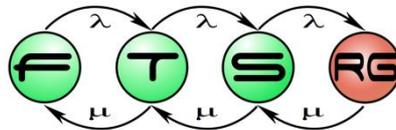


Concrete Syntax Design for Domain-specific Languages

Model Driven Software Development Lecture 5



Structure of DSMs

Graphical syntax



Abstract syntax



Well-formedness constraints

- Errors (4 items)
- File platform:/resource/fr.irisa.triskell.kerne
- File platform:/resource/fr.irisa.triskell.kerne
- Unable to set the type of fsm::State::step:
- Unresolved type 'Sstring'. (missing using ?)

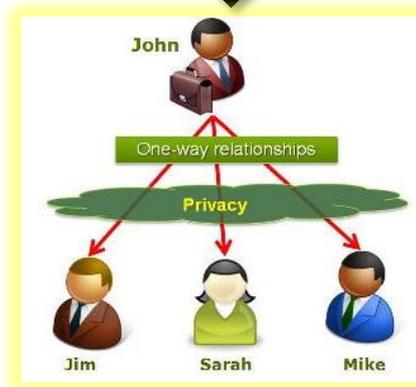
Behavioural semantics, simulation

Code generation

Mapping



Textual syntax

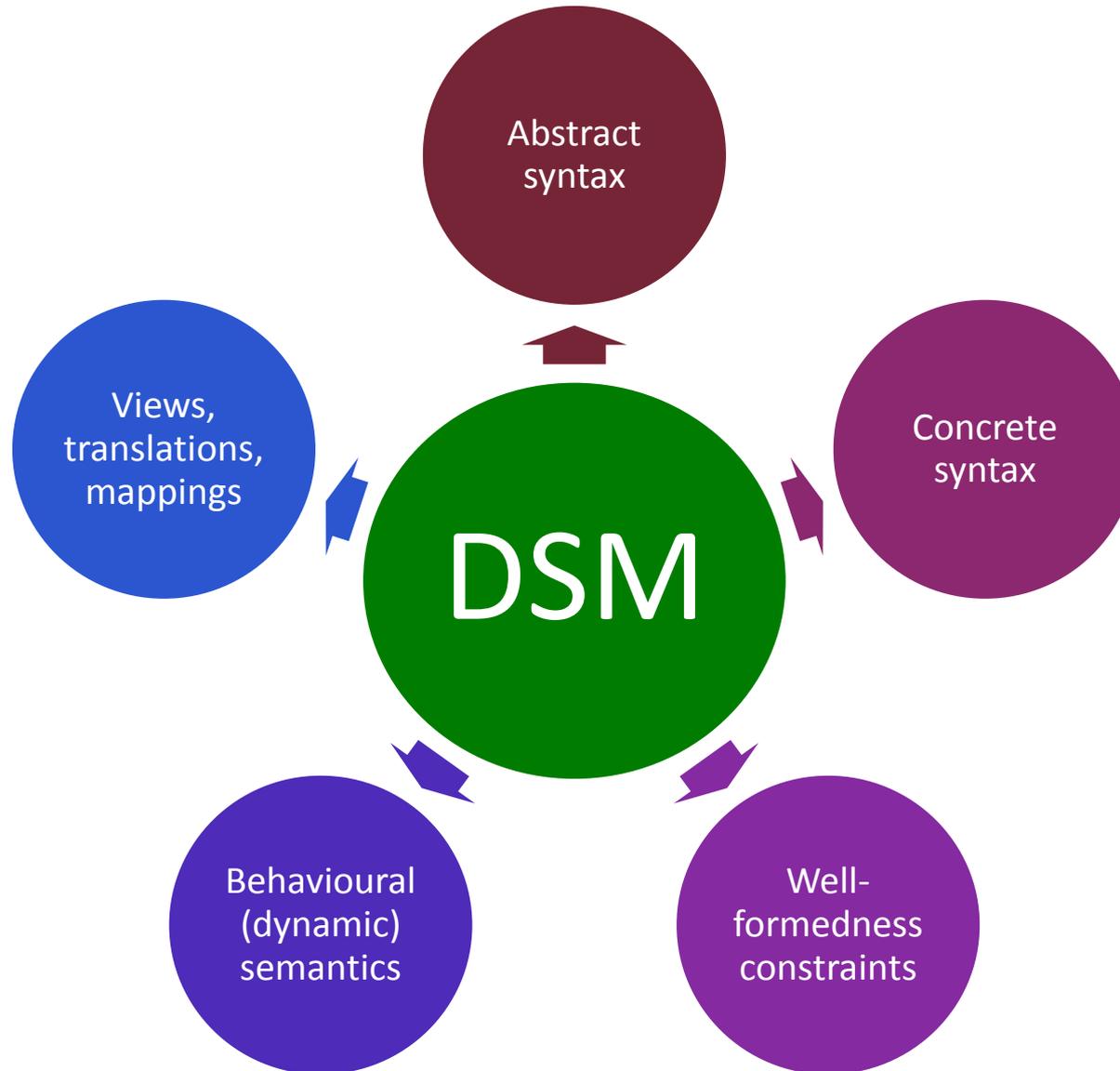


View

```
</membership>  
<profile defaultProvider="Sitefinity">  
  <providers>  
    <clear/>  
    <add name="Sitefinity" connectionS</add>  
  </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-facing parts of a modeling language
 - Performance
 - Robustness
 - Usability issues
- Creating model editors
 - Similar problems at programming languages
 - IDE extensions needed
- Viewers are also important!
 - ~read-only editors

Concrete Syntax Approaches

- Graphical
 - Focus of latter half of today's lecture
 - Typically graph-based modeling (Edges, Nodes)
- Textual
 - More details to come in next lecture
- Form-based
 - Tree views
 - Property sheets, combo / radio /etc.
 - Table/matrix approaches

Example: Petri net editor

The image shows a screenshot of a Petri net editor. The main window displays a Petri net diagram with three places (p1, p2, p3) and two transitions (t1, t2). Place p1 contains 2 tokens. Transitions t1 and t2 are connected to places p2 and p3 respectively. A red callout box points to the 'Outline' window on the right, which shows a tree-based view of the Petri net elements. The 'Outline' window is titled 'Outline' and contains a tree structure under 'PetriNet model elements'. The tree structure is as follows:

- PetriNet model elements
 - pn0
 - pn1
 - p1
 - p1_t1
 - p1_t2
 - token0
 - token1
 - p2
 - p3
 - t1
 - t1_p2
 - t2
 - t2_p3
- PetriNet diagrams
 - Example Petri net [PetriNetDiagram]
 - Example Petri net [PetriNetRoot]
 - pn1 [PetriNetFigure]
 - p1 [PlaceFigure]
 - token0 [TokenFigure]
 - token1 [TokenFigure]
 - p2 [PlaceFigure]
 - p3 [PlaceFigure]
 - t1 [TransitionFigure]
 - t2 [TransitionFigure]
 - p1_t1 [OutArcFigure]
 - p1_t2 [OutArcFigure]
 - t1_p2 [InArcFigure]
 - t2_p3 [InArcFigure]

Tree-based
outline view

Example: Social Network editor

The screenshot displays the Eclipse IDE interface for a Social Network editor. The main workspace shows a graph with nodes and edges. Nodes include 'J. Random', 'Jane Doe', and 'John Doe'. Edges connect 'J. Random' to 'Jane Doe', 'Jane Doe' to 'John Doe', and 'Jane Doe' to 'Bar Society'. 'Bar Society' contains a 'Baz Community' node. A 'Membership' node is also visible.

Callouts highlight the following features:

- Project Explorer extensions:** Shows a tree view of the project structure, including 'DSE', 'default', 'Social', 'Person Jane', 'Person John', and 'default.socialnetwork'.
- Graph outline view:** Shows a simplified graph structure with nodes and edges.
- Properties view:** Shows the properties of the selected node 'John Doe':

Property	Value
Name	John Doe
Sex	male
X	677
Y	240
- Palette:** Shows a list of available elements: Select, Marquee, Person, Community, Acquaintance, and Membership.

Advanced features

Viewer features

- Outlining / folding / abstraction
- Details / documentation overlay (e.g. Javadoc, „code mining“)
- Validation / task / etc. overlay
- Search, navigability
- Automatic layout/formatting

Editor features

- Templates/snippets/examples
- Guidance (content assist / snap)
- Composite operations/tools/refactorings
- Automatic fixes
- Undo&Redo, Transactionality

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
- JetBrains MPS

Human Aspects

Textual vs. Graphical
Visual Design
Layouting

Question: textual or graphical?

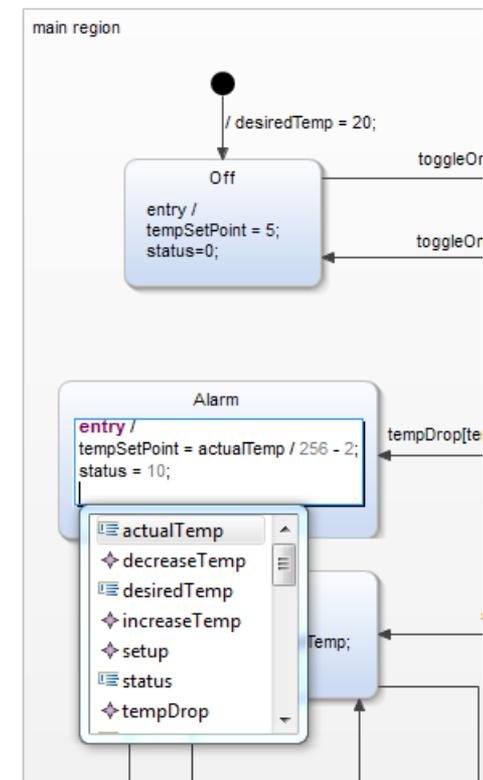
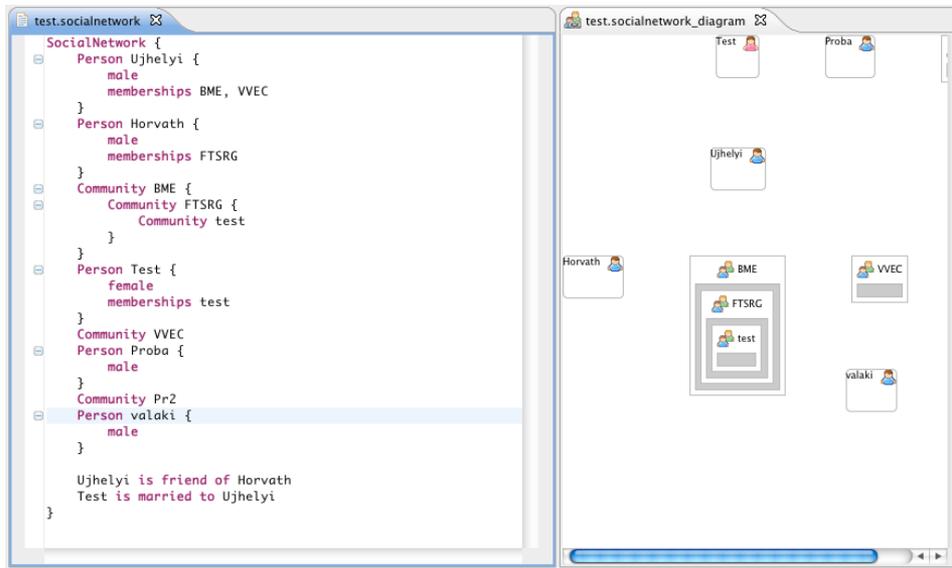
- No clear choice, just rules of thumb

Textual Languages (<i>raw editing</i>)	Graphical Languages
Quick and simple editing	More cumbersome editing
References as <i>string identifiers</i>	References displayed visually
Inconsistent during editing	Always syntactically correct
Trivial diff&patch, copy&paste, search&replace	Editing services require tool development effort
Typically better for behavior	Typically better for structure

- Simple languages: consider form-based as well
 - Like graphical, but cross-references poorly supported
- ...why not both?

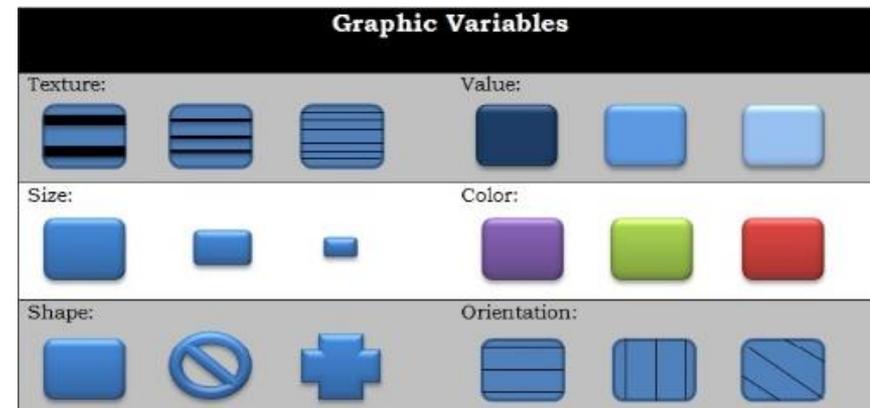
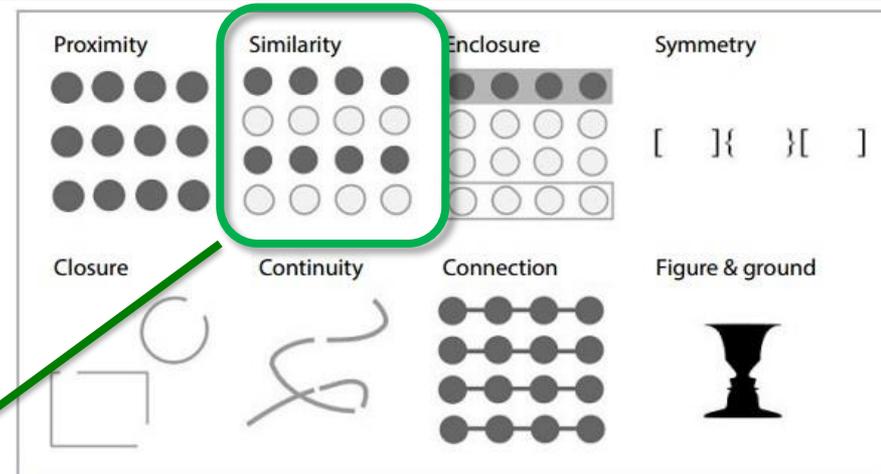
Textual + Graphical

- Same model, two syntaxes
 - Text editor + graphical view
 - Xtext Generic Viewer
 - Textual + graphical editors
 - Xtext + GMF side-by-side
- Different aspects of model
 - Diagram with text fields
 - Embedded Xtext support



Visual Design 101

- What belongs together?
„Gestalt principles of grouping”
 - E.g. which label belongs to which node?
- What is similar?
„Bertin’s visual variables”
 - Size, shape
 - Color hue, value, intensity
 - Line style / orientation / texture



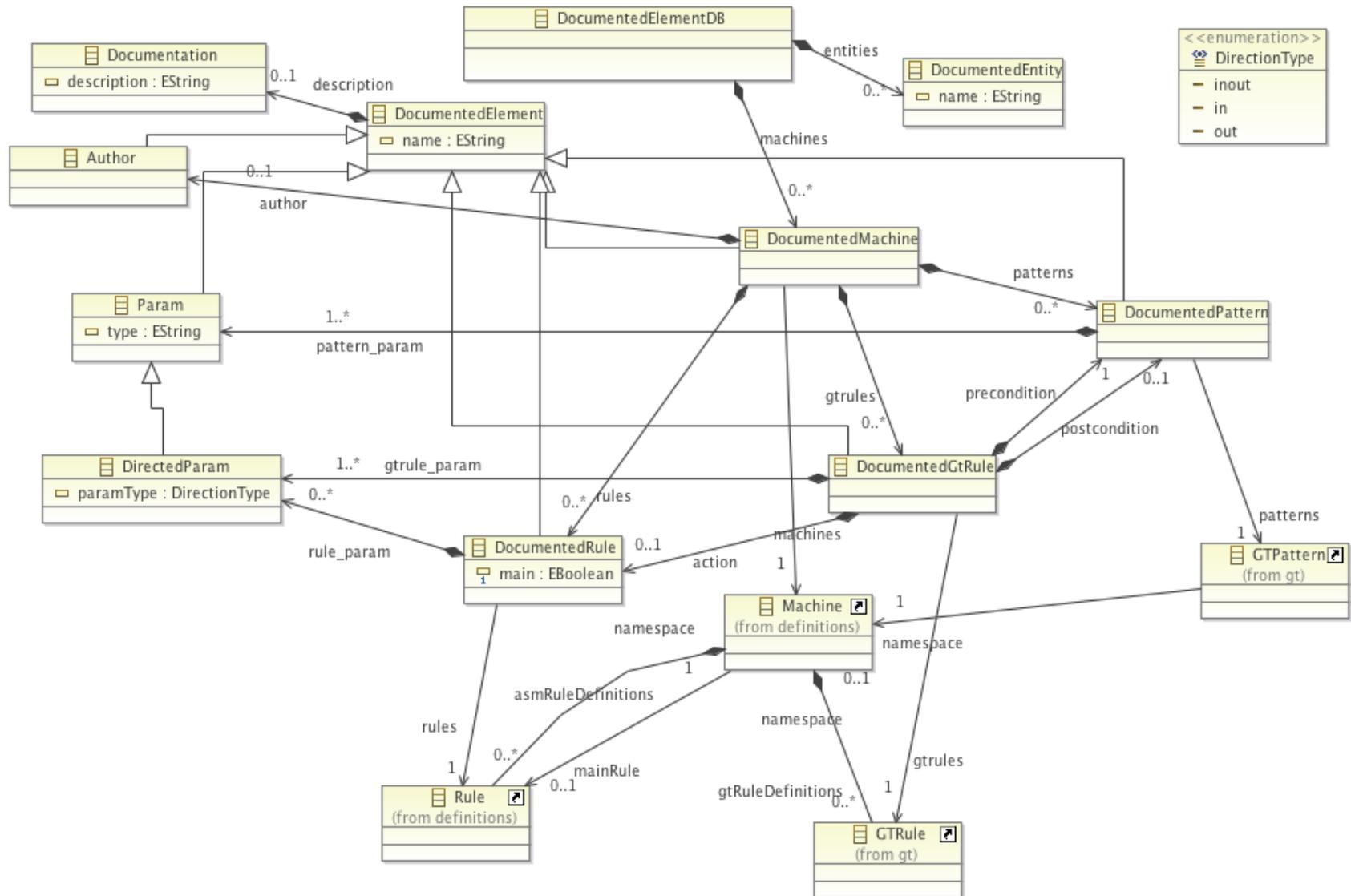
Sources: http://wiki.gis.com/wiki/index.php/Visual_variable

<https://www.fusioncharts.com/blog/how-to-use-the-gestalt-principles-for-visual-storytelling-podv/>

Scaling issues

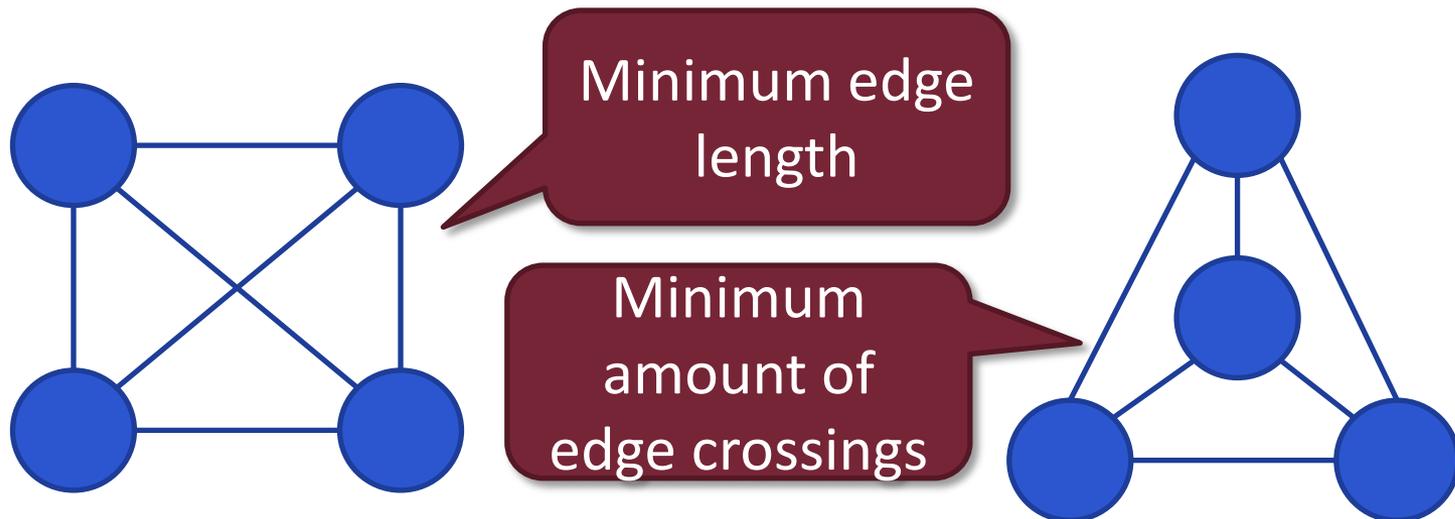
- Cumbersome editing
 - E.g., automatically reorganize diagram when inserting a node to the middle
- Handling large models
 - 20+ nodes on a diagram:
 - Logical structure, readability possible
 - But needs human support
 - 100-1000+ nodes on a diagram
 - Technological limitations
 - Usability limitations

Example: Layouting



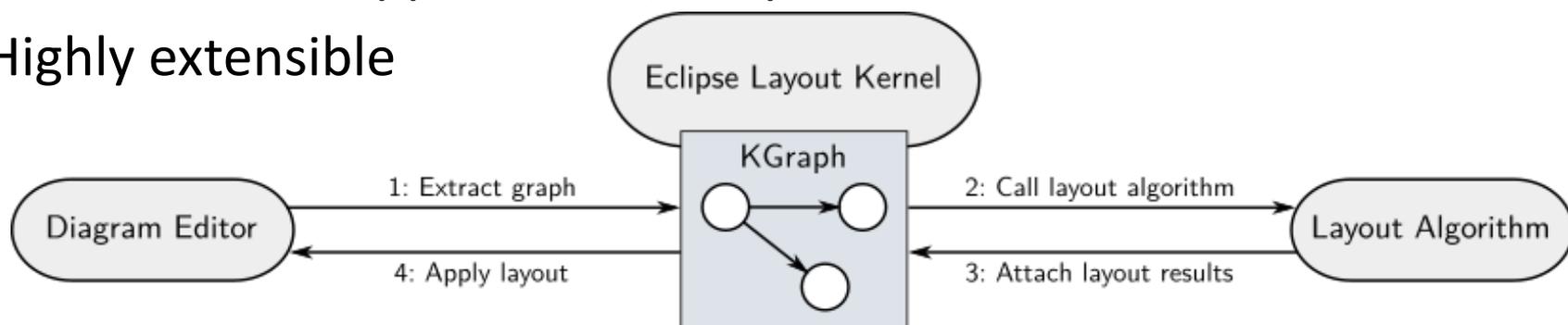
Layouting Support for Graphical Editors

- 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



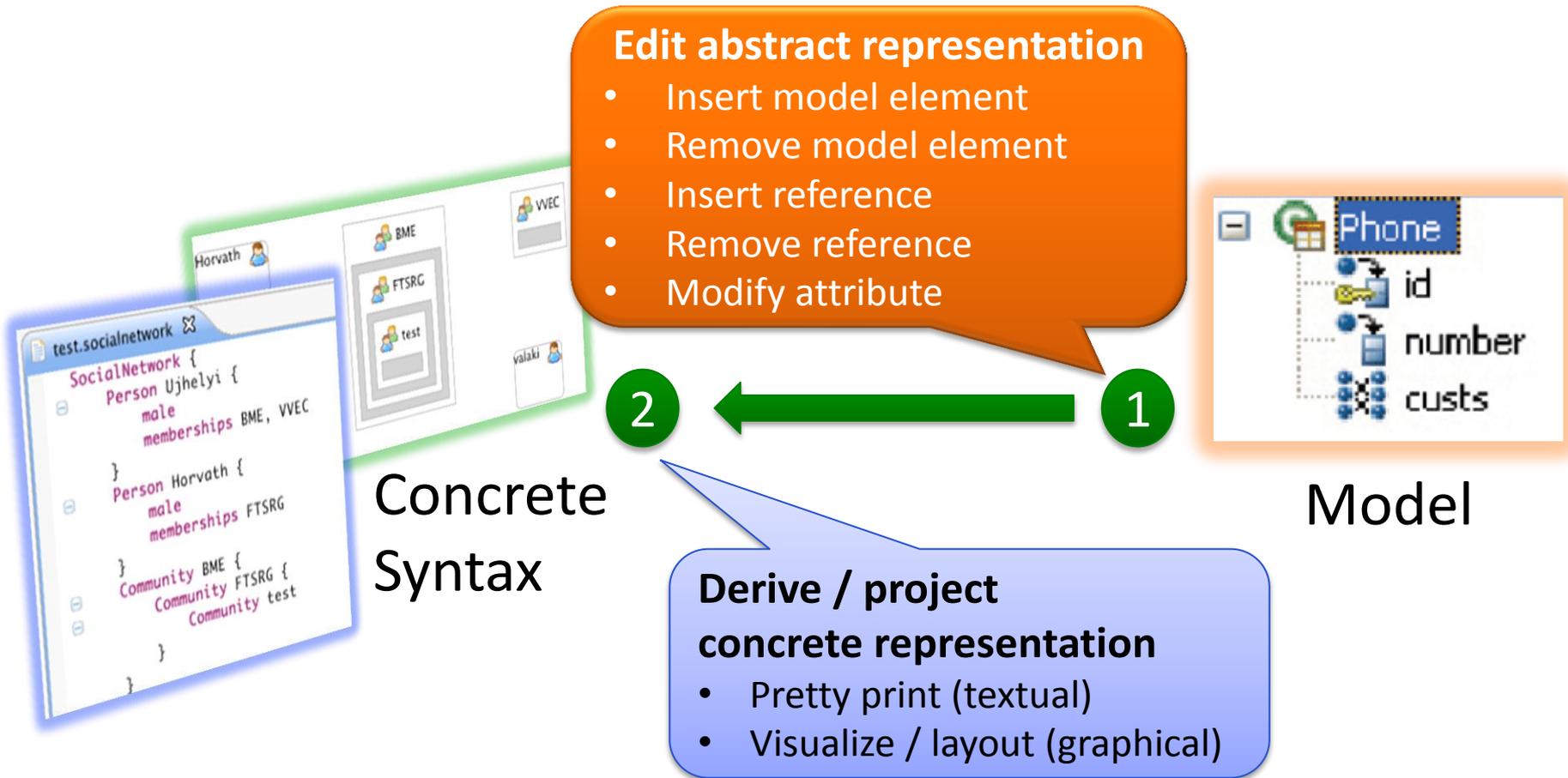
Editor Engineering

Editing Workflows
Transactionality
Notation Models

Projectional vs Raw

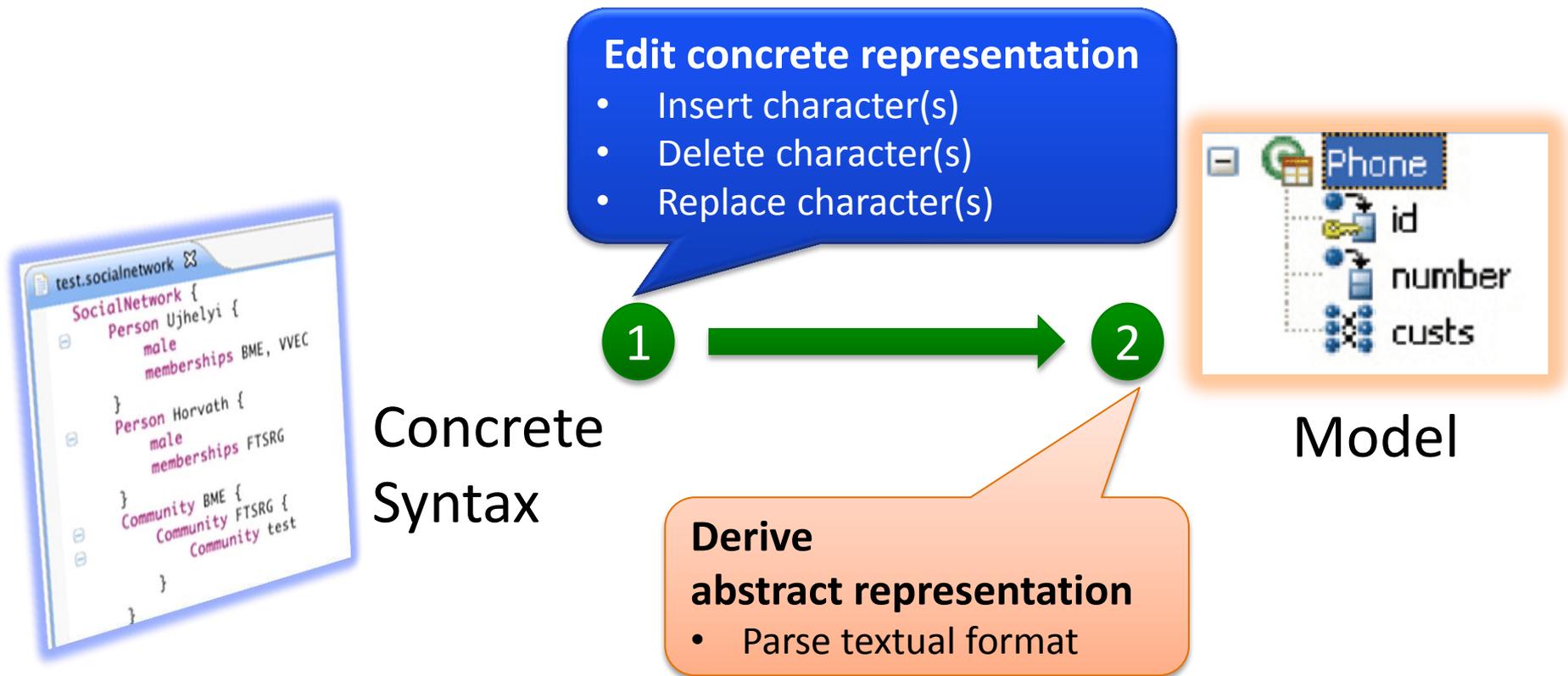
■ Workflow 1: **projectional editing**

- AKA syntax-driven editing, structural editing



Projectional vs Raw

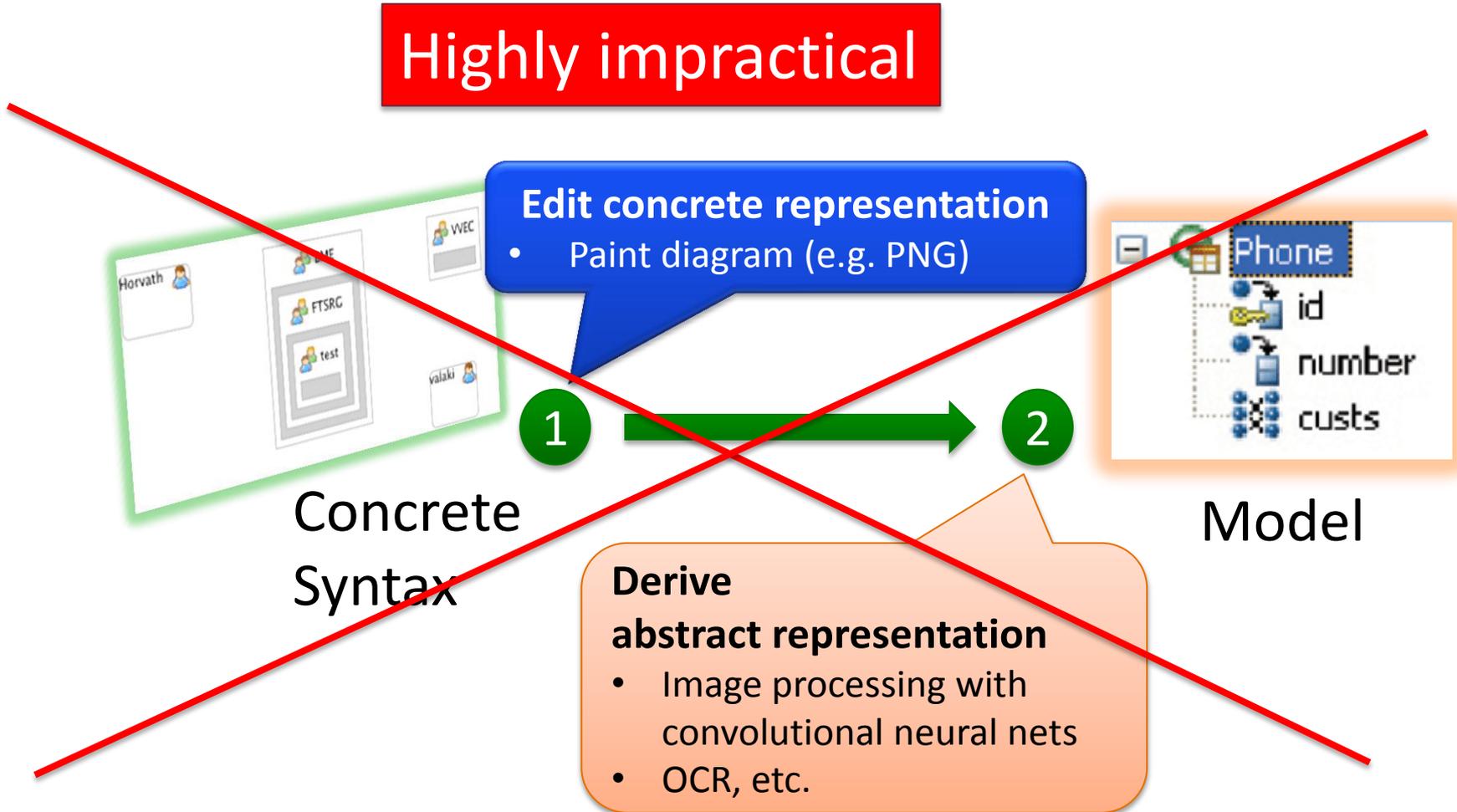
- Workflow 2: **raw editing** (w. textual syntax)
 - AKA source editing



Projectional vs Raw

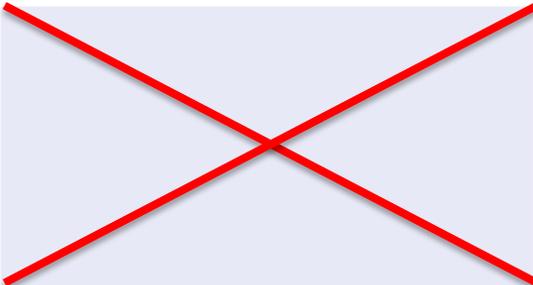
- Workflow 2: **raw editing** (w. graphical syntax)

Highly impractical



Projectional vs Raw

- „Feature matrix” + examples

	Graphical syntax	Textual syntax
Raw editing		Typical 
Projectional editing	Typical 	Rare 

Mixed workflow

```
ntFactory eINSTANCE = CrossContainment.in  
ew object  
user-doc -  
er-doc -->  
ew object  
ew object  
user-doc -  
er-doc -->  
ew object
```

```
changing 2 classes altogether  
Iterator contents = rSet.getAllContents();  
ntents  
Notif  
rrent
```

Derive / project
concrete representation

Complex manipulation of
abstract representation

- Quick fix
- Refactor
- M2M

```
test.socialnetwork  
SocialNetwork {  
  Person Ujhelyi {  
    male  
    memberships BME, VVEC  
  }  
  Person Horvath {  
    male  
    memberships FTSRG  
  }  
  Community BME {  
    Community FTSRG {  
      Community test  
    }  
  }  
}
```

Concrete
Syntax



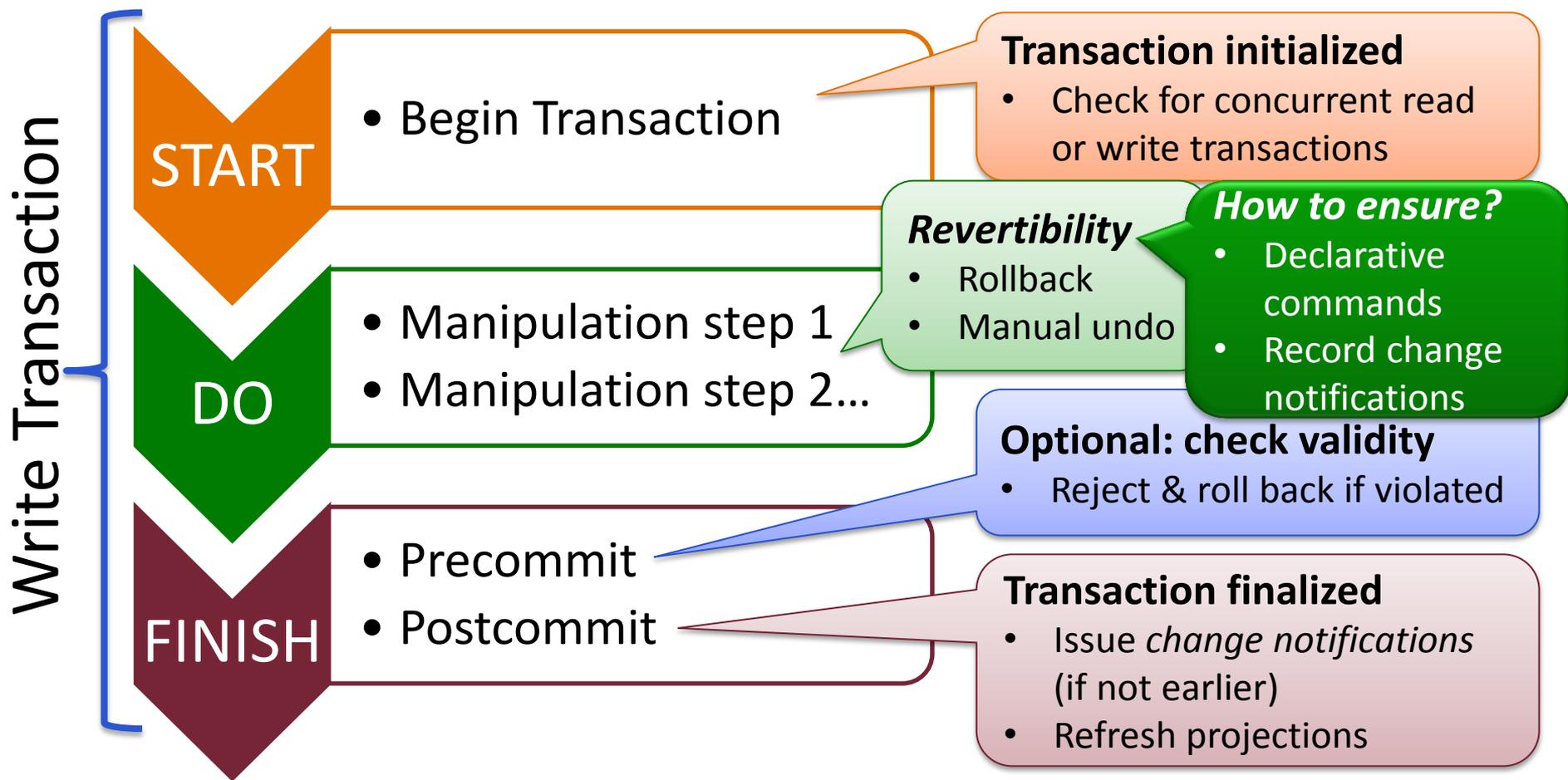
Model



Normal raw
editing workflow

Transactions in projectional editing

- Complex manipulation sequence as single action
 - „Extract subprocess”, „Drag&drop attribute” etc.



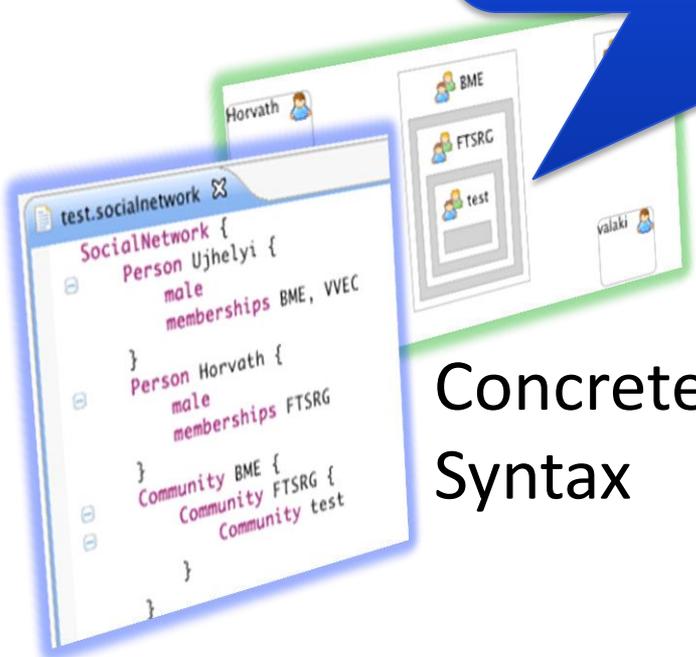
Superfluous notational parameters

■ Workflow 1: **projectional editing**

Must include *notational parameters*:

- Whitespace and comments, etc. (textual)
- Layout, edge routing, size, shape, etc. (graphical)

...even though not domain information



Concrete
Syntax

2



1



Model

Derive / project
concrete representation

- Pretty print (textual)
- Visualize / layout (graphical)

Deriving notational parameters

- Notational parameters can be...
 - ...”baked into” projection code
 - e.g. all lines are black, all fonts are 10pt (graphical)
 - e.g. apply this code formatting template (textual)
 - ...derived from domain information
 - e.g. shape determined by type, color by visibility

Problem 1:

Editable parameters cannot be a function of the domain model, must be stored

Problem 2:

Providing sane values is difficult for some parameters e.g. position in diagram

- ...**stored in the model**

Notation/view models

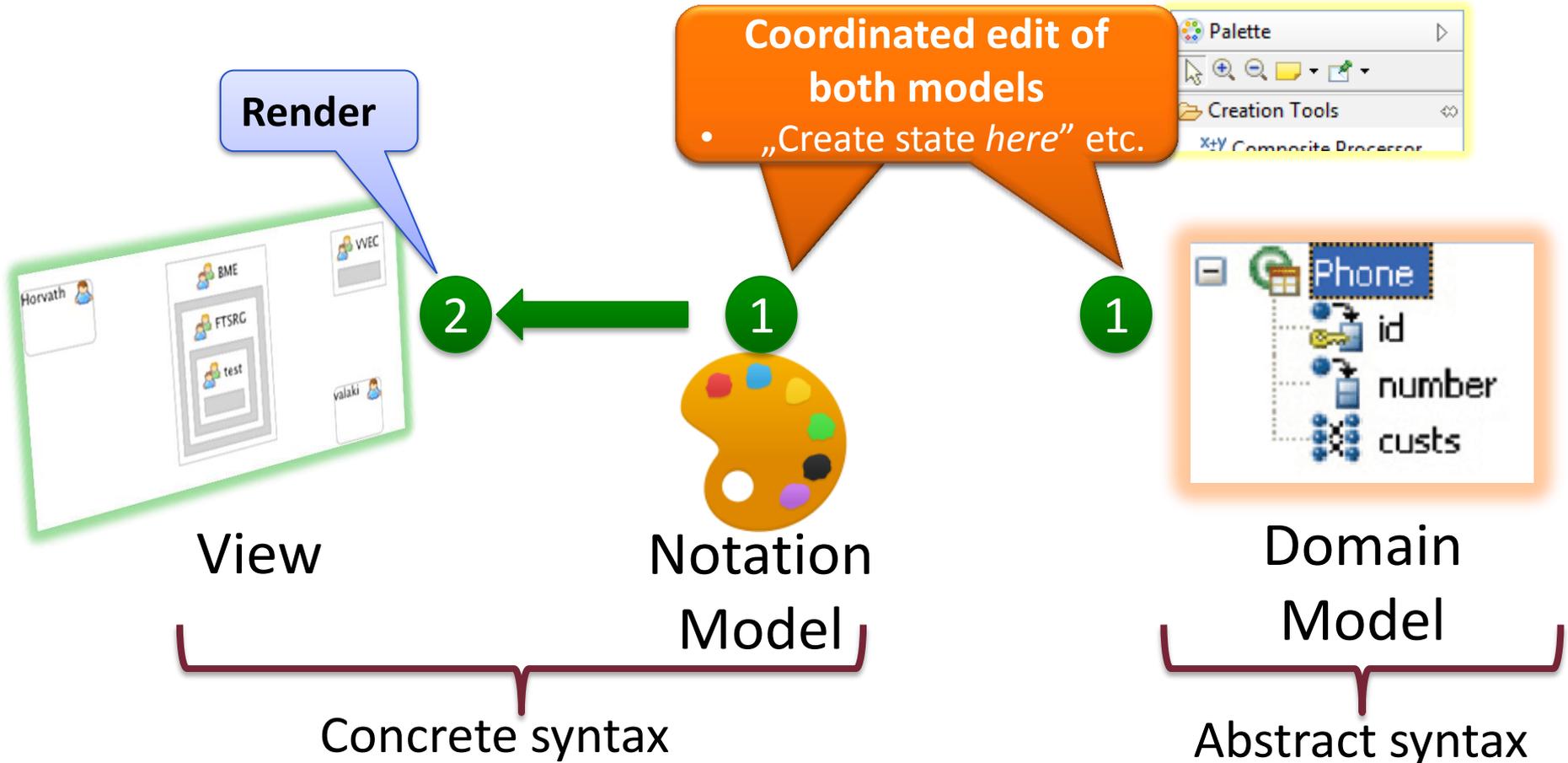
M.Fowler's
„Presentation Model”
architectural pattern

- Decompose model:
 - Domain / Semantic model (abstract syntax)
 - **Notation model** (view model): presentation state
 - may be editable by user
 - but still needs derivable defaults → see layouting
- Generic implementation in GMF and Graphiti
 - Based on EMF, in fact
- Often stored in external files
 - Separation of concerns
 - E.g. code generator not interested in view information

Editing workflow with notation models

■ Workflow 1: **projectional editing**

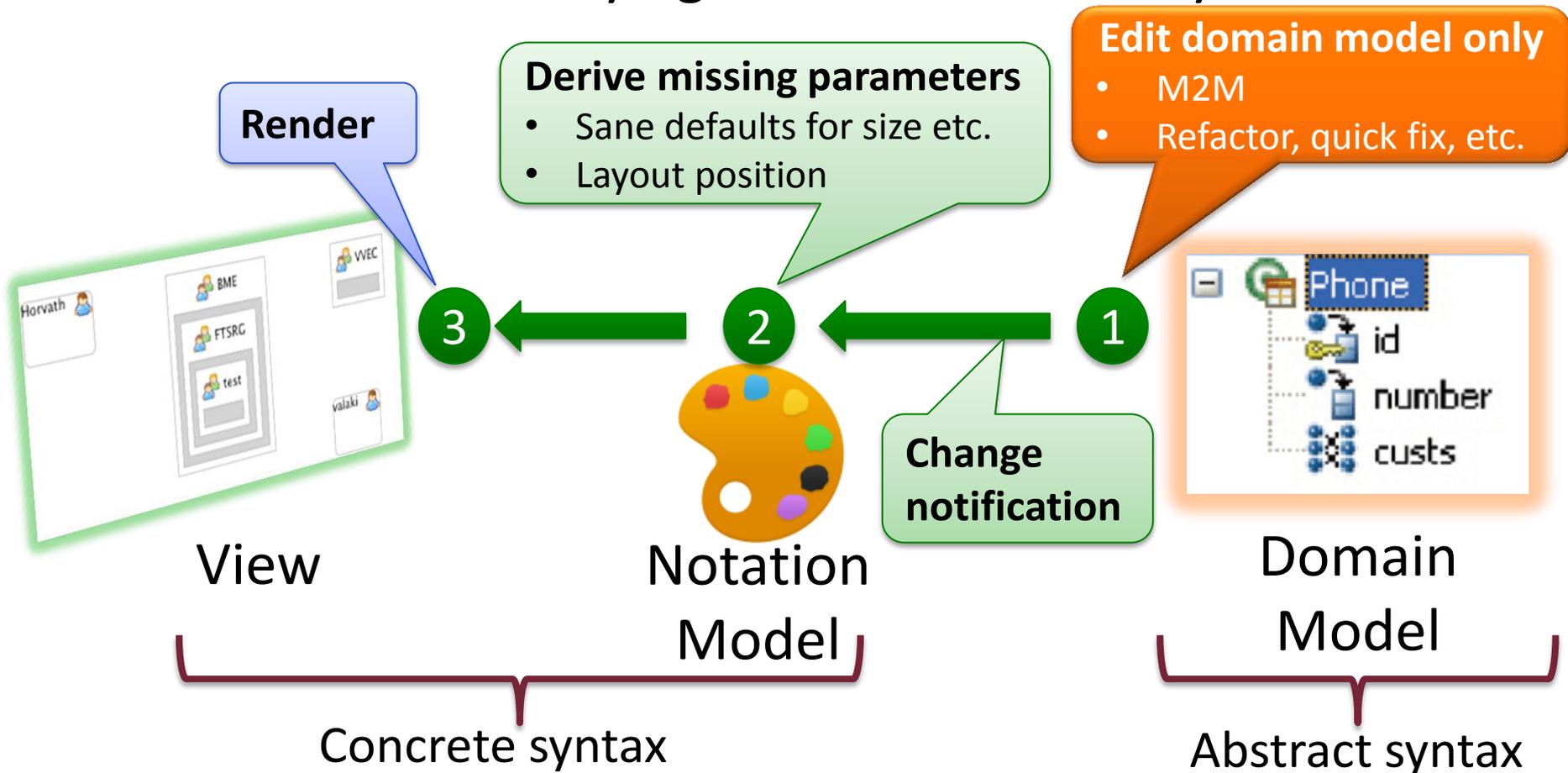
- Scenario A: co-modifying domain¬ation models



Editing workflow with notation models

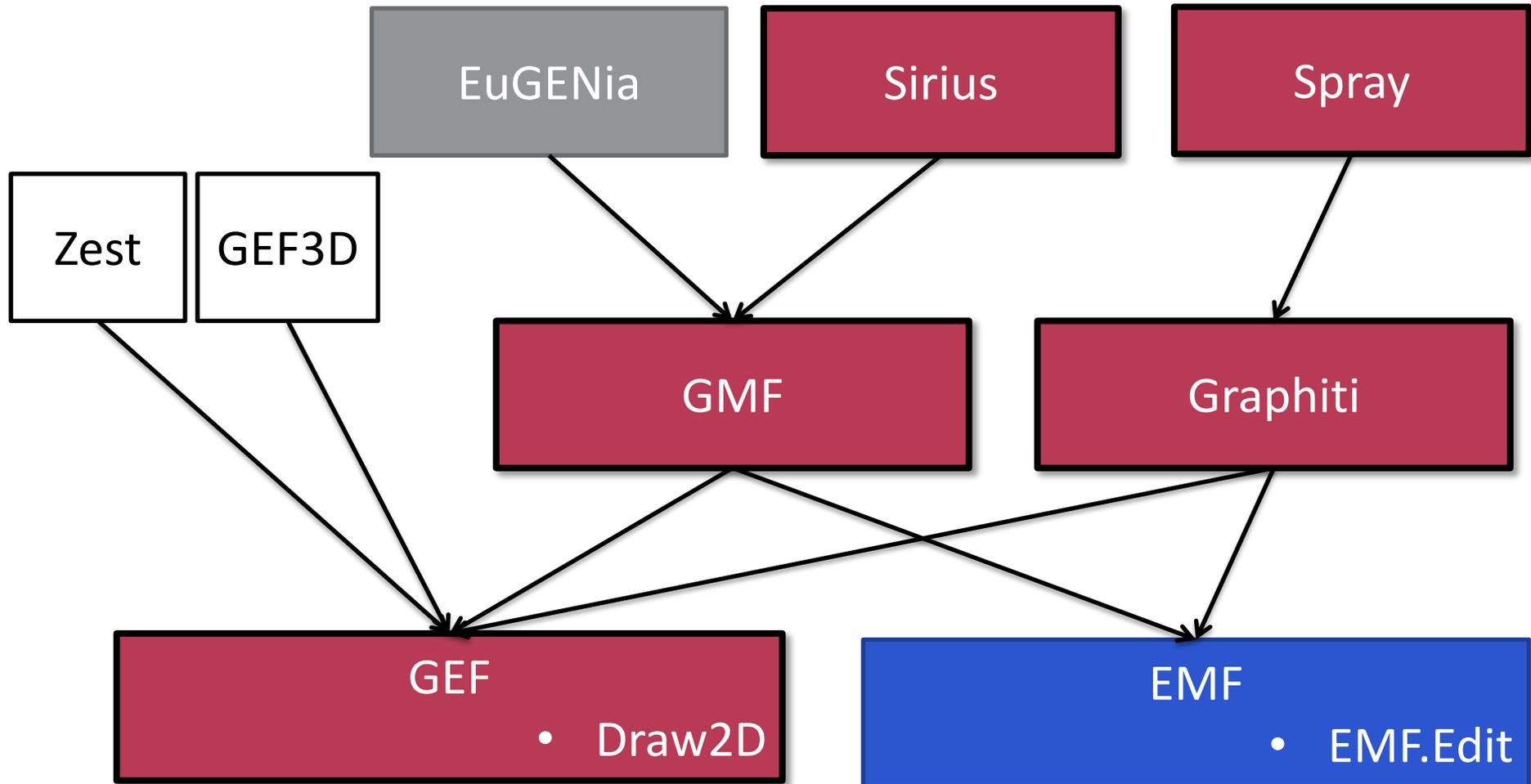
■ Workflow 1: **projectional editing**

- Scenario B: modifying domain model only



Graphical Editor Technologies

Graphical Editor Technologies



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 with *EMF Transactions*
 - Undo/redo support
- Notation/view model
 - Domain-independent implementation in GMF, Graphiti

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

The logo for Graphiti, featuring the word "Graphiti" in a white, stylized, 3D-effect font with a drop shadow, set against a dark blue rectangular background.

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



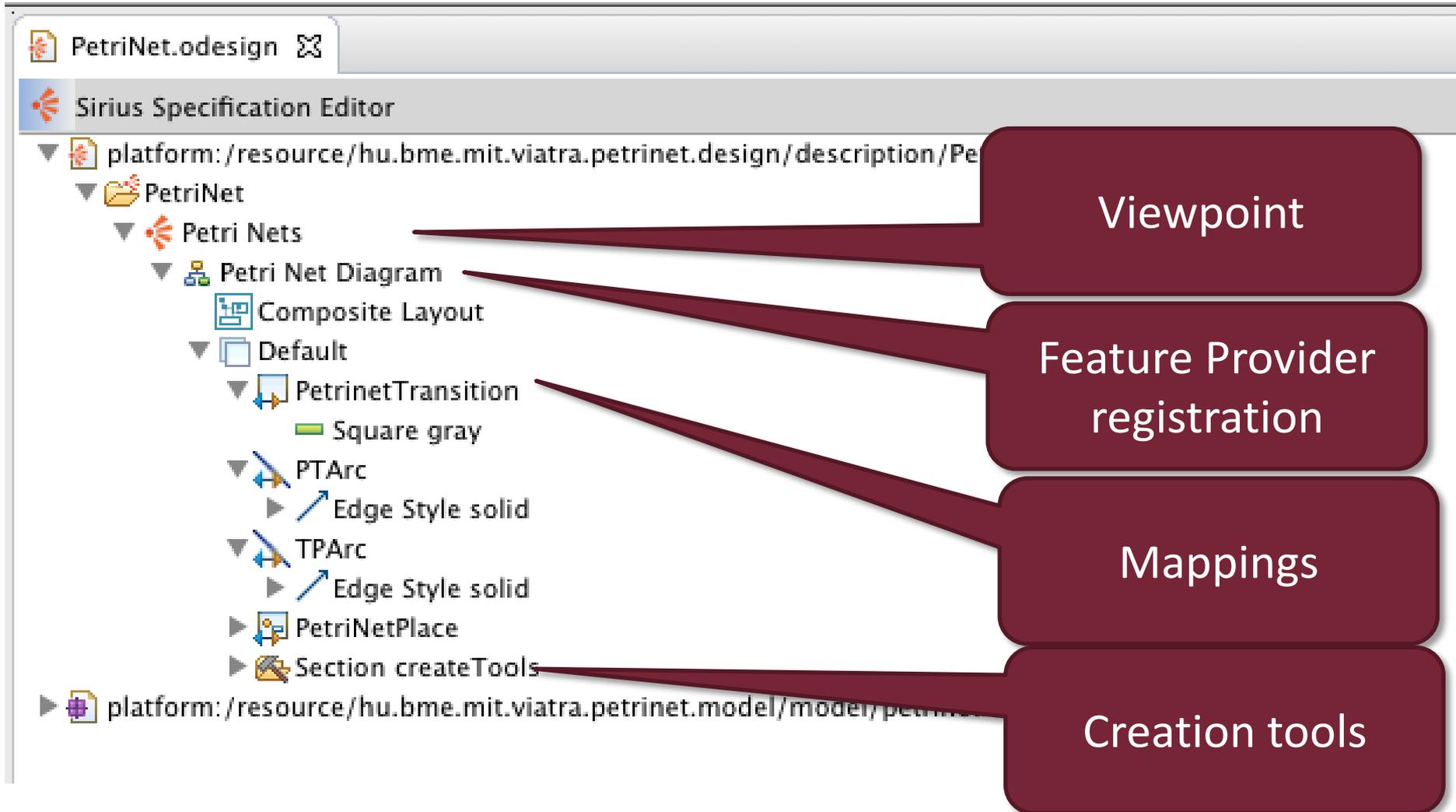
- New modeling project
 - Since 2013 on eclipse.org
 - Previously Obeo Designer – commercial tool
- How stable is it?
 - Old projects are to be migrated
 - Version history
 - 0.9: 2013-12-10
 - 1.0: 2014-06-25 (Kepler release train)
 - ...
 - 5.1: 2017-10-26
 - ...

Sirius Viewpoints

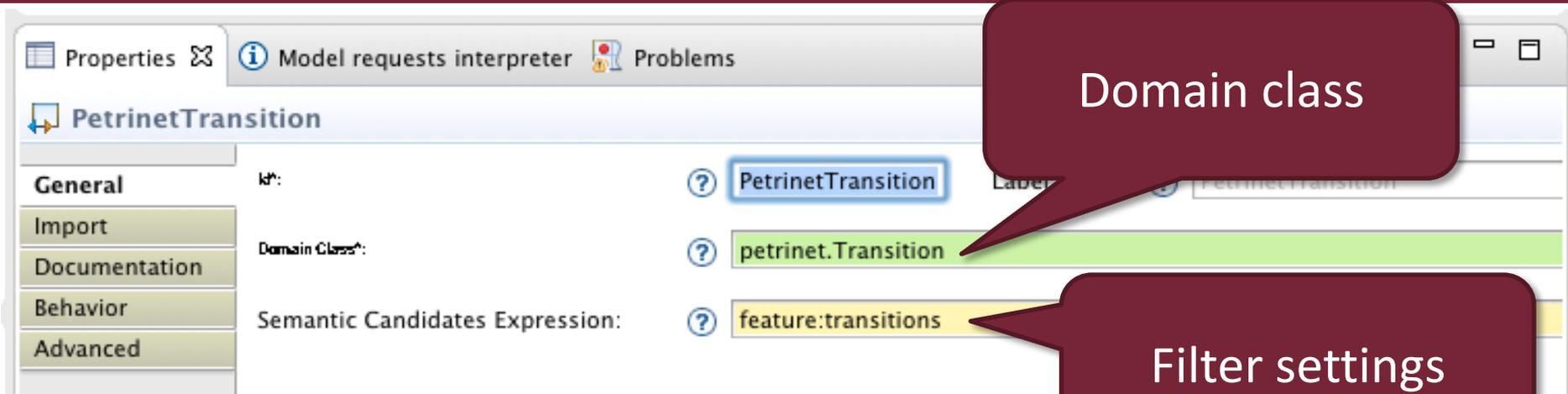


- Base concept:
 - Every diagram is a view of the model
 - With a defined syntax
 - **Graphical**
 - Table/Tree syntax
 - Xtext-based textual syntax
- Viewpoint definition
 - Viewpoint specification model

Viewpoint Specification Model



Node & Edge Mapping



Properties Model requests interpreter Problems

PetriNetTransition

General **id:** PetriNetTransition Label: PetriNetTransition

Import **Domain Class*:** petriinet.Transition

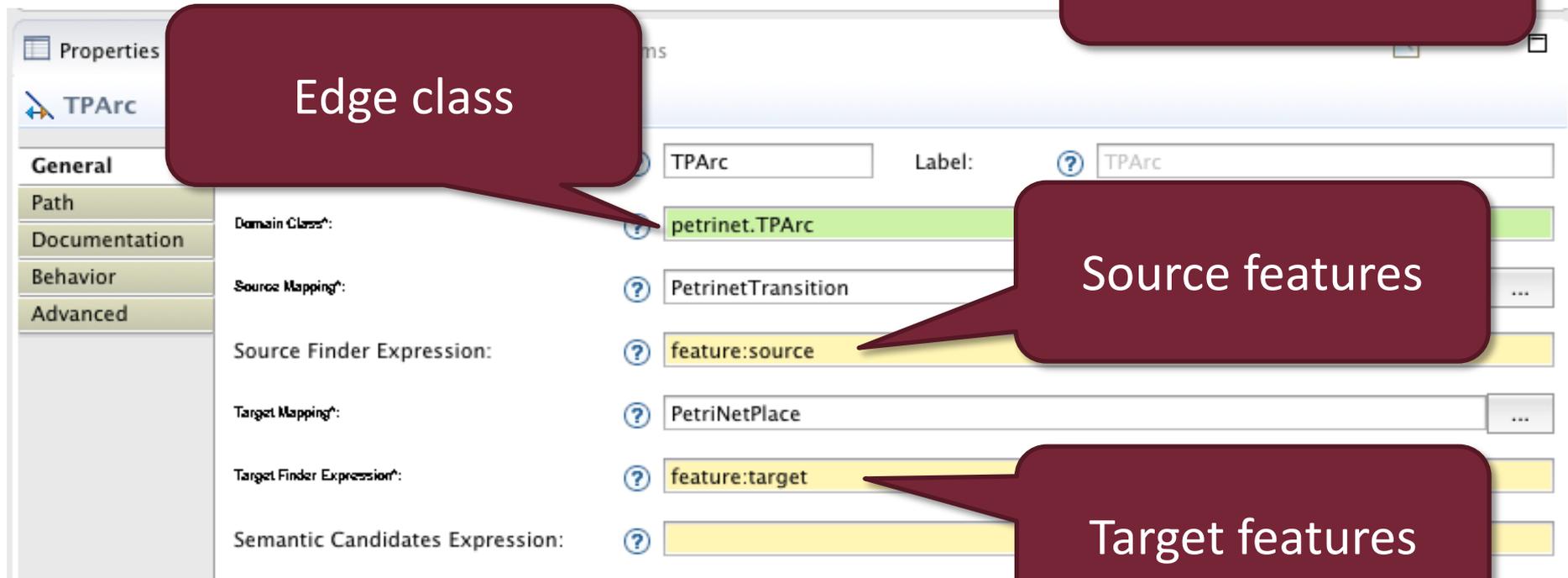
Documentation **Semantic Candidates Expression:** feature:transitions

Behavior

Advanced

Domain class

Filter settings



Properties ms

TPArc

General **id:** TPArc Label: TPArc

Path **Domain Class*:** petriinet.TPArc

Documentation **Source Mapping*:** PetriNetTransition

Behavior **Source Finder Expression:** feature:source

Advanced **Target Mapping*:** PetriNetPlace

Target Finder Expression*: feature:target

Semantic Candidates Expression:

Edge class

Source features

Target features

Feature Selection

■ Interpreted **model query** 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
 - (x)=Set name

Tool parameter variables

Model creation sequence

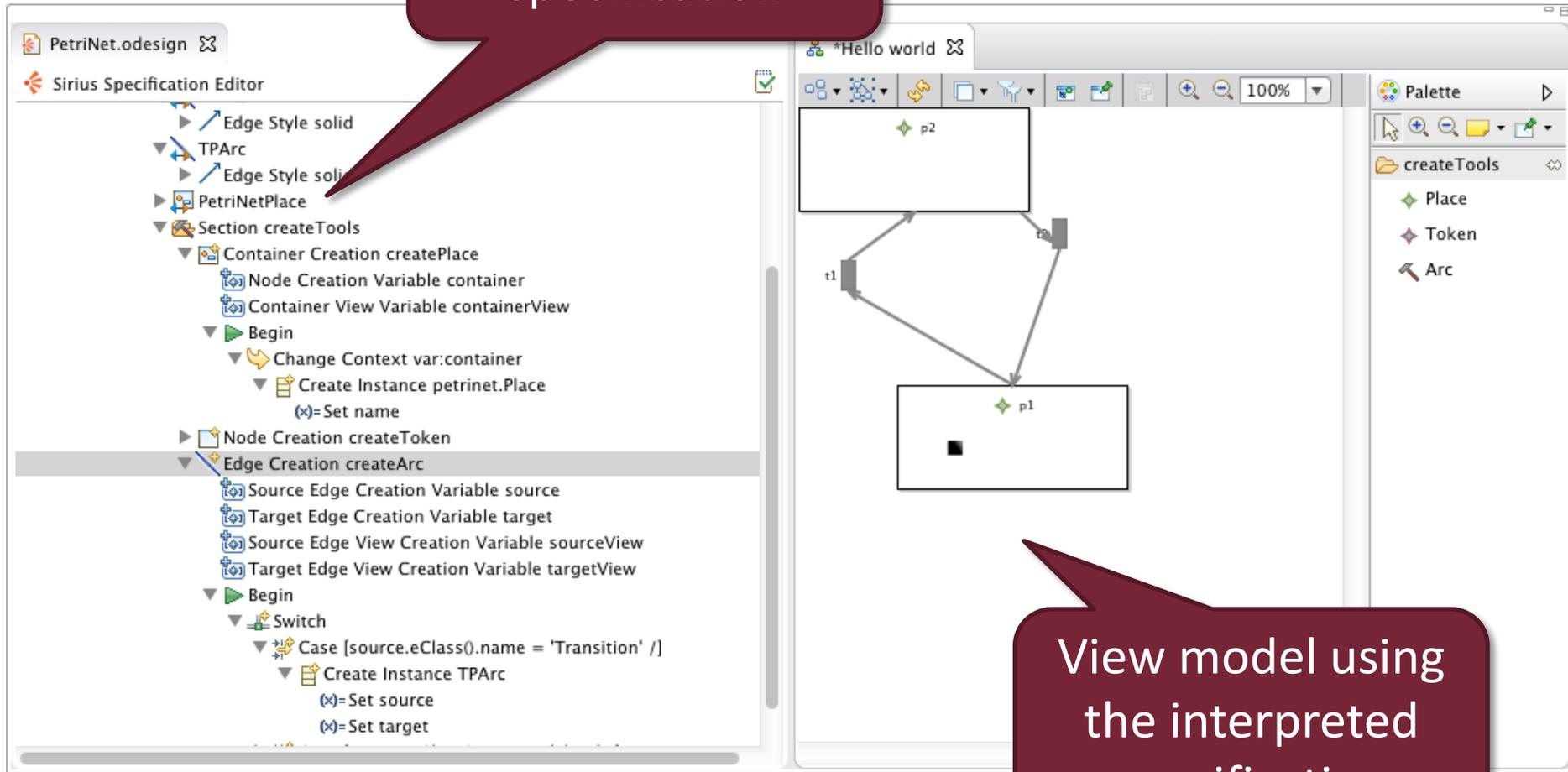
Different variables

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 TPArc
 - (x)=Set source
 - (x)=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

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