Meta-Modeling

About the Instructor - Background

Prof. Dr. Bedir Tekinerdogan received his MSc degree (1994) and a PhD degree (2000) in Computer Science, both from the University of Twente, The Netherlands. From 2003 until 2008 he was a faculty member at University of Twente, after which he joined Bilkent University until 2015. At Bilkent he has founded and led the Bilkent Software Engineering Group which aimed to foster research and education on software engineering in Turkey. Currently he is full professor and chair of the Information Technology group at Wageningen University, The Netherlands.

He has more than 20 years of experience in software engineering research and education. He has been active in dozens of national and international research and consultancy projects with various large software companies whereby he has worked as a principal researcher and leading software/system architect.

He graduated more than 40 MSc students and supervised/graduated more than 10 PhD students. He has reviewed more than 80 national and international projects and is a regular reviewer for around 20 international journals. He has also been very active in scientific conferences and organized more than 50 conferences/workshops on important software engineering research topics.

He has developed and taught more than 15 different academic software engineering courses and has provided software engineering courses to more than 50 companies in The Netherlands, Germany and Turkey. He can communicate in five languages (English, Dutch, Turkish, French, German).

https://nl.linkedin.com/in/bedir
About the Instructor - Research

Bedir Tekinerdogan

Years of experience: >20 years
# of publications: >200
# of projects: >30
# of PhD/MSc Students: >50

Software Engineering Research Topics
Bedir Tekinerdogan

- Aspect-Oriented Software Engineering
  # of publications: >40
- Aspect-Oriented Software Architecture Design
  # of publications: >70
- Software Product Line Engineering
  # of publications: >20
- Model-Driven Software Development
  # of publications: >10
- Model-Driven System Engineering
  # of publications: >20
- Software Analysis, Verification & Validation
  # of publications: >15

>20 years of experience in software engineering
More than 200 scientific publications (h-index 21 Google Scholar)

Contents

Modeling
Meta-Modeling
Example Metamodel
Process Modeling
Summary
Modeling

Question

• What is a model?
Motivation for Model(ing)

• Visualize any human activity before it is produced (implicit or explicit)
• Provide template for guiding the production
• Documents the decisions that are made
• Communicate our ideas
• Analyze
• ...

system S

representation of

model M
Model – Software Engineering - Definitions

- A model is a description of a (part of) system written in a well-defined language.
- A well-defined language is a language with well-defined form (syntax), and meaning (semantics),
- which is suitable for automated interpretation by a computer.

Different Kind of Models

- **Prescriptive model** – model is developed as a precursor to implementing the original

- **Descriptive model** – model is derived from an existing system

Models - Multiple Views of System

System

Model

Represented by:

1. Muscular System
2. Skeletal System
3. Digestive System
4. Respiratory System
5. Circulatory System

Bedir Tekinerdogan

Meta-Modeling
Different Kind of Models

- **Sketch** – simple drawing model; not precise or complete, nor is it intended to be. The purpose of the sketch is to try out an idea. The sketch is neither maintained nor delivered.

- **Blueprint** – document/design model describing properties needed to build the real thing. In other words, the blueprint is the embodiment of a plan for construction.

- **Executable** – software model that can be compiled and executed; can be automatically translated into other models or code.

- Fowler suggests a similar distinction based on three levels of models, namely *Conceptual Models*, *Specification Models* and *Implementation Models*.

Model-Based Software Development

- Application of models to software development (model-based) is a historical tradition.
- and has become even more popular since UML.
- Model-Based Software Development (MBSD) aims to use models to develop software.
- However, models are essentially separate from the code, and
- as such we are faced with ‘mere’ documentation.
- because the relationship between model and software implementation is only intentional but not formal.

![Graph of Traditional Lifecycle](image-url)
Model-Driven Software Development

- Model-Driven Software Development does focus on models (instead of ‘dropping’ these as in the agile approach)
- In contrast to Model-Based Software Development, however,
- Model-Driven Software Development adopts models as the basic abstraction.
- Models do not constitute documentation,
- but are considered equal to code,
- as their implementation is automated

Meta-Modeling
Models: geographical maps

System represented by Models

Model - Metamodel

Model has no meaning when separated from metamodel
Metamodel - Example

Metamodel

conforms to

Model

represented by

System

Metamodel – MS-Visio

Metamodels for drawing models in MS-Visio
Meta-Modeling

Meta-Modeling is also a ubiquitous activity....
What is Metamodelling?

• Modelling describes the concepts of a domain with the concepts provided by a modelling language.

• Metamodelling allows for the **modelling of modelling languages**.
  – Metamodel: A model that defines the language for expressing a model

• Metamodelling thus allows the **definition of tailored, or domain-specific** modelling languages

Metamodel

Domain

- Metamodel describes relevant concepts of a domain
- Domain
  - Bounded field of interest or knowledge
  - consisting of concepts
- Domain may itself be composed of subdomains

Domain Categorization

- **Horizontal domains**
  - Also called technical domains
  - applied to a large group of applications that share the same technical characteristic
  - E.g GUI layout and persistence

- **Vertical domains**
  - Also called in-house, business-oriented
  - usually indicates that the concepts in the domain are not computer science concepts but instead come from the business in which computer science is applied.
  - E.g. In insurance domain concepts like “product type”, “life insurance”, “vehicle”, “liability”, etc.
Abstract Syntax

Metamodel can be expressed in different notations:

- **Abstract Syntax**
  - The concepts and definition of the language
- **Concrete Syntax**
  - Realization of Abstract Syntax
  - Textual or Visual

The abstract syntax of a language describes the **vocabulary** of concepts provided by the language and how they may be combined to create models.

- It consists of a **definition of the concepts**, 
- the **relationships** that exist between concepts
Abstract Syntax - Process

There are a number of stages involved in the development of an abstract syntax model:

• concept identification;
• concept modelling;
• model architecting;
• model validation and model testing.

Domain and Domain Analysis

• **Domain**
  – An area of knowledge or activity
  – Characterized by a set of concepts and terminology
  – Understood by practitioners in that area.

  -G. Booch, J. Rumbaugh, I. Jacobson,

• **Domain Analysis**
  – is the *systematic activity* of collecting, organizing and storing domain knowledge
Domain Analysis

- **Domain Scoping**
  - Define the domain of interest wrt stakeholders and business context.

- **Domain Modeling**
  - Providing a domain model by data collection, data analysis, classification and evaluation.

Domain, Knowledge Source, Concept

```plaintext
<<domain>>
Domain

<<domain>>
Knowledge Source

<<domain>>
Concept

PCS System
```
Commonality Analysis

Example: Domain Lexicon

- Monitor,
  - The entity that monitors the driver and the engine performance
- Sensor
  - Entity that observes the controlled entity.
- Control Data
  - Data which represents the goal parameters
- Feedback
  - Reaction given to the driver by the monitor
- Display
  - Physical entity to represent the feedback of the monitor
Imagine a business modelling language suitable for modelling high level business rules about business data.

- An appropriate language for this domain might provide modelling concepts, such as:
  - "data model", "data entity", and "business rule".
- In addition there will be relationships between these concepts:
  - a "data model" may be composed of a number of "data entities".
- There will also be rules describing the valid models (static semantics) that may be constructed in the language, for instance,
  - "a datamodel cannot contain data entities with the same name"

It is important to emphasize that a language’s abstract syntax is independent of its concrete syntax and semantics.

- Abstract syntax deals solely with the form and structure of concepts in a language
- without any consideration given to their presentation (concrete syntax)
- or meaning (semantics)
Static Semantics

• Abstract Syntax is further constrained using well-formedness rules
• that state how the concepts may be legally combined (i.e. syntactically valid)
• Additional constraints on models that should conform to metamodel

Concrete Syntax

Metamodel can be expressed in different notations:

• Abstract Syntax
  – The concepts and definition of the language
• Concrete Syntax
  – Realization of Abstract Syntax
  – Textual or Visual
Concrete Syntax - Textual

- A textual syntax enables models or programs to be described in a structured textual form.
- A textual syntax typically consists of a mixture of declarations, which declare specific objects and variables to be available, and expressions, which state properties relating to the declared objects and variables.
- Ability to capture complex expressions.
- Can become difficult to comprehend.

```
public abstract class Thing
{
private String nameOfThing;
public String getName()
{return nameOfThings;}
}
```

Concrete Syntax - Visual

- Presents a model or program in a diagrammatical form.
- Consists of a number of graphical icons that represent views on an underlying model.
- Ability to express large amounts of detail in an intuitive and understandable form.
- Only certain levels of detail can be expressed beyond which it becomes overly complex and incomprehensible.
Formal Model

- Specified based on Concrete Syntax
- Sentence formulated in DSL
- ‘The program’

Semantics

- An abstract syntax conveys little information about what the concepts in a language actually mean.
- Therefore, additional information is needed in order to capture the semantics of a language.
- Defining a semantics for a language is important in order to be clear about what the language represents and means.
- Otherwise, assumptions may be made about the language that lead to its incorrect use.
Semantics

• Semantics must be either well-documented or be intuitively clear to the modeler. This is made easier by adopting concepts from the problem space so that a domain expert will recognize the ‘domain language’

• Different approaches for describing semantics including
  – Translational
  – Operational
  – Denotational

Abstract Syntax, Concrete Syntax and Semantics
Meta-Metamodel

- Metamodel describes concepts that can be used for modeling the model (i.e. instances of the metamodel)

- Consequently, the metamodel itself has a *meta metamodel*

Program, Grammar, EBNF

- Program conforms to Grammar
- Grammar conforms to EBNF
- EBNF conforms to Meta-Metamodel
Example OMG - MOF

- MOF stands for Meta Object Facility
  - enables meta-metamodeling of UML level metamodels

- MOF was first described in 1998

- MOF – a language used to define metamodels
  - Metamodels define language/constructs to build models
    - Relational for information sources
    - BPEL, BPM for business process
    - XML Schema for XML documents
    - UML for modeling applications

- MOF is now an international standard: ISO/IEC 19502:2005 Information technology -- Meta Object Facility (MOF)

How does the MOF look like?

M0: Instances
M1: Model
M2: Metamodel
M3: Meta-Metamodel
Run-time objects
Class model
UML
MOF

Stahl & Voelter, 2006

Modeling Constructs

In order to describe a particular kind of model we have to describe the set of modeling constructs that make up models of that kind:

• For relational data modeling
  – table, column, key, and so on.

• For workflow
  – activity, performer, transition, split, join, and so on.

• For UML class modeling
  – class, attribute, operation, association, and so on.

• For defining CORBA interfaces
  – interface, valuetype, and so on.

→ MOF is universal way of describing modeling constructs.

Metamodules defined using MOF
## Cardinality of Models....

<table>
<thead>
<tr>
<th>Level</th>
<th>MOF terms</th>
<th>Examples</th>
<th>Cardinality *</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>meta-meta-model</td>
<td>The “MOF” meta model</td>
<td>1</td>
<td>Standards developer</td>
</tr>
<tr>
<td>M2</td>
<td>metamodel, meta-meta-data</td>
<td>UML metamodel, CWM metamodel</td>
<td>10 ... 100</td>
<td>Developer for tools or infrastructures</td>
</tr>
<tr>
<td>M1</td>
<td>model, metadata</td>
<td>UML models such as class diagrams for a specific application, CWM metadata</td>
<td>1,000 ... 10,000</td>
<td>Application developer</td>
</tr>
<tr>
<td>M0</td>
<td>object, information</td>
<td>Instances of modeled class etc., data</td>
<td>100,000 ... 10,000,000</td>
<td>End user</td>
</tr>
</tbody>
</table>

## Summary

```
Summary

```

```
```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```
Summary

- **Domain** describes a bounded field of interest; Domain can be decomposed into sub-domains
- **Metamodel** defines a model for a modeling language. Meta-model describes relevant concepts of domain
- Metamodel consists of Abstract Syntax and Concrete Syntax
- **Abstract Syntax** consists of a definition of the concepts, the relationships that exist between concepts
- and defines the well-formedness rules that state how the concepts may be legally combined (**Static Semantics**)
- **Concrete Syntax** is the realization of the abstract syntax, and describes how the concepts are represented (textual, visual)
- **Semantics** defines the meaning of concepts in Abstract Syntax
- **Formal Model** is the program defined using Concrete Syntax

Meta-Modeling Goals
Question

• Why do we need metamodels in software development?

Metamodelling – Domain-Specific Modeling

- **Metamodel**: A model that defines the language for expressing a model

  • Metamodelling can focus on a particular domain and represent the model for the language in that specific domain

  • a domain-specific language (DSL) is a language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique.

  • DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes.

  • DSLs enhance productivity, reliability, maintainability, and portability.
Metamodelling – Validation

• **Model Validation**
  – models are validated against the constraints defined in the metamodel

• Validation ensures that the model meets its intended requirements

• The ultimate goal of model *validation is to make* the model useful in the sense that the model addresses the right problem, provides accurate information about the system being modeled, and to makes the model actually used

Metamodelling – Tool Integration

• **Tool adaptation and integration**
  – based on the metamodel, modeling tools can be adapted to the respective domain
Metamodelling – Interoperability

- Current systems are large scale and complex; not monolithic but consist of different modules
- Usually depending on different technologies
- How to achieve interoperability?

Interoperability - the ability of two or more systems or components to exchange information and to use the information that has been exchanged – IEEE

Metamodelling – Transformations

- Developer develops model(s) based on certain metamodel(s).
- Using code generation templates, the model is transformed to executable code.
- Optionally, the generated code is merged with manually written code.
- One or more model-to-model transformation steps may precede code generation.
### Metamodelling - Goals

- **Domain-specific modelling**
  - the metamodel describes abstract syntax of a DSL (see later)

- **Model Validation**
  - models are validated against the constraints defined in the metamodel

- **Tool adaptation and integration**
  - based on the metamodel, modeling tools can be adapted to the respective domain

- **Enable development and interoperability of model driven systems**

- **Model Transformations**
  - model transformations are defined as mapping rules between two metamodels

- **Code Generation**
  - Generation templates refer to the metamodel of the DSL

---

### Example Meta-Models
Question

• Assume that we are developing a metamodel for **StateMachine**
  
• Which concepts will the abstract syntax typically include?
  
• What is a typical concrete syntax for statemachines?
  
• Why would you need such a metamodel?

Description of the Domain

• **StateMachine**s have an initial state and an optional final state
• **StateMachine**s provide guards, actions and events.
• Guards are owned by transitions, and are boolean expressions that must evaluate to true before a transition can be invoked.
• Events are associated with transitions via an event.
• Events have a name and some optional parameters. It is the receipt of an event that causes a transition to be triggered.
• Events can be generated by **StateMachine** actions
Concepts from the Domain

- **State**: A named representation of the state of an object at a particular point in time.
- **Initial State**: The initial state that the object is in when it is created.
- **Final State**: The final state of the object - typically when the object has been destroyed.
- **Transition**: A state change. A transition has a source and target state.
- **Event**: A named event that causes a transition to occur.
- **Action**: An executable expression that may be attached to a transition or a state. Actions are invoked whenever the transition occurs, or a state is entered.
- **Guard**: A boolean expression which must be evaluated to be true before a transition can occur.

Abstract Syntax

- Is each concept a distinct abstract syntax concept?
- States and Transitions are clearly core to StateMachines, as they capture the central concepts of state and state change.
- However, initial states and final states could be argued not to be concepts in their own right.
- An initial state can be modelled as an attribute of the StateMachine, whilst a final state can be viewed as a property of the instance.
Concrete Syntax

How to represent each concept in the abstract syntax?

- How to represent State?
- How to represent Initial State?
- How to represent Final State?
- How to represent Transition?
- How to represent Action?
- How to represent Event?
- How to represent Guard?

Concrete Syntax - Graphical
Static Semantics - Well-Formedness Rules

- Once the concepts and relationship in the StateMachine language have been identified, well-formedness rules can be defined.
- Examples (OCL):
  - all states should have unique names

```
contextStateMachine
@Constraint StatesHaveUniqueNames
states->forall(s1 | states->forall(s2 | s1.name = s2.name implies s1 = s2))
```

- the initial state of the StateMachine must be one of the StateMachine’s states

```
contextStateMachine
@Constraint StatesIncludeInitialState
states->exists(s | s.name = initialState)
```

Semantics - Example

- Although we may have an intuitive understanding of what is meant by a state machine,
- it is likely that the detailed semantics of the language will be open to misinterpretation if they are not defined precisely. Consider, for example, the following questions:
  - What exactly is a state?
  - What does it mean for transition to occur?
  - What happens if two transitions leave the same state.
  - Which will be chosen?

- All these questions should be captured by the semantics of the language.

See for example (defining 16 different semantics):
Example – Music Metamodel

• Assume the following Case:

Music theory is a field of study that investigates the nature or mechanics of music. Every piece of music ....

• How to define the abstract syntax?
• How to define concrete syntax?
• How to define the static semantics? What are the well-formedness rules?
• Semantics?

Music Theory

• Music theory is the field of study that deals with how music works. It examines the language and notation of music.
• It identifies patterns that govern composers' techniques.
• In a global sense, music theory distills and analyzes the parameters or elements of music – rhythm, harmony (harmonic function), melody, structure, form, and texture.
• ...

http://en.wikipedia.org/wiki/Music_theory
Metamodel for Music

Music notation

Metamodel

instance of

Model

Music sheet

Music Metamodel – Abstract Syntax

Elements of music
• Melody
• Pitch
• Scales and modes
• Rhythm
• Harmony
• Consonance and dissonance
• Dynamics
• Texture
• Form or structure
Music – Concrete Syntax

Music in XML

<part id="P1">
<measure number="1" width="179">
<divisions>24</divisions>
<attributes>
=key>
<quint=3</quint>
<mode>major</mode>
</key>
<time>
<beats>2</beats>
<beat-type>4</beat-type>
</time>
<clef>
<sign>G</sign>
<line>2</line>
</clef>
</attributes>

Metamodel Elements

- **Domain** describes bounded field of interest; Domain can be decomposed into sub-domains
- **Metamodel** defines a model for a modeling language. Meta-model describes relevant concepts of domain
- Metamodel consists of Abstract Syntax and Concrete Syntax
- **Abstract Syntax** consists of a definition of the concepts, the relationships that exist between concepts
  - and defines the well-formedness rules that state how the concepts may be legally combined (Static Semantics)
- **Concrete Syntax** is the realization of the abstract syntax, and describes how the concepts are represented (textual, visual)
- **Semantics** defines the meaning of concepts in Abstract Syntax
- **Formal Model** is the program defined using Concrete Syntax
Example Standard Metamodels

- UML2 Metamodel
- CWM Metamodel
- Java Metamodel
- AspectJ Metamodel
- ...

UML Metamodel – Abstract Syntax
Organization of UML2

- In contrast to UML 1.*, UML 2.* includes not only one but four specification documents.
- The UML is structured using a metamodeling approach with four layers.
  - Infrastructure
  - Superstructure
  - Extensions (profiles)
- The Superstructure is structured into a tree of packages in turn.
  - (e.g. Actions, Activities, Common Behaviors, Classes, ...)

Model and Metamodel - Example
Metamodel – Abstract Syntax

- UML2 Metamodel for Use Cases

Model and Metamodel - Example

UML Meta-Model of Use-Cases

UML Model: Use-Case Diagram
Software Process Metamodeling

Software Process Engineering Model (SPEM)

- Standardized way of expressing any software development process
- Vendor, framework, methodology neutral
- The SPEM is a metamodel for defining processes and their components. A tool based on SPEM would be a tool for process authoring and customizing.
- Defines the minimal set of process modeling elements necessary to describe any software development process,
- without adding specific models or constraints for any specific area or discipline, such as project management or analysis.

SPEM Metamodel - Concepts

- Basic Elements (External Description, Guidance)
- Dependencies (Categorizes, Impacts, Import, Precedes, RefersTo, Trace)
- Process Structure (WorkProduct, WorkDefinition, Activity, Step, ProcessRole)
- Process Components (Package, ProcessComponent, Process, Discipline)
- Process Lifecycle (Lifecycle, Phase, Iteration, Precondition, Goal)
Process Metamodel – Abstract Syntax

Process Metamodel – Concrete Syntax

Activity (kind: Phase): Preliminary Analysis
Process: Information System Delivery

Activity (kind: Iteration): First Joint Requirements Planning (JRP) Workshop

TaskUse: Define Owner Requirements:
ProcessPerformer (kind: primary)
RoleUse: System Architect (kind: m)
WorkDefinitionParameter (kind: m)
WorkProductUse: EnterpriseArchitecture
WorkDefinitionParameter (kind: m)
WorkProductUse: Assessment of Current System (state: initial draft)
Steps
Step: Define objectives based on stated needs
Step: Define the key issues
Step: Determine the relevant enterprise principles

TaskUse: Draft Owner Models:
ProcessPerformer (kind: primary)
RoleUse: System Architect
WorkDefinitionParameter (kind: m)
WorkProductUse: Assessment of Current System (state: initial draft)
WorkProductUse: Owner Requirements (state: initial draft)
WorkDefinitionParameter (kind: m)
WorkProductUse: Business Structure (state: initial draft)
Steps
Common Warehouse Metamodel (CWM)

- OMG’s data warehouse standard
- A **data warehouse** is a repository of an organization's electronically stored data.
- Designed to facilitate reporting and analysis
- Enable easy interchange of
  - warehouse and business intelligence metadata between
  - warehouse tools
  - warehouse platforms and metadata repositories in distributed heterogenous environments.

- Defined using MOF
Common Warehouse Metamodel (CWM)

- Comprised of a number of constituent metamodels representing:
  - Data resources
  - Analysis
  - Warehouse management
  - Foundational components of a typical data warehousing/business intelligence environment

- Drawn using UML profile for MOF

Example: The Relational Metamodel
Summary

- Metamodel
- Model vs. Metamodel
- Motivation for Metamodel
- Software Language Engineering
- Abstract Syntax vs. Concrete Syntax
- Static Semantics
- Semantics
- Formal Model
- Standard Metamodels