Meta-Modelling: Why and What

Meta-Modelling is one major prerequisite for understanding the standard documents OMG (2007a, b), and the MDA ideas of the OMG. The idea is somewhat simple: if a modelling language is about modelling things, and if UML models are things, then why not model UML models using 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 (among others) 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)?

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

♦

∗

0..1

♦

0..1

♦

0..1

∗

type

0..1

0..1

0..1

type
Modelling vs. Meta-Modelling

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

\( C \vdash v : \text{Int} \)
\( S = (\{\text{Int}\}, \{\text{C}\}, \{v\}, \{\text{C} \mapsto v\}) \)
\( D \Rightarrow \Sigma \)

\( DS : C \vdash 0 \)

\( \sigma = \{u \mapsto \{v \mapsto 0\}\} \in \text{Meta-Model (M2)} \)

Model (M1)
Instance (M0)

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.

Constraint 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.

\( \text{allParents}() \rightarrow \text{includes}(\text{self}) \)

(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 \( \text{Inv}(M_U) \)), 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
A classifier is a classification of instances. It describes a set of instances that have features in common. A classifier is a namespace whose members can include features. Classifier is an abstract metaclass.

Classifiers can be used to group elements together that have a common set of features. A classifier is a classification of instances according to their features. Attributes and operations of a classifier are inherited by its instances.

Generalization relationships defined for that GeneralizationSet. In other words, a power type may not be an instance of itself nor may its instances also be its subclasses. Constraints applying to instances of the specific classifier. Any constraint applying to instances of the specific classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the specific classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the specific classifier are implicitly specified for instances of the specific classifier.

Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and itself nor may its instances also be its subclasses. Constraints applying to instances of the specific classifier. Any constraint applying to instances of the specific classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the specific classifier. Any constraint applying to instances of the specific classifier. Any constraint applying to instances of the specific classifier.

Classifiers in the generalization hierarchy. Subsets of classifiers that have been generalized to more general classifiers. Generalizations of classifiers. Generalizations of classifiers. Generalizations of classifiers. Generalizations of classifiers.

Semantic Variation Points

• powertypeExtent : GeneralizationSet
• substitution : Substitution
•maySpecializeType : Boolean

Constraints

• inherit = inhs
• maySpecializeType = self.oclIsKindOf(c.oclType)
• inheritableMembers = member->select(m | c.hasVisibilityOf(m))
• inheritableMembers = member->select(m | c.hasVisibilityOf(m))
• inheritableMembers = member->select(m | c.hasVisibilityOf(m))
• inheritableMembers = member->select(m | c.hasVisibilityOf(m))
• inheritableMembers = member->select(m | c.hasVisibilityOf(m))

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

ClassA::width is an attribute of type Integer. ClassA::area is a derived attribute with type Integer. It is marked as read-only. ClassA::area is a derived attribute with type Integer. It is marked as read-only. ClassA::area is a derived attribute with type Integer. It is marked as read-only. ClassA::area is a derived attribute with type Integer. It is marked as read-only.

ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle. ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle. ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle. ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle.

ClassA::area is a derived attribute with type Integer. It is marked as read-only. ClassA::area is a derived attribute with type Integer. It is marked as read-only. ClassA::area is a derived attribute with type Integer. It is marked as read-only. ClassA::area is a derived attribute with type Integer. It is marked as read-only.
Now you've been "tricked".

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.

This is Meta Object Facility (MOF), which (more or less) coincides with UML Infrastructure (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.

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. Buschermohle and Oelerink (2008).

In particular:

- Benefits for Modelling Tools
- Benefits for Language Design
- Benefits for Code Generation and MDA

And That's It!
Example: Object Diagrams for Documentation

1. OCL Semantics

How are system states and object diagrams related?

2. OCL Syntax

In what sense is OCL a three-valued logic? For what purpose?

Example: Object Diagrams

3. Content:

a. Class Diagrams I

b. OCL Semantics

c. OCL Syntax

4. Educational Objectives:

This Lecture:

Capabilities for these tasks/questions:

a. Basic Object System Signatures

b. UML class diagrams

c. System States

5. Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

6. Lecture 4

a. Example: Object Constraint Language (OCL)

b. Basic Object System Signatures

c. How do Basic Object System Signatures relate to UML class diagrams?

d. What is an object diagram? What are object diagrams good for?

e. Give a system state satisfying this constraint?

7. Last Lecture:

Contents & Goals

This Lecture:

What is a signature, an object, a system state, etc.?

8. Please formalise this constraint in OCL.

9. Please explain this OCL constraint.

10. Please un-abbreviate all abbreviations in this OCL expression.

11. When is an object diagram an object diagram (wrt. what)?

12. Does this OCL constraint hold in this system state?

13. Give a system state satisfying this constraint?

14. What is the purpose of signature, object, etc. in the course?

15. How do Basic Object System Signatures relate to UML class diagrams?

16. When is a set of OCL constraints said to be consistent?

17. What is the purpose of signature, object, etc. in the course?

18. Please formalise this constraint in OCL.
What is: Signal, Event, Ether, Transformer, Step, RTC.

•

OCL Consistency and Satisfiability

Object Diagrams

Capabilities for following tasks/questions.

What's the difference between “aggregation” and “composition”?

What’s the difference between “aggregation” and “composition”?

What makes a class diagram a good class diagram?

Educational Objectives:

Albert-Ludwigs-Universität Freiburg, Germany

Prof. Dr. Andreas Podelski,

Educational Objectives:

Capabilities for following tasks/questions.

C

Can you think of an object diagram which violates this OCL constraint?

Please un-abbreviate all abbreviations in this OCL expression.

What is “multiplicity”? How did we treat them semantically?

Please formalise this constraint in OCL.

Educational Objectives:

Capabilities for following tasks/questions.

C

In what sense is OCL a three-valued logic? For what purpose?

Which annotations of an association arrow are semantically related?

When is an object diagram called partial? What are partial ones good for?

Please un-abbreviate all abbreviations in this OCL expression.

What is the purpose of signature, object, etc. in the course?

Can you please model the following behaviour.

Representing class diagrams as (extended) signatures — for the moment without

•

Software Design, Modelling and Analysis in UML

Introduction: Motivation, Content, Formalia

This Lecture:

Contents & Goals

OCL Syntax

OCL Semantics

Content:

•

Associations syntax and semantics.

•

Associations: the rest.

•

Structures

•

Last Lecture:

Lecture 4

Last Lecture:

Lecture 9

Last Lecture:

Lecture 6

Last Lecture:

Lecture 7

Last Lecture:
Lecture 18

Hierarchical State Machines II

- Entry / exit / do actions, internal transitions
- Legal transitions

Lecture 13

- What is a cut, fired-set, etc.?
- Construct the TBA for this LSC.

Lecture 11

- What are constructive and reflective descriptions of behaviour?
- Capabilities for following tasks/questions.
- What is: Signal, Event, Ether, Transformer, Step, RTC.

Content:

- Last Lecture:
  - Capabilities for following tasks/questions.
  - What is a class diagram?
  - Associations in OCL syntax.
  - How did we treat "multiplicity" semantically?
- This Lecture:
  - Educational Objectives:
    - Study UML syntax.
    - Representing class diagrams as (extended) signatures — for the moment without
  - Map class diagram to (extended) signature cont'd.
  - How are Basic Object System Signatures related to UML class diagrams?
  - Please explain this OCL constraint.
  - What's the purpose of a behavioural model?
  - What is: Signal, Event, Ether, Transformer, Step, RTC.
  - Please explain this class diagram with associations.
- Last Lecture:
  - Educational Objectives:
    - What is a class diagram?
    - Representing class diagrams as (extended) signatures — for the moment without
  - Map class diagram to (extended) signature.
  - How are Basic Object System Signatures related to UML class diagrams?
  - Please explain this OCL constraint.
  - What's the purpose of a behavioural model?
  - What is: Signal, Event, Ether, Transformer, Step, RTC.
  - Please explain this class diagram with associations.
Lecture 21

Two approaches to obtain desired semantics

• Automaton construction
• LSC syntax

Cut Examples, Firedset

• Rhapsody code generation

Content:
• A closer look onto code generation

What is a cut, fired-set, etc.?

• What is a cut, fired-set, etc.?

• What about methods?

• What's the Liskov Substitution Principle?

• How are passive reactive objects treated in Rhapsody's UML semantics?

• What is the idea of deferred events?

• Last Lecture:
  • How to define what happens at "system / model startup"?

• What is a legal state configuration?

• Is this UML model consistent with that OCL constraint?

• What is a step / run-to-completion step?

• Deferred events

• Last Lecture:
  • How did we treat "multiplicity" semantically?

• What is visibility good for?

• Representing class diagrams as (extended) signatures — for the moment without Structures

• Could you please map this signature to a class diagram?

• Could you please map this class diagram to a signature?

• Educational Objectives:
  • Study UML syntax.
  • OCL Syntax
  • What is visibility as an extension of well-typedness.

• OCL Consistency and Satisfiability

• Transformers

• Capabilities for following tasks/questions.

• Last Lecture:
  • This Lecture:
  • This Lecture:

• Educational Objectives:
  • Contents & Goals

• Educational Objectives:
  • Contents & Goals

• Capabilities for these tasks/questions:

• Capabilities for these tasks/questions:

• Content:

• Content:

• Last Lecture:

• OCL Syntax

• OCL Semantics

• Last Lecture:

• Last Lecture:

• Last Lecture:

• How do Basic Object System Signatures relate to UML class diagrams?

• Please un-abbreviate all abbreviations in this OCL expression.

• Please explain this OCL constraint.

• Please formalise this constraint in OCL.

• Last Lecture:

• This Lecture:

• Last Lecture:

• This Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture:

• Last Lecture: