- What is it (definitely) not? Why?
- How does UML relate to programming languages?
- What are the intentions of UML?
- What is the history of UML? Why could it be useful to know that?
- Where can (what part of) UML be used in MBSE?
- For what purpose? To improve what?

Lecture 2:

- **Educational Objectives:**
  - Capabilities for these tasks/questions:
    - Why is UML of the form it is?
    - Shall one feel bad if not using all diagrams during software development?
    - What is a signature, an object, a system state, etc.? What's the purpose in the course?
    - How do Basic Object System Signatures relate to UML class diagrams?

Lecture 3 & 4:

- **Educational Objectives:**
  - Capabilities for these tasks/questions:
    - Please explain/read out this OCL constraint. Is it well-typed?
    - Please formalise this constraint in OCL.
    - Does this OCL constraint hold in this (complete) system state?
    - Can you think of a system state satisfying this constraint?
    - Please un-abbreviate all abbreviations in this OCL expression.
    - In what sense is OCL a three-valued logic? For what purpose?
    - How are \( D(C) \) and \( \tau_C \) related?
What's a system configuration?

- lecture 22: wrapup & questions
- consumption?
- lecture 21: meta-modelling, inheritance ii
- what's the difference: signal, signal event, event, trigger, reception
- lecture 20: inheritance i
- what is an ether? example? why did we introduce it?
- lecture 19: live sequence charts ii
- lecture 18: live sequence charts i
- please unabbreviate this abbreviated transition annotation
- lecture 17: hierarchical state machines ii
- what is: trigger, guard, action?
- lecture 16: hierarchical state machines i
- can you please model the following behaviour
- lecture 15: core state machines v, rhapsody
- lecture 14: core state machines iv
- capabilities for following tasks/questions.
- lecture 13: core state machines iii
- lecture 12: core state machines ii
- lecture 11: core state machines i
- lecture 10: constructive behaviour, state machines overview
- can you please model the following behaviour (and
- lecture 9: class diagrams iii
- lecture 8: class diagrams ii
- lecture 7: type systems and visibility
- what's the basic causality model?
- lecture 5: object diagrams
- what's the difference between reflective and constructive descriptions of
- lecture 4: ocl semantics
- lecture 3: object constraint language (ocl)
- lecture 2: semantical model
- lecture 1: motivation and overview
- wrapup: modelling behaviour, constructive
- wrapup: modellung behaviour, constructive
Wrapup: Inheritance

Lecture 20 & 21:

• Educational Objectives:
  - Capabilities for following tasks/questions.
  - What's the effect of inheritance on LSCs, State Machines, System States?
  - What's the Liskov Substitution Principle?
  - What is commonly understood under (behavioural) sub-typing?
  - What is the subset, what the uplink semantics of inheritance?
  - What is late/early binding?
  - What's the idea of Meta-Modelling?

Hmm...

Open book or closed book...?

References


