Introduction to MDE and Model Transformation

DTU Course 02291 – System Integration
1. Model Driven Engineering (MDE)
(including slides by M. Brambilla, J. Cabot, M. Wimmer)
Models in Software Engineering

- **Model-Driven Development (MDD):** a development paradigm that uses models as the primary artifact of the development process.
- **Model-driven Architecture (MDA):** the particular vision of MDD proposed by the Object Management Group (OMG).
- **Model-Driven Engineering (MDE):** a superset of MDD, going beyond pure development (e.g. automatic documentation generation).
- **Model-Based Engineering (MBE):** any development process using models, but not as the main process driver.
MDD (partially) automates software development.

Main principle of MDD: by iteratively refining an initial high-level model, it is possible to create a model of sufficient technical detail to allow accurate code generation.

Main challenge of MDD: generating not only code structure (i.e. packages, classes), but also behavior.

Model transformations automate the refinement process.
Motivation for adopting MDE

The adoption of MDE as a software development paradigm can bring important benefits to many types of development projects and to many development-related activities.

Some examples:

1. **Code generation (MDD)**
   - Generating an application’s actual implementation from models.

2. **System interoperability**
   - Using models to align the data formats exchanged by communicating systems.

3. **Legacy systems re-implementation**
   - Many development projects have the goal of re-implementing obsolete IT systems, which often need to be reverse-engineered.
   - The reverse engineering process can be facilitated by using models as an intermediate step, followed by updated code generation.
Systems interoperability

Legend:

- **System A**
  - Conformance relationship (C2)
  - Model-to-Model Transformation (Single or composite)
  - Projection (Model-to-Text or Text-to-Model)
Legacy systems re-implementation

**Legacy artifacts:**
- source code
- configuration files
- tests
- databases
- etc.

- Models provide a homogenous representation of all legacy components.
- Model transformations are used to reverse-engineer models from legacy artifacts, as well as to generate new artifacts.
MDE adoption in practice

- MDE has high initial costs:
  - Deploying the technical infrastructure (including modeling tools, model manipulation and transformation tools, code generation tools)
  - Training software engineers accustomed to other development paradigms
  - Creating or adapting Domain Specific Modeling Languages (DSMLs)

- After the initial adoption effort, MDE delivers (among others):
  - Increased development productivity and efficiency,
  - Stronger correctness guarantees,
  - Faster updates and bug fixes.

- As a result, MDE is mostly popular with large companies facing strict regulatory compliance requirements.
  - Defense industry, avionics, automotive industry
MDE adoption in practice

- **Technology Trigger**
  - R&D

- **Peak of Inflated Expectations**
  - Startup companies first round of venture capital funding
  - First-generation products, high price, lots of customization needed

- **Trough of Disillusionment**
  - Mass media hype begins
  - Early adopters investigate

- **Slope of Enlightenment**
  - Supplier consolidation and failures
  - Second/third rounds of venture capital funding
  - Second-generation products, some services

- **Plateau of Productivity**
  - High-growth adoption phase starts: 20% to 30% of the potential audience has adopted the innovation
  - Third-generation products, out of the box, product suites

- **Climbing the Slope**
  - Negative press begins

- **Entering the Plateau**
  - Methodologies and best practices developing
2. Model Driven Architecture (MDA)
(including slides by M. Brambilla, J. Cabot, M. Wimmer)
Model Driven Architecture

- A standard by the Object Management Group (OMG) prescribing how to apply MDE practices to software development.

Principles of MDA:

1. Models are expressed using well-defined notations.
2. System specifications are organized around a set of models and associated transformations.
3. Models conform to metamodels.
4. Competition among tools implementing the MDA standard will increase the acceptance and adoption of MDE.
Modeling levels in MDA

CIM (Computation - Independent Model)
- Business view of the solution
- Context and requirements, independent from how they are implemented

PIM (Platform - Independent Model)
- Description of the system
- Information and algorithms, independent from the implementation technology

PSM (Platform - Specific Model)
- Technology-aware detailed specification of the system
- Aware of the technical implementation platform

Mappings
MDA example: CIM fragment

1. New customer arrives to counter
2. Check customer identity
3. Retrieve account number
4. Ask customer about operation to perform
5. Execute operation on account
MDA example: PIM fragment

```
«business entity»
Account
- number : Integer {unique}
- balance : Float
+ getNumber() : Integer
+ setNumber(number : Integer)
...
```

--- English
Account number must be between 1000 and 9999

--- OCL
context Account inv:
number >= 1000 and number <= 9999
MDA example: PSM fragment

AccountImplBean

- number : Short {unique}
- balance : Float

+ ejbCreate(number : Short, balance : Float) : Integer throws CreateException
+ ejbPostCreate(number : Short, balance : Float)
+ ejbActivate()
+ ejbLoad()
+ ejbPassivate()
+ ejbRemove()
+ ejbStore()
+ findByNumber(Short number) : Account
  throws RemoteException, FinderException;
+ getNumber() : Integer
+ setNumber(number : Integer)
...

--- English
Account number must be between 1000 and 9999

--- OCL
class Account inv:
  number >= 1000 and number <= 9999
MDA and UML

- MDA is based on the Unified Modeling Language (UML), another OMG standard.

- MDA typically requires a Domain-Specific Modeling Language (DSML) for expressing the PSM.

Options for specifying DSMLs based on OMG standards:

1. UML Profiles: collections of Stereotypes extending the notation and semantics of standard UML elements.
2. Full DSMLs based on the Meta-Object Facility (MOF), the metamodeling language to which UML itself conforms.
3. Model Transformations
(including slides by H. Störrle, M. Brambilla, J. Cabot, M. Wimmer)
Definition

- A model transformation is an operation which:
  - takes as input one or more source models and a transformation definition,
  - produces target models, or other artifacts (e.g. code, documentation, deployment scripts) based on these inputs.

- Model transformations can be classified along several dimensions:
  - Horizontal vs. Vertical
  - Endogenous vs. Exogenous
  - Model-to-Text vs. Model-to-Model
Model transformation (MT)

- Source model
  - Reads
  - Conforms to
  - Refers to

- Transformation engine
  - Executes
  - Refers to

- Target model
  - Writes
  - Conforms to

- Source metamodel
  - Refers to

- Target metamodel
  - Conforms to

- Transformation definition
  - Refers to

Model transformations and MDA
Despite the different purpose, the same notation is used at each level.

If you look closely, you can spot the differences.
**CM to RDBM translation**

- **1. Package-to-schema**
  - Every package in the class model is mapped to a schema with the same name.

- **2. Class-to-table**
  - Every persistent class is mapped to a table with the same name.
  - Furthermore, the table should have a primary-key column with the type `NUMBER` and the name being the class name with `_tid` appended.

- **3. Attribute-to-column**
  - The class attributes have to be appropriately mapped to columns, and some columns may need to be related to other tables by foreign key definitions.
Code generation

- **Code generation is a Model-to-Text (M2T) transformation.**
  - Other M2T transformations produce artifacts such as documentation, test cases, and model serialization formats (XMI).

- **Code generation must produce human-readable code expressed in various languages such as Java or SQL.**
  - The human-readable condition is important, since the generated code will almost always require editing and additions by a developer.

- **Generated code does not live in isolation.**
  - It must usually be integrated with existing frameworks and APIs to be useful.
  - Providing this integration automatically is feasible for code generated from Domain Specific Modeling Languages (DSML).
  - Conversely, code generated from general purpose modeling languages such as UML is mostly boiler-plate.
Overview of generation techniques

Most M2T transformation approaches are based on templates.

- Templates are widely used in many areas of software development (e.g. text processing, Web development).
- As an example, consider the following e-mail template:

<table>
<thead>
<tr>
<th>Template text</th>
<th>E-mail text</th>
</tr>
</thead>
</table>
| Dear «firstName» «lastName»,
Thank you for your interest in ... |
| Dear John Doe,
Thank you for your interest in ... |

Components of a template-based M2T transformation approach:

- **Templates**
  - Consist of text fragments with embedded meta-markers.

- **Meta-Markers**
  - Have to be interpreted and evaluated based on the model.
  - Use declarative query languages (e.g. OCL, VMQL, XPath) to extract model information.

- **Template Engine**
  - Replaces meta-markers with model data at runtime and produces output files.
Template-based code generation

- **Source Model**
  - Person
  - Customer
  - ...

- **Template Engine**
  - **Input**
    - Query
    - **Result**
  - **Produced Text**
    - public class Person { String id, ... }
    - public class Customer { String id, ... }
    - ...

- **Text fragment**
  - «context class»
  - public class «name» { String id, ... }
4. Model Transformation Languages
(including slides by H. Störrle)
## Model Transformation Paradigms

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Description</th>
<th>Strengths, Weaknesses, Examples</th>
</tr>
</thead>
</table>
| **Operational**     | Imperative language to create (sets of) model elements, based either on a meta-modeling formalism extended with facilities for expressing computations, or based on a query language (e.g. OCL) with imperative constructs. | PRO: Concrete, may be efficient  
CON: Low-level, very verbose  
EX: QVT Operational mappings, Kermeta, MTL, XMF-Mosaic’s executable MOF, C-SAW, etc. |
| **Relational / Declarative** | Mathematical relations between source/target element types using constraints, executable semantics e.g. in PROLOG  
Relational approaches are side-effect-free, support multidirectional rules, can provide backtracking | PRO: Higher-level  
CON: Difficult to execute  
EX: QVT Relations, VMTL, MTF, Kent Model Transformation Language, Tefkat, AMW, mappings in XMF-Mosaic, etc. |
| **Graph-transformation** | Graph transformations, are operates on typed, attributed, labeled graphs, consisting of a pair of patterns  
The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place | PRO: Well-understood formal semantics  
CON: Not efficient, complex rules / hard to debug  
EX: AGG, AToM3, VIATRA, GReAT, UMLX, BOTL, MOLA, Henshin etc. |
| **Hybrid**          |                                                                                                                                                                                                                                    | PRO: Pragmatic approach  
CON: No (simple) semantics, not necessarily better  
EX: ATLAS, Epsilon, TRL, XDE, ... |
Epsilon Transformation Language (ETL)

- Part of the Epsilon family of model management languages
  - Based on the common foundation of the Epsilon Object Language (EOL)
  - The Epsilon family includes languages for model validation, code generation, model comparison, model migration, and model merging.

- ETL is a hybrid textual model transformation approach.
  - It combines an imperative programming language having a Java-inspired syntax with declarative OCL-like constructs.

- ETL is based on the Eclipse Modeling Framework (EMF).
  - It is implemented as an Eclipse plug-in containing an editor and interpreter.
  - Apart from Ecore, EMF, and GMF models, it can operate on many model formats, including XML and Excel.
ETL abstract syntax
ETL concrete syntax

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

(@abstract)?
(@lazy)?
(@primary)?

rule <name>

transform <sourceParameterName>:<sourceParameterType>

to <rightParameterName>:<rightParameterType>

(, <rightParameterName>:<rightParameterType>)*

(exextends <ruleName>(, <ruleName>))*? { 

(guard (:expression)|({statementBlock}))? 

statement+

}
ETL example: tree to graph

- Transform a tree model to a graph model.
- The Tree and Graph meta-models are shown below.
ETL example: tree to graph

- Graph nodes maintain the labels of corresponding tree nodes.
- Graph edges are created between every tree node and its parent.

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

    n.label = t.label;

    if (t.parent.isDefined()) {
        var edge = new Graph!Edge;
        edge.source = n;
        edge.target = t.parent.equivalent();
    }
}
```
Henshin: a graph-based MTL

- Also based on the Eclipse Modeling Framework (EMF).

- A transformation consists of one or more rules. Each rule is expressed as an object diagram reflecting the abstract syntax of the host modeling language.
  - E.g.: When transforming a UML model, nodes included in rules instantiate elements of the UML metamodel.

- Model elements included in rules have one of these stereotypes:
  - <<create>> – the element will be created in the target model;
  - <<delete>> – the element must exist in the source model and will be deleted in the target model;
  - <<forbid>> – the element must not exist in the source model;
  - <<preserve>> – the element must exist in the source model and will be preserved in the target model.
Control flow: Units

- In Henshin, control flow is specified using units.

- Units can be arbitrarily nested.
  - A rule is also a unit.

- The following unit types are available:
  - **Sequential units**: sub-units are executed in the given order;
  - **Priority units**: the sub-unit with highest priority is executed;
  - **Independent units**: one sub-unit is randomly selected for execution;
  - **Loop units**: the only sub-unit is executed as often as possible;
  - **Iterated units**: the only sub-unit is executed for a fixed number of times;
  - **Conditional units**: have either two or three sub-units: if, then, (and else). If a match for the if unit can be found, the then unit is executed. Otherwise, if present, the else unit is executed.
The Henshin metamodel
Henshin example: bank accounts

- An *endogenous* transformation operating on instances of the *Bank Metamodel* shown below.
Example rule: create a bank account
Example rule: transfer money
The Visual Model Transformation Language (VMTL)

- A declarative MTL allowing users to express model-to-model transformations using their existing model editors.

- Transformations consist of one or more rules, each containing:
  - A Find Pattern describing where the rule can be applied;
  - A Produce Pattern describing the effects of the rule;
  - The Find Pattern and the Produce Pattern can be merged, forming an Update Pattern;
  - Optionally, Require Patterns and Forbid Patterns describe positive and negative rule application conditions, respectively.

- VMTL patterns are model fragments enhanced by textual annotations.
  - The annotations are typically expressed as comments.
The VMTL metamodel
Elements of the VMTL metamodel are mapped to elements of the host modeling language metamodel. A different mapping exists for each VMTL-compatible host language.

There are four elements of the VMTL metamodel that must be mapped to elements of the host language: *transformations*, *rules*, *patterns*, and *annotations*. 
VMTL example: process refactoring

- In a Business Process Model and Notation (BPMN) Process Diagram, the execution order of Tasks is determined by directed Sequence Flows.

- Although legal, the absence of outgoing Sequence Flows from a Task may be considered a design anti-pattern, since the execution of this task will implicitly lead to the termination of the Process.

- Process termination can be made explicit by applying a refactoring which ensures that an End Event is executed after each Task with no outgoing Sequence Flows.
Source model
VMTL example: process refactoring

**Update Pattern**
- $task$
- [VM Annotation]
  - create

**Forbid Pattern**
- $task$
- [VM Annotation]
  - type := FlowNode