# DPF Text

A textual graph-based modeling framework for education and research

08.04.2015

# What is DPF Text?

- A modeling framework very close to theory
- Graphs and morphisms are core concepts
- Simple and easy definitions of models
- No hidden internal constraints

# Why did I implement it?

- Fun & wanted to understand the formal definitions of algebraic graph transformations

- Wanted to have something for prototyping that behaves according to the theory of algebraic graph transformations (http://www.ckrause.org/2014/08/why-model-transformations-graph.html)

- Wanted to learn „Scala“ (https://typesafe.com/community/core-projects/scala)

# What can I already do?

- Create models called „specifications“ according to the theory of generalized sketches / DPF (http://dpf.hib.no)
- Arbitrary deep meta-model hierarchies
- Constraints can be defined in OCL templates & models can be validated accordingly
- Export specifications to images (GraphViz), Ecore and XMI
- Core concepts for transformations have been implemented but not completed yet

# For whom is the framework?

- The repository on github contains what I thought may be useful for other researcher (its an extract of what I implemented to play around)

- … hence the tool has been designed for researchers, not for professionals planning to use it in production…

- The tool is based on a grammer in XText and some algebraic data types (case classes) in Scala, therefore I hope it is easy to change for own ideas

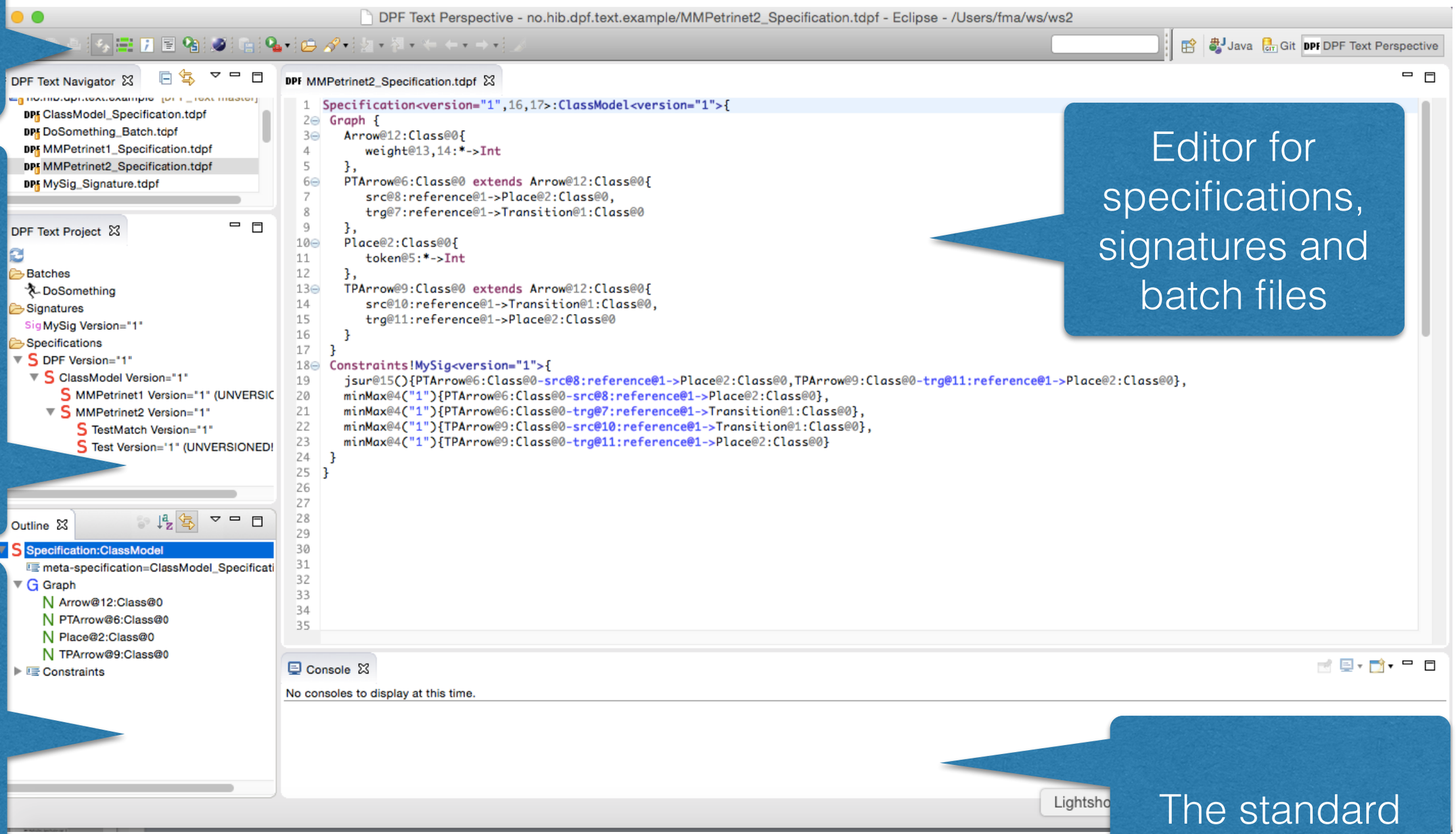
# Tool Overview

## DPF Text Perspective

Navigator showing all projects

A project navigator showing the basic concepts in the selected project

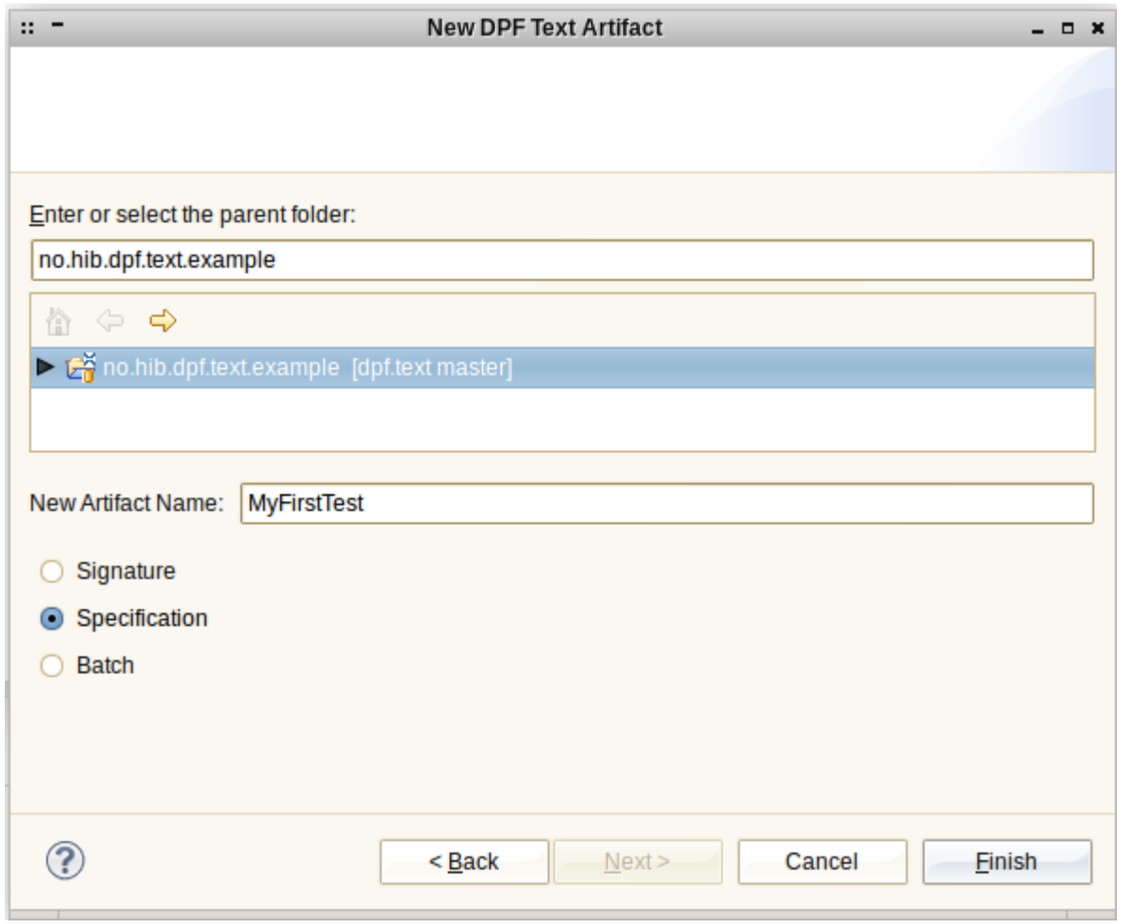
An outline of the current file in the editor




Editor for specifications, signatures and batch files

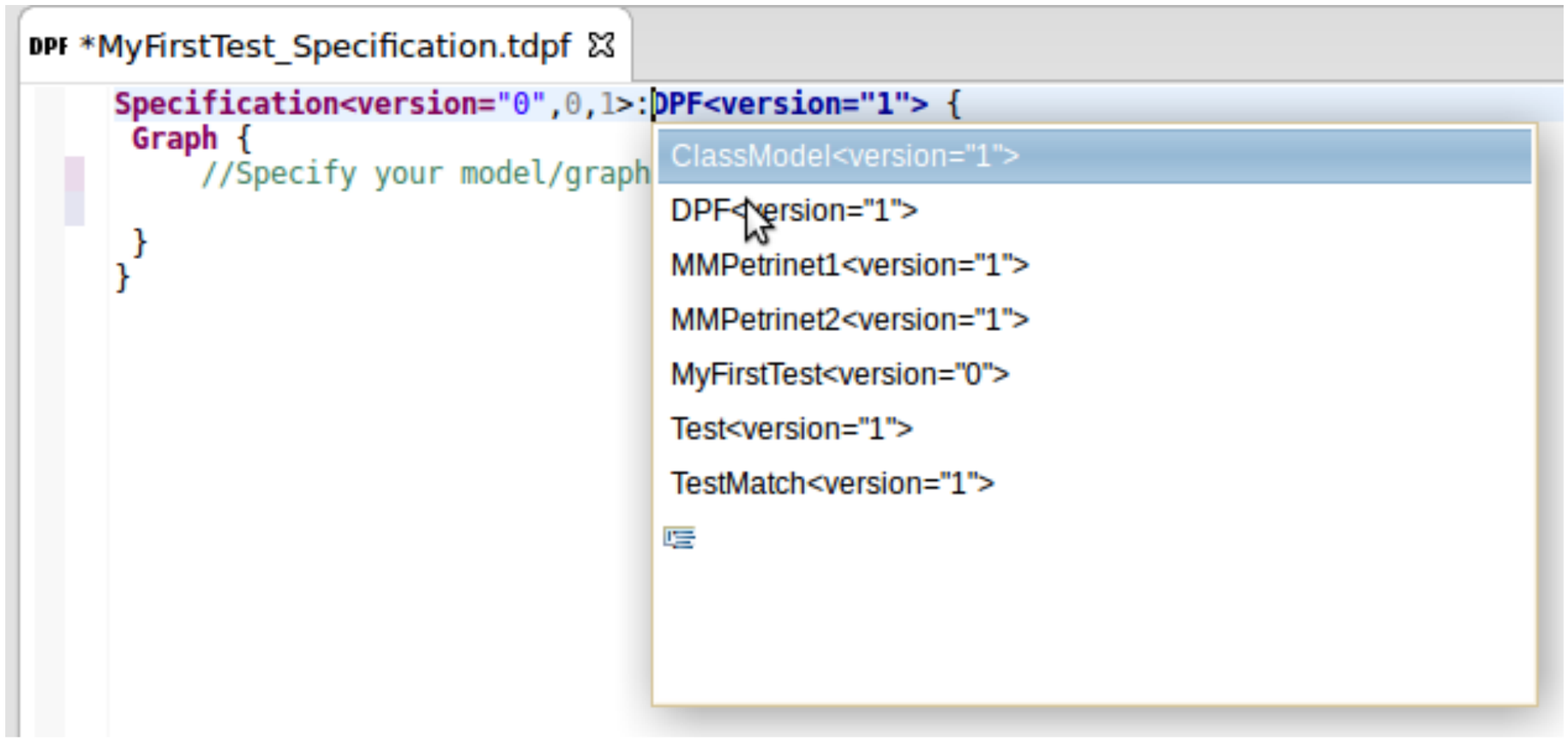
The standard console.

# Create New Specification (1)




Specifications are stored in plain text files having a specific naming convention. Specifications are our models.

# Create New Specification (2)




The editors support auto-completion to a large extend. If the „meta-model“ should not be the default DPF model, any available (i.e. in the same directory) can be chosen

# Create New Specification (3)

DPF *MyFirstTest_Specification.tdpf

```
Specification<version="0",0,1>:ClassModel<version="1">{
 Graph {
    Car:Class@0 extends Vehicle:Class@0{
        isForRacing:*->Boolean
    }
 }
}
```

Graphs are specified as AIGraphs (http://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2008/2008-07.pdf) i.e. they support inheritance and attributes

# Create New Specification (4)

DPF MyFirstTest_Specification.tdpf

```
Specification<version="0",0,4>:ClassModel<version="1">{
 Graph {
   Car@1:Class@0 extends Vehicle@2:Class@0{
      isForRacing@3:*->Boolean
   },
   Vehicle@2:Class@0
 }
}
```

When saved, specifications are stored in a default serialization. IDs are automatically assigned to all elements.

DPF MyFirstTest_Specification.tdpf

```
Specification<version="0",0,4>:ClassModel<version="1"><PLAIN>{
 Graph {
   Car@1:Class@0,
   Car@1:Class@0-isForRacing@3:*->Boolean,
   Car@1:Class@0-|>Vehicle@2:Class@0,
   Vehicle@2:Class@0
 }
}
```

By adding <PLAIN> alternatively the model is presented as a list of nodes and edges.

# A Signature

DPF MySig_Signature.tdpf

```
Signature<version="1",15,16><OCL> {
    abstract@1(){x:_}="context #x# inv: false" errorMsg="instance of an abstract element",
    irr@2(){x:_-y:_->z:_}="context #x# inv: not #y#->includes(self)" errorMsg="[irr] constraint violated",
    jsur@15(){x1:_-y1:_->z:_,x2:_-y2:_->z:_}="context #z# inv: (not O#y1#->isEmpty()) or (not O#y2#->isEmpty())" errorMsg="[jsur] constraint violated",
    min@3(min){x:_-y:_->z:_}="context #x# inv: #y#->size() >= #min#" errorMsg="[min] constraint violated",
    minMax@4(minMax){x:_-y:_->z:_}="context #x# inv: #y#->size() = #minMax#" errorMsg="[min_max] constraint violated",
    sur@5(){x:_-y:_->z:_}="context #z# inv: not O#y#->isEmpty()" errorMsg="[sur] constraint violated"
}
```

- Signatures are used to define templates for constraints
- Constraints can be assigned to „graphs“ see http://www.kirj.ee/public/proceedings_pdf/2013/issue_1/Proc-2013-1-3-15.pdf (A visual DPF tool implemented by our students)
- Graphs are specified by lists of nodes and arrows. Elements (in the signature) having the same name are considered to be identical.

# A Signature used in a Specification

DPF MySig_Signature.tdpf  DPF MMPetrinet2_Specification.tdpf

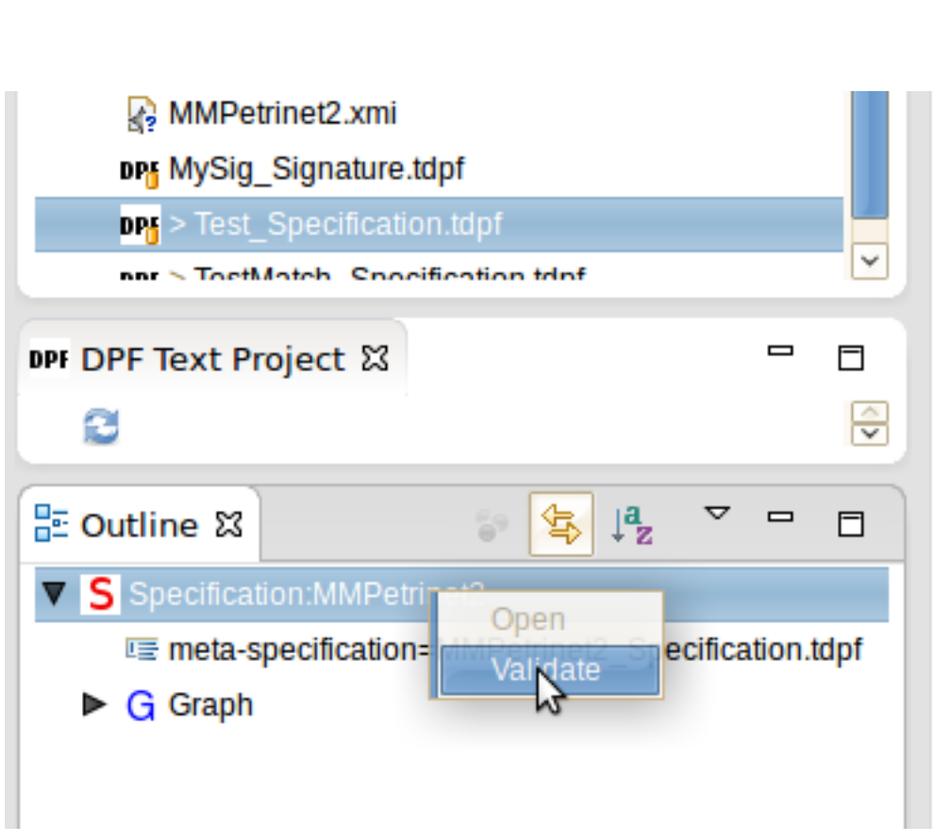
```
Specification<version="1",16,17>:ClassModel<version="1">{
 Graph {
   Arrow@12:Class@0{
      weight@13,14:*->Int
   },
   PTArrow@6:Class@0 extends Arrow@12:Class@0{
      src@8:reference@1->Place@2:Class@0,
      trg@7:reference@1->Transition@1:Class@0
   },
   Place@2:Class@0{
      token@5:*->Int
   },
   TPArrow@9:Class@0 extends Arrow@12:Class@0{
      src@10:reference@1->Transition@1:Class@0,
      trg@11:reference@1->Place@2:Class@0
   }
 }
 Constraints!MySig<version="1">{
   jsur@15(){PTArrow@6:Class@0-src@8:reference@1->Place@2:Class@0,TPArrow@9:Class@0-trg@11:reference@1->Place@2:Class@0},
   minMax@4("1"){PTArrow@6:Class@0-src@8:reference@1->Place@2:Class@0},
   minMax@4("1"){PTArrow@6:Class@0-trg@7:reference@1->Transition@1:Class@0},
   minMax@4("1"){TPArrow@9:Class@0-src@10:reference@1->Transition@1:Class@0},
   minMax@4("1"){TPArrow@9:Class@0-trg@11:reference@1->Place@2:Class@0}
 }
}
```

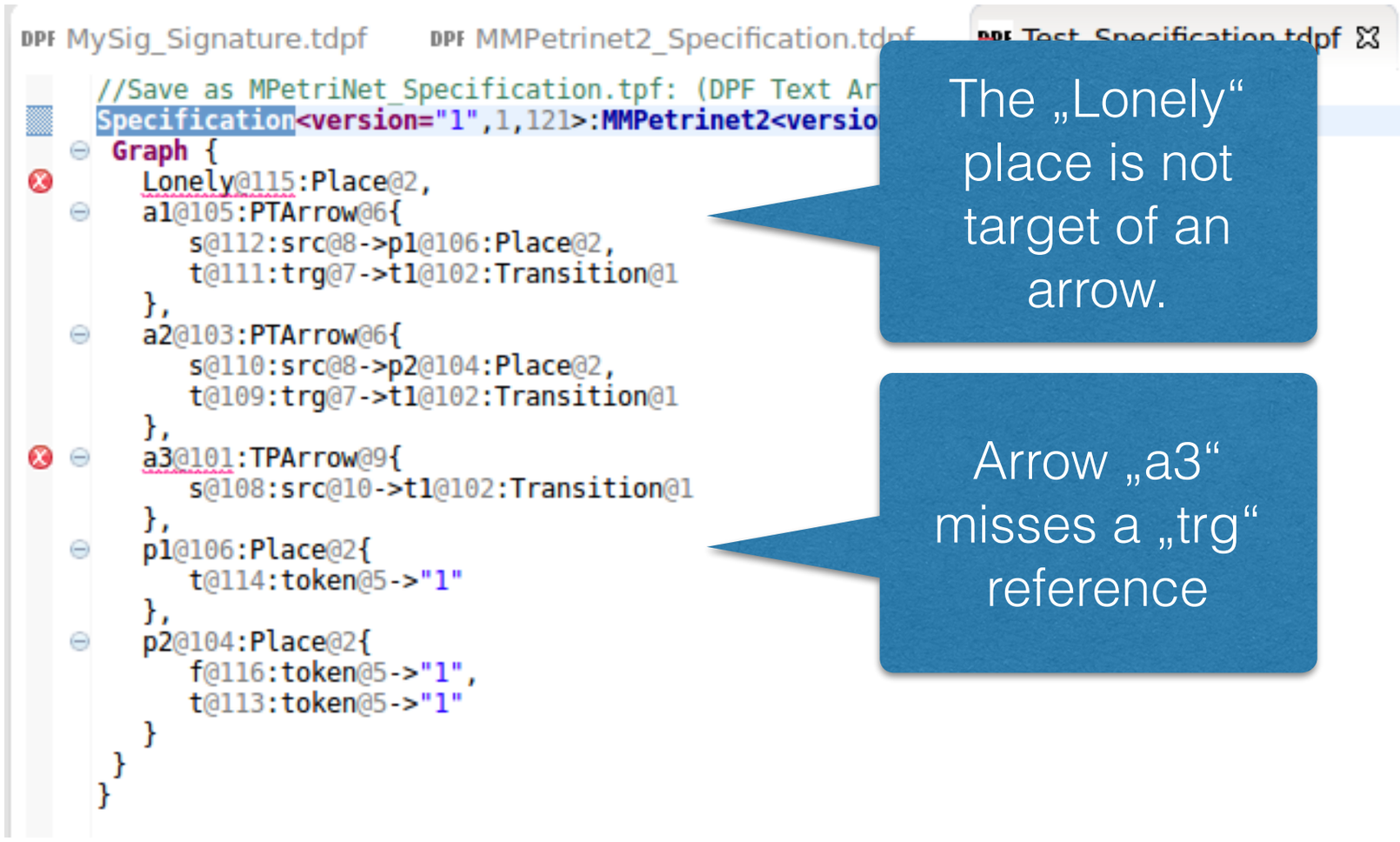
A simple PetriNet meta-model

**Constraints are assigned to the specification:**

1. *[minMax]*:*src* and *trg* references of *Arrow*s have cardinality =1,
2. *[jsur]*: each *Place* must be the source or target of an arrow.

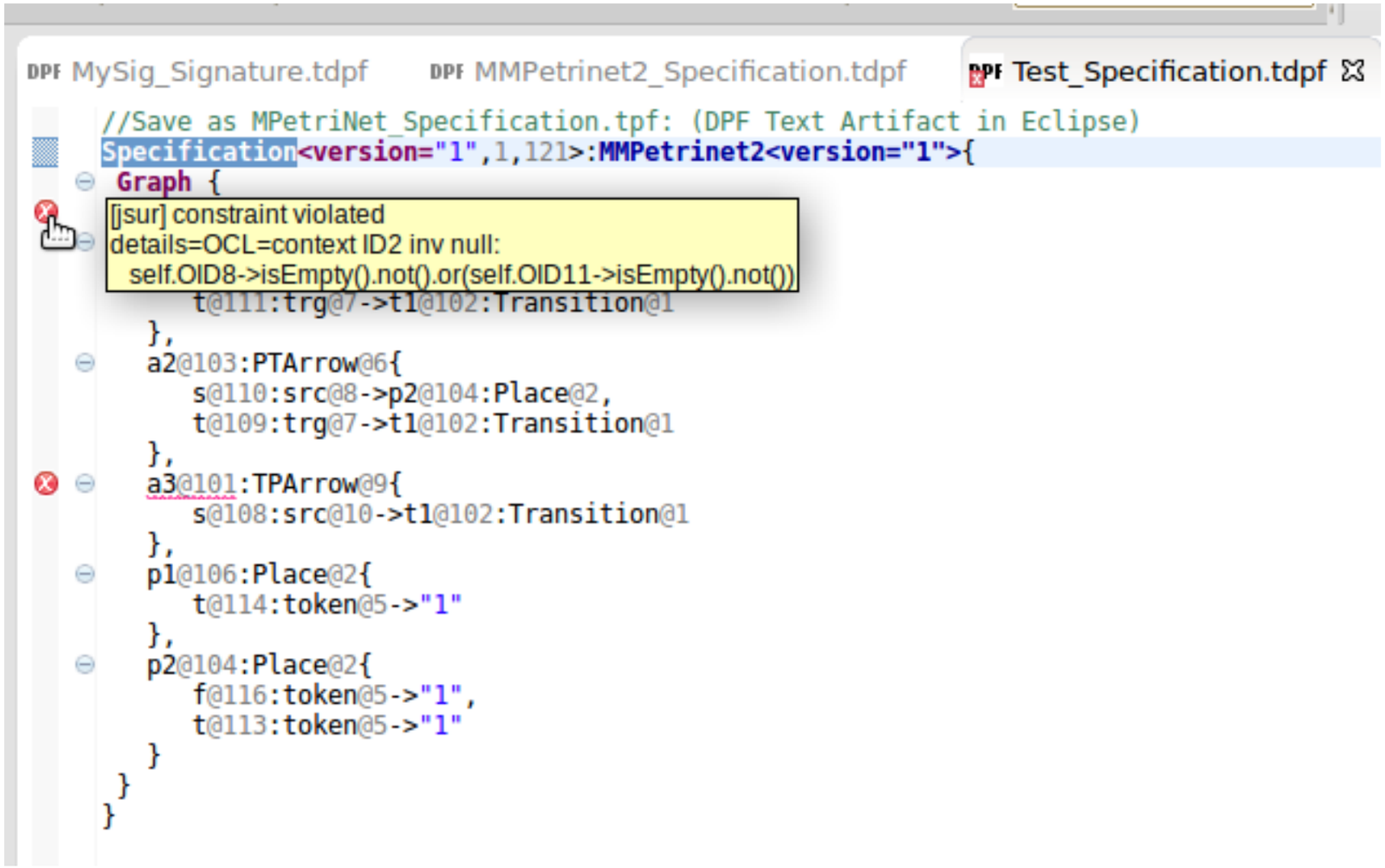
# Specification Validation (1)




Also on *save* models are automatically validated.

# Specification Validation (2)

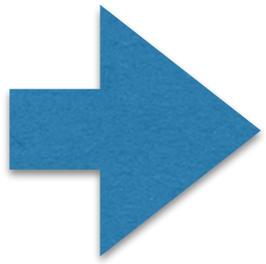


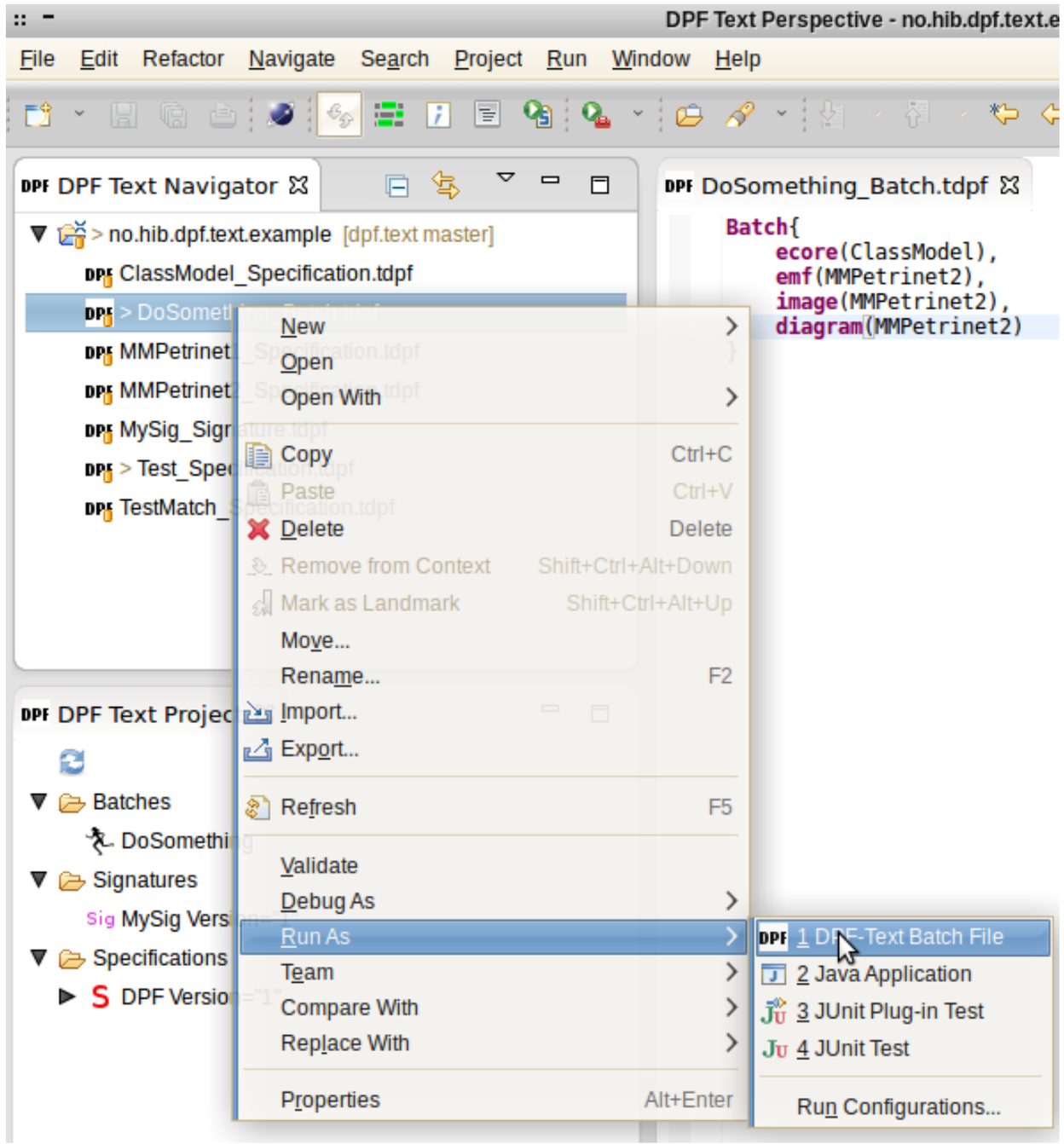
# Specifications can be exported into different Formats

DPF DoSomething_Batch.tdpf

```
Batch{
    ecore(ClassModel),
    emf(MMPetrinet2),
    image(MMPetrinet2),
    diagram(MMPetrinet2)
}
```

input

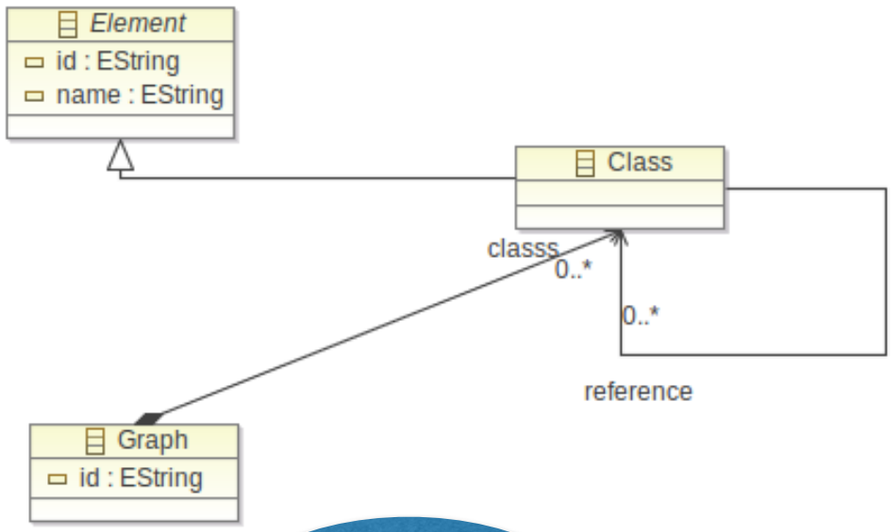




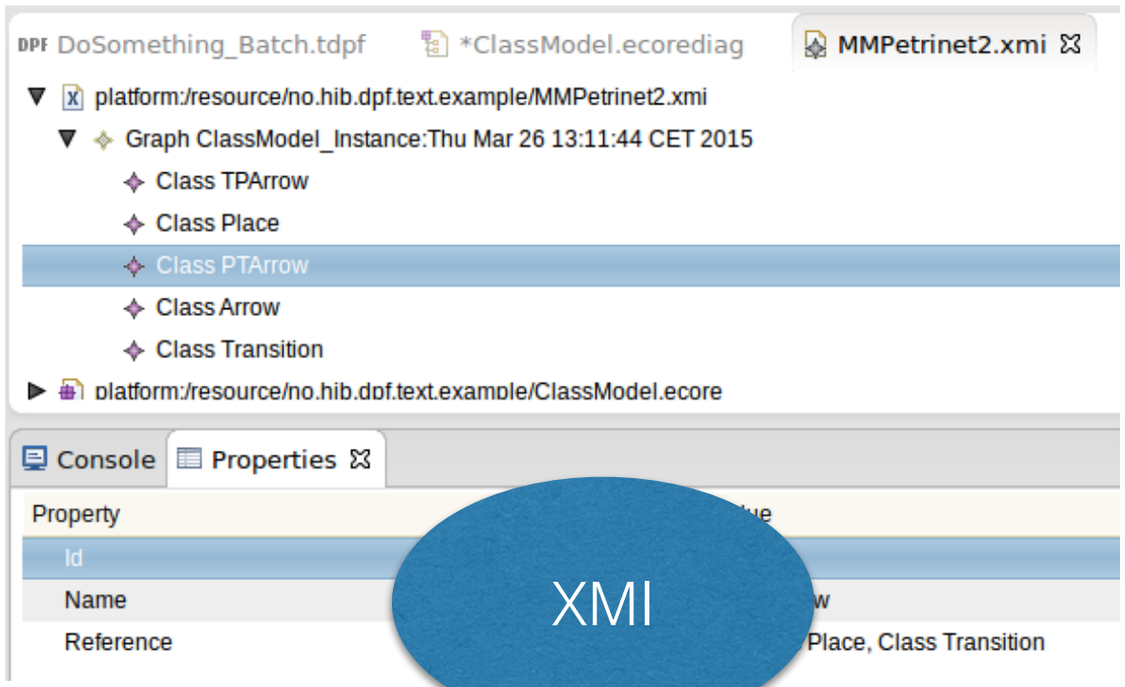

- > no.hib.dpf.text.example [dpf.text master]
  - > debug
  - ClassModel_Specification.tdpf
  - ClassModel.ecore
  - diagram_MMPetrinet2.dot
  - diagram_MMPetrinet2.jpg
  - > DoSomething_Batch.tdpf
  - image_MMPetrinet2.dot
  - image_MMPetrinet2.jpg
  - MMPetrinet1_Specification.tdpf
  - MMPetrinet2_Specification.tdpf
  - MMPetrinet2.xmi
  - MySig_Signature.tdpf
  - > Test_Specification.tdpf
  - TestMatch_Specification.tdpf

output

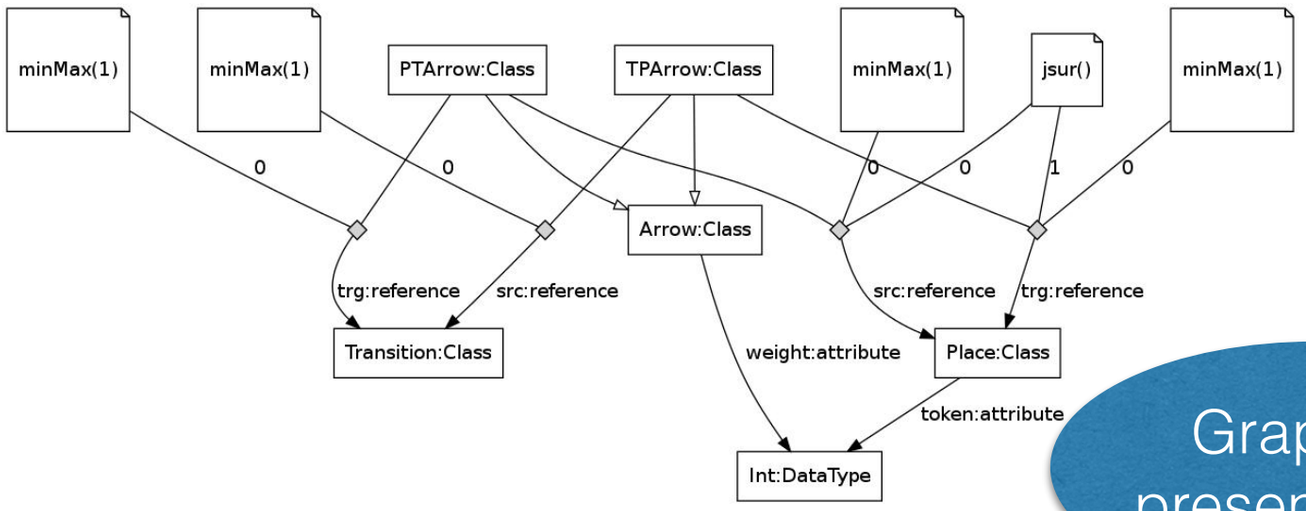
# Exported Specifications



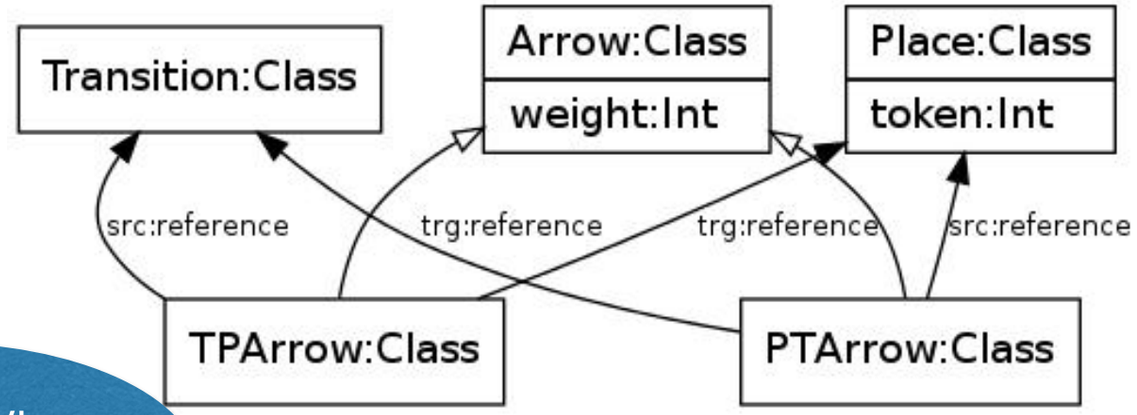

Ecore




XMI




GraphViz presentations



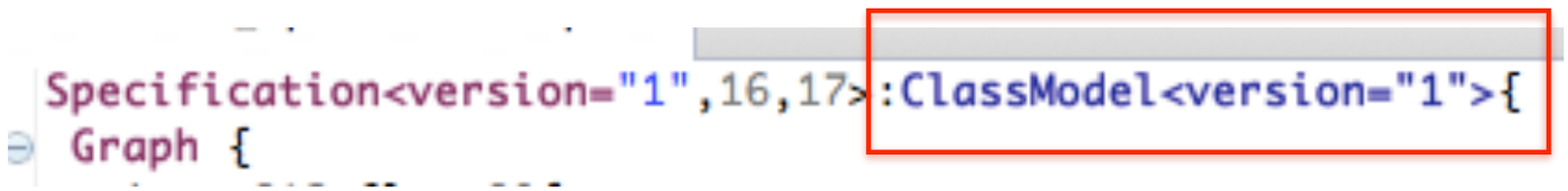
# If a Meta-specification got lost…

```
//*************************************************************************
//Automatically generated from specification:/home/hyperion/git/dpf.text/no.hib.dpf.text.example/GenerateMetamodel/T
//Date:Thu Mar 26 13:26:12 CET 2015
//*************************************************************************

Specification<version="?",0,11>:DPF<version="1">{
 Graph {
   PTArrow@6:Vertex@0{
      src@8:edge@1->Place@2:Vertex@0,
      trg@7:edge@1->Transition@1:Vertex@0
   },
   Place@2:Vertex@0{
      token@5:*->String
   },
   TPArrow@9:Vertex@0{
      src@10:edge@1->Transition@1:Vertex@0
   }
 }
}
```

*…. is a new one generated*
*from the specification trying to load it…*

*… however specifications should be versioned and stored in a repository …*

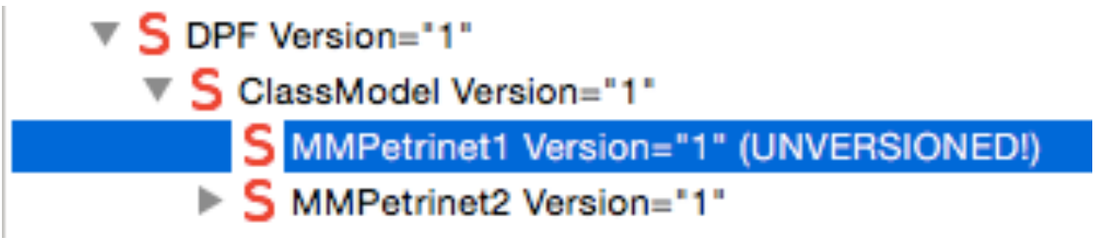
# Prepared for „Versioning“

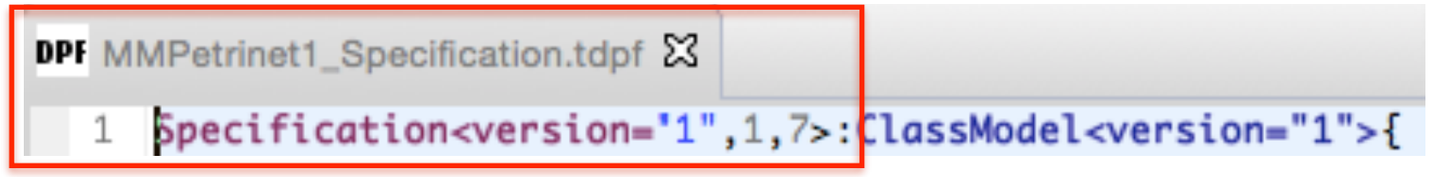



„Specifications and signatures are supposed to be stored in a version control system (VCS). Therefore each reference to a specification or signature need to specify its version.“

**Note:** before storing a specification *set the number after version and the ID counter to (ID counter - 1).* This make it possible to track which IDs have been published. In changed specifications/signatures all new ids should be set to numbers larger than the one  of the head version. *Also assign a new version name!*

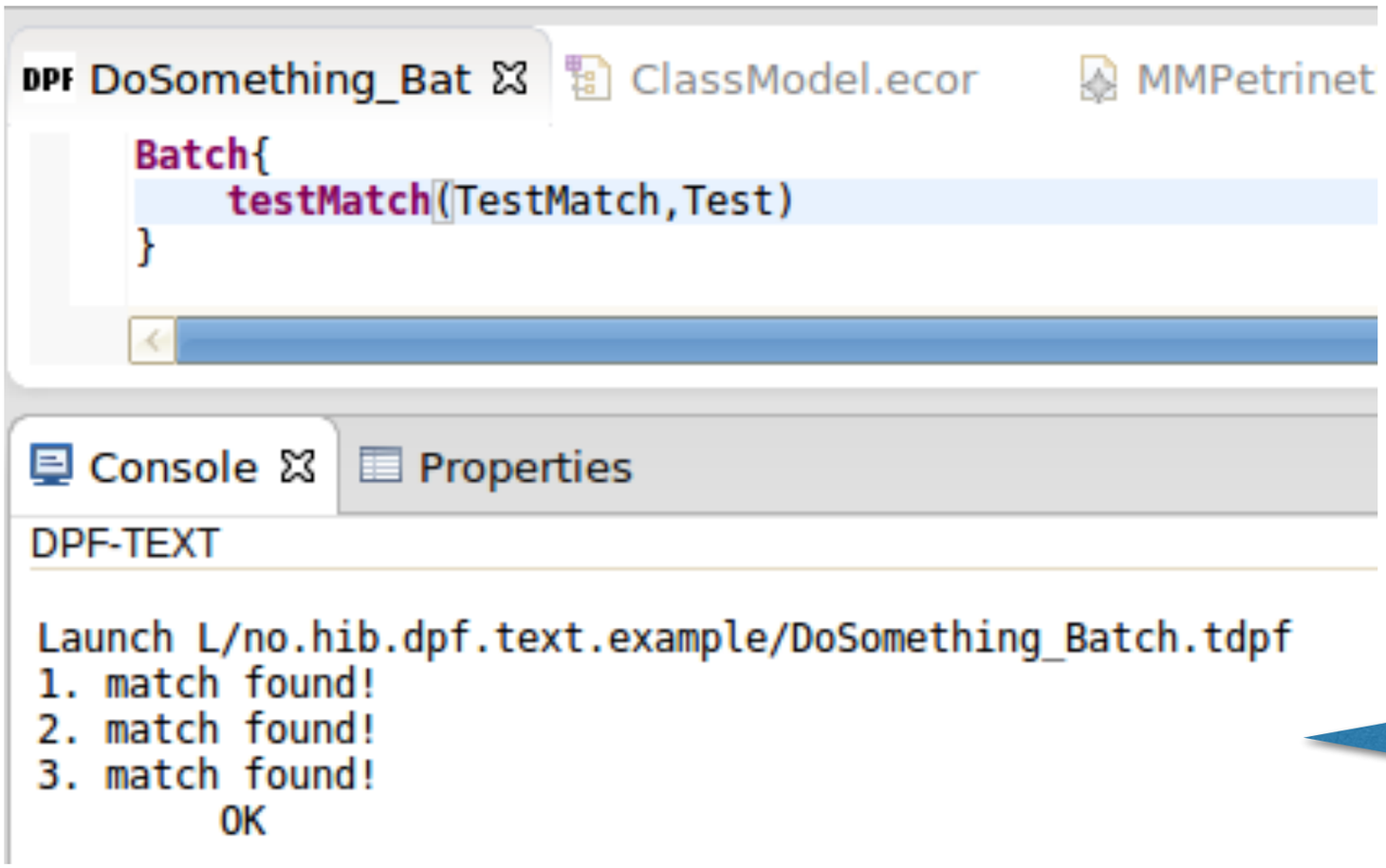
Tool support for versioning has been planned but not implemented yet (Diff + Merge).




7 - 1 ≠ 1 (next - lastFromVersion)

# Prepared for implementing Algebraic Graph Transformations



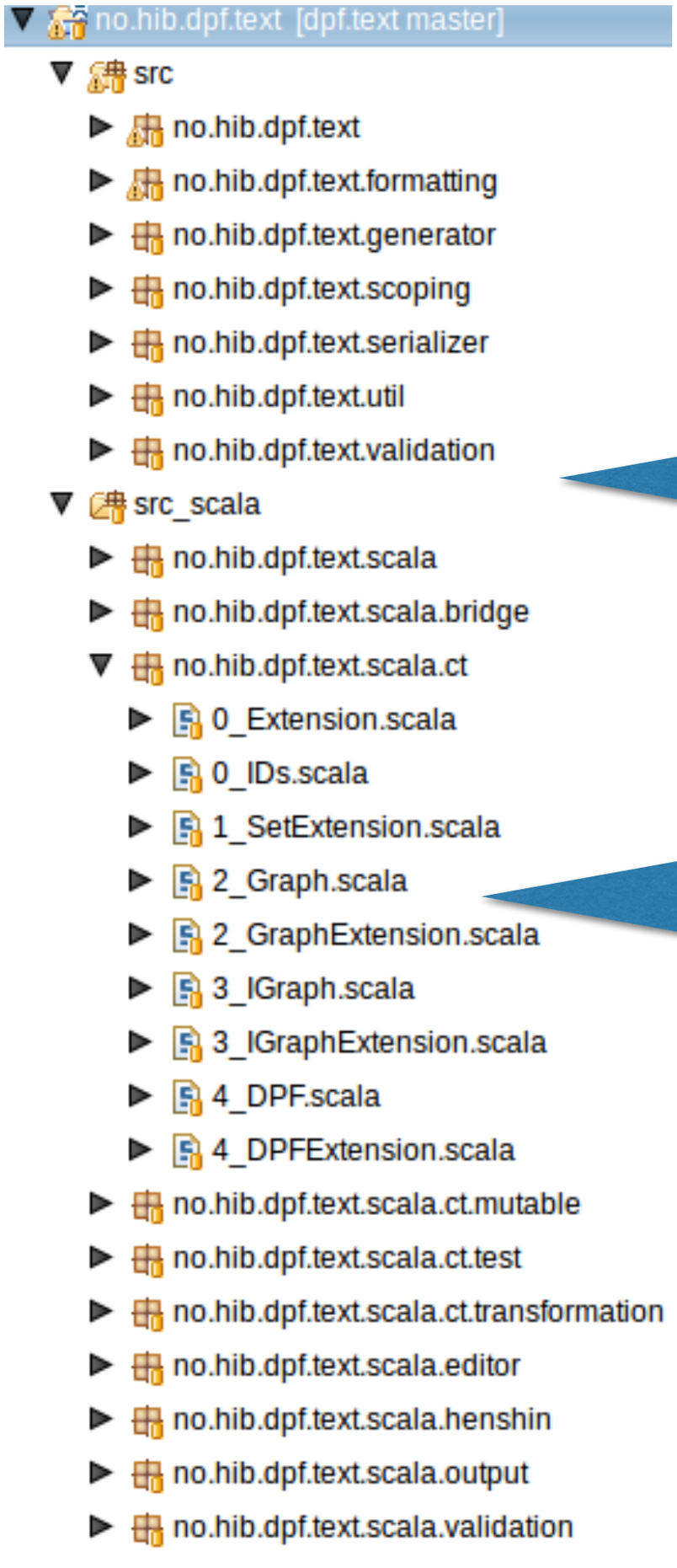

Graphs are internally translated to EMF, and Henshin Rules (https://www.eclipse.org/henshin/)

# More Features

- All elements have IDs which allow to compare different versions of specifications easily.
- IDs are sets of Integers values: this allows to trace even „merges“ of elements. The ID of the new element can consist of the IDs of the old elements.

```
Specification<version="1",16,
 Graph {
    Arrow@12:Class@0{
       weight@13,14:*->Int
    },
```

- Morphisms like the typing morphism are inferred by IDs only i.e. you can **change names of elements** as you like. Instances will find their new type names automatically.




Code is „strictly“ divided into a Java part and a Scala part

Categorical constructions are implemented as defined in the literature
(not very efficient but good for education!)

# Code Examples (1)

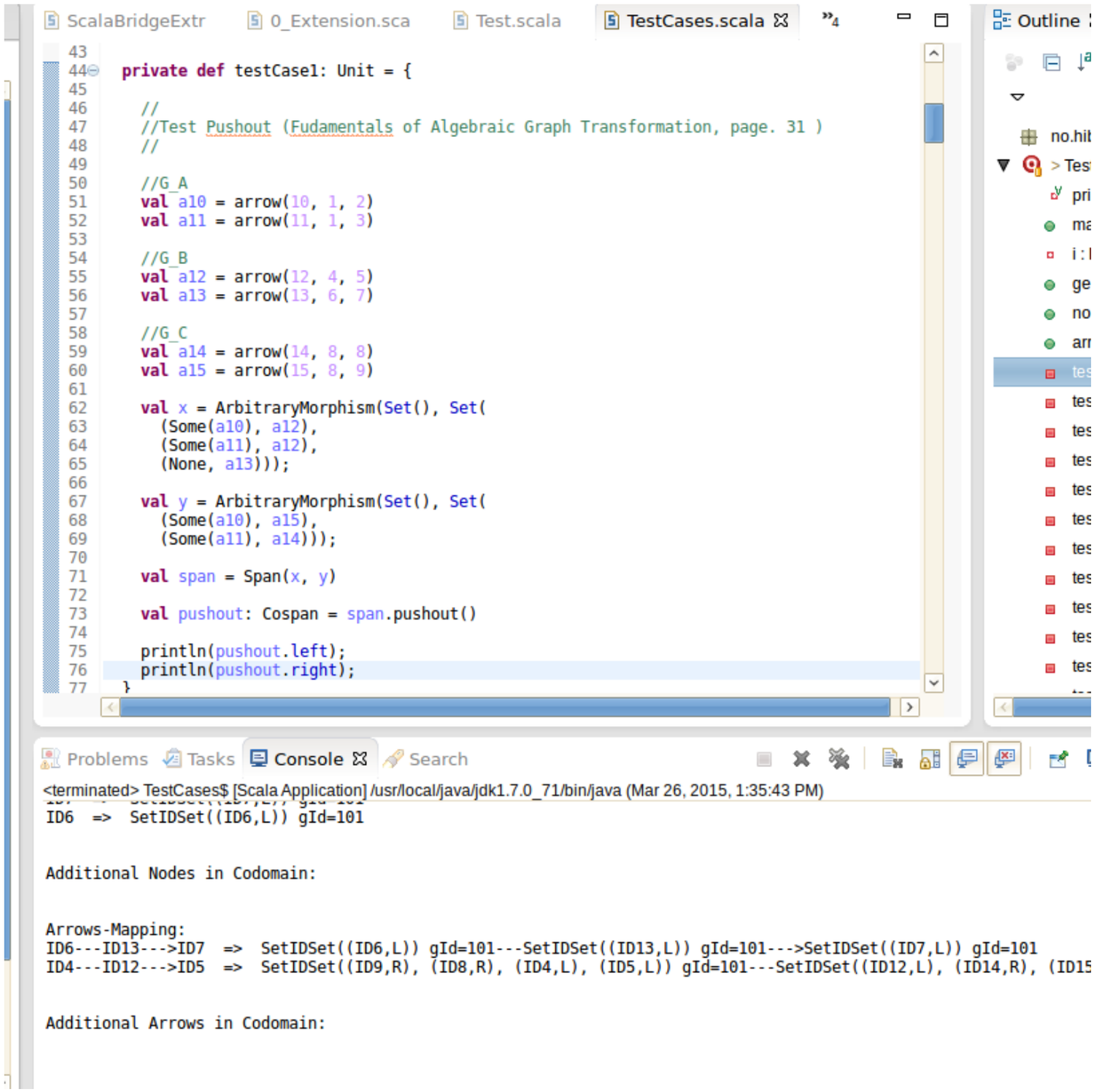
DPFTextCore.jav | ScalaBridgeExtr | 0_Extension.sca

```scala
package no.hib.dpf.text.scala.ct;

/**
 * Group Ids are used to create new Ids in the constructions.
 * They help to construct name spaces.
 */
trait TMorphism{

  val domainGId:Option[Int]=None
  val codomainGId:Option[Int]=None

  def isMono(): Boolean
  def isEpi(): Boolean
  def isIso(): Boolean
  def validate(): Boolean

}

trait TSpan{
  def pushout(gId: Int = GroupIdGen.gen()): TCospan
  def validate(): Boolean
}

trait TCospan{
  def pullback(gId: Int = GroupIdGen.gen()): TSpan
  def validate(): Boolean
}

trait TComposition{
 def finalPullbackComplementMono(gId: Int = GroupIdGen.gen()): TComposition
 def pushoutComplement(gid: Int = GroupIdGen.gen()): TComposition
 def validate(): Boolean
}

```

# Code Examples (2)

```
43  /**
44   * A directed multi-graph
45   */
46  case class Graph(override val tKey: FKey,
47    override val nodes: IMap[EId, Node],   //Set
48    override val arrows: IMap[EId, Arrow], //Set
49    override val in: IMap[Node, IMap[TypeArrow, ISet[Arrow]]],
50    override val out: IMap[Node, IMap[TypeArrow, ISet[Arrow]]],
51    override val names: IMap[EId, String]) extends AbstractGraph() {
52    override lazy val toString = super.toString
53  };
```

# Code Examples (3)



# Installation

1. Install Eclipse **Luna in the „modeling version“**

2. Install **XText** via „Install modeling components“

3. Update XText to version **2.8.1** (Simply search for updates)

4. Install **Eclipse Henshin** via the update site
   http://download.eclipse.org/modeling/emft/henshin/updates/release

5. Install **Scala IDE** via the update site
   http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site

6. Install **GraphViz** and make the dot command available to run it from the console/terminal
   (http://www.graphviz.org/)

7. Install the **DPF Text** plugin from your local update site
   {LOCAL PATH ON YOUR COMPUTER}/no.hib.dpf.text.updatesite/target/site

8. Open the DPF Perspective and try example:  „no.hib.dpf.text.example“

**Have fun :-)**

# Development

1. Follow the **installation instruction Item 1 to 6**

2. ***Do your changes! Add your extension!***

3. You can try your changes already via **run as Eclipse Application**

4. Make a new update site:

    1. **Install Maven 3 on your computer**

    2. Simply open a terminal and run
       **„mvn clean install“** in the directory no.hib.dpf.text.releng

       **Note:** works only if the plugin has been build in the Eclipse IDE before!
       This means the projects need to be imported into your workspace.

    3. Now you can **install your plugin as in Item 7 of the installation instruction**