

M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics  
Department of Measurement and Information Systems  
Fault Tolerant Systems Research Group

Critical Architectures Laboratory  
Spring Semester 2015/2016

## **Automatic Analysis of Domain-Specific Languages**

Syllabus  
v1.2

**Author: Csaba Debreceni**  
(debreceni@mit.bme.hu)

February 20, 2016

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.1.1	Cyber-Physical Systems . . . . .	1
1.1.2	Laboratory . . . . .	2
<b>2</b>	<b>Technologies</b>	<b>2</b>
2.1	EMF Validation Framework . . . . .	2
2.2	Ecore . . . . .	3
2.3	Reflective EMF API . . . . .	3
2.4	Acceleo . . . . .	4
2.5	Alloy . . . . .	5
<b>3</b>	<b>List of Questions</b>	<b>5</b>
<b>4</b>	<b>Tasks</b>	<b>6</b>
4.1	Extend the existing DSL . . . . .	6
4.2	Create an instance model for the DSL . . . . .	6
4.3	Provide additional validation rules . . . . .	6
4.4	Create an Acceleo code generator . . . . .	6
4.5	Create an Acceleo UI Launcher . . . . .	7
4.6	Alloy Analyzer . . . . .	7

## 1 Introduction

This document is the syllabus for the *Automatic Analysis of Domain-Specific Languages* session of the *Critical Architectures Laboratory* course. The purpose of this session is to demonstrate a method on:

1. how to validate our *Domain-Specific Languages* (created with EMF) using the *EMF Validation Framework*,
2. how to generate source code using the *Acceleo* engine, and
3. how to use the *Reflective EMF API*.

### 1.1 Background

This section shows the running example used during the laboratory.

#### 1.1.1 Cyber-Physical Systems

*Cyber-Physical Systems* (CPS) are on one hand close to *embedded systems* as they are also built from sensors, controllers and actuators, where the sensors gather heterogeneous information from the environment, the controllers observe the gathered information and order the actuator to modify the environment according to the observed information. On the other hand, CPS systems are aiming to harvest the benefits of *elastic cloud based* resources to provide more sophisticated automation services.

The metamodel of a simplified *Cyber-Physical System* is depicted on Figure 1. In our terminology, the sensors, controllers and actuators are Task instances and the nodes in the cloud infrastructure are Computer instances. Each Task can be allocated to one dedicated Computer (allocatedTo reference) and uses some resources of the node defined as slots (reqSlot). Additionally, different type of tasks have different level of severity (low, medium, high, critical). The Computer class has two attributes where the attribute availableSlots shows the currently unused slots, while the defaultSlots represents the maximum number of resources the node has.

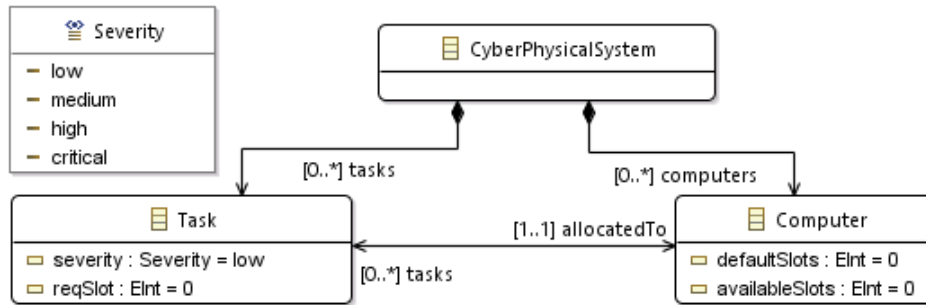


Figure 1: Simplified Cyber-Physical System metamodel

### 1.1.2 Laboratory

During this session, your task are the following:

1. extend the metamodel based on some novel requirements,
2. define additional validation rules to the extended metamodel using the *EMF Validation Framework*, and
3. create an *Acceleo* code generator to derive *Alloy* code.

## 2 Technologies

### 2.1 EMF Validation Framework

The *Eclipse Modeling Framework (EMF)* [6] is considered as the *de facto* industry standard for modeling. During the past years, EMF was extended with additional services such as the *EMF Validation Framework* [8]. The purpose of this framework is to provide language-independent validation on EMF models. The framework supports the definition of *batch* and *live* validation rules. In the case of using *batch* rules, users have to explicit execute the validation, while the *live* rules are triggered by a *change notification* service provided by EMF.

The basic overview of *validation framework* is described in [9]. An advanced validation tutorial is explained in [3].

In this session, we will use a pre-configured validation plug-in. The main task is to create additional constraints by extending the *AbstractModelConstraint* class and inserting a reference for the created classes into the *plugin.xml* under the *org.eclipse.emf.validation.constraintProviders* extension point. The validation can be executed by the **Validate** button in the context menu of any EMF tree editor.

## 2.2 Ecore

The *Object Management Group* defines a four-layered architecture standard for model-driven engineering that separates the different conceptual level for defining a model:

- M3: meta-metamodel,
- M2: metamodel,
- M1: instance model,
- M0: reality.

The EMF framework defined its own M3 variant called *Ecore*. Its simplified structure is depicted on Figure 2.

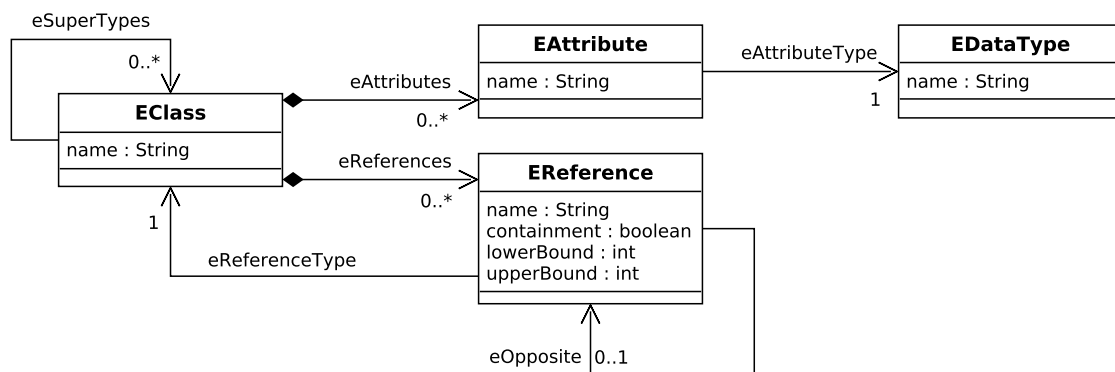


Figure 2: Ecore metamodel – OMG M3 standard

An *EClass* consists of multiple *EAttributes* and *EReferences*. Each *EAttribute* has an *EDataType* type while *EReferences* use an *EClass* as types. An *EClass* can also have multiple super types (*eSuperTypes*) to describe inheritance. For more details about the *Ecore* metamodel, check out [7].

## 2.3 Reflective EMF API

The *Reflective EMF API* provides the ability to traverse model elements and their features in a generic way. One of the key interfaces in EMF is *EObject*. All modeled objects implement this interface in order to provide several important features:

- The *eClass()* method is used to retrieve the metadata (*EClass*) of the instance.
- The *eGet()* and *eSet()* methods are used to access the data of the instance.
- The *eContainer()* method is used the container (parent) object of the instance.

The following example shows how to get all the EClasses of the metamodel from an instance.

```
public Collection<EClass> getEClasses(CyberPhysicalSystem cps) {
    final List<EClass> ret = new ArrayList();
    for (EClassifier ec : cps.eClass().getEPackage().getEClassifiers()) {
        if (ec instanceof EClass)
            ret.add((EClass) ec);
    }
    return ret;
}
```

Listing 1: Getting all the EClasses in Java

## 2.4 Acceleo

*Acceleo* [1] is a pragmatic implementation of the Object Management Group (OMG) Model to Text Language (MTL) standard. It is a template-based code generator with an advanced IDE. The framework defines its own language. The following simple example shows the syntax:

```
[comment encoding = UTF-8 /]
[module generate('/hu.bme.mit.cps/model/cps.ecore')]

[template public generate(cps : CyberPhysicalSystem)]
    [comment @main /]
    [file ('cps.txt', false, 'UTF-8')]

    Hello ACCELEO World
    [for (task : Task | cps.tasks)]
        [if (task.severity = Severity.low)]
            [task.eClass.name] low
        [elseif (task.severity = Severity.medium)]
            [task.eClass.name] medium
        [elseif (task.severity = Severity.high)]
            [task.eClass.name] high
        [else (task.severity = Severity.critical)]
            [task.eClass.name] critical
    [/if]
[/for]
[/file]
[/template]
```

Listing 2: Acceleo example

The generator files consist of modules and templates. Acceleo can be interpreted as an object-oriented language where the modules are the classes and the templates are the methods inside a module. The first line defines the character coding of the generated file, then at least one metamodel path has to be connected to this module as it is in the second line. The main entry point of the generator will be the template with @main comment. The language uses only the if-else, let, for structures because it supports the definition OCL queries. The following query returns all the EClasses of the metamodel.

```
[query public getEClasses(cps : CyberPhysicalSystem) : Set(EClass) =
    cps.eClass().ePackage.eClassifiers->filter(EClass)/]
```

Listing 3: Acceleo query example

The "Getting Started" documentation can be found at [2]. Use it wisely for your preparation.

## 2.5 Alloy

Alloy is a language for describing structures and defining constraints for them. It is also a tool for generating instances that satisfies the constraints on the given structure. The following example shows the main elements of the language that will be used in the current lab session.

```
enum MyEnum {
    literal1, literal2
}

abstract sig AbstractMyClass {
    attr1 : one MyEnum,
    attr2 : one Int,
    ref1 : many AbstractMyClass
}

sig MyClass extends AbstractMyClass {}

run {} for 2 MyClass
```

Listing 4: Alloy language example

Enumerations can be defined with the `enum` keyword and classes (signatures) with the `sig` keyword. After every properties of a structure (enum or class) put a comma (",") except the last one. To execute traversals on the search space, we can use the `run` command where the number of required instances can also be defined. For further information, check out [5]. The Alloy Analyzer is already available on the virtual machines and can be downloaded from [4] as well.

## 3 List of Questions

1. What kind of validation modes are supported in the EMF Validation Framework? What is the difference between them?
2. What languages are supported languages in the EMF Validation Framework for defining constraints?
3. What is a module in Aceleo? What is a template in Aceleo?
4. What kind of control flow statements are supported in Aceleo?
5. What is the purpose of an Aceleo UI Launcher project?
6. What are the most important parts of the Ecore metamodel?
7. What is an EClassifier in Ecore? Please list some known subinterfaces!
8. What is an EStructuralFeature in Ecore? Please list some known subinterfaces!

## 4 Tasks

### 4.1 Extend the existing DSL

Extend the existing DSL with the following elements: the *motion detector*, *alarm*, *smoke detector* and *controller* elements are inherited from Task while the *server* and *mainframe* elements are inherited from Node. Additionally, the *controller* has one reference for at least two other tasks. Provide that neither Task nor Computer classes can be initiated.

### 4.2 Create an instance model for the DSL

In the runtime Eclipse application, initiate an instance model for your extended DSL. It should contain at least 10 model elements.

### 4.3 Provide additional validation rules

Create at least 3 validation rules from the following list (extra points for all):

- Every Computer instance should contain only one *critical* task.
- Every Task instance should have the correct severity level.
  - *motion detector*: low
  - *alarm*: medium
  - *smoke detector*: high
  - *controller*: critical
- Every *controller* instance should control tasks with different type.
- The *availableSlots* attribute of every Computer should be calculated as follows:

$$\text{defaultSlots} - \sum \text{tasks.reqSlots}$$

where the tasks are the allocated Task instances on the actual Computer.

Validate your instance model (see Section 4.2).

### 4.4 Create an Acceleo code generator

This task has more sub-tasks to provide some refinement steps but it is possible to implement the whole generator in a single step. The requirements for the generator are the following: (1) generate enum and class signatures with properties and references, (2) also generate the run command with the proper number of the existing classes in your model. A simple output is shown in Listing 5.

```
enum MyEnum {
    literal1, literal2
}

abstract sig AbstractMyClass {
    attr1 : one MyEnum,
    attr2 : one Int,
    refl  : many AbstractMyClass
}

sig MyClass extends AbstractMyClass {}

run {} for 2 MyClass
```

Listing 5: Simple example output

The generator has to be well-commented and well-formatted. It will be awarded with some extra points if you use multiple *templates* and/or *queries*.

Sub-tasks:

- Generate enumerations with their literals
- Generate classes (abstract/not abstract, define ancestor)
- Generate properties, references (EStructuralFeature) with respected to types ( $EInt \rightarrow Int$ )
- Generate run command according to your instance model

Additional task for extra point:

- Order the generation as follows: (1) enumerations, (2) classes without ancestors, and (3) classes with previously defined ancestors

## 4.5 Create an Acceleo UI Launcher

Generate Alloy code from the previously created instance model in a runtime eclipse.

## 4.6 Alloy Analyzer

Download the Alloy Analyzer and try it with your generated als code.



## References

- [1] Acceleo. <https://www.eclipse.org/acceleo/>.
- [2] Acceleo Getting Started. [http://wiki.eclipse.org/Acceleo/Getting\\_Started](http://wiki.eclipse.org/Acceleo/Getting_Started).
- [3] Advanced EMF Validation Tutorial. [http://publib.boulder.ibm.com/infocenter/rsahelp/v7r0m0/index.jsp?topic=/org.eclipse.emf.validation.doc/references/extension-points/org\\_eclipse\\_emf\\_validation\\_constraintProviders.html](http://publib.boulder.ibm.com/infocenter/rsahelp/v7r0m0/index.jsp?topic=/org.eclipse.emf.validation.doc/references/extension-points/org_eclipse_emf_validation_constraintProviders.html).
- [4] Alloy Analyzer. <http://alloy.mit.edu/alloy/download.html>.
- [5] Alloy language. <http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf>.
- [6] Eclipse. <http://www.eclipse.org/>.
- [7] Ecore package details. <http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details>.
- [8] EMF Validation Framework. <http://www.eclipse.org/modeling/emf/?project=validation>.
- [9] EMF Validation Overview. [http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.emf.doc%2Freferences%2Foverview%2FEMF.Validation.html&cp=17\\_0\\_2](http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.emf.doc%2Freferences%2Foverview%2FEMF.Validation.html&cp=17_0_2).