

Model Driven Engineering

1 Introduction	1
1.1 Why model driven engineering	1
1.2 Model Driven Architecture	2
1.3 Model	2
1.4 Metamodels	3
1.5 Model Driven Engineering	3
2 Models and Metamodels	3
2.1 General Characteristics of Models	3
2.2 Metamodel	4
2.3 Definition of a Metamodel	4
2.4 Metalevels	5
2.5 Meta Object Facility	5
3 Object Constraint Language	5
3.1 Why OCL	5
3.2 Applications	5
3.3 OCL Metamodel	6
3.4 Iterations on collections	6
3.5 Predefined Operations	6
4 DSLs and XTEXT	6
4.1 Domain-Specific Modelling Language (DSML)	7
4.2 Language Metamodel	7
4.3 XText	8
5 Model Transformations / Atlas Transformation Language	9
5.1 Model Transformations	9
5.2 Approaches for writing transformation definitions.	9
5.3 Features of Model Transformation Languages	9
5.4 Atlas Transformation Language	10

6 QVT Transformation language & Java Transformations	11
6.1 QVT Terminology	11
6.2 QVT Operational Context	11
6.3 Original QVT requirements	11
6.4 QVT Architecture	12
6.5 Operational Mappings Language	12
6.6 Mapping operation	12
6.7 Helpers	13
6.8 Resolution of object references	13
6.9 Putting it together	13
6.10 Other facilities	14
6.11 Tool support	14
6.12 Take-home messages (QVT)	14
6.13 Java Transformations	14
7 Model-to-Text Transformations / Acceleo	15
7.1 Compiling models in MDE	15
7.2 Code generation	15
7.3 Acceleo	15
7.4 Example: Simple Activity Language	16
7.5 Acceleo module	16
7.6 Polymorphic queries	16
7.7 Take-home messages (M2T)	17

1 Introduction

1.1 Why model driven engineering

In the 90's → software was becoming expensive to build, even harder to maintain, failed to meet planned targets. How do you cope with different business requirements, changing personnel and changing technologies? High impact on software systems. There is a need to port software to new technologies. Business logic and functionality are too intertwined.

Since requirements and technologies change, you can decouple them (using technology agnostic models). Models are stable assets (business models) and then you use a strategy to generate the technological aspects (the implementation).

1.2 Model Driven Architecture

Model engineers would define the models, and developers would ignore them and implement the functionality however they wanted (model burning party). There was no way to generate the implementation from the model.

The OMG¹ proposed model driven architecture. They eventually even supported UML (from version 1.3).

DEF: Model Driven Architecture

An approach to IT system specification that separates the specification of functionality from the specification of the implementation of that functionality on a specific **platform**.

Separate functionality from implementation.

Initially java code was generated from class diagrams, it was seen as a sophisticated way to use UML. But it's also a well founded model support, and it increases the **level of abstraction**.

The idea of MDA was to have models that went beyond the implementation. Models are specified at a higher abstraction level than implementation logic. The implementation(s) are then obtained from models via model transformations. This way, when a new technology is adopted, you just re-implement the transformation. This works well if there are different instances of a system being generated.

¹Object Management Group

1.3 Model

DEF: Model

A model is a formal specification of the function, structure and/or behavior of an application or system.

A model is information selectively representing some aspect of a system based on a specific set of concerns (current OMG definition)

A model is an abstraction (also called representation or denotation) of an object system (also called system under study) expressed in some language. An interpretation of a model gives the meaning of the model relative to the object system.

Typical MDA-Based development

1. Computation independent business model: concerns the business aspects independently of automated support.
2. Platform independent model: concerns the application independently of the specific platform
3. Platform specific model: concerns the application and is targeted to a specific platform
4. Programming language specific code: code written in a specific programming language.

DEF: Platform

A set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem that depends on the platform can use without concern for the details of how the functionality provided by the platform is implemented' (old definition)

Trough a platform independent models, you have application architectural stability.

1.4 Metamodels

Four layer meta-modelling architecture:

M0: System (real world)→M1: model (UML model)→M2: Model of a model (UML metamodel)→M3: Model of a model of a model (MOF)

(You can use UML to specify UML itself)

DEF: Model Transformation

The process of converting one model (source model) to another model (target model) of the same system (OMG, MDA guide).

For example, UML to Java.

Still, MDA was not a methodology. It was a collection of viewpoints. Any practical use of MDA requires the adoption of a development process.

- MDD is a development paradigm with models as primary artifact
- MDA is an MDD approach based on OMG standards
- MDE goes beyond development, including other engineering tasks
- MBE is more relaxed with respect to the role of models (not necessarily primary artifacts)
- MDSE is about MDE for Software Systems

1.5 Model Driven Engineering

MDE is an engineering approach in which models are the primary artifacts. Beyond development, involving engineering tasks like model and system evolution, reverse engineering,...

MDE should be used to

- raise the level of abstraction via modelling
 - helps manage complexity
- model languages intended for the domain experts, not for developers
 - better communication between problem domain experts solution domain experts.

- Rigorous techniques for reasoning on models
 - Increase system quality
- Better automation via model transformations
 - Improve the productivity and time to market.
- Mature engineering disciplines employ strong modelling techniques

Modern MDE applications are:

- Digital Threads: collection of system models related to each other.
- Low Code Programming is supported under the hood by DSLs and code generation.

2 Models and Metamodels

DEF: Model

A model is purposely abstracted, clear, precise and unambiguous **conception**.

A model **denotation** is a precise and unambiguous representation of a model, in some appropriate formal or semi-formal language.

A model is a **representation of a concept**. The representation is purposeful: the model purpose is used to abstract from the reality the irrelevant details.

2.1 General Characteristics of Models

- They represent something in the real world (some system)
- Simplification (abstraction)
- Conception **and** concrete representation
- Purpose: often descriptive, prescriptive or predictive
- Desired qualities: precise, unambiguous, allows analysis

A model requires a part of the real world that is modelled (a system being modelled, **modelOf** relation). A model can be seen as a role or may be the subject of modelling (model of a model).

Abstraction is a powerful tool for mastering complexity. In models, some of the characteristics are ignored (abstracted from)

2.1.1 Nature of the model relation.

-Denotation: some of the properties of the system are represented or denoted in the model.

- Demonstration: knowledge is obtained from the model in the terms of the model elements.
- Interpretation: the obtained knowledge is translated in terms of the system.

2.1.2 Working definition of model

A model:

- represents a part of the reality called the object system;
- is expressed in a modelling language;
- provides knowledge for a certain purpose.

This knowledge can be interpreted in terms of the object system

2.1.3 Purpose of models

1. Sketches: used for communication, often partial/incomplete views of the object system.
2. Blueprints: used to provide detailed and complete specification as pre-description of what should be built.
3. Programs: used to develop the system, as opposed to code

2.1.4 Software Systems as Models

A running software system can be considered a model of a system (in OOP). However, it may interact and change the real world (they **control** it).

2.1.5 Software Artifacts as Models

The original OMG definition is too general

2.2 Metamodel

DEF: Metamodel

A metamodel is a model of a model.

Metamodel as a model of a modelling language. A meta model is a model of models expressed in a given modelling language.

In MDE, the view that a metamodel represents a modelling language is widely accepted. We assume that a metamodel is a model of a modelling language.

2.3 Definition of a Metamodel

Language is a set of sentences (models). A metamodel models the valid members of the set. A metamodel constrains the valid models expressible in a given modelling language.

Language has concrete syntax, abstract syntax and semantics. A metamodel should focus on the concepts that can be expressed with the language and their relationships. It very roughly corresponds to the language's abstract syntax².

A metamodel is a model of a model. Different characteristics of a model can in principle be chosen to be modelled in a metamodel. What are the concepts that can be used to define the model, and how are these related between each other.

In MDE, meta-modelling considers the types of the model elements and consequently the abstract syntax of the modelling language as the characteristic represented in the metamodel!

A model is an instance of a meta-model, because it can only use elements defined by the meta model?

²Syntax elements without considering how they are represented

There are different (language-specific) `instanceOf` relations. It allows to interpret a metamodel formally (systematically).

2.4 Metalevels

Hierarchy of metamodels, can be infinite ($\mathcal{L}$). In practice, we stop at level 3. There are two ways to stop the recursive tower.

1. Assume that some language is just given (i.e. XML)
2. Model the top language in itself (most popular)³.

Real world → Model → Metamodel → Metametamodel

2.5 Meta Object Facility

Metadata management framework, data about data. The idea was to have a language to define new languages. UML metamodel is defined via MOF. MOF 2.0 is a subset of UML 2.0. The idea was to use UML tools to express a metamodel.

Ecore is used in Eclipse, and is compatible with MOF 1.x, and allows EMOF metamodels to be imported via XML. Ecore is a metamodeling language. You can specify classes of objects, its attributes, relationships and operations, simple constraints (essentially a subset of the UML class diagram).

3 Object Constraint Language

Used to define DSLs.

3.1 Why OCL

Many modelling languages require an extra notation to express all the constraints over its sentences (models). For example in XML.

Graphical specification languages such as UML often allow users to describe partial aspects of a system. OCL used to be part of UML but they are detached now.

³For example, you use UML to specify UML.

Formal languages/mathematical interpretations are preferred (set theory over UML objects or semantics)⁴.

The original purpose of OCL was to provide formal, precise and unambiguous specification of constraints over models.

It was also meant to be usable by a large number of users.

OCL is a declarative language, not a programming language (its not executable): it defines constraints, conditions.

An OCL constraint limits the set of valid models.

- It also has no side effects in the functional style. The evaluation of OCL expression returns a value, while the model remains unchanged.
- OCL is also not a programming language, there is no control flow. It only has queries.
- OCL is a typed language. Each OCL expression has a type, and OCL includes a set of predefined types.
- The evaluation of OCL expressions is instant. The states of objects cannot change during execution.

3.2 Applications

Original application was to specify constraints for model elements in UML and MOF models (Invariants, Pre and post conditions, initial or derived values).

It can also be a query language to obtain values from models.

Navigation language in model transformation languages like Xpath for XML documents.

```
context Employee inv: self.age >= 18
```

⁴I.e. no “Please no underaged employees”.

inv invariant constraint must be true for all instances of the constrained type.

pre and **post** conditions must be true before and after the execution of an operation⁵.

3.3 OCL Metamodel

A metamodel defines the abstract syntax of a language.

OCL has primitives, collections, let-in expression allows to define a variable to be used in one constraint. If-then-else (implication)

```
context Employee inv:
let annualIncome : Integer = wage * 12 in
  if self.isUnemployed then
    annualIncome < 5000
  else
    annualIncome >= 5000
  endif
```

Accessing objects and properties: `self.age`, `self.getWage()`. Enumerations are accessed with `::`.

Navigation: going from one class to another through a relation. If the navigation is ≥ 1 , the association ends are an ordered set.

`->size()` means applying a function/transformation to a collection.

3.4 Iterations on collections

- **select** and **reject** create a subset of a collection based on a boolean condition.
- **collect** specifies a collection that is derived from some other collection. Generates a bag of **A** where **A** is the type of the attribute that you are collecting by.

- **forall** specifies a boolean condition that must hold for all objects in a collection. **forall** operations can be nested⁶.
- **exists** returns **true** if the expression is true for at least one element of the collection.

3.5 Predefined Operations

- **oclIsTypeOf**(*t*: OclType) : Boolean. Result is **true** if *t* is either the direct type or one of the supertypes of the object.
- **oclAsType**(*t*: OclType) : T similar to casting in programming language.
- **allInstances()** get the collection of all instances of that type that exist at the time when the expression is evaluated.

4 DSLs and XTEXT

We want to automate some problem domain with a computer system. But there is an abstraction gap between the computation domain and the problem domain.

To reduce the abstraction gap you can make a computational model closer to the problem domain. To increase the implementation gap you can distance yourself from the executable domain process (but is familiar for domain experts).

There is a tradeoff: you offer something to the domain experts, but increase the effort needed to translate the executable domain.

UML was an effort to have higher level abstraction.

What is the purpose of these languages?

Java is a general-purpose programming language (GPL)

UML is a general-purpose modelling language (GPML)

- More abstract than Java

⁵This refers to the real code, not the OCL model

⁶`forall(e | e.age >= 18)` can be considered a lambda in java like `forall(e->e.age >=18)`, where the type of *e* is inferred from the collection.

- Generally not executable
- Allows modelling in any domain
- Still does not talk in a domain language

DSLs are highly expressive for a specific domain. It is familiar for the domain experts, there is no (or a small) abstraction gap.

Often, domains have intersections, or interact at some point.

How do you execute a model? Depends on the type of model, usually it is executed by a model interpreter. A model is a sentence in a modelling language.

Compiling a model means transforming it (in MDE), for example model-to-text.

The details of model-to-text and model-to-model transformation make it possible to provide the information necessary to perform code execution.

4.1 Domain-Specific Modelling Language (DSML)

Domain-Specific Language (DSL)

DSL components (like in any language)

- Abstract syntax (metamodel)
- Several concrete syntaxes (typically textual or diagrammatical)
- Diagrammatical = Visual, Graphical

Semantics: All components of a DSL are models!

4.2 Language Metamodel

The language metamodel plays a central role: a DSL is a coordinated set of models built around it.

What we know so far:

- Models are abstractions
- Models are symbolic entities expressed in a language
- The metamodel is the core of the language definition

- It defines the abstract syntax (the main concepts)
- It does **not** define the concrete representation
- Models are depicted with symbols: text, diagrams, audio, etc.

4.2.1 Concrete Syntax

Graphical concrete syntax: takes the form of diagrams. Eclipse offers frameworks for defining the syntax and generating graphical syntax editors: GMF (Graphical Modeling Framework, deprecated), Graphiti and Sirius.

Textual concrete syntax: defines how a model is textually serialised. The modeller defines the syntax, and a parser and an editor are generated. Examples: EMFText (University of Dresden) and XTEXT.

Concrete Syntax as Model (in EMFText)

Concrete Syntax is a template that maps the metamodel classes to textual syntax (.cs file). The metamodel is an Ecore metamodel, possibly defined with the EMFText Ecore editor.

4.2.2 Semantics

Semantics is a crucial component for any language:

- It allows building interpreters/compiler
- It allows reasoning and proving properties on models
- It allows simulating and debugging models

Semantics definition remains an interesting topic in MDE!

4.2.3 Take-home messages (DSLs)

- Raising the level of abstraction helps developers and domain experts, but increases the implementation gap
- Model transformations can be used to bridge this gap
- A DSL is a powerful tool, but it is limited to a certain domain
- Language infrastructure is important!
- Multiple DSLs may be needed to describe a system

- Risk: language cacophony, “Babel tower” of languages

4.3 XText

XText is a set of DSLs and modern APIs to describe the different aspects of a language. It is useful to define DSLs. It allows language designers to define the concrete syntax of their languages in the way they prefer.

It's the most popular environment for DSL development.

It can generate a parser, a type-safe abstract syntax tree (AST), a serializer and code formatter. It provides scoping, linking, compiler checks and static analysis (validation), and code generator or interpreter.

It also provides a runtime architecture and a full-blown Eclipse IDE specifically tailored for the language being designed.

You can start development from the grammar (originally designed for this), but also from a metamodel.

It is composed of a grammar language and a Modeling Workflow Language 2 (MWL 2).

Workflow (default) — the one used in the tutorials

1. Create Xtext project (comes with a default grammar and a default MWE2 workflow)
2. Get default grammar
3. Edit default grammar
4. Run MWE 2 workflow
5. Get language artifacts: parser, validator, editor, etc. **and an Ecore metamodel!**

Workflow with Ecore model

1. Ecore project must be converted to Xtext project (right-click → Configure → Convert to Xtext project)
2. Ecore model, genmodel and model code
3. Generated grammar

4. Adjust generated grammar (optional)
5. Language grammar
6. Run MWE 2 Workflow
7. Get language artifacts: parser, validator, editor, etc.

4.3.1 Concrete syntax of an Ecore metamodel

Procedure to define the concrete syntax of an Ecore metamodel (example: an `Sql` metamodel):

1. Convert the EMF project to an Xtext project (with Configure...) and generate the genmodel
2. Run the wizard to create an Xtext project from the metamodel, choosing a class as root (e.g. `Sql`)
 - It generates a default grammar (`.xtext` file) and a default workflow (`.mwe2` file)
 - In the `.xtext` file, all concrete metaclasses have a textual representation, and abstract classes (inheritance) are used as choices

4.3.2 Grammar elements

In the default grammar:

- A rule creates an object of the corresponding metaclass (e.g. the `Sql` rule creates an `Sql` object)
- Keywords are the literal strings that appear in the text
- Alternative (0 or 1): optional element
- Multiple (0 or *): repeated element
- Choice: one among several alternatives (used for abstract classes)
- Attribute value: the text is assigned to an attribute of the created object
- Reference: a link to another element, e.g. a `Type`, via its name

Modifying the grammar: after modifying the grammar, run the default workflow (`.mwe2` file) as an MWE2 Workflow to generate the editor.

More information on the language: [Xtext Grammar Language documentation](#).

Running the editor: run the Xtext project as an Eclipse Application, create a project and inside this project create a file with the language extension (e.g. `.sql`).

4.3.3 Take-home messages (Xtext)

- Xtext allows fast definition of concrete syntax
- Xtext generates the required tools
- Xtext can start both from a grammar or from a metamodel
 - Metamodel or syntax (or both) reengineering may be needed
- EMFText is a more “metamodel-based” alternative, but it is not properly supported any more

5 Model Transformations / Atlas Transformation Language

5.1 Model Transformations

DEF: Model Transformation

Produce different models or artifacts from a model based on a transformation pattern. — MDA Guide

We use a more concrete and general definition

DEF: Model Transformation

Process through which target models are automatically generated from source models, according to a transformation definition expressed in a model transformation language.

Multiple target models don't have to be of the same system. (main system + testing system).

Source models → Transformation → Target model

Transformation definition is based on the metamodel of the target and source. Target and source metamodel are instances of the metamodel.

5.2 Approaches for writing transformation definitions.

1. Through a GPL
2. In a domain specific transformation language

The OMG approach was to have a domain specific transformation language (ATL & QVT).

5.3 Features of Model Transformation Languages

- Programming Style:
 - Declarative specification: specify the relations between models elements without a specific execution order
 - Imperative: specifies an explicit sequence of steps that produce the result.
 - Hybrid: a mix of declarative and imperative constructs.
- Transformation direction can be mono or bidirectional
- Input and Output Cardinalities: 1-1, 1-N, N-1, M-N

Each transformation refines the input model (applies patterns, adds platform specific details). These are vertical transformations: each step moves to a lower level of abstraction (details are added).

Model composition is used to combine different models.

How is the transformation chain defined?

Both source and target models may evolve, changes can propagate in both directions.

In refactoring the models can be instance of the same metamodel.

The MDE platform provides a metamodeling language and at least one transformation language.

EMF has software for developing open source transformation languages (model to model, m2m), and model to text transformation (m2t).

5.4 Atlas Transformation Language

For the project we can choose between ATL and QVT (preferred).

ATL Characteristics

Distinct source and target models source models are read only (can only be navigated), and target models are write only (cannot be navigated).

Hybrid language declarative & imperative.

Declarative you look for patterns in the source metamodels, recognize them and then know how to generate the corresponding pattern in the target metamodel.

Imperative called rules and action blocks.

ACL data types are similar to OCL but are not exactly the same.

Recommended Style: declarative

A declarative rule specifies a source pattern to be matched in the source models and a target pattern to be created in the target models for each match during rule application.

An imperative rule is a procedure called by name, possibly with arguments. It consists of a declarative target pattern and/or an action block (a sequence of statements).

5.4.1 Transformation Rules

Rule `Class2Table`

- Table is created from each class;
- Column corresponding to the key of the table is created
- Columns of the table correspond to the single-valued attributes of the class

- Rule `SingleValuedAttribute2Column`: Column is created from each single-valued attribute
- ...

5.4.2 Declarative Rules

Source pattern consists of

- Labeled set of types from the source metamodels
- Guard (boolean expr) used to filter matches

Match holds for tuple of elements from the source model that matches the types specified in the source pattern (one element of each type).

Attributes are resolved into columns by the automatic traceability support!

`objectId` needs a type and this is defined based on `Integer`, assuming this type is defined in the source model. Officially an attribute helper that defines a constant value (is computed only once) with default context (the whole transformation).

5.4.3 Types of ATL rules

Matched rules applied once for each match. A given set of elements may only be matched by one matched rule

Lazy rules applied as many times for each match as it is referred to from other rules (possibly never for some matches)

Unique lazy rules applied at most once for each match (return always the same target element) and only if it is referred to from other rules

Called rules do not have a matching (from clause) and must be called in order to be executed.

Unique lazy rules only execute once.

5.4.4 Execution order of declarative rules

- Declarative ATL frees the developer from specifying execution order.
- Order in which rules are matched and applied is not specified (non-deterministic)
- Match of (unique) lazy rule must be referred to before rule is applied
- Order in which bindings are applied is not specified (non-deterministic)
- Execution of a rule cannot change source models → cannot change a match
- Target elements are not navigable → execution of a binding cannot change the value of another

Declarative ATL frees the developer from specifying execution order, the execution is non-deterministic, you won't know the order

Called/imperative rules have the same structure as declarative rules, but without a from clause and possibly with an imperative code section (do)

Rule inheritance: helps structure transformation and reuse rules. Child rule matches a subset of what its parent rule matches. Child rule specializes target elements of its parent rule.

6 QVT Transformation language & Java Transformations

QVT (Query/View/Transformation) is the OMG standard language for model transformations.

6.1 QVT Terminology

DEF: Query

Expression that is evaluated over a model. The result of a query is one or more model elements, which are instances of types defined in the source model, or defined by the query language.

DEF: View

Model completely derived from another model (the base model). A live connection exists between the view and the base model: when the base model changes, the view changes with it.

DEF: Transformation

Process of automatic generation of a target model from a source model, according to a transformation definition.

6.2 QVT Operational Context

The source metamodel, the target metamodel and the QVT language itself are all defined with MOF. A transformation definition is written in QVT and refers to the elements of the source and target metamodels. The QVT engine executes it: it reads the source model (instance of the source metamodel) and produces the target model (instance of the target metamodel).

Source model → QVT engine (runs the transformation) → Target model

So: the abstract syntax of QVT is a MOF 2.0 metamodel, transformations are defined on MOF 2.0 metamodels, and they are executed on instances of those metamodels.

6.3 Original QVT requirements

Mandatory

- Query language (→ OCL)
- Transformation language
- Abstract syntax, based on MOF 2.0
- Paradigm → declarative

Optional

- Bidirectionality
- Traceability

- Reusability
- Model update

6.4 QVT Architecture

QVT is a layered architecture with three transformation languages:

Core (declarative) small, low-level language to define relations between source and target models. It has the same expressive power as Relations, but transformations are much more verbose and traceability links must be handled manually. It is mainly used as a reference to define the semantics of Relations.

Relations (declarative) describes more complex relations between model elements, with object patterns that are matched in one model and created in another. Traceability links are handled automatically and transformations can be multidirectional. It supports check-only mode (just verify that two models are related, without creating anything), uni- and multidirectional transformations, and incremental (in-place) updates of existing models.

Operational Mappings (imperative) extends Relations with imperative constructs.

Relations is translated into Core (by the RelationsToCore transformation). Operational Mappings and **Black Box** (a mechanism to call external programs, e.g. code written in another language) extend both of them.

The 3 QVT languages collectively provide one **hybrid** language.

Core and Relations are not relevant for the course, the focus is on Operational Mappings.

6.5 Operational Mappings Language

Explained with the **flattening UML class hierarchies** example: given a UML model, produce another UML model where only the leaf classes

(classes not extended by other classes) are kept, and each of them contains everything it inherited.

Rules:

1. Copy the primitive types
2. Copy only the leaf classes of the source model
3. Include the inherited attributes and associations
4. Attributes with the same name override the inherited attributes

Source and target use the same metamodel, **SimpleUML**: a simplified UML class diagram with packages, classes (with their attributes), primitive types, generalizations (each one points to the superclass, called **general**) and associations (from a source class to a target class).

Example: **Person** is extended by **Student** and **Employee**, which are extended by **PhDStudent** and **Professor**. After flattening, only **PhDStudent**, **Professor** and the other leaf classes (**Course**, **Address**, **Car**, ...) remain. **PhDStudent** now directly has **name**, **ssn** and **school**, and the inherited associations (**attends**, **residesAt**, **supervisor**). **Professor** keeps its own **name : FullName**, which overrides the inherited **name : String**.

6.5.1 Structure of a transformation

A QVTo file declares:

- the metamodel it uses (**modeltype**, by its URI);
- the signature of the transformation: its input and output models;
- the entry point **main()**. In the example, it takes the root **Model** of the input and applies the top-level mapping to it;
- then helpers and mapping operations.

6.6 Mapping operation

A mapping operation maps one or more source elements onto one or more target elements. It's always unidirectional, and selects source elements based on their type and a boolean condition (the guard, **when**). It executes operations in its body to create the target elements, and it may invoke other

mapping operations (and be invoked). Mapping operations may be related by inheritance.

A mapping is defined on a type: the source element it is applied to is available as `self`. It can have parameters (`in`, `out`, `inout`) and returns one or more results, the created target elements. Its body has three sections:

- `init`: executed before the result elements are created;
- `population`: the result elements are created automatically and then filled in. It is the default section: what you write directly in the body goes here;
- `end`: executed before exiting the operation.

6.6.1 Transforming leaf classes

The mapping for classes has a guard that only accepts classes that are not the `general` (superclass) of any generalization, i.e. the leaf classes. When the body runs, `self` is the source class and the new target class has already been created. The body just copies the name and the abstract flag⁷. The attributes are handled by a separate mapping.

6.7 Helpers

Operations attached to a type, used to perform complex navigations over the source models. They have input parameters and a body. A `query` is side-effect free, while a `helper` may modify its input parameters.

The example uses a recursive query to compute the **derived attributes** of a class (its own plus the inherited ones):

- if the class has no superclass, they are just its own attributes;
- otherwise, take the derived attributes of each superclass, remove the ones with the same name as one of the class's own attributes (this implements overriding), and add the class's own attributes.

Collecting over several superclasses gives a set of sets, so the result has to be flattened into a single set.

⁷`abstract` is a reserved word, so the property has to be escaped as `_ 'abstract'` .

6.8 Resolution of object references

Associations in the target model must connect the **new** classes, not the original ones. To find them, the transformation engine keeps a **trace**: every time a mapping runs, it records which source element produced which target element. The operation `resolveIn` (or `resolveoneIn` for a single result) looks up in the trace what a given mapping created from a given source element.

To transform associations, for each leaf class we collect its own associations plus those inherited from its superclasses (recursively, as for attributes), and copy each one. The copy keeps the name; its source becomes the new leaf class, and its target becomes the new class created from the original target. Both are found through the trace.

E.g. `residesAt` goes from `Person` to `Address`. Both `PhDStudent` and `Professor` inherit it, so the output contains two `residesAt` associations.

6.9 Putting it together

The top-level mapping (applied to the `Model`) works in three phases:

1. `init`: create the copies of the primitive types, the new classes (the leaf class mapping is applied to all classes, and its guard skips the non-leaf ones) and the associations.
2. Body: name the new model `flattened_` + original name, and add all the created elements to it.
3. `end`: assign the properties of every leaf class. This must happen last, because the type of a property must point to a class or primitive type that has already been created.

To assign the properties, a mapping finds (through the trace) the class already created for the leaf class, and gives it all its derived attributes. Since its result is set to this existing class in `init`, no new class is created. Each attribute is copied by another mapping, which copies the name and sets the type through the trace (to the copied primitive type or to the new class).

6.10 Other facilities

- Objects can also be created explicitly inside a mapping with the `object` operation, setting their properties directly (they can refer to the source object, `self`).
- Imperative constructs for control flow: `compute`, `while`, `forEach`, `break`, `continue`, `if-then-else`.
- Not covered (see the QVT specification): transformation libraries, rule inheritance and merging, disjunctions of mapping operations, constructor operations, intermediate data, reusing and extending transformations, post conditions (`where` clause).

6.11 Tool support

- Eclipse MMT (Model to Model Transformation) project: aims at a full implementation of QVT and ATL, provides the Operational Mappings engine, and brings together existing tools (Borland, Compuware, INRIA).
- Commercial tools (possibly inactive): MediniQVT (IKV++), ModelMorf (Tata Consultancy), SmartQVT (France Telecom).

In Eclipse, a transformation is run with an *Operational QVT Interpreter* run configuration: choose the `.qvto` file, the input model and the output model. Optionally, it can also generate a trace file.

6.12 Take-home messages (QVT)

- QVT is the OMG standard language for model transformations
- The requirement of **views** over models is not explicitly addressed
- The query language is based on OCL
- QVT is a family of three transformation languages:
 - Core: declarative language, simplified notation
 - Relations: declarative language
 - Operational Mappings: imperative language that extends Relations
- Collectively, the QVT languages form a hybrid language

6.13 Java Transformations

Transformations can also be written in a GPL such as Java (especially for not complex transformations). A pure Java implementation has no overhead, so in theory it should be the fastest.

6.13.1 SiTra library

SiTra (Simple Transformations in Java) is a minimal library to help write model transformations in Java (shown as an example, not as the best or only way).

- Each transformation rule implements a `Rule` interface with three methods: **check** (can this rule transform this source object?), **build** (create the target object) and **setProperties** (fill in the properties of the target object).
- A **transformer** object applies the rules: given a source object (or a list), it finds a rule that accepts it and runs it. SiTra ships with a simple transformer.

Creating and filling in the target are separate steps, so the target object already exists when its properties are set, and other rules can refer to it.

6.13.2 Example: SimpleClass to SimpleRDBMS

Classes are transformed into database tables. The source metamodel has packages containing classes (with attributes, an optional parent class and a persistence flag), primitive types and associations. The target metamodel has tables, with columns, a primary key and foreign keys that reference other tables.

They defined both metamodels, generated the genmodel and the model code for each, and implemented the transformation both with SiTra and in “pure” Java (inspired by SiTra).

In the pure Java version, each rule is a class with `transform` methods (e.g. `Class2Table` transforms a class into a table). Since Java is imperative:

- each `transform` method must be called explicitly by another one, following the structure of the transformation (package → its classes → ...);

- traces are not automatic: a `HashMap` (from class to table) is needed to remember what has already been transformed.

The goal was to compare transformation languages with pure Java: for some types of models, performance depends a lot on the tracing mechanism.

6.13.3 Take-home messages (Java)

- SiTra makes it easier to implement transformations, but here we built our own code inspired by SiTra
- This works for simple transformations, but more complex scenarios (multiple sources/targets, very different metamodels, complex mappings) require much more complex code
- With [PyEcore](#) something similar can be done in Python

7 Model-to-Text Transformations / Acceleo

Model-to-text (M2T) transformations are “the missing piece” of MDE: so far we transformed models into other models, now we turn models into text (e.g. code).

7.1 Compiling models in MDE

In MDE, compiling is a translation, so it's a **transformation**. A model can be compiled into a model that is directly executable, or into a model (program code) that can be executed. There are two ways:

- **Model-to-model** (e.g. QVT, ATL): transform the model, possibly in several steps, until we get a model that can be executed.
- **Model-to-text**: transform the model into code (e.g. Java, C#, SQL, HTML), which is then executed.

Models are abstractions, so behavioural details needed for execution may be missing. The model-to-model and model-to-text transformations must add this missing information, so that the whole process is fully automated.

Typical chain: Model → Platform model → Code → Executable

1. Model-to-model: the result is still a model, but it uses platform concepts (e.g. OOP constructs like classes and methods).
2. Model-to-text: the result is a textual model, i.e. code (e.g. Java). This is the step of this lecture.
3. Compilation, done by a normal compiler: the result can be executed (e.g. Java bytecode).

7.2 Code generation

Code generation is a model-to-text transformation. In MDE it is treated as a special type of transformation, and it is supported by specialised M2T languages (e.g. Jamda, OptimalJ, JET, AndroMDA, XPand). Here we use **Acceleo**, developed by Obeo and part of the Eclipse Model to Text (M2T) project.

7.3 Acceleo

- Tool for model-to-text transformation based on **templates**.
- Pragmatic implementation of the OMG MOF Model to Text Language (MTL) standard.
- Can generate any text, in particular programming language code.
- Relatively easy to use (much easier than e.g. XPand).
- Has the usual IDE features: content assist, outline, navigation to declarations, quick fixes, refactoring, syntax highlighting, etc.

How it works: Acceleo takes two inputs, the model (instance of an Ecore metamodel) and a template. A template is text with “gaps”: the fixed parts are copied as they are into the output, while the gaps are expressions over the metamodel elements, which are filled in with the values found in the model. The output is text (code, a report, etc.).

Model + Template → Acceleo → Text (code, report, ...)

7.4 Example: Simple Activity Language

A model represents a behaviour, like a flowchart: a start action, intermediate actions (variable declarations, assignments, condition tests) and a stop action. Each action has a label and points to the next action, while a condition points to a “yes” action and a “no” action. Assignments use integer expressions (variables, constants and sums), conditions use a “less than” expression.

The language has a textual syntax where each line is an action: its label, what it does, and the label of the next action.

Example: a model that computes the 9th Fibonacci number ($F_n = F_{n-1} + F_{n-2}$, with $F_0 = 0$, $F_1 = 1$). It declares the variables `a`, `b`, `r`, `n`, `fn`, initialises them, then loops: while `n < fn`, it sets `a = b`, `b = r`, `r = a + b` and increments `n`. When the condition fails, it goes to the stop action (at the end, `r = 21`).

7.5 Acceleo module

An Acceleo module (a `.mtl` file) declares its name and the metamodels it uses, and contains templates and queries. Inside a template, everything between square brackets is Acceleo code (expressions, loops, ...), and everything else is text copied to the output⁸.

The **main template** (marked with `@main`) is the entry point. It is applied to the `Model` element and opens a **file** block: everything generated inside it is written to an output file (here `Fibonacci.java`). Inside, it writes a Java class named after the model.

The generated class follows this idea:

- each variable declaration becomes an `int` field;
- each action becomes a method named after its label;
- “go to the next action” becomes a call to the method of the next action;
- a condition becomes an `if/else` that calls the “yes” or the “no” method;

⁸Since `[]` starts Acceleo code, literal brackets in the output (e.g. `String[]`) must be written as string expressions.

- an assignment becomes the Java assignment, followed by the call to the next method;
- the stop action becomes a method that prints the final value of every variable;
- a `main` method creates an object of the class and calls the start method.

To generate the same text for every element of a certain type (e.g. one method per assignment), the template uses a `for` loop over the model’s actions, filtered by type.

So the loop of the Fibonacci model becomes a chain of method calls (check $\rightarrow$ iterate $\rightarrow$... $\rightarrow$ check).

7.6 Polymorphic queries

The expressions in the model (e.g. `a + b`, `n < fn`) must be turned into Java text. This is done by **queries**: side-effect free operations that compute a value, here a string.

They are **polymorphic**: there are several versions with the same name for different types, and Acceleo picks the one matching the actual type of the element (like method overriding in Java):

- a variable gives its name, a constant gives its value;
- a sum gives “left + right”, calling the query again on both sides (so chained sums also work);
- a less-than gives “left < right”;
- the versions for the abstract types return an empty string, as a fallback.

E.g. the assignment `r = a + b` becomes `r = a + b`; in Java, and the condition `n < fn` becomes `if(n < fn)`.

Running it: with an Acceleo run configuration, choose the module file, the model file (`Fibonacci.xmi`) and the destination folder. Running it generates the Java file there.

7.7 Take-home messages (M2T)

- M2T transformations are often used to generate executable code from models, but they can also generate other textual representations, such as reports.
- A template-based language lets developers define code patterns for meta-model elements, and how these patterns are filled in (generated) with the elements of the models.
- Usually M2T transformations are the last step of a transformation chain (just before compilation).