

# 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 grammar 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

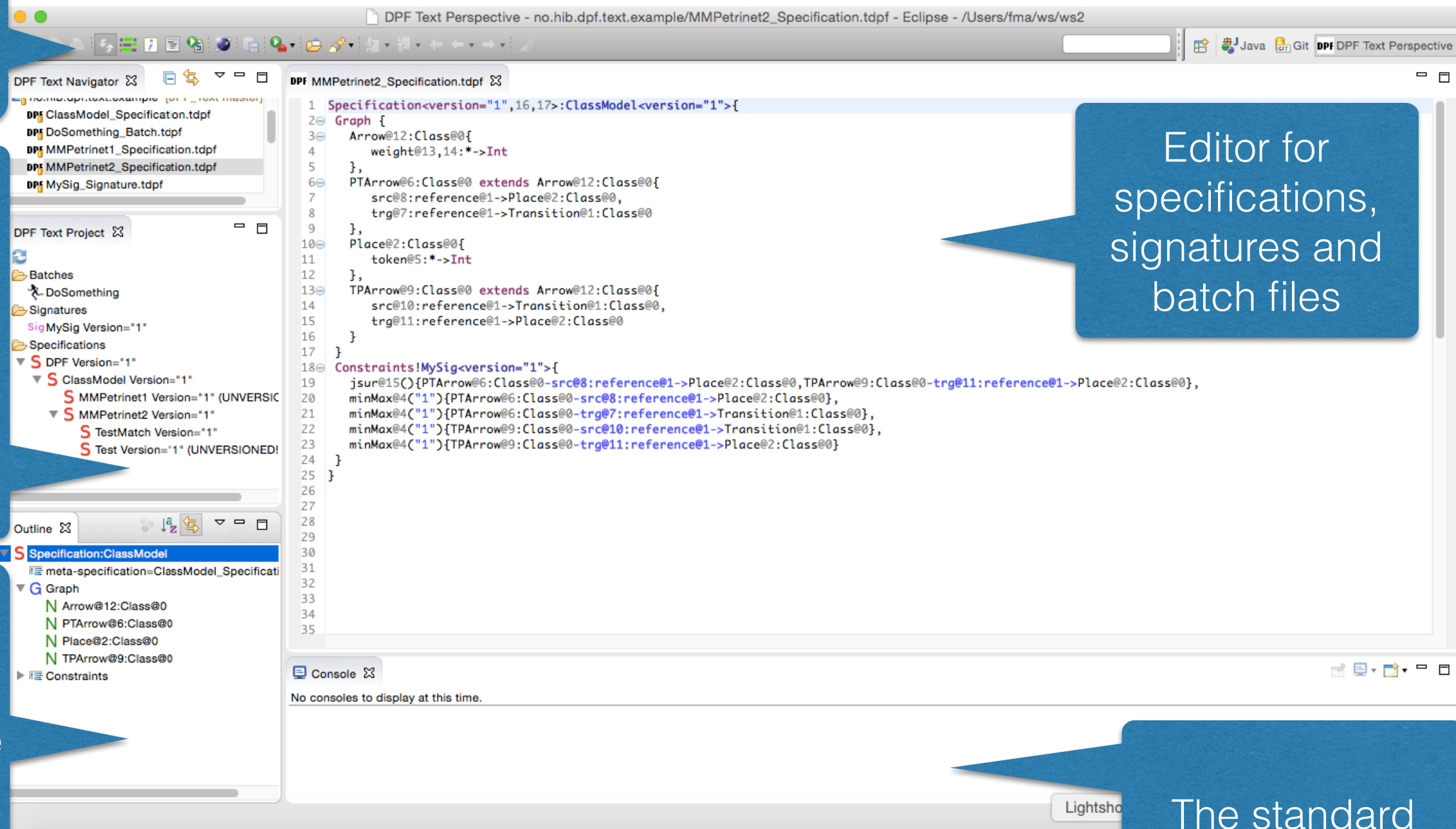
Navigator showing all projects

A project navigator showing the basic concepts in the selected project

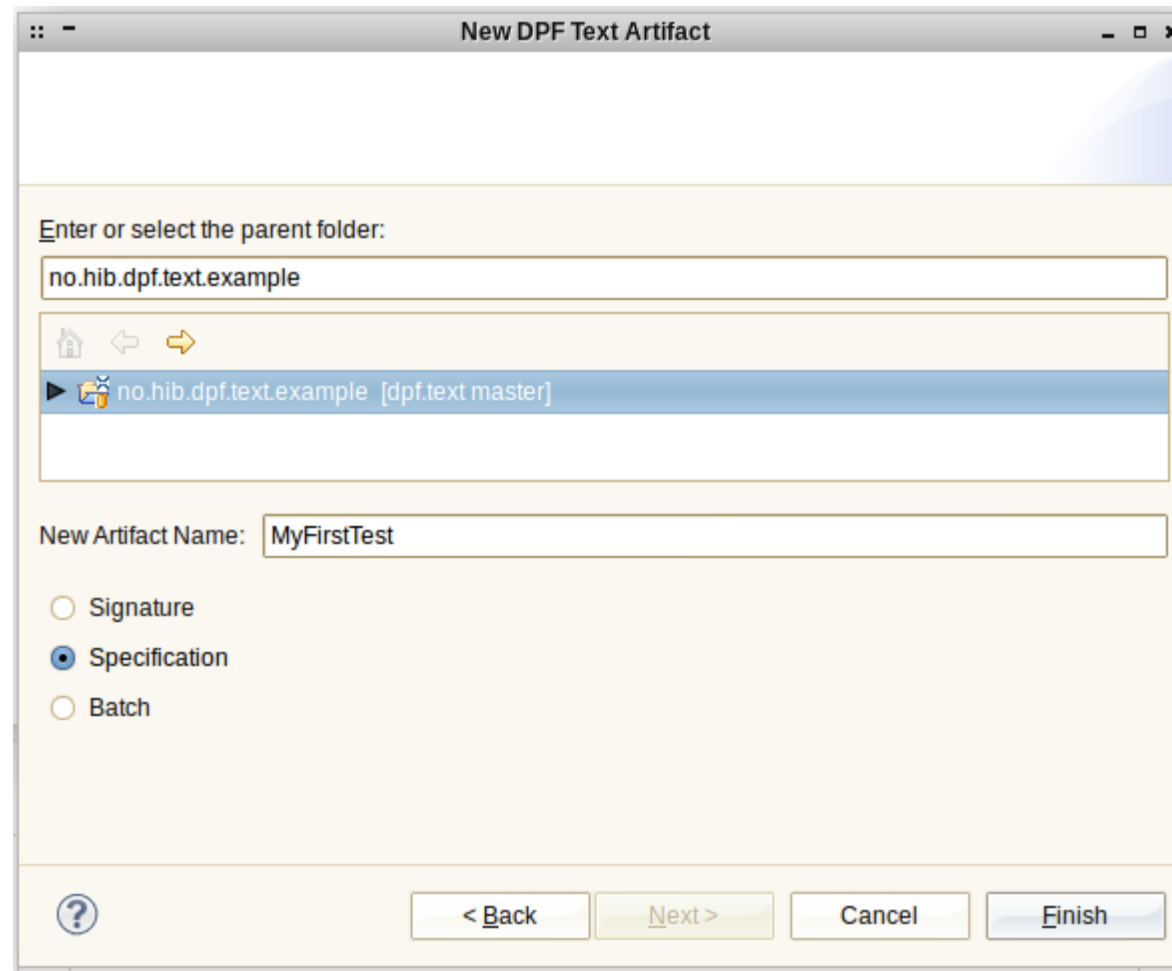
An outline of the current file in the editor

Editor for specifications, signatures and batch files

The standard console.

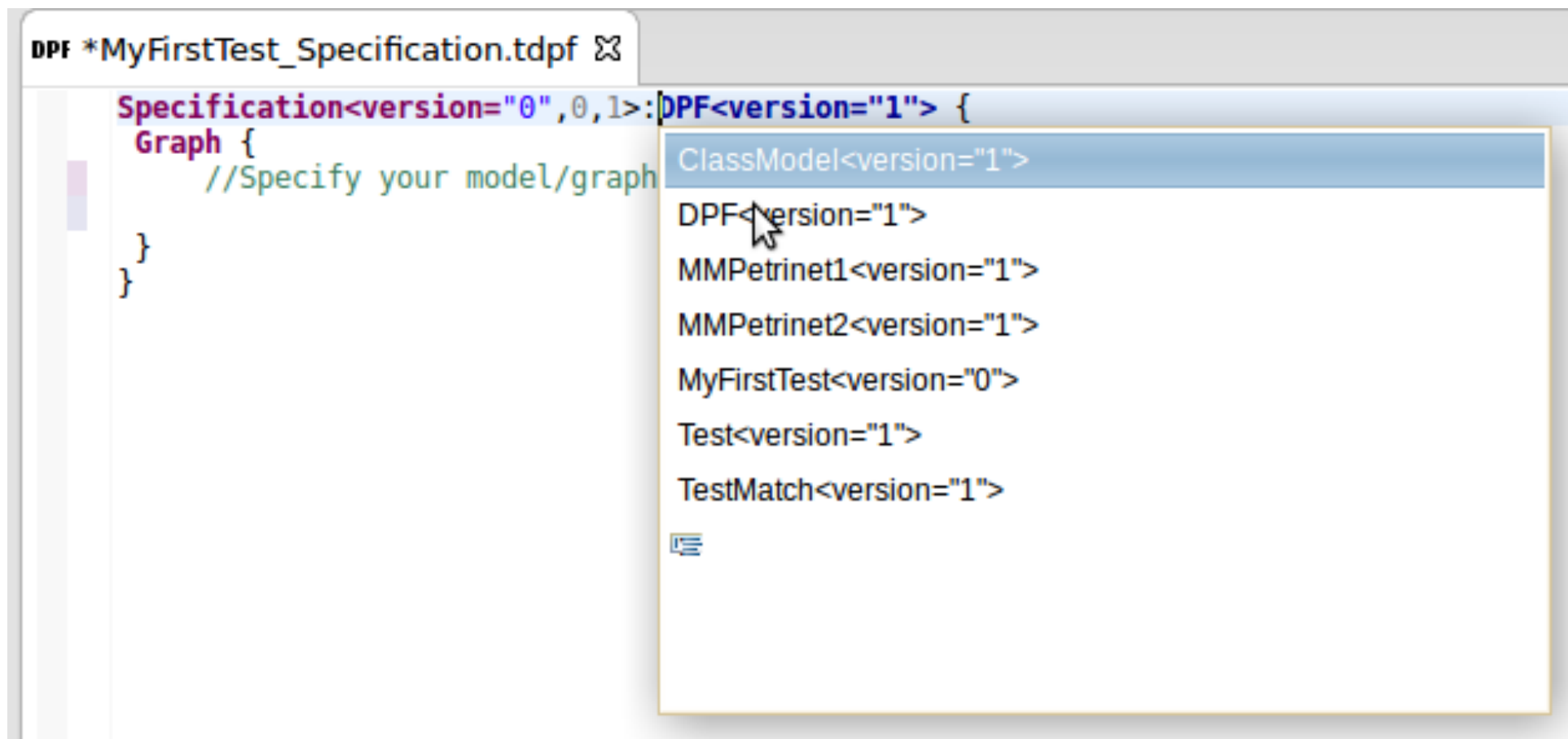


# Create New Specification (1)



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

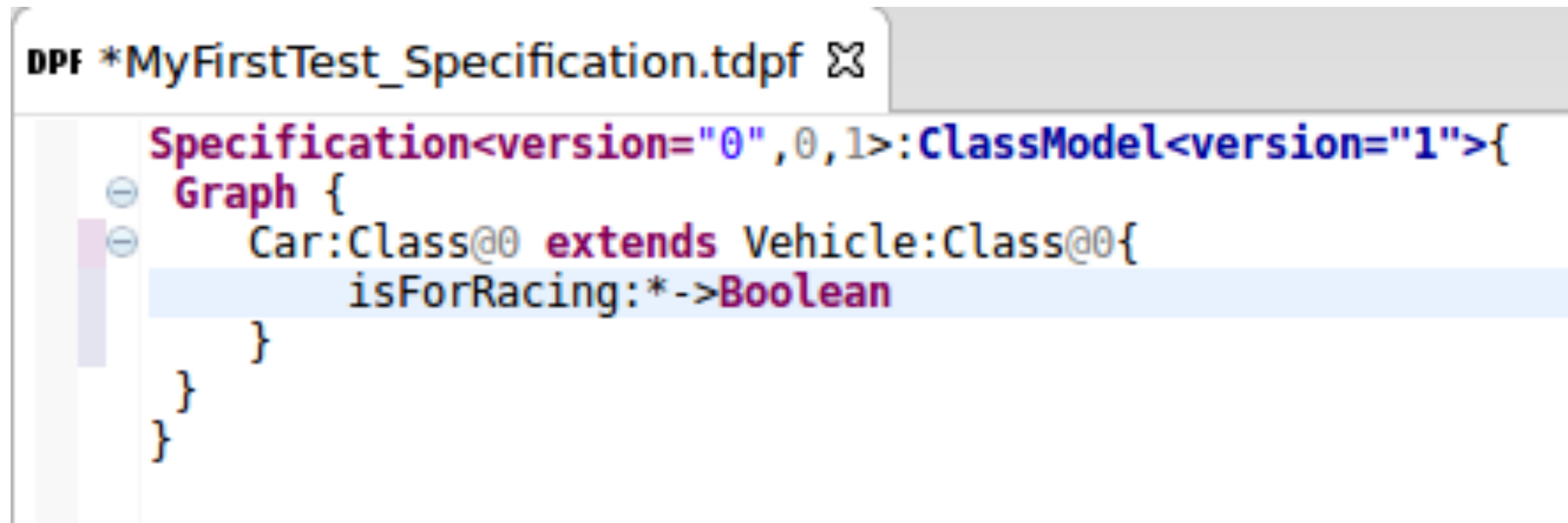
# Create New Specification (2)



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

```
1 Signature<version="1",15,16><OCL> {
2   abstract@1(){x:_}="context #x# inv: false" errorMsg="instance of an abstract element",
3   irr@2(){x:_-y:_->z:_}="context #x# inv: not #y#->includes(self)" errorMsg="[irr] constraint violated",
4   jsur@15(){x1:_-y1:_->z:_ ,x2:_-y2:_->z:_}="context #z# inv: (not 0#y1#->isEmpty()) or (not 0#y2#->isEmpty())" errorMsg="[jsur] constraint violated",
5   min@3(min){x:_-y:_->z:_}="context #x# inv: #y#->size() >= #min#" errorMsg="[min] constraint violated",
6   minMax@4(minMax){x:_-y:_->z:_}="context #x# inv: #y#->size() = #minMax#" errorMsg="[min_max] constraint violated",
7   sur@5(){x:_-y:_->z:_}="context #z# inv: not 0#y#->isEmpty()" errorMsg="[sur] constraint violated"
8 }
```

- 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](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  
    },  
    PTA@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  
    },  
    TPA@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(){PTA@6:Class@0-src@8:reference@1->Place@2:Class@0,TPA@9:Class@0-trg@11:reference@1->Place@2:Class@0},  
    minMax@4("1"){PTA@6:Class@0-src@8:reference@1->Place@2:Class@0},  
    minMax@4("1"){PTA@6:Class@0-trg@7:reference@1->Transition@1:Class@0},  
    minMax@4("1"){TPA@9:Class@0-src@10:reference@1->Transition@1:Class@0},  
    minMax@4("1"){TPA@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 *Arrows* have cardinality = 1,
2. *[jsur]*: each *Place* must be the source or target of an arrow.

# Specification Validation (1)

The screenshot displays a software interface for Petri net modeling. On the left, a project explorer shows a file named 'Test\_Specification.tdpf' selected. Below it, an 'Outline' pane shows a tree structure with 'Specification:MMPetrinet2' expanded, and a 'Graph' element. A context menu is open over the 'Graph' element, showing 'Open' and 'Validate' options, with 'Validate' being the active choice. The main area on the right is a code editor showing the content of 'Test\_Specification.tdpf'. The code is a Petri net specification in a text-based format. It includes a header comment, a version declaration, and a 'Graph' block. The 'Graph' block contains several elements: a 'Lonely' place, two 'PTArrow' elements ('a1' and 'a2'), and a 'TPArrow' element ('a3'). The 'Lonely' place is defined as 'Lonely@115:Place@2,'. The 'a1' arrow is defined as 'a1@105:PTArrow@6{ s@112:src@8->p1@106:Place@2, t@111:trg@7->t1@102:Transition@1 },'. The 'a2' arrow is defined as 'a2@103:PTArrow@6{ s@110:src@8->p2@104:Place@2, t@109:trg@7->t1@102:Transition@1 },'. The 'a3' arrow is defined as 'a3@101:TPArrow@9{ s@108:src@10->t1@102:Transition@1 },'. The 'p1' place is defined as 'p1@106:Place@2{ t@114:token@5->"1" },'. The 'p2' place is defined as 'p2@104:Place@2{ f@116:token@5->"1", t@113:token@5->"1" },'. The code is color-coded, with places in blue, transitions in green, and arrows in red. There are two red 'X' marks in the left margin of the code editor, indicating errors. Two blue callout boxes provide explanations for these errors. The first callout box points to the 'Lonely' place and states: 'The „Lonely“ place is not target of an arrow.' The second callout box points to the 'a3' arrow and states: 'Arrow „a3“ misses a „trg“ reference'.

```
//Save as MPetriNet Specification.tpf: (DPF Text Ar
Specification<version="1",1,121>:MMPetrinet2<versio
Graph {
  Lonely@115:Place@2,
  a1@105:PTArrow@6{
    s@112:src@8->p1@106:Place@2,
    t@111:trg@7->t1@102:Transition@1
  },
  a2@103:PTArrow@6{
    s@110:src@8->p2@104:Place@2,
    t@109:trg@7->t1@102:Transition@1
  },
  a3@101:TPArrow@9{
    s@108:src@10->t1@102:Transition@1
  },
  p1@106:Place@2{
    t@114:token@5->"1"
  },
  p2@104:Place@2{
    f@116:token@5->"1",
    t@113:token@5->"1"
  }
}
```

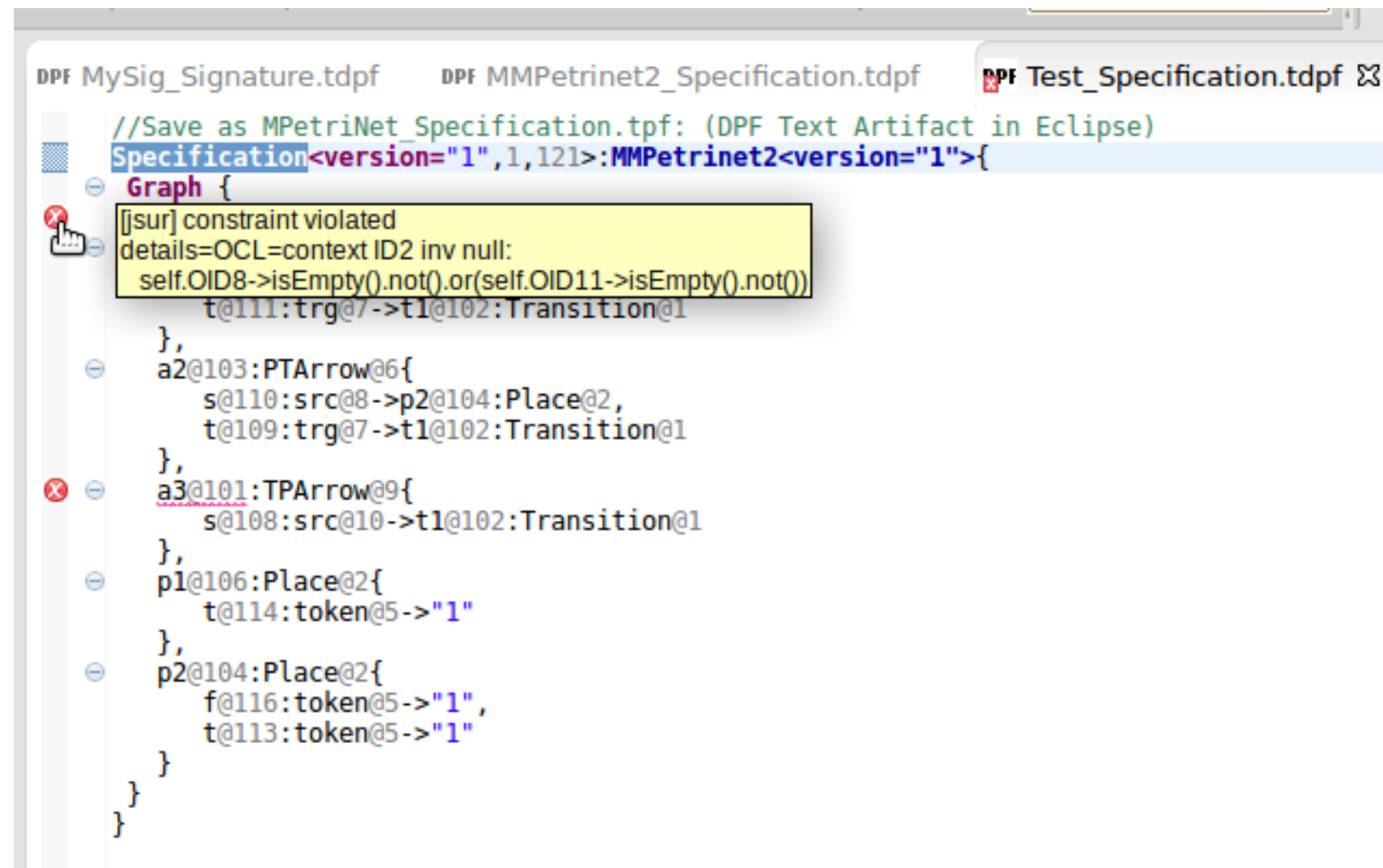
The „Lonely“ place is not target of an arrow.

Arrow „a3“ misses a „trg“ reference

Also on *save* models are automatically validated.



# Specification Validation (2)



The screenshot shows the Eclipse IDE with three tabs: 'DPF MySig\_Signature.tdpf', 'DPF MMPetrinet2\_Specification.tdpf', and 'DPF Test\_Specification.tdpf'. The 'MMPetrinet2\_Specification.tdpf' tab is active, displaying a DPF specification. A yellow tooltip is visible over the 'Graph' section, indicating a constraint violation. The specification text is as follows:

```
//Save as MPetriNet_Specification.tpf: (DPF Text Artifact in Eclipse)
Specification<version="1",1,121>:MMPetrinet2<version="1">{
  Graph {
    [jsur] constraint violated
    details=OCL=context ID2 inv null:
    self.OID8->isEmpty().not().or(self.OID11->isEmpty().not())
    t@111:trg@7->t1@102:Transition@1
  },
  a2@103:PTArrow@6{
    s@110:src@8->p2@104:Place@2,
    t@109:trg@7->t1@102:Transition@1
  },
  a3@101:TPArrow@9{
    s@108:src@10->t1@102:Transition@1
  },
  p1@106:Place@2{
    t@114:token@5->"1"
  },
  p2@104:Place@2{
    f@116:token@5->"1",
    t@113:token@5->"1"
  }
}
```

# Specifications can be exported into different Formats

DPF DoSomething\_Batch.tdpf

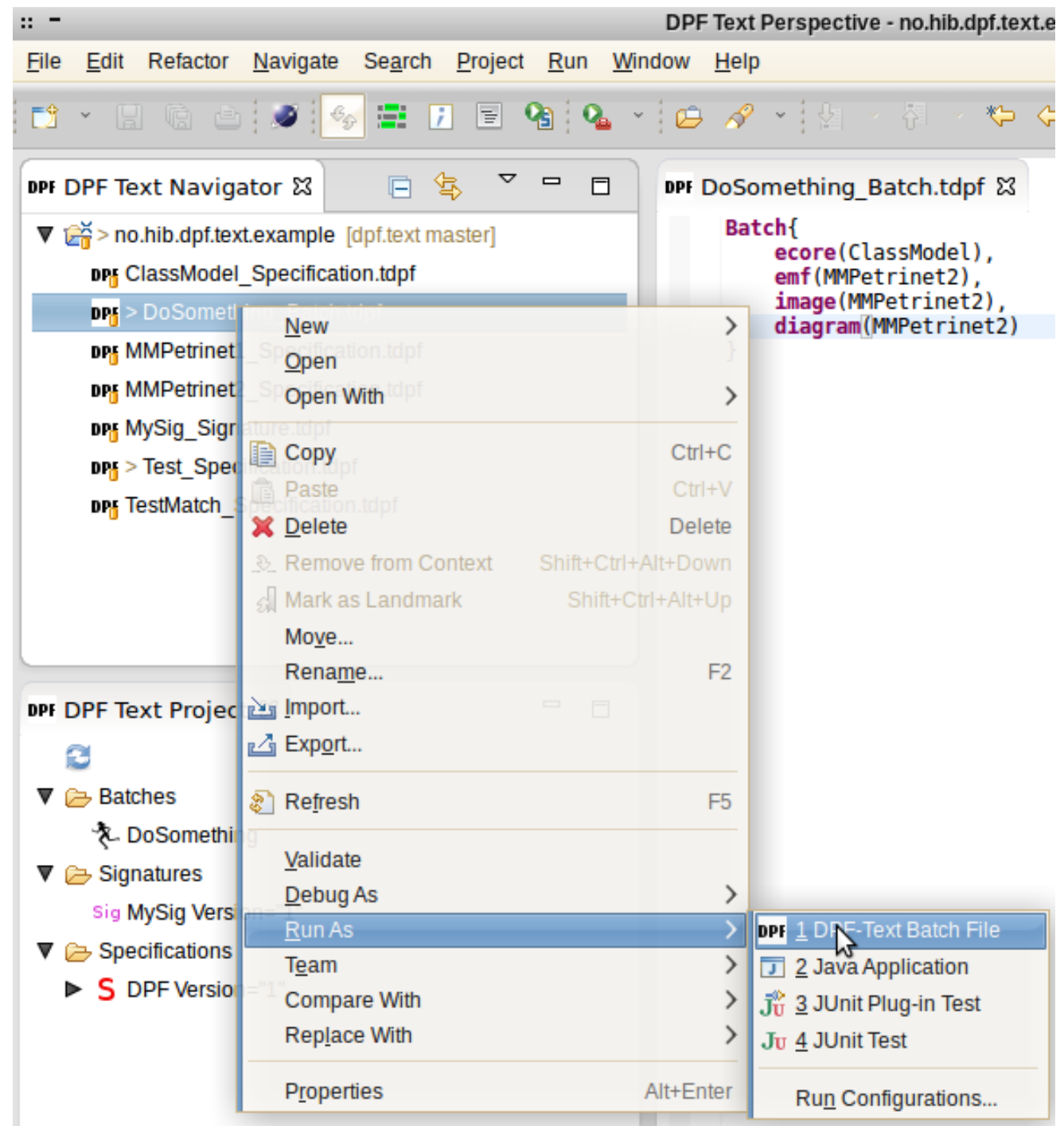
```
Batch{  
    ecore(ClassModel),  
    emf(MMPetrinet2),  
    image(MMPetrinet2),  
    diagram(MMPetrinet2)  
}
```

input

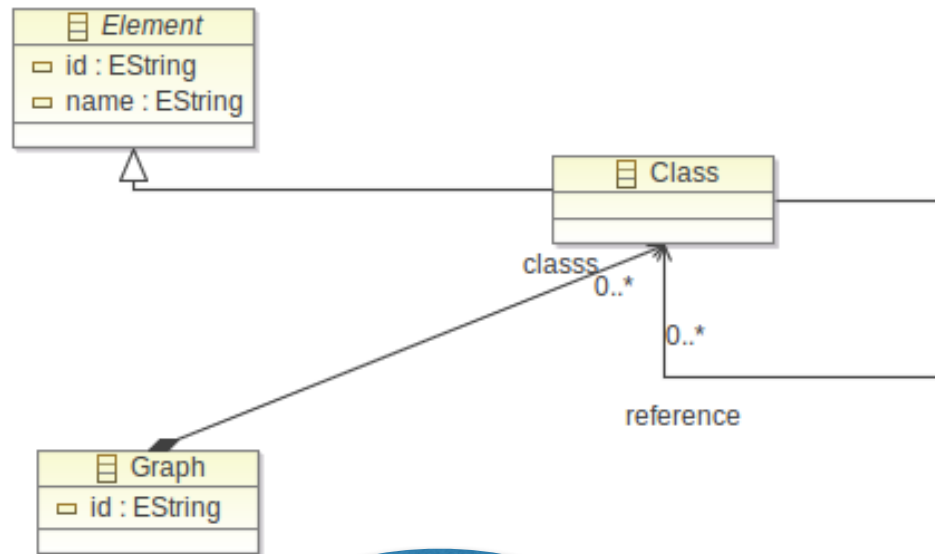
no.hib.dpf.text.example [dpf.text master]

- 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

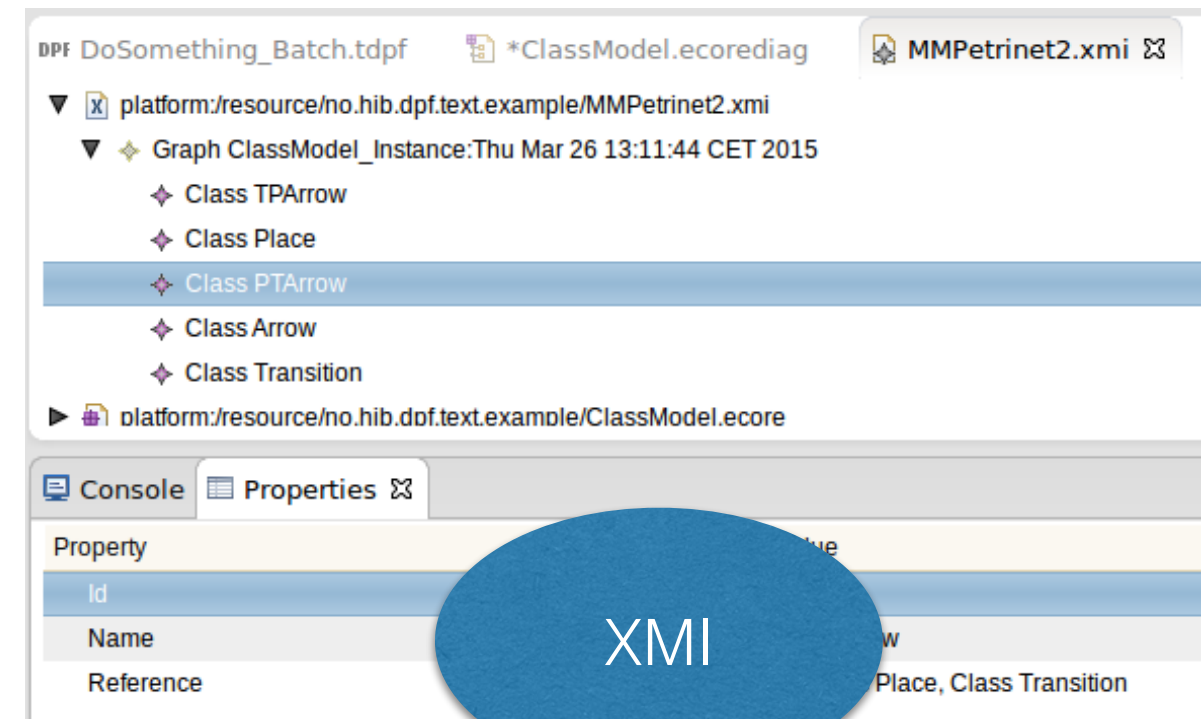
output



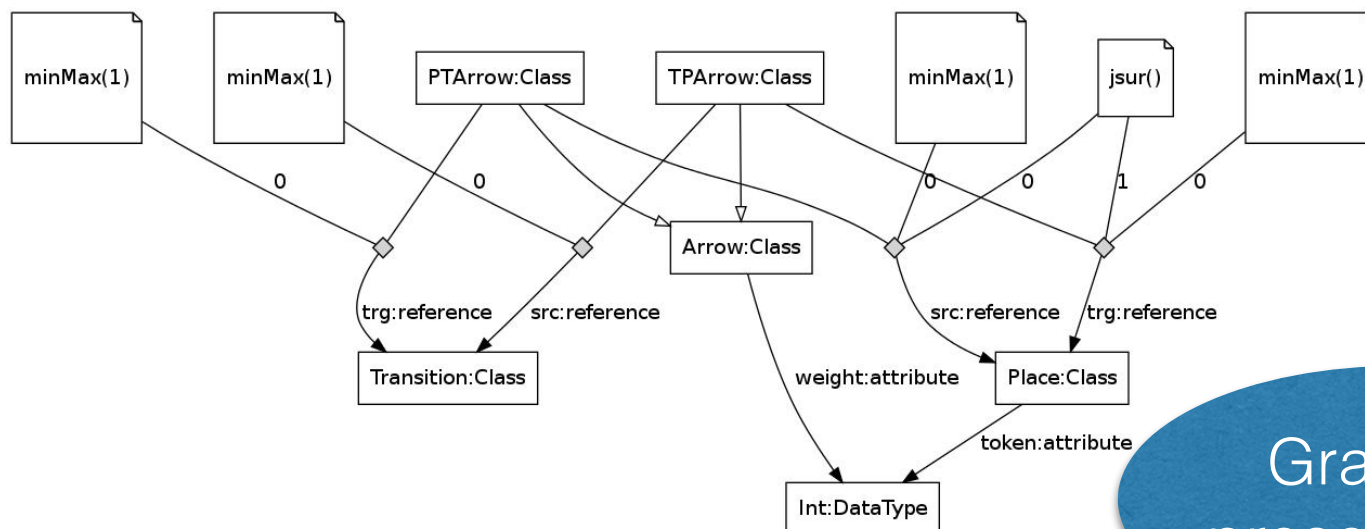
# Exported Specifications



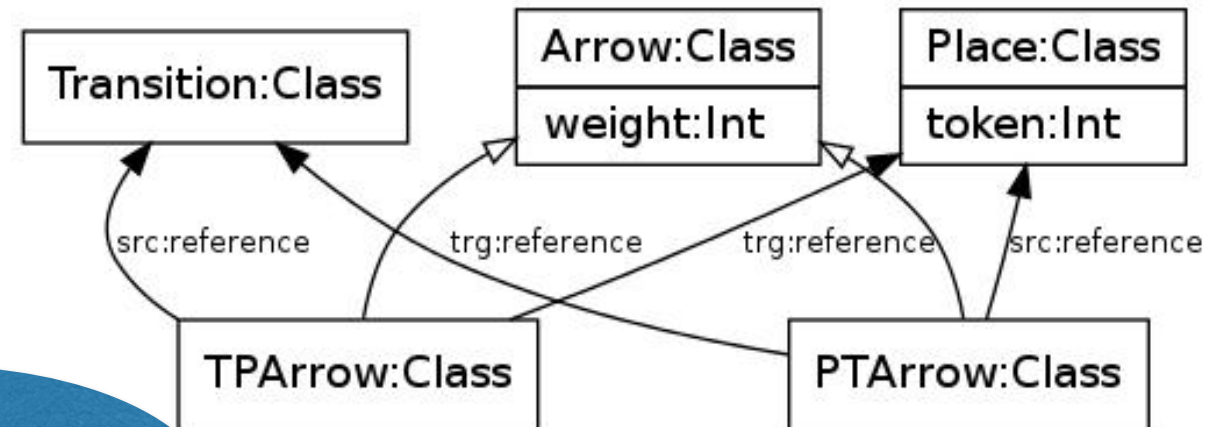
Ecore



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 {  
    PTAArrow@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  
    },  
    TPAArrow@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“

```
Specification<version="1",16,17>:ClassModel<version="1">{  
  Graph {
```

„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!*

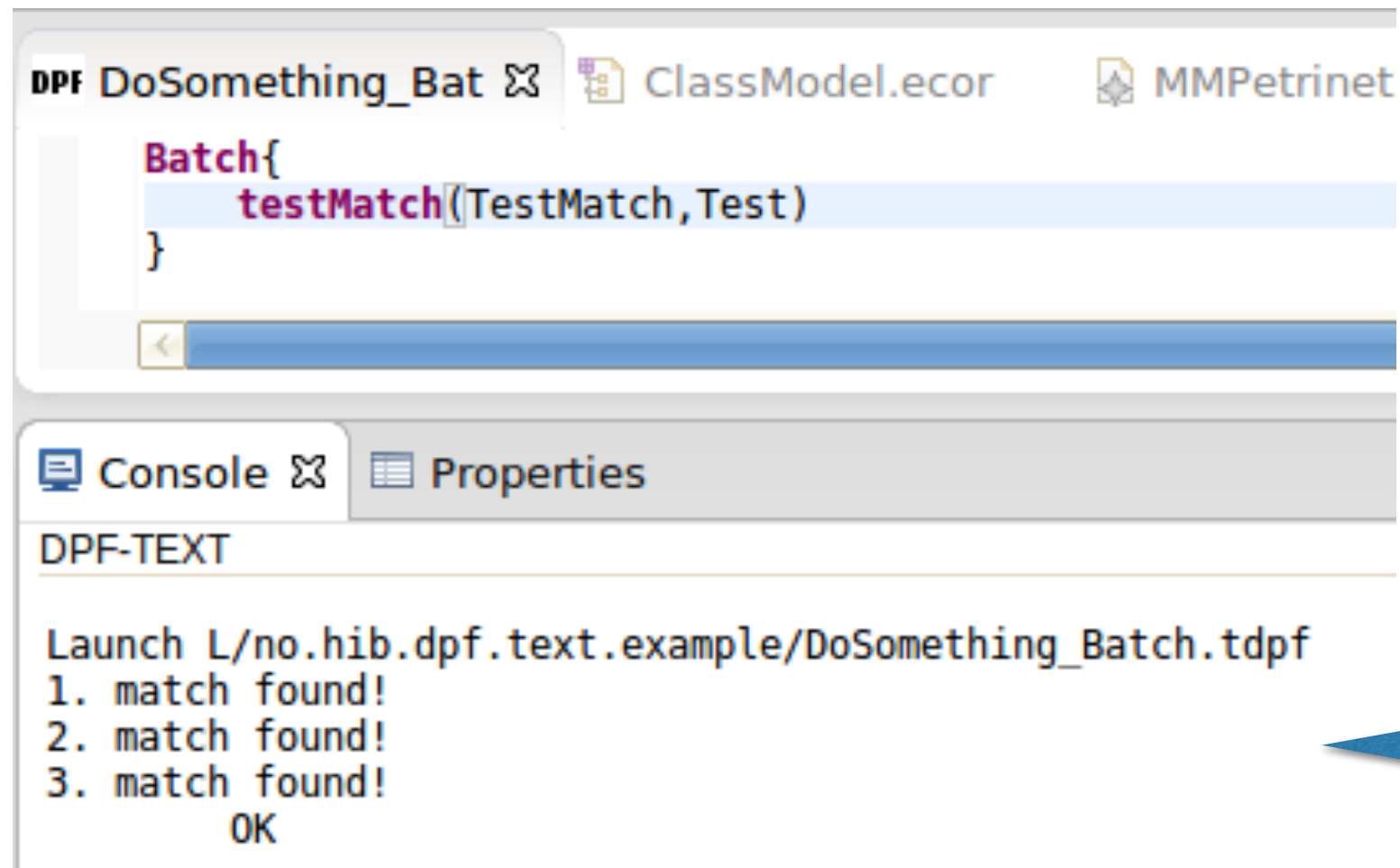
Tool support for versioning has been planned but not implemented yet (Diff + Merge).

```
▼ S DPF Version="1"  
  ▼ S ClassModel Version="1"  
    S MMPetrinet1 Version="1" (UNVERSIONED!)  
    ► S MMPetrinet2 Version="1"
```

```
DPF MMPetrinet1_Specification.tdpf ✕  
1 Specification<version="1",1,7>:ClassModel<version="1">{
```

$7 - 1 \neq 1$  (next - lastFromVersion)

# Prepared for implementing Algebraic Graph Transformations



```
DPF DoSomething_Bat ✕ ClassModel.eor MMPetriNet

Batch{
  testMatch[TestMatch,Test]
}

Console ✕ Properties
DPF-TEXT

Launch L/no.hib.dpf.text.example/DoSomething_Batch.tdpf
1. match found!
2. match found!
3. match found!
OK
```

Graphs can be  
searched in other  
graphs!

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.

▼ no.hib.dpf.text [dpf.text master]

▼ src

- ▶ no.hib.dpf.text
- ▶ no.hib.dpf.text.formatting
- ▶ no.hib.dpf.text.generator
- ▶ no.hib.dpf.text.scoping
- ▶ no.hib.dpf.text.serializer
- ▶ no.hib.dpf.text.util
- ▶ no.hib.dpf.text.validation

▼ src\_scala

- ▶ no.hib.dpf.text.scala
- ▶ no.hib.dpf.text.scala.bridge
- ▼ no.hib.dpf.text.scala.ct
  - ▶ 0\_Extension.scala
  - ▶ 0\_IDs.scala
  - ▶ 1\_SetExtension.scala
  - ▶ 2\_Graph.scala
  - ▶ 2\_GraphExtension.scala
  - ▶ 3\_IGraph.scala
  - ▶ 3\_IGraphExtension.scala
  - ▶ 4\_DPF.scala
  - ▶ 4\_DPFExtension.scala
- ▶ no.hib.dpf.text.scala.ct.mutable
- ▶ no.hib.dpf.text.scala.ct.test
- ▶ no.hib.dpf.text.scala.ct.transformation
- ▶ no.hib.dpf.text.scala.editor
- ▶ no.hib.dpf.text.scala.henshin
- ▶ no.hib.dpf.text.scala.output
- ▶ no.hib.dpf.text.scala.validation

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!)



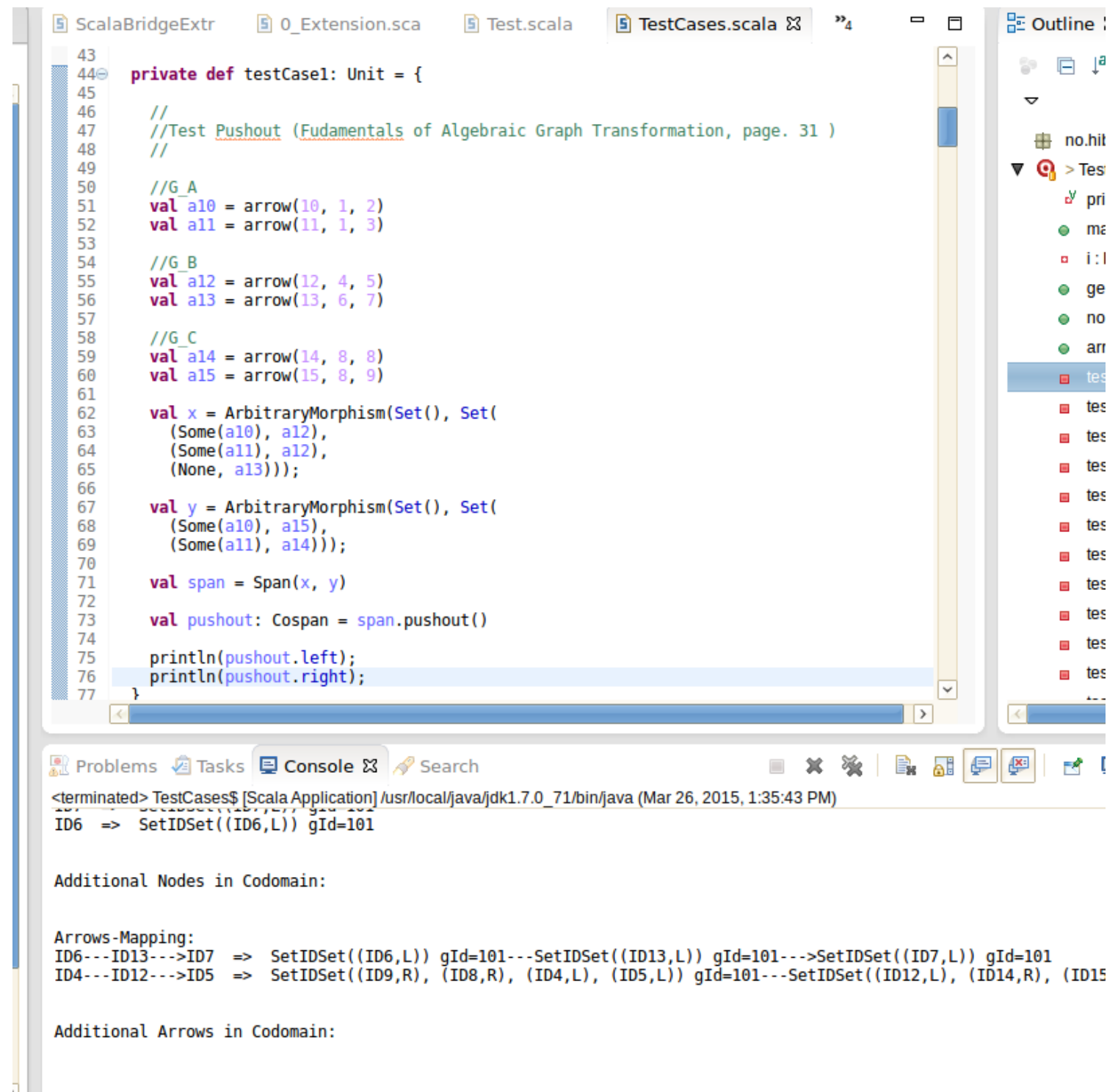
# Code Examples (1)

```
DPFTextCore.jav  ScalaBridgeExtr  0_Extension.sca  »3
1 package no.hib.dpf.text.scala.ct;
2
3 /**
4  * Group Ids are used to create new Ids in the constructions.
5  * They help to construct name spaces.
6  */
7 trait TMorphism{
8
9     val domainGId:Option[Int]=None
10    val codomainGId:Option[Int]=None
11
12    def isMono(): Boolean
13    def isEpi(): Boolean
14    def isIso(): Boolean
15    def validate(): Boolean
16
17 }
18
19 trait TSpan{
20     def pushout(gId: Int = GroupIdGen.gen()): TCospan
21     def validate(): Boolean
22 }
23
24 trait TCospan{
25     def pullback(gId: Int = GroupIdGen.gen()): TSpan
26     def validate(): Boolean
27 }
28
29 trait TComposition{
30     def finalPullbackComplementMono(gId: Int = GroupIdGen.gen()): TComposition
31     def pushoutComplement(gid: Int = GroupIdGen.gen()): TComposition
32     def validate(): Boolean
33 }
34
```

# 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)



```
ScalaBridgeExtr  O_Extension.sca  Test.scala  TestCases.scala  »4  Outline
```

```
43
44 private def testCase1: Unit = {
45
46     //
47     //Test Pushout (Fundamentals of Algebraic Graph Transformation, page. 31 )
48     //
49
50     //G_A
51     val a10 = arrow(10, 1, 2)
52     val a11 = arrow(11, 1, 3)
53
54     //G_B
55     val a12 = arrow(12, 4, 5)
56     val a13 = arrow(13, 6, 7)
57
58     //G_C
59     val a14 = arrow(14, 8, 8)
60     val a15 = arrow(15, 8, 9)
61
62     val x = ArbitraryMorphism(Set(), Set(
63         (Some(a10), a12),
64         (Some(a11), a12),
65         (None, a13)));
66
67     val y = ArbitraryMorphism(Set(), Set(
68         (Some(a10), a15),
69         (Some(a11), a14)));
70
71     val span = Span(x, y)
72
73     val pushout: Cospan = span.pushout()
74
75     println(pushout.left);
76     println(pushout.right);
77 }
```

no.hit

▼ Q > Tes

- pri
- ma
- i:l
- ge
- no
- arr
- tes
- tes
- tes
- tes
- tes
- tes
- tes
- tes
- tes
- tes
- tes
- tes
- tes
- tes
- tes

Problems Tasks Console Search

<terminated> TestCases\$ [Scala Application] /usr/local/java/jdk1.7.0\_71/bin/java (Mar 26, 2015, 1:35:43 PM)

ID6 => SetIDSet((ID6,L)) gId=101

Additional Nodes in Codomain:

Arrows-Mapping:

ID6---ID13--->ID7 => SetIDSet((ID6,L)) gId=101---SetIDSet((ID13,L)) gId=101--->SetIDSet((ID7,L)) gId=101

ID4---ID12--->ID5 => SetIDSet((ID9,R), (ID8,R), (ID4,L), (ID5,L)) gId=101---SetIDSet((ID12,L), (ID14,R), (ID15

Additional Arrows in Codomain:



# 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**