Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze
dell'Informazione e Matematica

Corso di Laurea in Informatica

Generation of Textual Modelling Environments for Metamodel-specific Languages

Relatore:
Prof.
Alfonso Pierantonio

Candidato:
Stefano Valentini
Matr.: 227718

Introduction

- This thesis describes an automated process that generates web-based text modeling environments. In particular, starting from a meta-model, it has been defined:
  - *a canonical mapping for the definition of the textual (concrete) syntax;*
  - *the generation of its modeling environment.*

The whole process is carried out by a Java command line application that uses the interaction of models transformation engines and frameworks such as *Acceleo* and *Xtext*.

Model Driven Engineering

- **Model Driven Engineering** technologies and tools have been used. MDE is a software development methodology based on key concepts such as:
  - **Models:** given a purpose, we can define a model as an artefact that represents a certain system by abstracting the details that are not useful for achieving that purpose.
  - **Meta-models**: a meta-model is a formal description for the creation of models.
  - **Models transformations**: programs for the generation of new models starting from existing models and/or textual artefacts.

- MDE refers to a meta-modeling hierarchy for which :
  - in the lower level, **M0**, there is the system that needs to be modeled;
  - in the **M1** level there is the model (or models) that represent the system;
  - for each model, in the **M2** level, there is the respective meta-model to which it conforms;
  - similarly, in the **M3** level, for each meta-model, there is the meta meta-model to which it conforms.

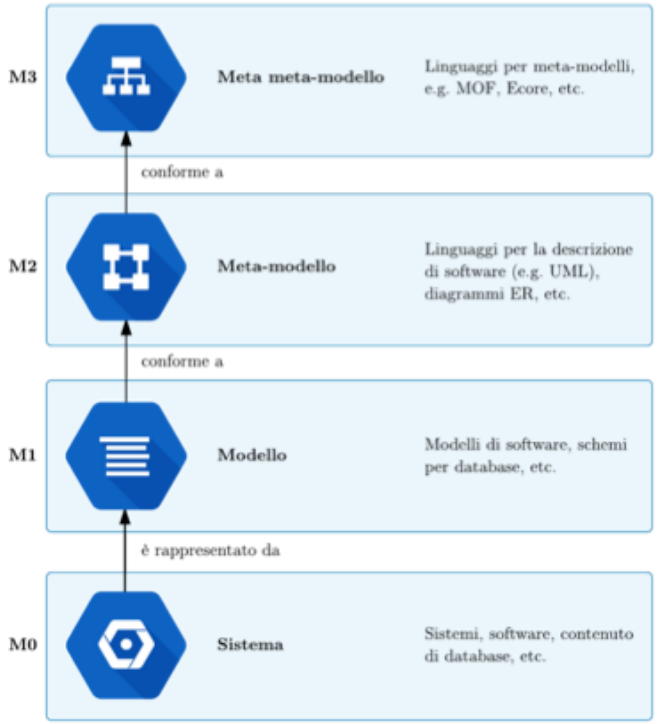

Fig. 1. Meta-modeling Hierarchy.

Eclipse Modeling Framework

- The tools that have been used are part of the *Eclipse Modeling Framework* (EMF). **EMF** is a framework that exposes tools for model manipulation. In particular, EMF provides a meta-model for the creation of models called *Ecore*. The main concepts of the *Ecore* standard are the *EClasses*, *EStructuralFeatures*, *EReferences* and *EAttributes*. In general, an *EClass* may contain more *EStructuralFeatures* or *EReferences* and/or *EAttributes*.

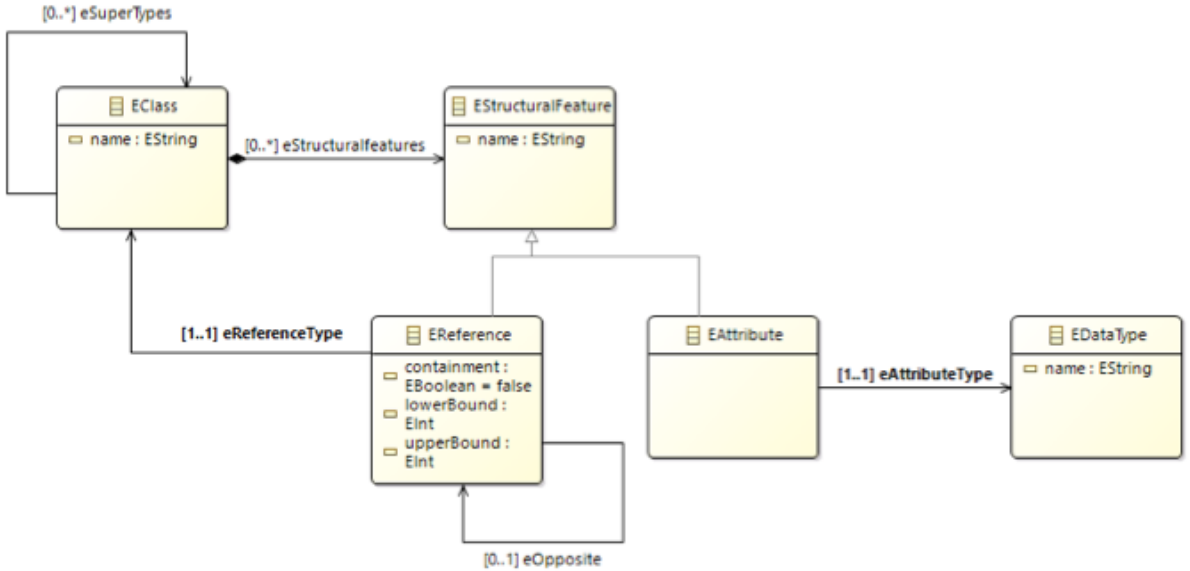

Fig. 2. Structure of the Ecore standard.

Acceleo

- Acceleo is a M2T model transformation language. A M2T transformation in Acceleo is composed by one or more *modules*, that are *.mtl* files that contains **templates** and **queries**.
- **Templates** are portions of code that are delimited by the `[template... ][/template]` tag that contains text generation instructions.
- **Queries** are portions of code that are delimited by the `[query... /]` tag and are used to query and then extract information from the input models.
- Exemple of a ìn Acceleo template that generates a JavaBean for each class that is contained in the UML input model.

```
1  [comment encoding = UTF-8 /]
2  [module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]
3
4  [template public generate(aClass : Class)]
5  [file (aClass.name.concat('.java'), false)]
6      public class [aClass.name.toUpperFirst()/] {
7      [for (p: Property | aClass.attribute) separator('\n')]
8          private [p.type.name/] [p.name/];
9      [/for]
10
11     [for (p: Property | aClass.attribute) separator('\n')]
12         public [p.type.name/] get[p.name.toUpperFirst()/]() {
13             return this.[p.name/];
14         }
15     [/for]
16
17     [for (o: Operation | aClass.ownedOperation) separator('\n')]
18         public [o.type.name/] [o.name/]() {
19             // TODO should be implemented
20         }
21     [/for]
22     }
23 [/file]
24 [/template]
```

Fig. 3. Example of an *.mtl* file

Xtext

- Xtext is an Eclipse framework used to implement programming languages and DSLs starting from a **grammatical specification**. From this specification Xtext generates an Ecore meta-model representing the entities expressed in the grammar and the associated parser; Xtext also offers the possibility to integrate the project with support for text editors. The editors are implemented in JavaScript. Language-specific resources and support services (such as **syntax checks**, **syntax highlighting**, and **code completion**) are provided via HTTP requests to the server component (also generated by Xtext).

- The **grammatical specification** describes the concrete syntax and how it is represented in memory. It consists of several types of rules:
  - **Terminal rules:** describe DSL tokens, they are usually used to express basic data types such as INT, STRING etc.
  - **Parsing rules:** they describe the entities of the DSL, they are used as a pattern for the production of EClasses in the Ecore model derived from the grammar.
  - **Type rules**: they are used to express complex data types, lead to the generation of EDataTypes instances instead of Eclasses,
  - **Enumeration rules**: can be seen as shortcuts for type rules; allow to define a set of possible "options" for a given rule.

```
1  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
2  ...
3  terminal INT returns ecore::EInt:
4      ('0'..'9')+;
```

Fig. 4.1. Example of a terminal rule.

```
1  Entity:
2      'entity' name = ID ('extends' superType=[Entity])? '{'
3          attributes += Attribute*
4          '}';
```

Fig. 4.2. Example of a parsing rule.

```
1  QualifiedName returns ecore::EString:
2      ID ('.' ID)*;
```

Fig. 4.3. Example of a type rule.

```
1  enum METHOD returns METHOD
2      GET = 'GET' | POST = 'POST' ;
```

Fig. 4.4. Example of an enumeration rule.

Implementation

- The developed application performs the following steps: after having *loaded* and *registered* the *Ecore meta-model*, and after having *compiled* the *Acceleo template*, performs the *M2T transformation* that generates the .xtext file containing the *grammatical specification* produced from the input meta-model. This file is inserted in a *dedicated Xtext project* created by the application itself; the compilation of this project leads to the generation of the *web editor* and the server component that provides support services.

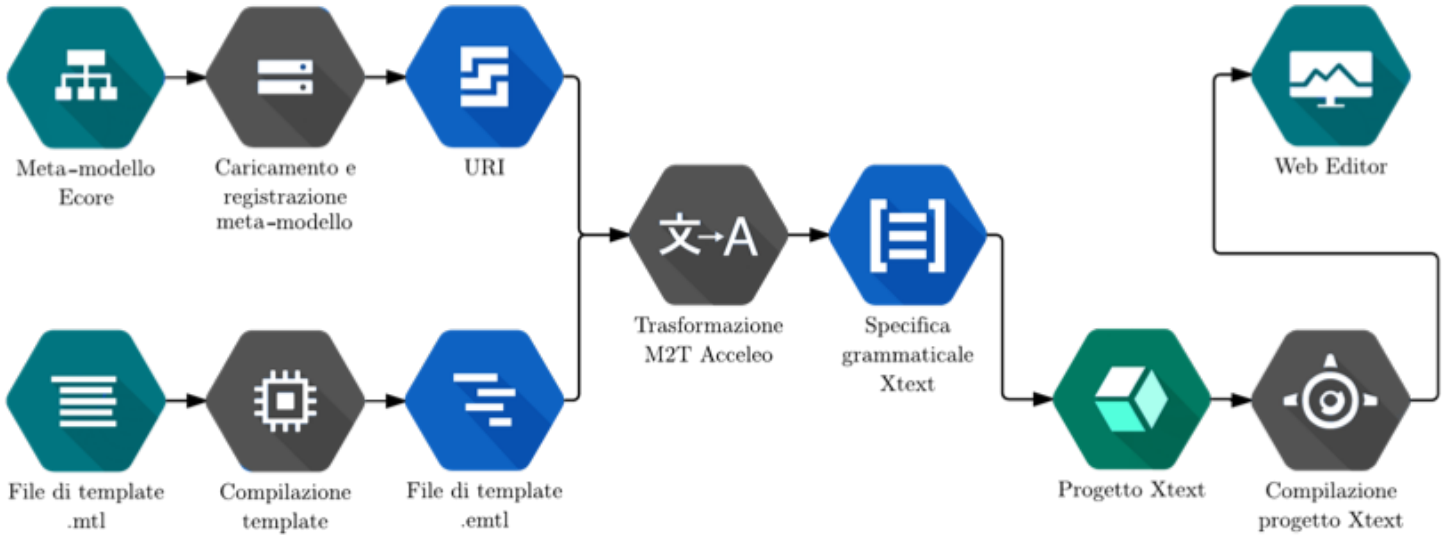

Fig. 5. Application execution flow.

generate.mtl Acceleo Template

- The essence of the application resides in the template; to obtain a *.xtext* file, it is necessary to provide the Acceleo template with the URI of the Ecore meta-model in input and the *entry point*. The *entry point* is the name of the EClass from where the generation starts, it can be considered as the main entity of the meta-model or the entity that somehow "encloses" the other entities. The Acceleo template generates grammar rules this way:
  - the **entry point** EClass is analyzed, generating parsing rules that reflect its characteristics (attributes, relationships, compositions, considering the case in which it can be a superclass of other EClasses and/or an interface/abstract EClass).
  - following the same procedure the parsing rules for the other **EClasses** are generated.
  - for each **EDataType** in the input meta-model a **type rule** is generated.
  - **Enumeration rules** are generated for each **EEnum** in the input meta-model.

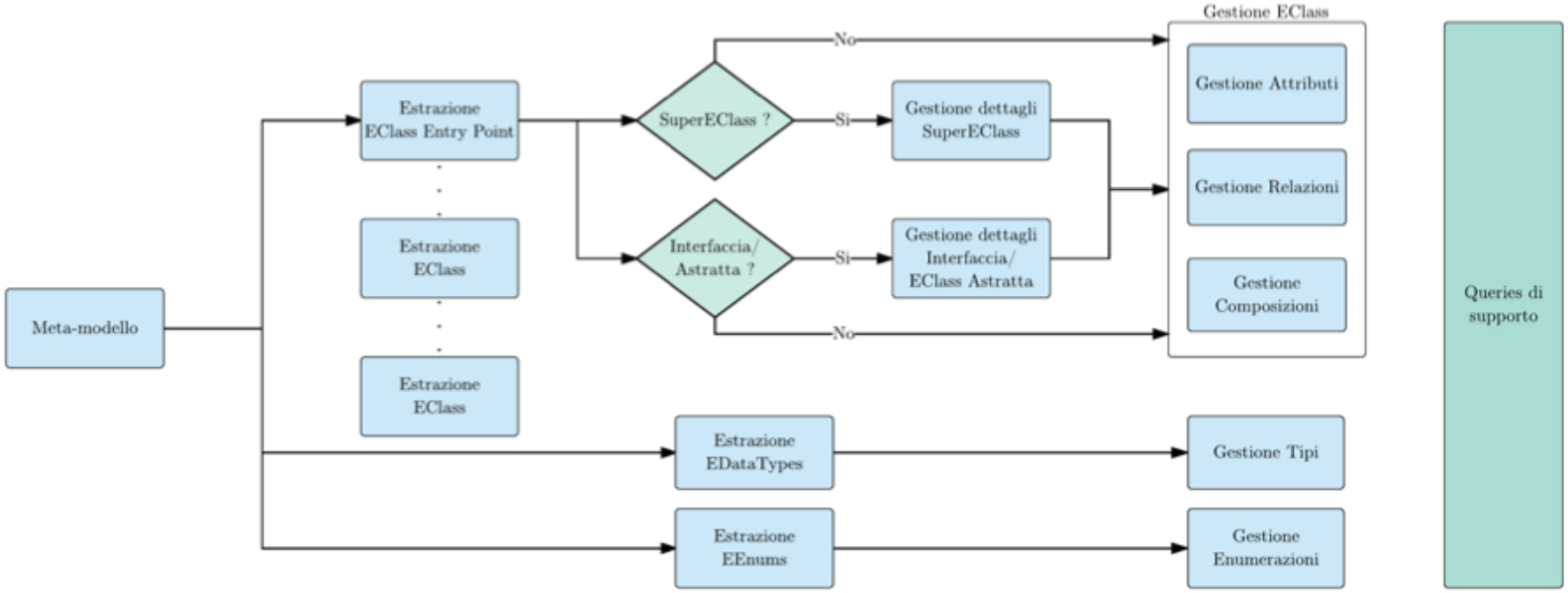

Fig. 6. *generate.mtl* Acceleo template structure

Running example

- Let's take a look at the *school.ecore* meta-model in Fig. 7. From this meta-model, through the *M2T Acceleo transformation*, we obtain the *grammatical specification* in Fig. 8. This file is subsequently placed inside the *dedicated Xtext project*. After compiling the project, we get the *web editor* in Fig. 9.

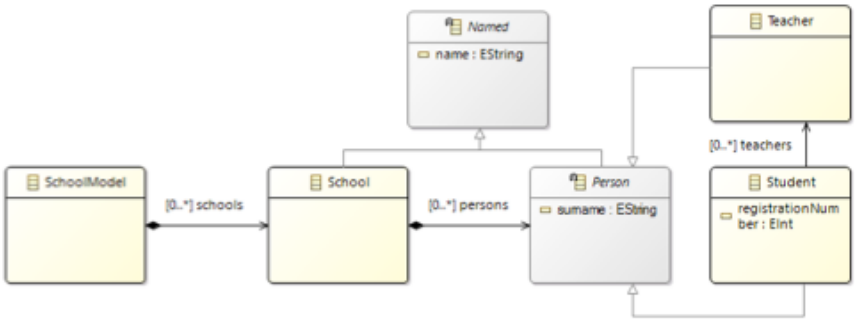

Fig. 7. *school.ecore* meta-model.

```
1  grammar org.xtext.schoolDsl.SchoolDsl
2      with org.eclipse.xtext.common.Terminals
3  generate schoolecoreDsl "http://www.xtext.org/schoolecoreDsl"
4  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
5
6  SchoolModel returns SchoolModel:
7      {SchoolModel}
8      'SchoolModel'
9      '{'
10         ('schools' ':' '{' schools+=School ( "," schools+=School)* '}' )?
11     '}';
12
13 Named returns Named:
14     School | Student | Teacher ;
15 Person returns Person:
16     Student | Teacher ;
17
18 School returns School:
19     {School}
20     'School'
21     '{'
22         ('name' '=' name = EString   )?
23         ('persons' ':' '{' persons+=Person ( "," persons+=Person)* '}' )?
24     '}';
25
26 Student returns Student:
27     (Student)
28     'Student'
29     '{'
30         ('name' '=' name = EString)?
31         ('surname' '=' surname = EString)?
32         ('registrationNumber' '=' registrationNumber = EInt)?
33         ('teachers' '{' teachers += [Teacher|STRING]
34                     ( "," teachers += [Teacher|STRING])* '}' )?
35     '}';
36
37 Teacher returns Teacher:
38     (Teacher)
39     'Teacher'
40     '{'
41         ('name' '=' name = EString)?
42         ('surname' '=' surname = EString)?
43     '}';
44 EInt returns ecore::EInt:
45     '-'? INT;
46 EString returns ecore::EString:
47     STRING | ID;
```

Fig. 8. Grammar generated from the *school.ecore* meta-model.

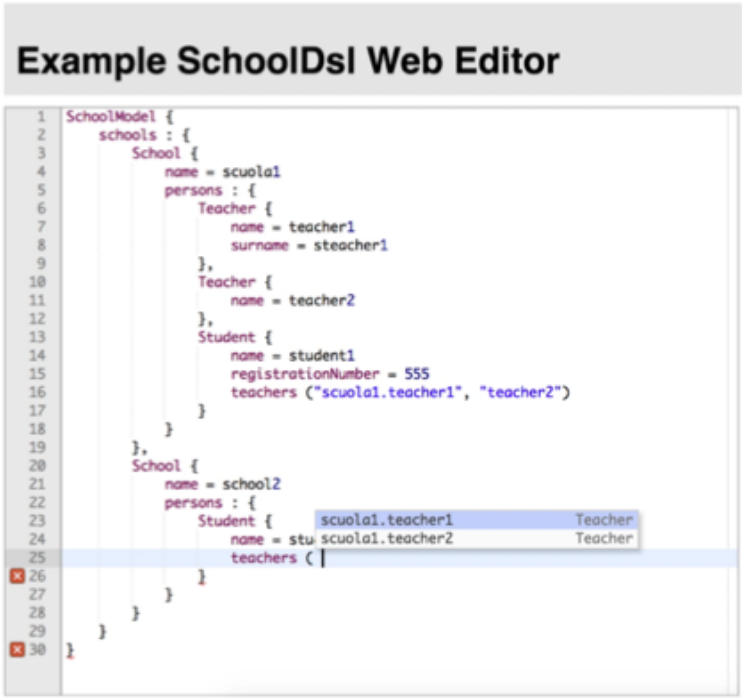

Fig. 9. Web editor for the *school.ecore* meta-model.

Case study: generation of the web modeling environment for the meta meta-model *ecore.ecore*

- Having a generator of modeling environments we can experiment with more complex meta-models; in particular, being the **meta meta-model for the Ecore standard** considered in any case a meta-model, we can generate, using the developed application, a textual development environment for the Ecore meta-models themselves. Considering as an entry point the EClassifier *EPackage*, what we obtain is a grammatical specification that reflects the modeling constraints expressed by the standard itself; that is, we get a web-based modeling environment for generic meta-models that conform to the Ecore standard. Fig. 10 shows a practical example of modeling; in detail, through the generated web editor, the meta-model *school.ecore* (shown in Fig. 7 of the previous slide) has been constructed.

Fig. 10. *ecore.ecore* meta meta-model web editor.