Overview of workshop

- Model driven software engineering in general
- Grammars, signatures and meta-models
- Model transformations and code generation
- DSL Design
Model transformations

- Models increase the level of abstraction
  - used for both hardware and software design
  - often manually translated into design documents and code
  - no guarantee for consistency between model, design and resulting code

- Models increase the quality of the code
  - via model checking

- Models increase the efficiency of software development
  - code generation
Model transformations

• Mappings are defined on meta-models
  • The mapping is defined with meta-models; one meta-level higher than the input and output models of the transformation

• Transformations are defined on models
  • A transformation implements a mapping
Model transformations

• How are the meta-models of Xtext mapped to meta-models describing the abstract syntax?

• General architecture:
  • Parse/construct an input model (Xtext/GMF)
    – Map concrete syntax model to abstract syntax model
  • Apply transformation from input to output model
  • Unparse the output model
Model transformations

- **Mapping and transformation**

  ![Diagram of model transformations]

  - **Source Meta-model**
  - **Target Meta-model**
  - **Input Model**
  - **Output Model**

  Mapping and transformation are based on the relationship between the source and target meta-models, transforming the input model into the output model.
Model transformations

• A model transformation takes input and
  • changes the input, or
  • produces output
  • transforms according to a predefined mapping
  • is used in “run time”

• Two main categories of transformations
  • Vertical or horizontal transformations
  • Exogenous or endogenous transformations
Model transformations

• Vertical transformations
  • Source model is at a different level of abstraction than the target model
  • Examples of vertical transformation
    – refinement (specialization)
    – abstraction (generalization)
Model transformations

• Horizontal transformations
  • Source model has the same level of abstraction as target model
    − not to be confused with “meta-levels”

• Examples of horizontal transformation
  − refactoring
  − merging
Model transformations

• **Endogenous transformations:**
  • between same meta-models

• **Exogenous transformations:**
  • between different meta-models
# Model transformations

## Taxonomy

<table>
<thead>
<tr>
<th></th>
<th>Horizontal</th>
<th>Vertical</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Endogenous</strong></td>
<td><em>Refactoring</em></td>
<td><em>Refinement</em></td>
</tr>
<tr>
<td><strong>Exogenous</strong></td>
<td><em>Migration</em></td>
<td><em>Code generation</em></td>
</tr>
</tbody>
</table>

**Further distinction: syntactic vs. semantic transformation**
Model transformations

- Model transformation formalisms
  - ATL
  - Xtend
  - Xtext
  - Xpand
  - QVT Relations
  - QVT Operations
  - QVT Core
  - ASF+SDF
  - Stratego/XT
  - VIATRA
  - Tefkat
  - ETL (Epsilon)
  - GrGen
  - ...

- Platform: Eclipse and EMF
Model transformations

- Model transformation ontology

```
Model Transformation

Monolithic Transformation
- Complete, disjoint
  - Viatra
  - GReAT
  - OptimalJ

Compositional Transformation
- Complete, disjoint
  - JET
  - AndroMDA

Vertical Transformation
- Complete, disjoint
  - QVT
  - ATL

Horizontal Transformation

Endogenous Transformation
- Complete, disjoint
  - MMa=MMb

Exogenous Transformation
- Complete, disjoint
  - MMa!=MMb

Declarative Transformation

Imperative Transformation

General purpose programming language
```
Model transformations

• **Applications:**
  • Elaboration: generating detailed models or code from less detailed models.
  • Synchronization: ensuring consistency between models at the same or different levels of abstraction.
  • View creation: producing query-based views.
  • Model evolution (including refactoring)
  • Abstraction: generating less detailed models from more detailed ones.
Model transformations

• are not:
  • … necessarily semantics preserving.
    – They can be, but there are useful transformations that are “lossy”.

  • … necessarily refinements.
    – They can be (especially update-in-place transformations) but many useful ones are not.

  • … necessarily specified in a way that allows interesting properties to be checked of them.
    – Sometimes they must be transformed!
Model transformations

Approach

- Meta-metamodel
- Transformation Language
- Transformation Definition
- Target Metamodell
- Target Model
- Transformation
- Source Model
- Source Metamodell

Input

Output

Conforms to

Instance of
Model transformations

• Tasks of model management:
  • Validation, transformation and generation constitutes tasks that can be applied to models:
    - Transforming models
    - Generating text from models
    - Refactoring models
    - Merging models
    - Validating models
    - Comparing models
    - Migrating models as a metamodel changes
    - Querying and modifying models
    - Workflows
Model transformations

• How can we support these model management tasks?
  • Obviously we do not want to transform, validate and generate HTML manually!
  • It’s error prone, time consuming, expensive, and pointless – these are automatable tasks.
Model transformations

- So how can we automate these tasks (transform, generate, ...)?
  - We could write stand-alone tools in Java.
    - Not great, because we would have to largely reimplement our tool for each different problem/project!
  - We could encode everything in XML and use XQuery etc.
    - Not great because this exposes implementation details!
  - We could use existing languages for MDE.
    - E.g., OCL for validation, ATL for transformation, Xpand for generation.
    - Not great, there are problems with this.
Model transformations

- **Languages for MDE**
  - Inconsistent syntaxes
    - Different dialects of OCL
    - Different ways to perform model navigation/modification
    - End up writing the same code in many languages
  - Poor integration and interoperation
    - E.g. validation -> transformation -> generation
  - Recurrence of bugs / missing features
Model transformations

- Epsilon has been developed to support:
  - validation
  - transformation
  - generation

- Epsilon is a family of integrated (domain specific) languages for managing models
Model transformations

- **Features of Epsilon**
  - Languages for a range of model management tasks
  - Languages have consistent syntaxes
  - Can manage models from different meta-models / modeling technologies
  - Can call methods of Java objects
  - Strong integration with EMF and GMF
  - Eclipse-based development tools
    - Editors, Launching facilities
Model transformations

- **Epsilon Object Language (EOL)**

```java
-- In this script, we query Ecore to find out:
-- ... how many classes it has
EClass.all.size().println('All classes: ');

-- ... how many abstract classes it has
EClass.all.select(c|c.abstract)
  .size().println('Abstract classes: ');

-- ... the names of its classes and
-- features each one has
'Class names: '.println();
for (c in EClass.all) {
    var toPrint = ' ' + c.name;
    toPrint = toPrint + '->' +
    c.eStructuralFeatures.size.
    toPrint.println();
}
```
Model transformations

- Epsilon Generation Language

Overview of our world

Our world has 4 people:

- George
- Ian
- Bob
- Alan

and we know the following facts:

- George and Ian are friends
- Ian and Bob are enemies
- Bob and Alan are friends
- Alan and George are enemies
Model transformations

- Epsilon Transformation Language
Model transformations

- Epsilon Validation Language
Model transformations

• Basic scenario:
  • A model (in one language) is transformed into a model in a (possibly) different language.
  • Multiple inputs, multiple outputs also possible.

• A typical MDE workflow:
  • Construct an abstract model.
  • Successively transform it until a sufficiently detailed model is produced.
  • Generate code from the detailed model.
Model transformations

• What do model management operations have in common?
  • i.e., transformation, query, merge, generate code, validate, etc.

• After some reflection:
  1. They all require the ability to **navigate** models (e.g., to go from class to class, node to node, line of code to line of code).
  2. Many require the ability to **modify** models.
Model transformations

• EOL: Overview
  • Dynamically and strongly typed
  • Object-oriented
  • Modular
  • Primitives, collections and model elements are objects
Model transformations

- Navigation is something that the OMG’s Object Constraint Language (OCL) is really good at.
  - It is abstract and declarative.
  - It allows very concise navigation expressions to be written.
  - e.g., `self.processor_rack.process`
- But it’s restricted to the OMG’s standards.
- It also does not allow model modification.
Model transformations

- **EOL**
  - Borrows navigation expressions (and some basic operations) from OCL.
  - Borrows conceptually from Javascript.
  - Adds assignment statements, sequencing, and multiple model access.
  - … plus other features.
Model transformations

• EOL: types
  • Four primitive types:
    – String, Integer, Real and Boolean
  • Four collection types:
    – Bag, Sequence, Set and OrderedSet

• Universal type: Any
  – isDefined(), isUndefined()
  – isOfType(type:Type),
    isKindOf(type:Type)
Model transformations

- **EOL: operations**
  - Collection types provide (side-effect free) higher-order operations
    - `select()`, `reject()`
    - `collect()`, `exists()`

- Ability to define custom operations
  - Can have context, i.e. called using dot notation
  - Optional return type
EOL: other useful functionality

- User input: `System.getUser()`
  - `.inform()`, `.choose()`, `.chooseMany()`, `.confirm()`, `.prompt()`, `.promptInteger()`, `.promptReal()`

- Platform independent.
Model transformation

```javascript
var model : UML!Model;
model = new UML!Model;
model.name = 'TestModel';
var i : Integer;
for (i in Sequence{1..3}){
    var class : new UML!Class;
    class.name = 'TestClass' + i;
    class.visibility = UML!VisibilityKind#vk_public;
    class.namespace = model;
}
```

Define a model variable
Instantiate the model variable
Assign a name to the model
Define an iterator
for i = 1 to 3
Define and instantiate a class variable
Assign a name to the class
Set the class visibility to public
Set the namespace of the class to be the model
Model transformation

- Access to multiple models
  - Many model management tasks, such as transformation or comparison, require simultaneous access to multiple models.
  - To support this, EOL employs the `<Model Name>!<Meta-class>` syntax.

```java
for (class in UML!Class.allInstances()){
    if (DBMS!Table.allInstances().exists(table|table.name == class.name)){
        ('Found matching table for class ' + class.name).println();
    }
    else {
        ('Not found matching table for class ' + class.name).println();
    }
}
```
Model transformations

- **EOL summary**
  - Effectively, all model management can be done in EOL.
    - (Or Java... but...)
  - Operational model transformations can (and have) been written in EOL.
  - But EOL does not possess task-specific constructs for transformations.
    - There are repeated patterns that arise with any transformation.
Model transformations

- ETL: overview
  - Model-to-model transformation language
- Hybrid language (declarative and imperative parts)
- Arbitrary number of source/target models
- Traceability
Model transformations

• ETL: overview
  • Rule-based
    – Optional guards
    – Reuse via rule extension
    – Abstract, primary, lazy annotations
    – Can be interactive

• Pre and post blocks
Model transformations

- Concrete syntax of an ETL rule

```
(@abstract)?
(@lazy)?
(@primary)?
rule <name>
  transform <sourceParameterName>:<sourceParameterType>
  to (<rightParameterName>:<rightParameterType>
      (, <rightParameterName>:<rightParameterType>)*
      (extends (<ruleName>,)*<ruleName>))*
  (guard (:expression)|{{statement+}})?

  statement+
```

/ Faculteit Wiskunde en Informatica
Model transformations

- Concrete syntax of pre and post blocks
  - executed before and after ETL transformation rules

```
(pre|post) <name> { 
  statement+
}
```
Model transformations

• Example Tree2Graph

• Tree.emf:

```java
package Tree;

class Tree {
    val Tree[*] children;
    ref Tree parent;
    attr String label;
}
```
Model transformations

• Example Tree2Graph

• Graph.emf:

```kotlin
package Graph;

class Graph{
    val Node[*] Nodes;
}
class Node {
    attr String name;
    val Edge[*] outgoing;
    ref Edge[*] incoming;
}
class Edge {
    ref Node source;
    ref Node target;
}
```
Model transformations

- **Example Tree2Graph**
- **Tree2Graph.etl:**

```plaintext
rule Tree2Node
  transform t : Tree!Tree
to n : Graph!Node {

  n.name = t.label;

  // If t is not the top tree
  // create an edge connecting n
  // with the Node created from t's parent
  if (t.parent.isDefined) {
    var e : new Graph!Edge;
    e.source ::= t.parent;
    e.target = n;
  }
}
```
Model transformations

• ETL: overview
  • Execution
    – Pre blocks
    – Non-abstract, non-lazy (applicable) rules
    – Post blocks

• `.equivalents()` and `.equivalent()`
  – Resolves source elements to their target counterparts
  – Invokes both lazy and non-lazy rules
  – Shorthand : :=
Model transformations

- See Epsilon book for more details
What is a model transformation

A model transformation is a function from set $A$ to set $B$:

$A2B: A \rightarrow B$

Meta-model defines a set of models

Model transformation is a function from set $A$ to set $B$
Constituents of a model transformation

\[ A \rightarrow B = (A \downarrow 1 \rightarrow B \downarrow 1 ) \times (A \downarrow 2 \rightarrow B \downarrow 2 ) \ldots (A \downarrow n \rightarrow B \downarrow n ) \]
How to compose a model transformation
How to compose a model transformation

Decomposition along the tree-structure of the input meta-model

\[ A2B = (A\downarrow 1 \ 2B\downarrow 1) \times (A\downarrow 2 \ 2B\downarrow 2) \times \ldots \times (A\downarrow n \ 2B\downarrow n) \]
How to compose a model transformation

A chain of transformations

A

B

C

D
A2B(m) = D2B \circ C2D \circ A2C(m)

A2B: A \rightarrow B
A2C: A \rightarrow C
C2D: C \rightarrow D
D2B: D \rightarrow B
Chain of transformations

• With intermediate metamodels

Metamodel A  \rightarrow\text{MT}\rightarrow Metamodel C  \rightarrow\text{MT}\rightarrow Metamodel D  \rightarrow\text{MT}\rightarrow Metamodel B

C = A’

• Without intermediate metamodels

Metamodel A  \rightarrow\text{MT}\rightarrow Metamodel A’  \rightarrow\text{MT}\rightarrow Metamodel B’  \rightarrow\text{MT}\rightarrow Metamodel B

D = B’
Chain of transformations without intermediate metamodels

\[ A_2 B(m) = D_2 B \circ C_2 D \circ A_2 C(m) \]
Chain of transformations (without intermediate metamodels)

- Import of other transformation or library
- Intermediate classes
- Intermediate properties
- Predefined types from standard library:
  - Set, Dict, Sequence, OrderedSet
Two dimensions of composing transformations

\[ A_2B(m) = D_2B \circ C_2D \circ A_2C(m) \]

Transformation along the structure of the input metamodel

\[ A_2B(m) = A \downarrow 1 \ 2B \downarrow 1 (m \downarrow 1) \times A \downarrow 2 \ 2B \downarrow 2 (m \downarrow 2) \ldots \times A \downarrow n \ 2B \downarrow n (m \downarrow n) \]
How to compose a model transformation

- Choose a structure along which to compose the transformation
  - Use one of your models, usually the most ‘dense’ one (with the highest granularity of objects in it)
- Identify the delta(s) between this structure and the input / output models
  - For each delta you need to create a transformation
- Split a transformation into a chain of transformations
  - What is the flow of the information between your models
- Repeat these steps till the transformations are simple enough
What is a trace?

```plaintext
model2RDBModel (SELF: Model, RESULT: Model)
  in self : Model = Model
  +  Model model
  out result : Model = Model
     Model model
package2schema (SELF: Package, RESULT: Schema)
  in self : Package = Package
     Package class
  out result : Schema = Schema
     Schema class
persistentClass2table (SELF: Class, RESULT: Table)
  in self : Class = Class
     out result : Table = Table
primitiveAttribute2column (SELF: Property, PARAMS: Class, RESULT: TableColumn)
  in self : Property = Property
     Property callId
        name : EString
           callId
        stereotype : EString
           taggedValue : TaggedValue
           type : Type
           owner : DataType
```

<table>
<thead>
<tr>
<th>mapping</th>
<th>param1</th>
<th>param2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapping</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mapping</td>
<td>param1</td>
<td>param2</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Technische Universität Eindhoven
University of Technology
Model resolving
mapping Project :: Project2Model() : Model

mapping MMA::Automaton :: Automaton2Automaton() : MMB::Automaton

mapping Location :: Location2Mode() : Mode

mapping Edge :: Edge2Transition() : Transition
Where to find more information

  - Use more mappings than helpers
  - Specify pre- and post-conditions
  - Use less overloading
Conclusions

• Domain specific for model transformations

\[ A2B: A \rightarrow B \]

• Traceability and resolving models

• Not mature
  • Tools, documentation, pragmatics