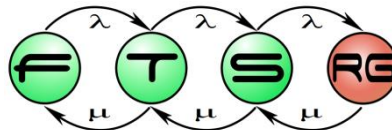


# Graphical Concrete Syntax Design for Domain-specific Languages

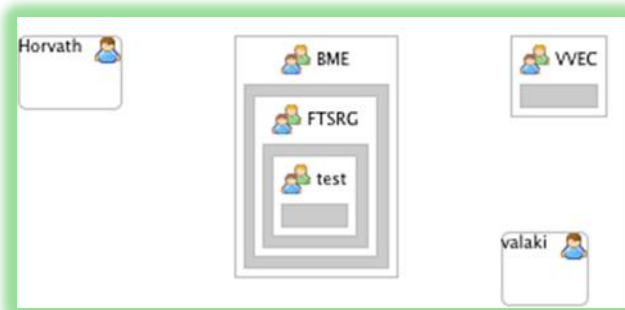
## Model Driven Software Development Lecture 6





# Structure of DSMs

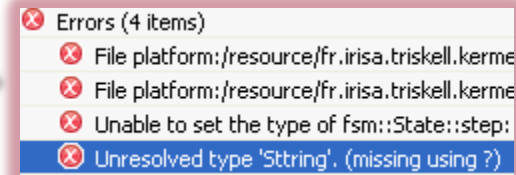
Graphical syntax



Abstract syntax



Well-formedness constraints



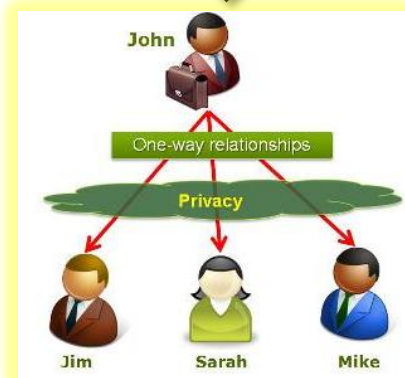
Behavioural semantics, simulation

Code generation

Mapping



Textual syntax



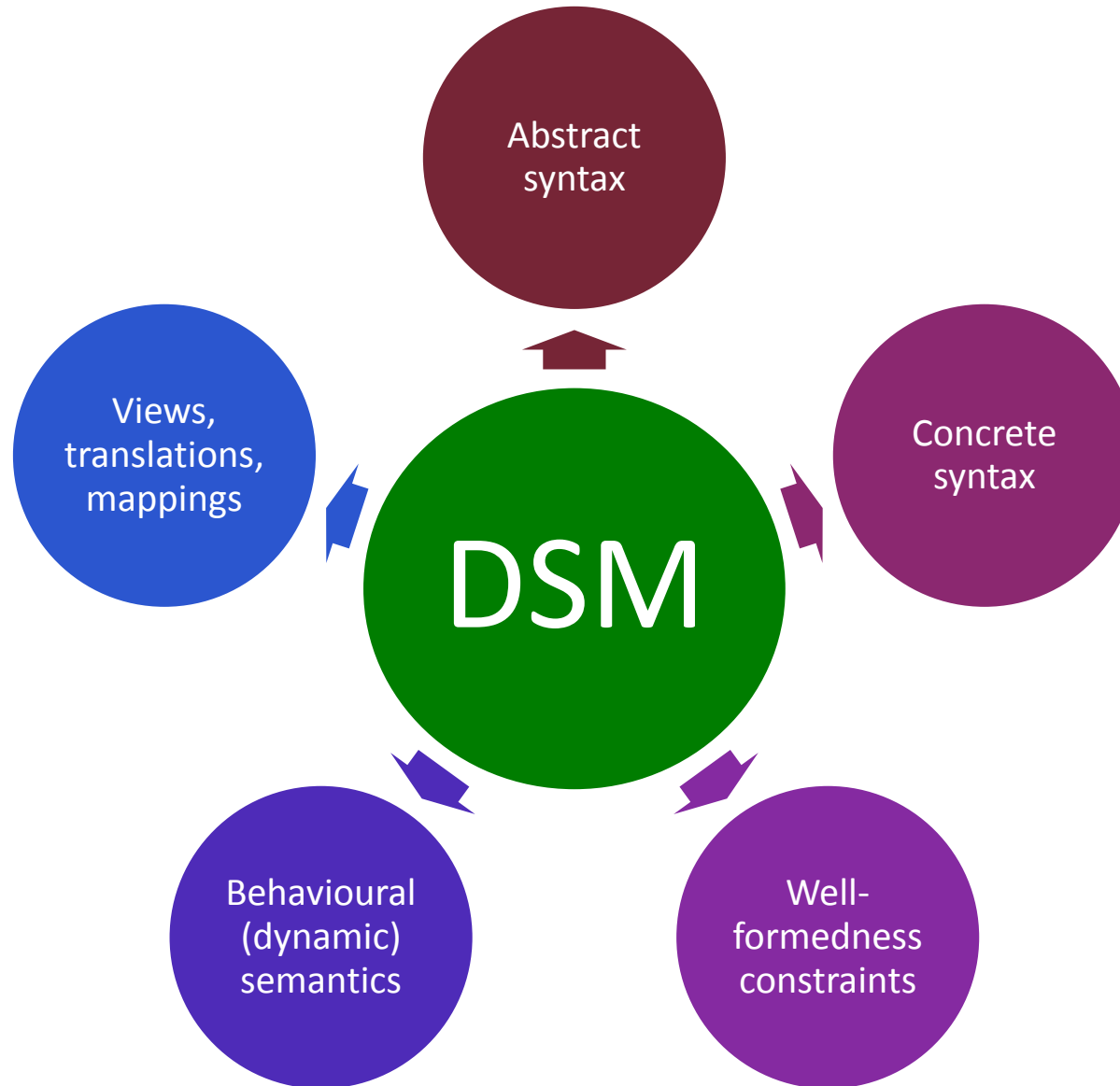
View

```
</membership>
<profile defaultProvider="Sitefinity">
  <providers>
    <clear/>
    <add name="Sitefinity" connectionS
  </providers>
  <properties>
    <add name="FirstName"/>
    <add name="LastName"/>
    <!-- SNP specific properties -->
    <add name="NickName" />
    <add name="Gender" />
  </properties>
</profile>
```

Code  
(documentation,  
configuration)



# DSM aspects





# Concrete Syntax Design

- User-managed parts of a modeling language
  - Performance
  - Robustness
  - Usability issues
- Creating model editors
  - Similar problems at programming languages
  - IDE extensions needed



# Approaches

- Textual syntax
  - Character-based edit operations (unless *projectional*)
  - Abstract syntax: traditional AST
- Graphical syntax
  - Editing operations: translated to abstract syntax
  - Abstract syntax: based on metamodel
- Form-based entry
  - Less common
  - Behaves similar to graphical syntax



# Advanced features

## High level editing support

- Outline view
- Documentation display (e.g. Javadoc)
- Templates/snippets/examples
- Content assist
- Validation, automatic fixes

## Project-level integration

- Code generation
- Wizards to create projects/files
- Integration with manually written code in programming language



# Technology

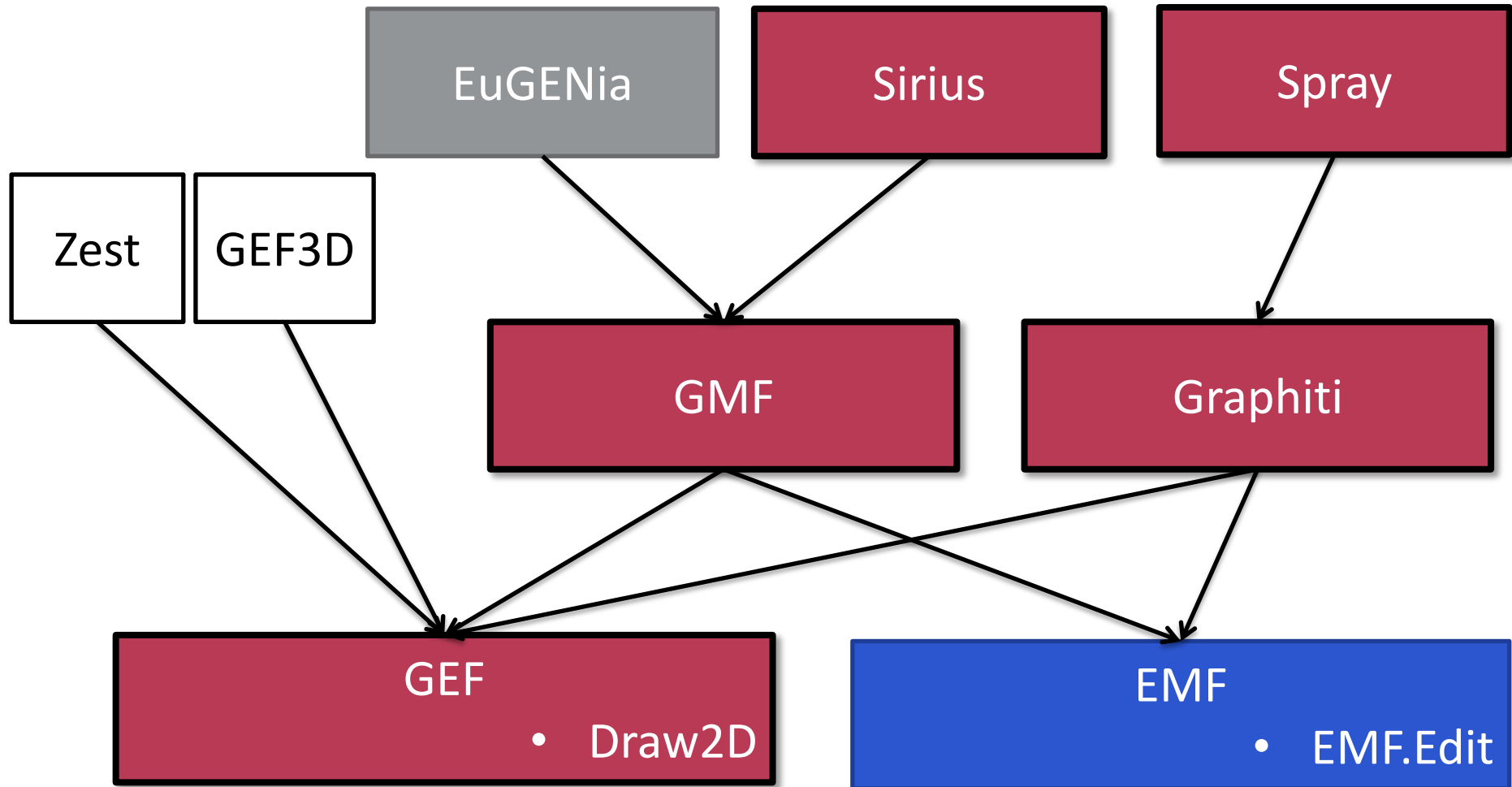
- Eclipse Modeling Tools
  - Several related subprojects
  - Each supports a single aspect
  - Examples of today
- Microsoft Visual Studio 2010 Visualization & Modeling SDK
  - DSL modeling framework from Microsoft
  - Own metamodeling core
  - Focuses on graphical modeling



# Graphical Editors



# Graphical Editor Techniques



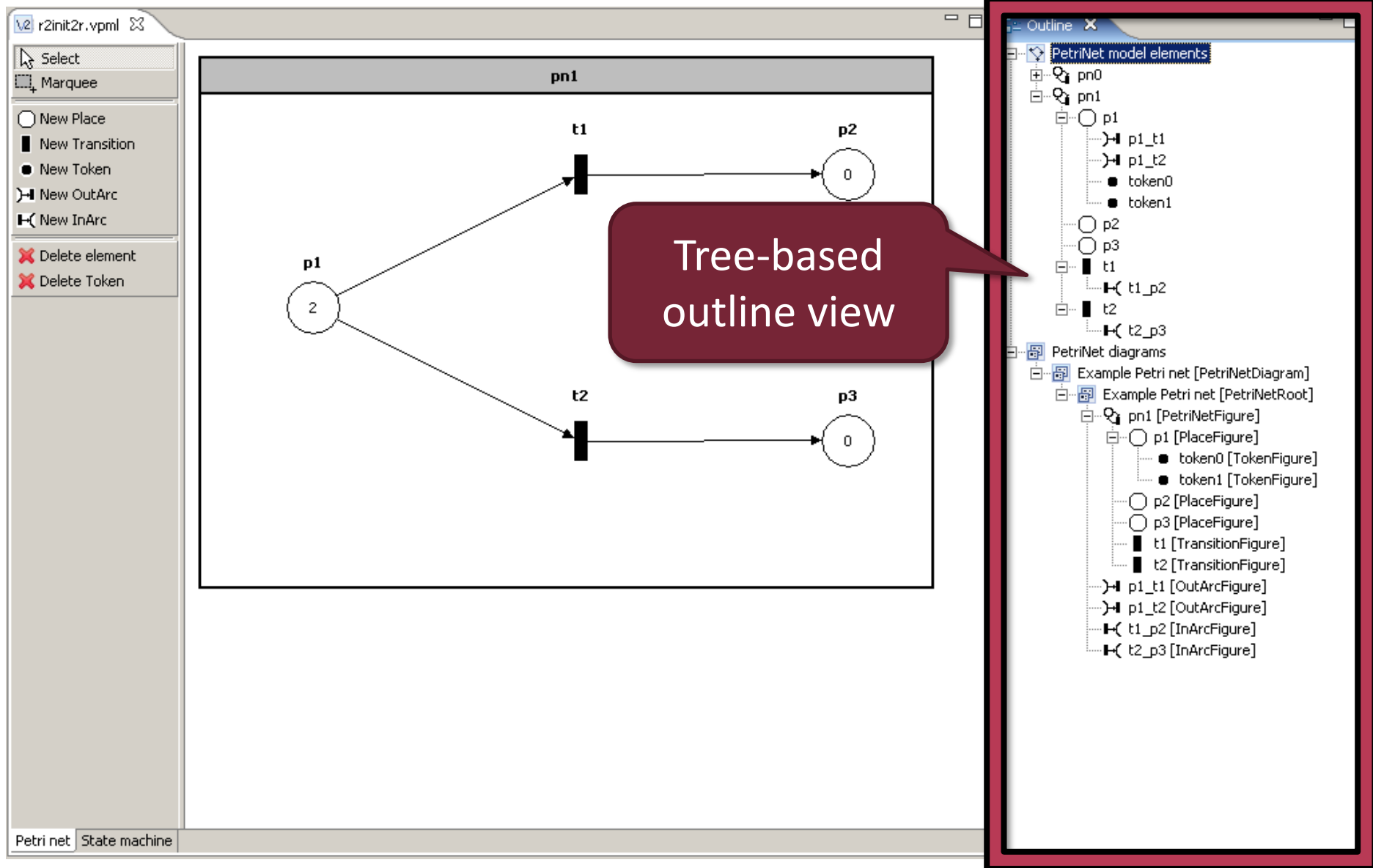


# Graphical Modeling

- Model
  - Typically graph-based modeling (Edges, Nodes)
  - In our case EMF
- Idea
  - Display and editing as a graph model



# Example: Petri net editor





# Example: Social Network editor

The screenshot displays the Eclipse IDE interface with a Social Network editor. The main workspace shows a graph with nodes like 'J. Random', 'Jane Doe', and 'John Doe', and containers like 'Bar Society' and 'Baz Community'. The interface includes several views and callouts:

- Project Explorer extensions:** A callout pointing to the Project Explorer view on the left, which shows a tree structure of the project.
- Graph outline view:** A callout pointing to the Outline view on the left, which shows a hierarchical list of the graph elements.
- Properties view:** A callout pointing to the Properties view at the bottom, which displays the attributes of the selected node.
- Palette:** A callout pointing to the Palette on the right, which contains tools for creating and editing graph elements.

The Properties view shows the following data:

Property	Value
Name	John Doe
Sex	male
X	677
Y	240



# Implementation

## ■ Presentation

- Based on a Canvas
- Using vector-graphic libraries (GEF/Draw2d)

## ■ Model manipulation

- EMF Edit model manipulation commands
  - Atomic operations: create/modify/remove node/edge
- Transactional modifications
  - Undo/redo support
  - Replayability



# Implementation 2.

## ■ View models

- Modeling for view-specific information
  - Coordinates
  - Size
  - Colors and fonts
  - ...
- Generic implementation in GMF and Graphiti
- Often stored in external files
  - Separation of concerns
  - E.g. code generator not interested in view information



# Technologies 1. - GEF

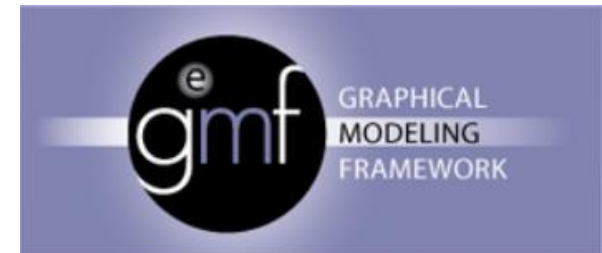
- Graphical Editing Framework (GEF)
  - “Low level” editor framework
  - Not EMF-specific
- Model-View-Controller approach
- Generic graph-based editor framework
  - Including undo/redo support
  - Graphical outlines
- Manual coding for every possible element
- GEF4 FX – JavaFX-based replacement of the core





# Technologies 2. – GMF

- Graphical Modeling Framework
- Based on GEF and EMF
- Well-separated view and domain models
  - Generic view model
  - Synchronization provided by GMF framework
- Relatively old technology
  - Widely used
  - Very complex to start





# Technologies 2. – GMF

- Model-driven development environment
  - Common model for graphical editors, using
    - Figure definition model
      - Basic symbol definition of the graphical language
    - Tooling model
      - Defining model manipulation commands
    - Mapping model
      - Mapping figures and tools to domain model
  - Fully functional editor can be generated
    - Problematic manual modifications
- Or a high-level editor framework
  - Manual coding





# Technologies 3. - Graphiti

- Newer high level graphical editor framework
  - Based on EMF and GEF
  - But: different approach then GMF
    - Simplified programmatic API
    - Manual coding
  - Idea
    - All Graphiti based editors should
      - Look similar
      - Behave similar





# Technologies 3. - Graphiti

- Development methodology
  - Coding over a high-level Java framework
    - Much simpler then GMF
    - Repetitive code needed
- Spray project
  - Textual modeling environment for graphical editors
  - Generates code over the Graphiti framework





# Technologies 4. - Sirius



- New modeling project
  - Since 2013 on eclipse.org
  - Previously Obeo Designer – commercial tool
- How stable is it?
  - Old projects are to be migrated
  - Version history
    - 0.9: 2013-12-10
    - 1.0: 2014-06-25 (Kepler release train)
    - ...
    - 4.0: 2016-06-22
    - ...



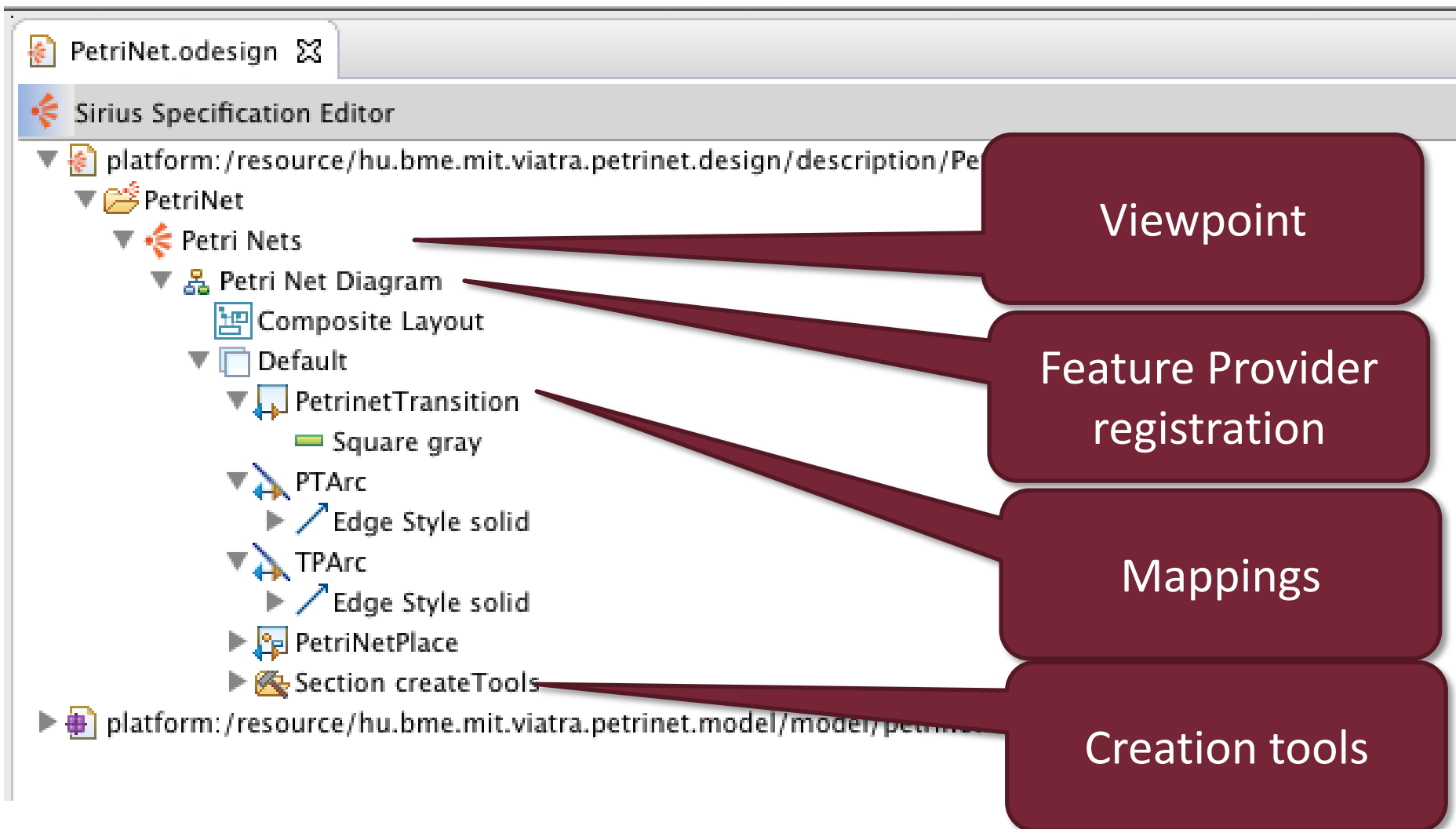
# Sirius Viewpoints



- Base concept:
  - Every diagram is a view of the model
  - With a defined syntax
    - **Graphical**
    - Table/Tree syntax
    - Xtext-based textual syntax
- Viewpoint definition
  - Viewpoint specification model



# Viewpoint Specification Model





# Node & Edge Mapping

The image displays two screenshots of a software interface for configuring Petri net transitions and arcs. The top screenshot shows the 'PetrinetTransition' properties window. The 'General' tab is active, showing fields for 'id:', 'Domain Class:', and 'Semantic Candidates Expression:'. The 'Domain Class' is set to 'petrinet.Transition', and the 'Semantic Candidates Expression' is 'feature:transitions'. The bottom screenshot shows the 'TPAParc' properties window. The 'General' tab is active, showing fields for 'id:', 'Domain Class:', 'Source Mapping:', 'Source Finder Expression:', 'Target Mapping:', 'Target Finder Expression:', and 'Semantic Candidates Expression:'. The 'Domain Class' is 'petrinet.TPAParc', 'Source Mapping' is 'PetrinetTransition', 'Source Finder Expression' is 'feature:source', 'Target Mapping' is 'PetriNetPlace', and 'Target Finder Expression' is 'feature:target'. Five callouts point to specific elements: 'Domain class' points to 'petrinet.Transition', 'Filter settings' points to 'feature:transitions', 'Edge class' points to 'petrinet.TPAParc', 'Source features' points to 'feature:source', and 'Target features' points to 'feature:target'.

**PetrinetTransition**

General

id: ? PetrinetTransition

Domain Class: ? petrinet.Transition

Semantic Candidates Expression: ? feature:transitions

**TPAParc**

General

id: ? TPAParc

Domain Class: ? petrinet.TPAParc

Source Mapping: ? PetrinetTransition

Source Finder Expression: ? feature:source

Target Mapping: ? PetriNetPlace

Target Finder Expression: ? feature:target

Semantic Candidates Expression: ?



# Feature Selection



## ■ Interpreted expressions

### ○ Special interpreters

- **var**: accessing specification model variables
- **feature**: accessing EMF model features
- **service**: accessing service methods

### ○ Acceleo

- Acceleo expressions
  - Basic operations
  - Comparison with single '=' symbols
- Syntax: **[theExpression/]**

### ○ Raw OCL

- Not recommended, Acceleo provides superset features

### ○ Custom interpreter



# Node & Edge Tool

- ▼ Section createTools
    - ▼ Container Creation createPlace
      - Node Creation Variable container
      - Container View Variable containerView
    - ▼ Begin
      - ▼ Change Context var:container
        - ▼ Create Instance petrinet.Place
- = Set name

Tool parameter  
variables

Model creation  
sequence

Different  
variables

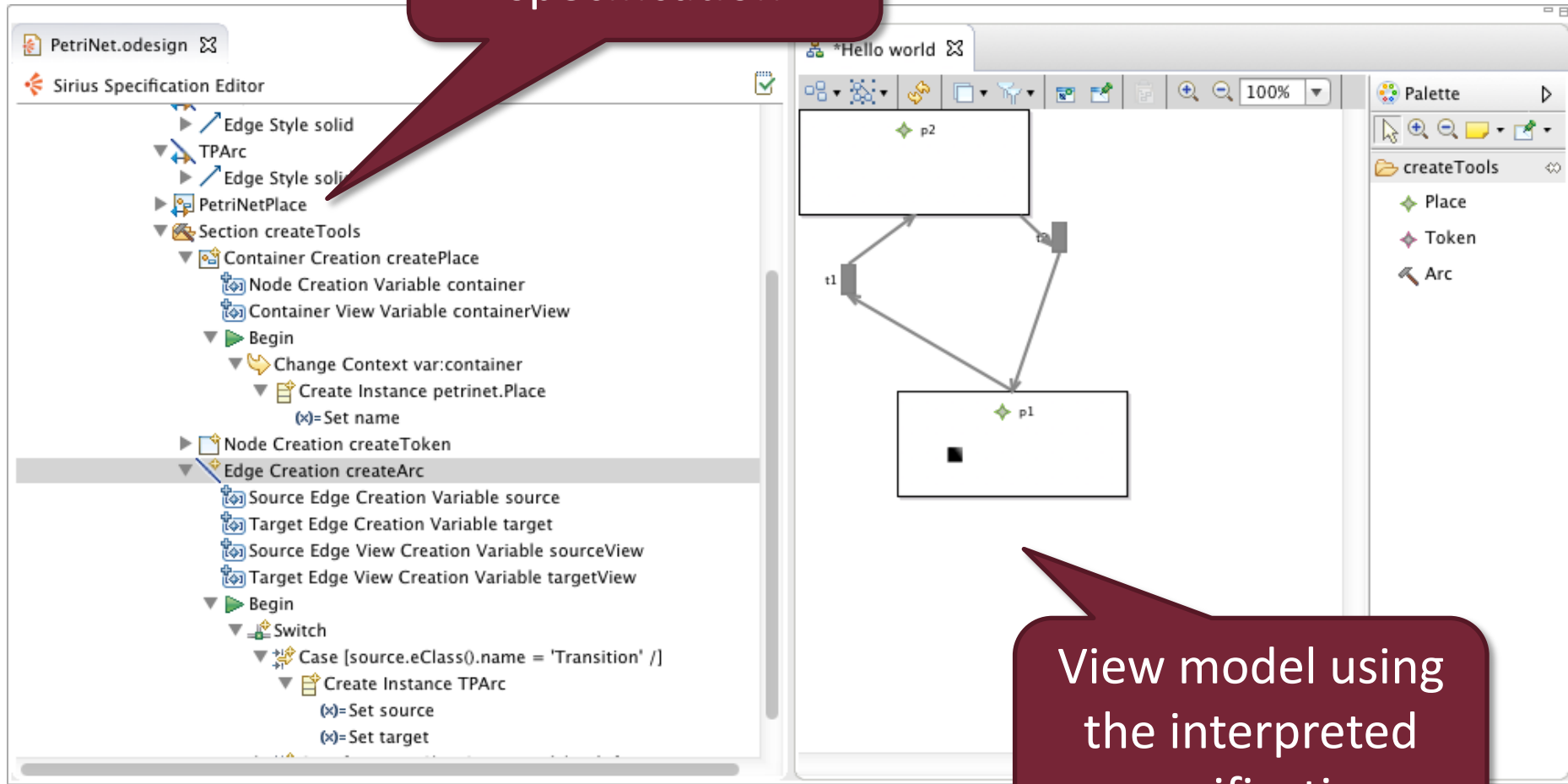
More complex  
creation steps

- ▼ Edge Creation createArc
  - Source Edge Creation Variable source
  - Target Edge Creation Variable target
  - Source Edge View Creation Variable sourceView
  - Target Edge View Creation Variable targetView
- ▼ Begin
  - ▼ Switch
    - ▼ Case [source.eClass().name = 'Transition' /]
      - ▼ Create Instance TPACrc
        - = Set source
        - = Set target
    - Case [source.eClass().name = 'Place' /]



# Interpreted Modeler Development

Viewpoint  
specification



View model using  
the interpreted  
specification



# Technology Comparison

	GEF	GMF	Graphiti	Sirius
Model	Arbitrary	EMF	EMF	EMF
Non graph-based presentation	Manageable	Large amount of customization needed	Not supported	Tree, Table
Code size	Large, repetitive code	Mostly modeling, some coding	Smaller amount, but repetitive code	Negligible
Development workflow	Only coding	Modeling and coding	Coding	Modeling

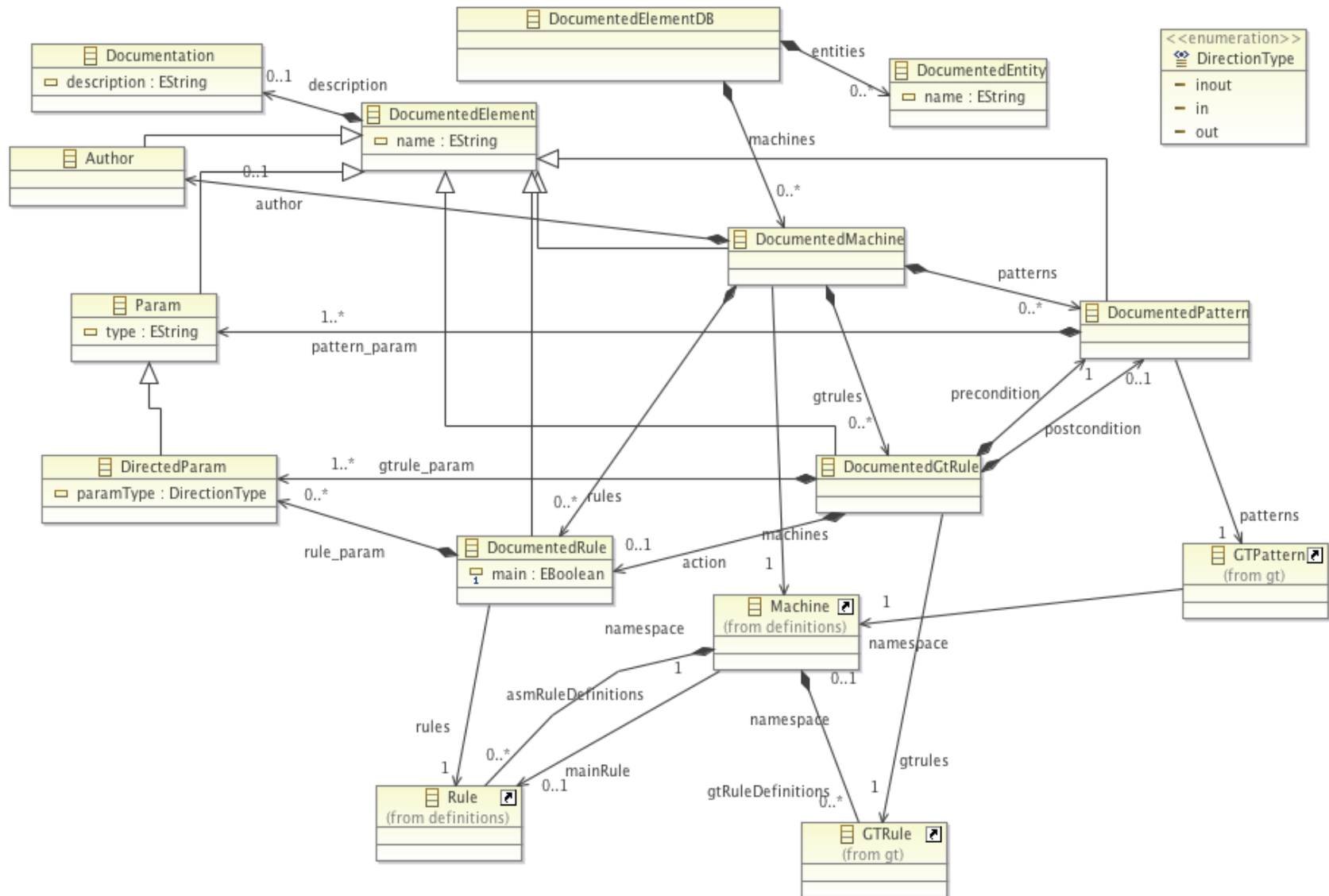


# Advanced issues

- Cumbersome editing
  - E.g., reorganization to insert a node to the middle
- Handling large models
  - 20+ nodes on a diagram:
    - Logical structure, readability possible
    - But needs human support
  - 100-1000+ nodes on a diagram
    - Technological limitations
    - Usability limitations

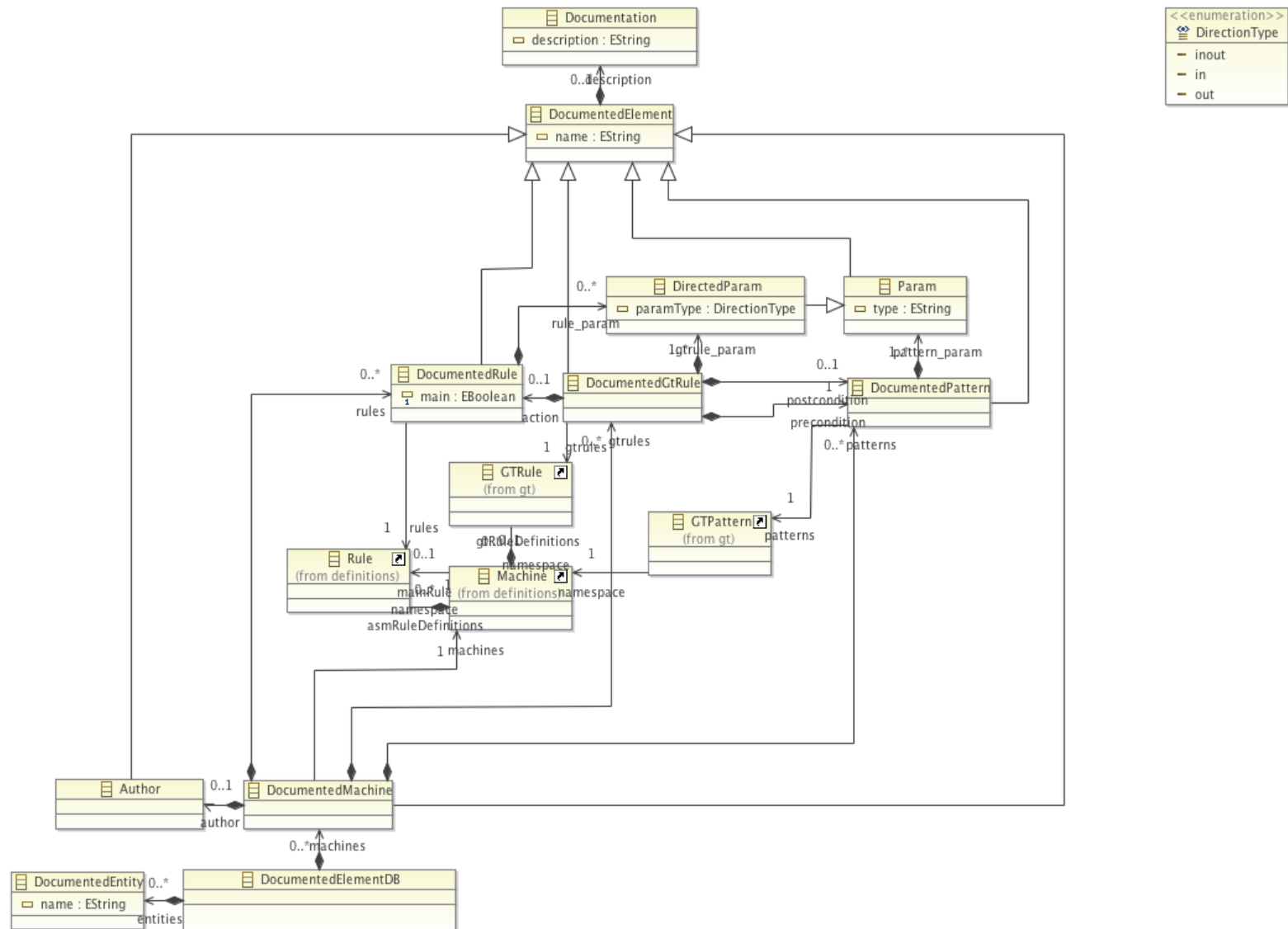


# Example: Layouting





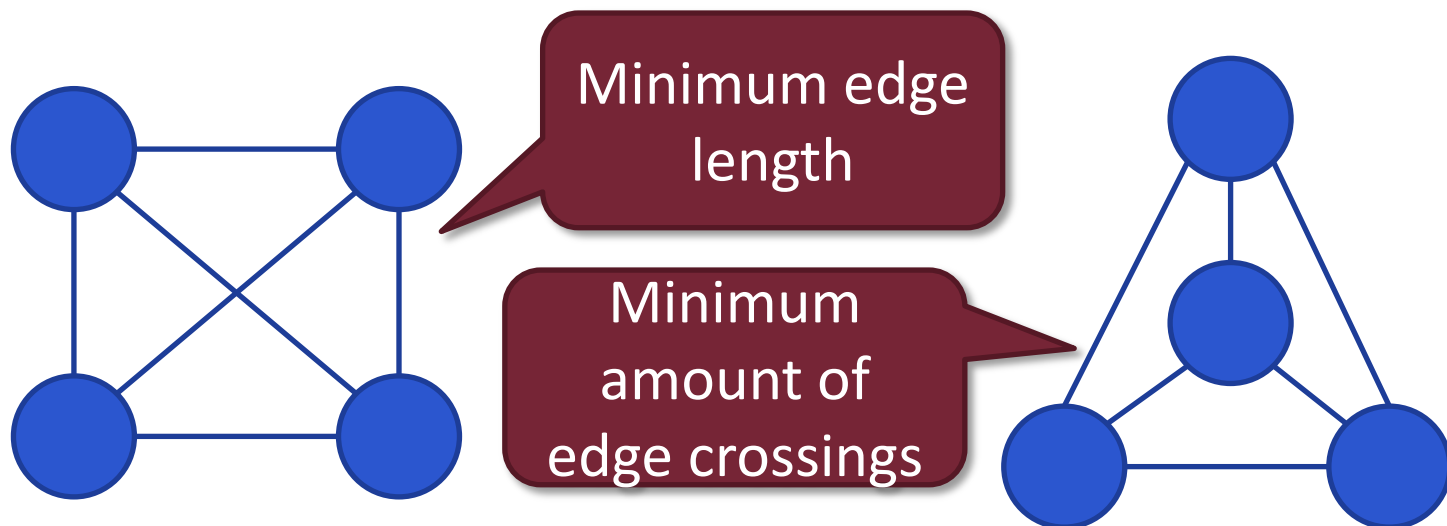
# Example: Layouting





# Layouting Support for Graphical Editors

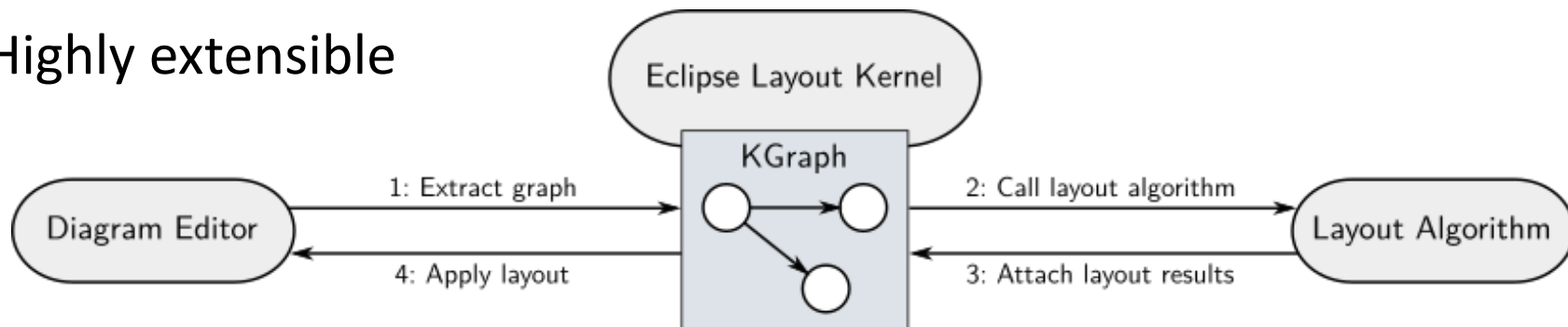
- Computation of the position of nodes
  - Possible to do automatically
  - For a given metamodel
    - No unified visual requirements possible
    - We have to decide what is important to show





# Layouting Support for Graphical Editors

- **GraphViz** - <http://graphviz.org>
  - Layouting project with high quality layout algorithm
  - Hard to integrate into Eclipse applications
- **Zest** - <http://wiki.eclipse.org/index.php/Zest>
  - Easily Eclipse integration (SWT-based graph widget)
  - So-so layout algorithms
- **ELK** (née ~~KIELER~~) - <https://www.eclipse.org/elk/> (relatively new)
  - Eclipse Layout Kernel
  - Some built-in support: GMF, Graphiti
  - Highly extensible





# Textual or graphical?



# Comparison

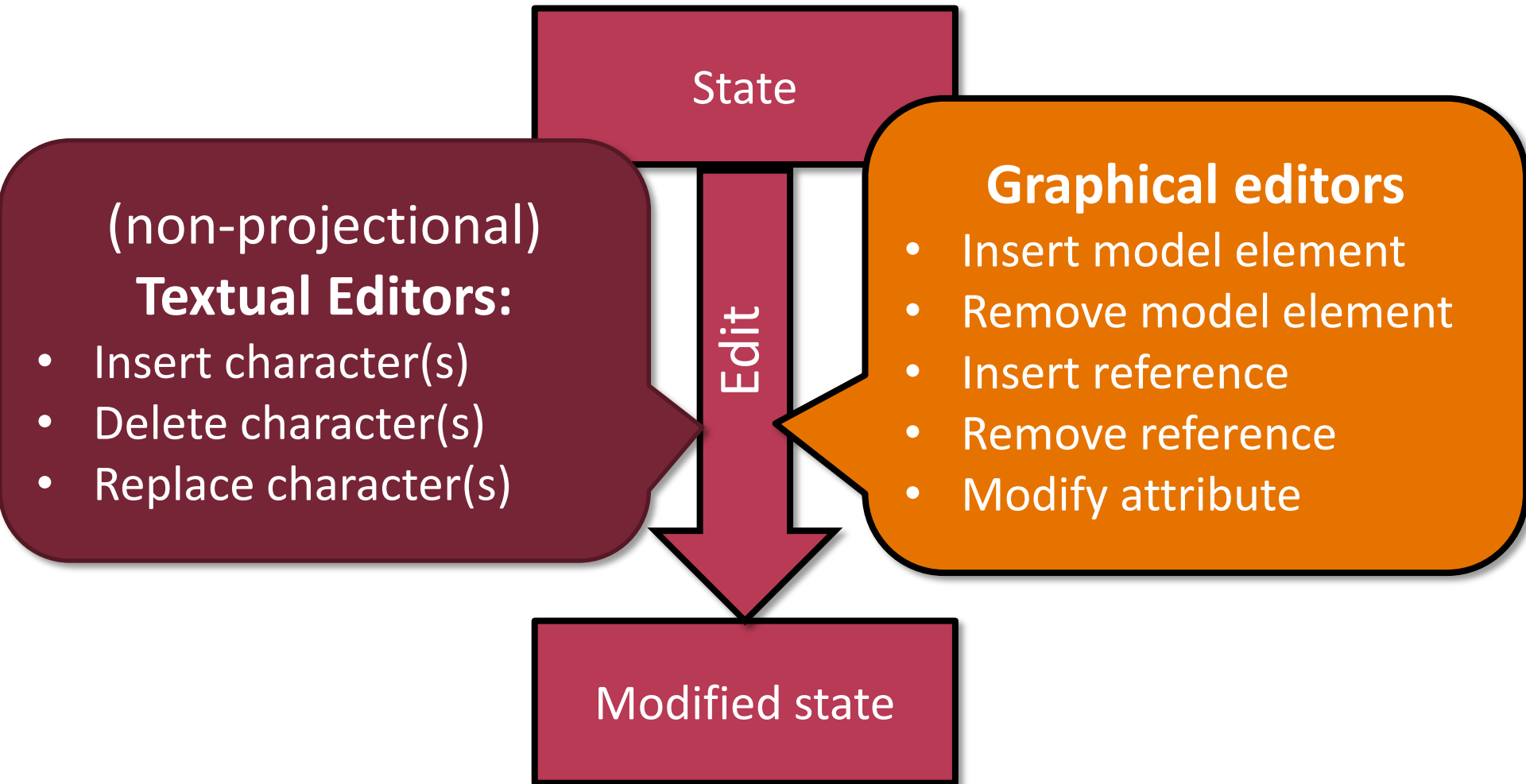
Textual Languages	Graphical Languages
Quick and simple editing	More cumbersome editing
References described as <i>string identifiers</i>	References displayed visually
Inconsistent models during editing	Models always syntactically correct
Automatic formatting	Automatic layouting
Content assist	Tool list to add nodes/edges

Displaying validation errors, offering quick fixes

Both are supported with EMF-based technologies



# Editing



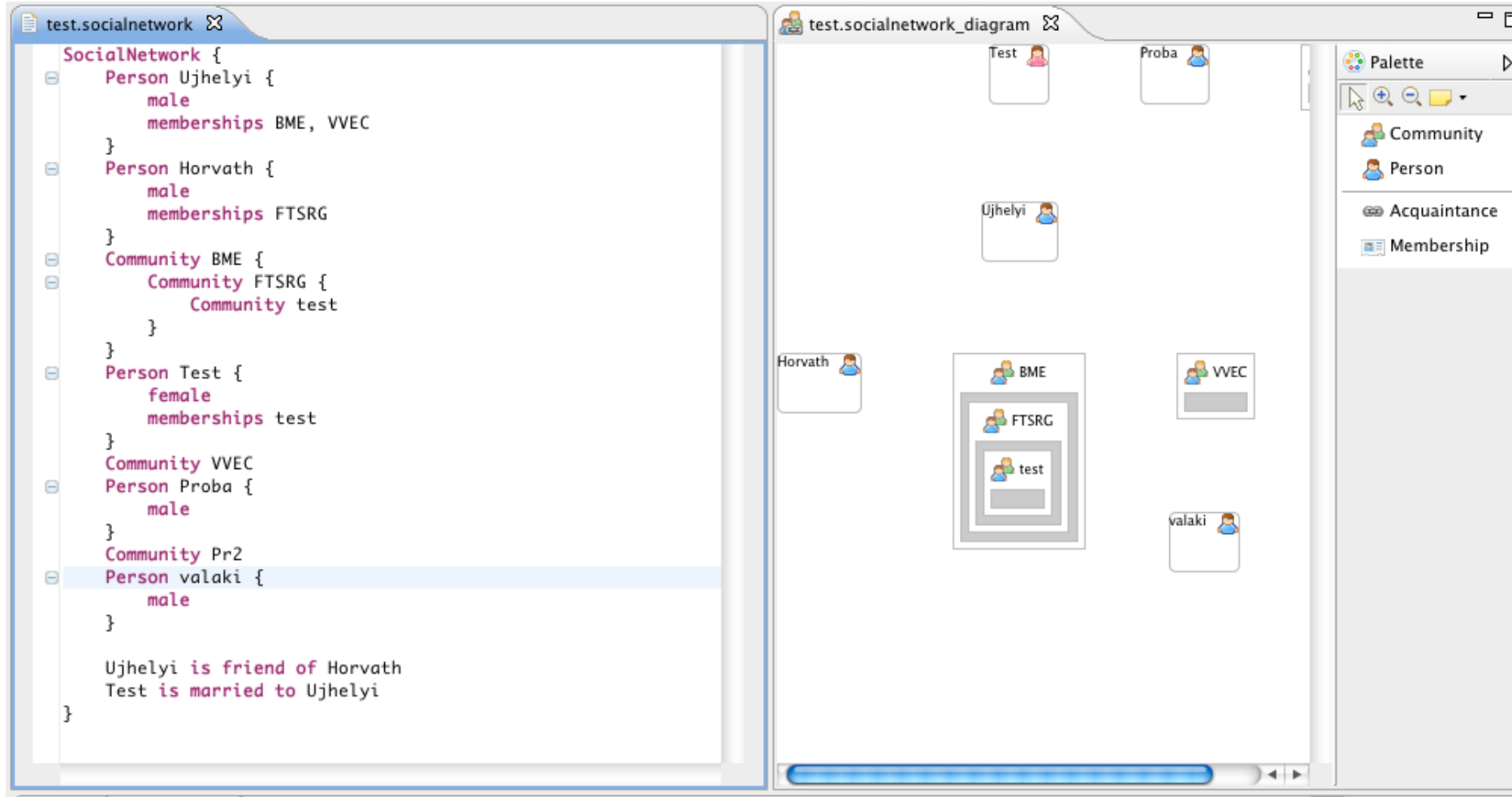


# Question: textual or graphical?

- No clear choice
- Rules of thumb
  - Behaviour description is usually simpler in textual
  - For structural information graphical is often better
- For simple languages
  - Form-based editing might also be an alternative



# Xtext and GMF on the same instance model





# Derived Graphical viewer support

- Xtext Generic Viewer component
  - Created by Xtext developers
  - Independent from the main Xtext development
  - Requires an extra language
    - to define uni-directional mapping
    - to define format
- See “A fresh look at graphical modeling” for details
  - <http://www.slideshare.net/schwurbel/a-fresh-look-at-graphical-editing-10068461>



# Concrete Syntax Design

Conclusion



# Concrete Syntax Design

- Multiple approaches
  - Textual and/or graphical syntaxes
  - Combinable
- Large amount of development work needed
  - Directly used by users
  - Usability issues
- Not everything is coded in an editor
  - Editor + corresponding views form the interface