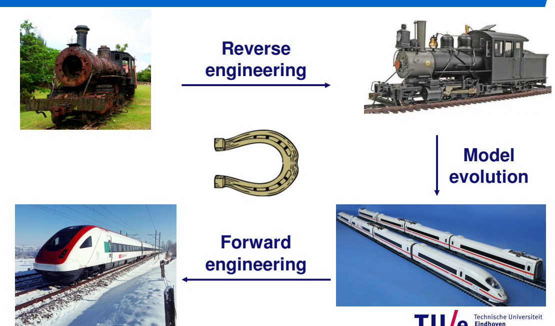
2IS55 Software Evolution

Implementing evolution: Model-Driven Engineering

Marcel van Amstel

Recap



Model-Driven Engineering

Goal:
- Raising the level of abstraction
  … from the computing domain to the problem domain

MDE combines:
- Domain-specific modeling languages
- Model transformations

[1] D. C. Schmidt: *Model-driven engineering*
IEEE Computer, vol. 39, no. 2, pp. 25 – 31, 2006.

Domain-Specific Languages

Domain-Specific Language:

" a language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain " [2]

[2] A. van Deursen, P. Klint, and J. Visser: *Domain-specific languages: An annotated bibliography*
SIGPLAN Notices, vol. 35, no. 6, pp. 26 – 36, 2000.

Domain-Specific Languages

Examples

- HTML
- SQL
- MediaWiki
- (La)TeX
- PROMELA (SPIN)
- YACC
- ATL
- ASF+SDF
- EBNF
- Flash
- …

Domain-Specific Languages

Generic	Specific
General solutions for many problems	Solutions for a smaller set of problems
Possibly suboptimal	Better solutions

Domain-Specific Languages

Pros	Cons
Expression of the solution in terms of domain concepts	Cost of DSL implementation and education
Enhanced productivity, reliability, maintainability, and portability	Difficulty of finding the right scope
Domain knowledge contained in language	Difficulty of balancing between domain-specificity and general-purpose constructs
Mostly concise and largely self-documenting	Potentially less efficient code

Domain-Specific Languages

Domain-specific language design:

1. Domain analysis
2. Language implementation
3. Use of the language

Domain-Specific Languages

" A problem domain is defined by consensus, and its essence is the shared understanding of some community "[3]

Domain Analysis:

1. Identify domain concepts
2. Define semantic notions
3. Define syntactic carrier

[3] G. Arango: *Domain analysis: – From art form to engineering discipline –* SIGSOFT Software Engineering Notes, vol. 14, no. 3, pp. 152 – 159, 1989.

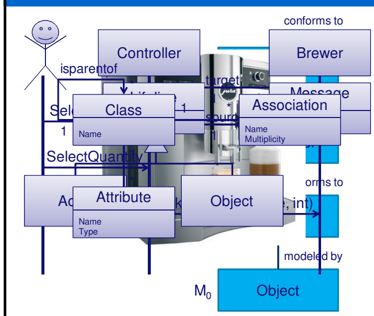
Domain-Specific Languages

Language implementation

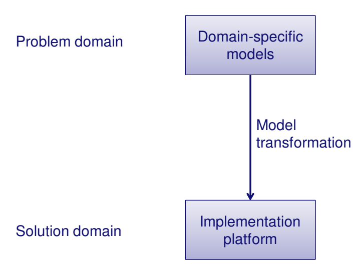
Implementation Strategies

- Stand-alone
- Embedded
- Translation

Domain-Specific Languages



Model Transformation



Model Transformation

Approaches

- Direct model manipulation
- Intermediate representation
- Transformation engine

[4] S. Sendall , W. Kozaczynski: *Model Transformation: The Heart and Soul of Model-Driven Software Development*
IEEE Software, vol . 20, no. 5, pp. 42 – 45, 2003.

Model Transformation

Types of model transformations

- Text to Model
- Model to Model
- Model to Text
- Text to Text

Model Transformation

Model Transformation formalisms

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

Model Transformation

Taxonomy

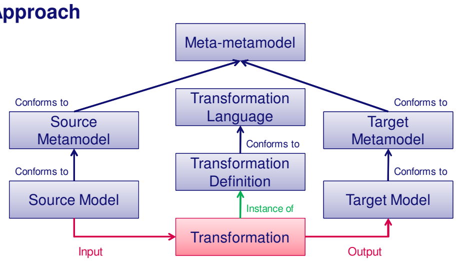
	Horizontal	Vertical
Endogenous	*Refactoring*	*Refinement*
Exogenous	*Migration*	*Code generation*

Further distinction: syntactic vs. semantic transformation

[5] T. Mens, P. van Gorp : *A Taxonomy of Model Transformation*
In Proceedings of the International Workshop on Graph and Model Transformation (GraMoT'05)
Electronic Notes in Theoretical Computer, vol. 152, pp. 125 – 142, 2006.

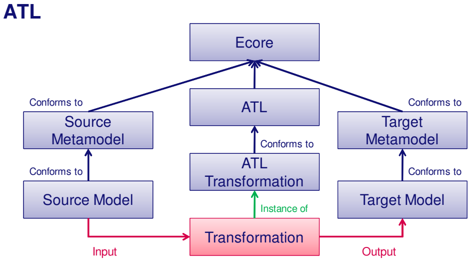
Model Transformation

Approach



Model Transformation

ATL



ATL

Assigning attributes

```
rule example1{
  from in:    MM1!MetaClassA
  to   out:   MM2!MetaClass1(
                    attr <- in.attr
              )
}
```

ATL

Assigning references

Three cases:

- Model target element generated by current rule
- Default target model element generated by another rule
- Non-default target model element generated by another rule

ATL

Model target element generated by current rule

```
rule example{
  from in:    MM1!MetaClass
  to   out1:  MM2!MetaClass1(
                    RefToMetaClass2 <- out2
              ),
      out2:   MM2!MetaClass2(
              )
}
```

ATL

Default target model element generated by another rule

```
rule example1{
  from in:    MM1!MetaClassA
  to   out:   MM2!MetaClass1(
                    RefToMetaClass2 <- in.RefToMetaClassB
              )
}

rule example2{
  from in:    MM1!MetaClassB
  to   out:   MM2!MetaClass2(
              )
}
```

ATL

Non-default target model element generated by another rule

```
rule example1{
  from in:    MM1!MetaClassA
  to   out:   MM2!MetaClass1(
                    RefToMetaClassN <-
      thisModule.resolveTemp(in.RefToMetaClassB, 'out_n')
              )
}

rule example2{
  from in:    MM1!MetaClassB
  to  out:    MM2!MetaClass1(
              ),
      out_n:  MM2!MetaClassN(
              )
}
```

ATL

Assigning enumerations

TypeEnum
- Integer = 0
- IntegerArray = 1

```
rule example{
  from in:    MM1!MetaClassB
  to   out:   MM2!MetaClass1(
                    enum <- #"Integer"
              )
}
```

- **Quotes are only needed in case of reserved words**

ATL

Rules

– Matched rules

– Lazy matched rules

– Unique lazy matched rules

– Called rules

Rules

Matched rules

```
rule rule_name{
  from in:     MM1!MetaClass(<matching condition>)
  using{<variable definitions>}
  to    out1:  MM2!MetaClass1(
               <bindings1>
               ),
        out2:  MM2!MetaClass2(
                <bindings2>
               )
  do{<imperative block>}
}
```

Matches on all model elements of type `MM1!MetaClass`

ATL

Lazy matched rules

```
lazy rule rule_name{
  from in:     MM1!MetaClass
  using{<variable definitions>}
  to    out1:  MM2!MetaClass1(
               <bindings1>
               ),
        out2:  MM2!MetaClass2(
                <bindings2>
               )
  do{<imperative block>}
}
```

- Generates new target elements for every call to the rule
- Invoked from other rules as follows:

```
thisModule.rule_name(<model element of type MM1!MetaClass>)
```

ATL

Unique lazy matched rules

```
unique lazy rule rule_name{
  from in:     MM1!MetaClass
  using{<variable definitions>}
  to    out1:  MM2!MetaClass1(
               <bindings1>
               ),
        out2:  MM2!MetaClass2(
                <bindings2>
               )
  do{<imperative block>}
}
```

- Always returns the same target elements for a given source element, i.e., target elements are generated only once per source element

ATL

Called rules

```
rule rule_name(<parameters>){

  using{<variable definitions>}
  to    out1:  MM2!MetaClass1(
               <bindings1>
               ),
        out2:  MM2!MetaClass2(
                <bindings2>
               )
  do{<imperative block>}
}
```

- For generating target elements from imperative code

ATL

Helpers

- Helper with context

```
helper context MM!MetaClass def: helper_name(<parameters>): return_type =
  let <variable definition>
  in  <expression>;
```

  – Invocation:

```
<model element of type MM!MetaClass>.helper_name(<parameters>)
```

  – The context should never be of a collection type

- Helper without context

```
helper def: helper_name(<parameters>): return_type =
  let <variable definition>
  in  <expression>;
```

  – Invocation:

```
thisModule.helper_name(<parameters>)
```

- For OCL functions refer to the ATL user guide

Model Transformation Quality

- Model driven engineering is becoming increasingly important
- Model transformations are similar to traditional software artifacts
  - Used by multiple developers
  - Changed according to changing requirements
  - Reused if possible

Model Transformation Quality

- Model Transformations become the next maintenance *nightmare*
- Make the quality of model transformations measurable

$$M \xrightarrow{t} M'$$

[6] M.F. van Amstel: *The Right Tool for the Right Job: Assessing Model Transformation Quality*
To appear in *Proceedings of the Fourth IEEE International Workshop on Quality Oriented Reuse of Software* (QUORS'10)

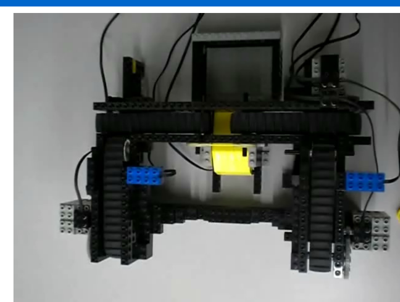
Model Transformation Quality

Quality:

- Understandability
- Modifiability
- Reusability
- Modularity
- Completeness
- Consistency

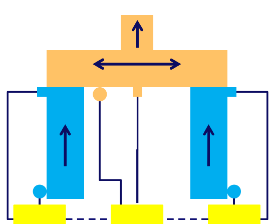
[7] Boehm, et al.: *Characteristics of Software Quality*
North-Holland, 1978

MDE Case Study



MDE Case Study

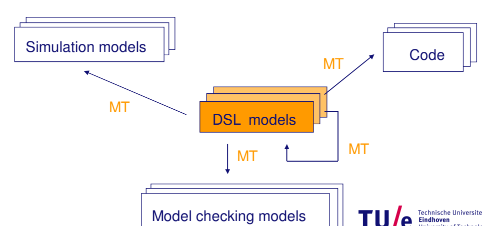
- Concurrent objects
  - Controllers
  - Hardware
    - Conveyors
    - Motors
    - Sensors
- Communication
  - Wireless
  - Wired



MDE Case Study

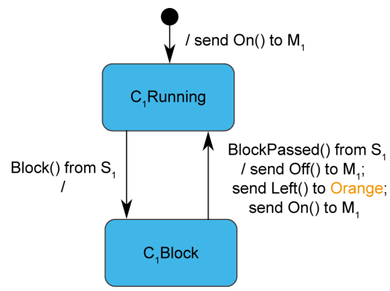
Goal:

- Simulation, verification and implementation of the same model

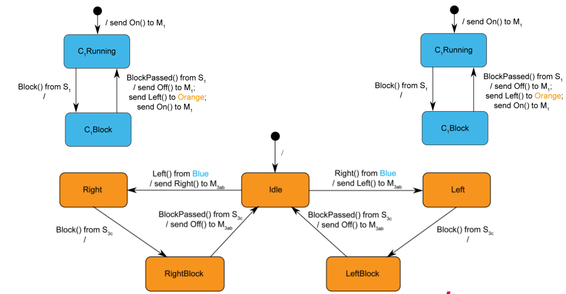


MDE Case Study

DSL



MDE Case Study



MDE Case Study

Platforms:

- **Simulation**
  - **POOSL**
- **Execution**
  - **NQC**
- **Verification**
  - **PROMELA/SPIN**

	Communication Primitives		#Objects
	Synchronous/ Asynchronous	Lossless/ Lossy	
DSL	Both	Both	Unlimited
POOSL	Synchronous	Lossless	Unlimited
NQC	Asynchronous	Lossy	Limited
PROMELA	Both	Lossless	Unlimited

MDE Case Study

Model Transformations