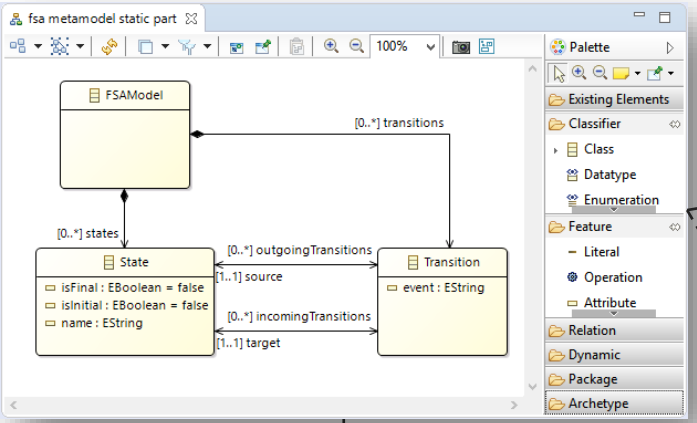
M6: Graphical Modeling Languages

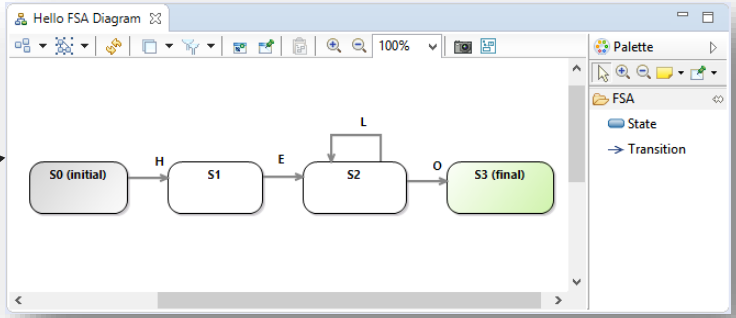
Business Informatics Group
Institute of Software Technology and Interactive Systems
TU Wien
Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at

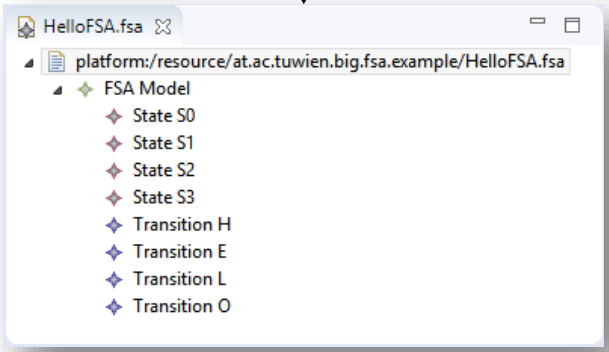
Recap

Metamodel
Defines Abstract Syntax



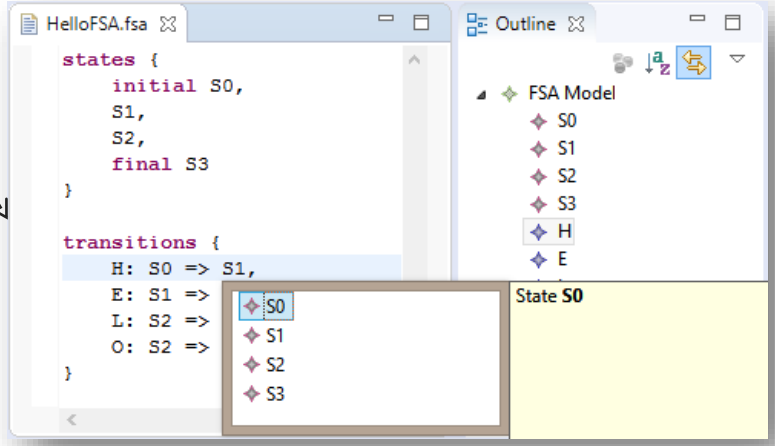
Graphical Editor
Follows Graphical Concrete Syntax





Generic EMF Tree Editor

Textual Editor
Follows Textual Concrete Syntax



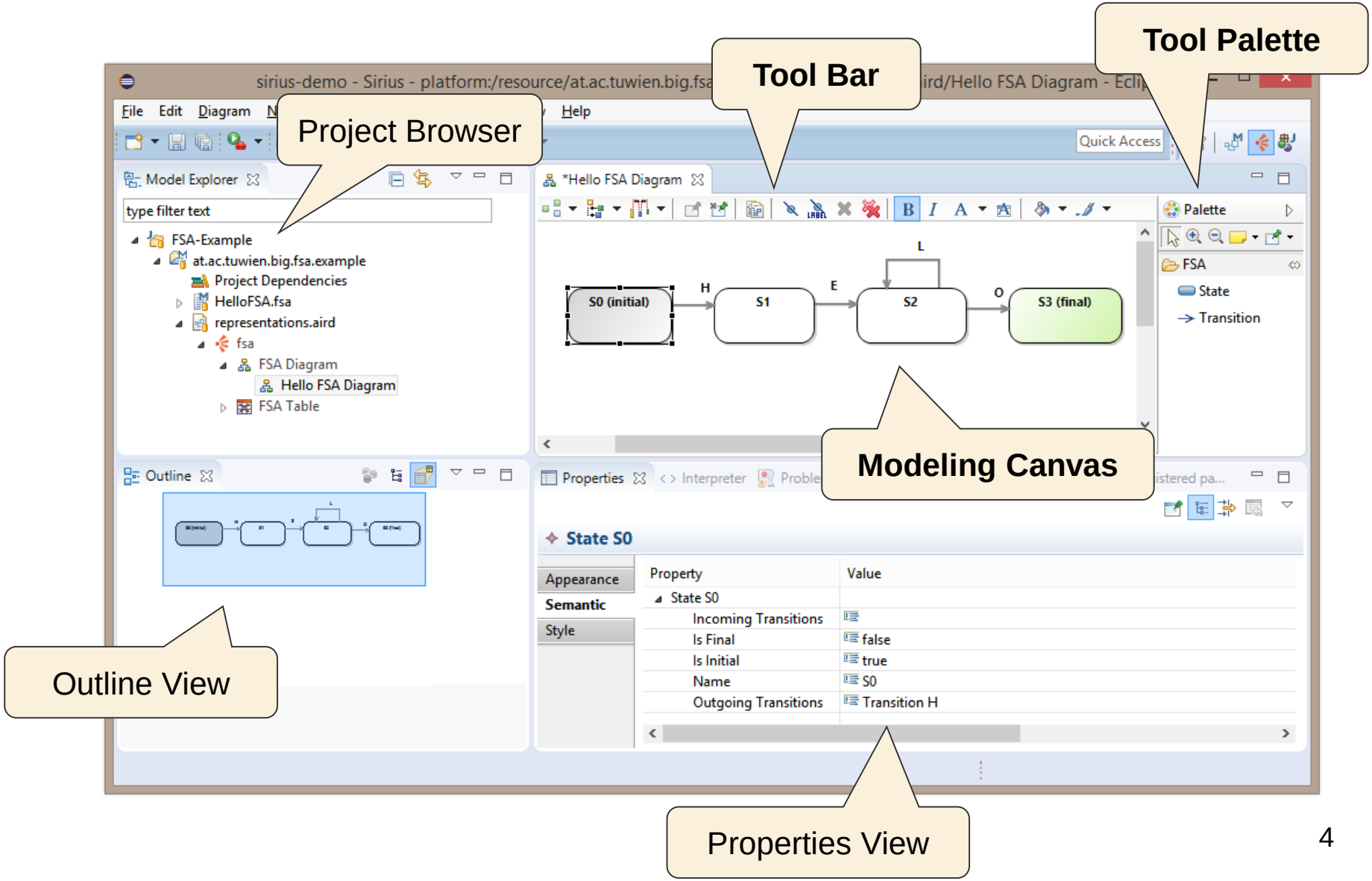
M7: Graphical Modeling Languages

Contents

- **Anatomy of Graphical Modeling Languages**
- Graphical Concrete Syntax Approaches
  - Mapping-based Approach
  - Annotation-based Approach
  - API-based Approach
- Eclipse Sirius Framework

Graphical Modeling Editors

Components

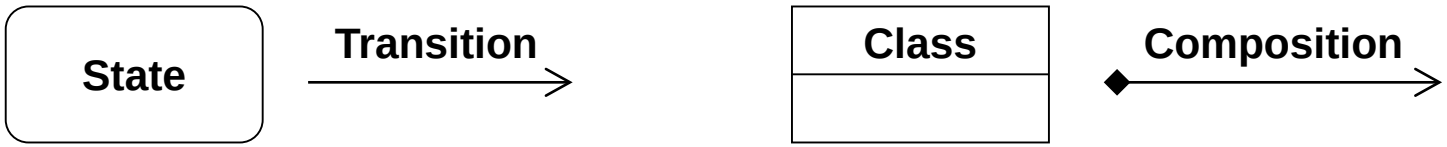


Graphical Modeling Editors

Specification

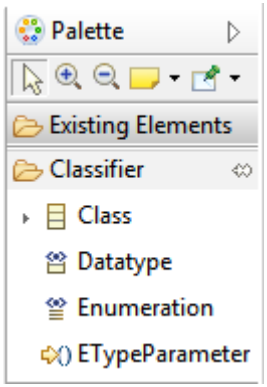
- A graphical modeling editor requires the specification of two main components:

  1. **Graphical concrete syntax** of the modeling language

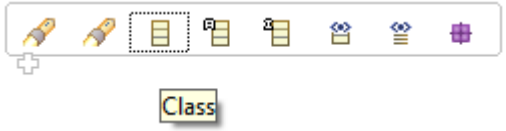


  2. **Editing behavior** specific to the modeling language

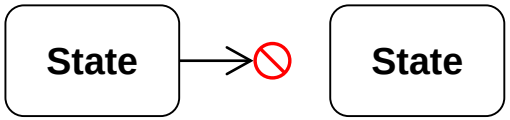
Tool Palette



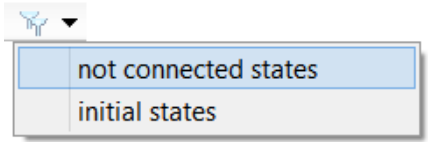
Action Bar



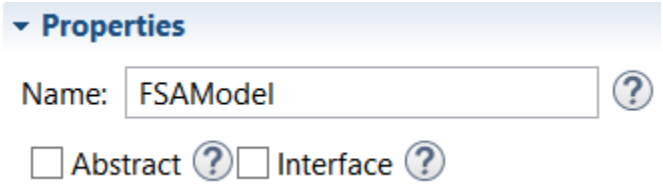
Connection Handlers



Filter



Properties View

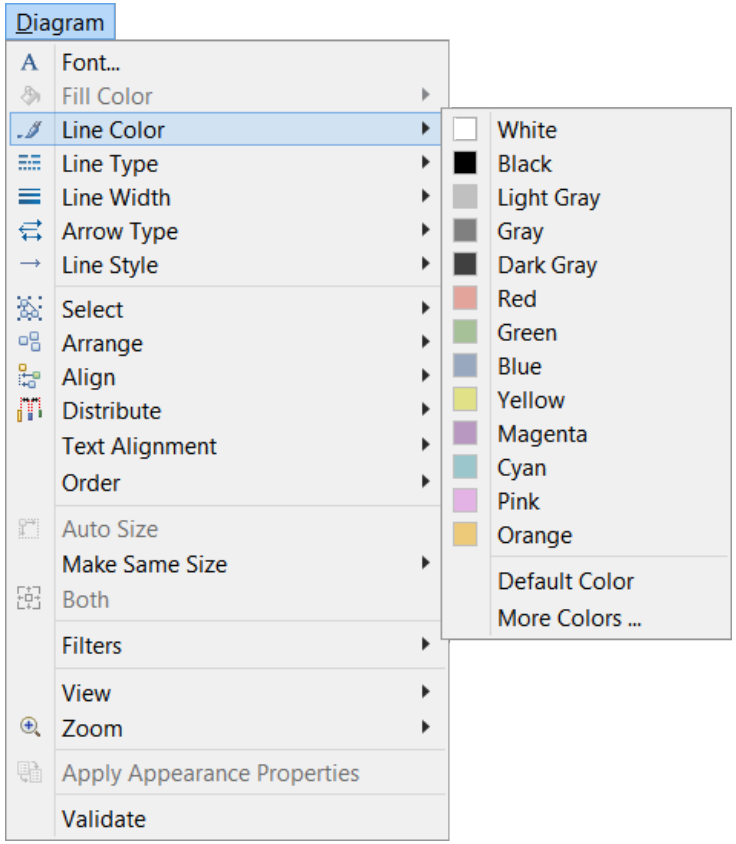


Graphical Modeling Editors

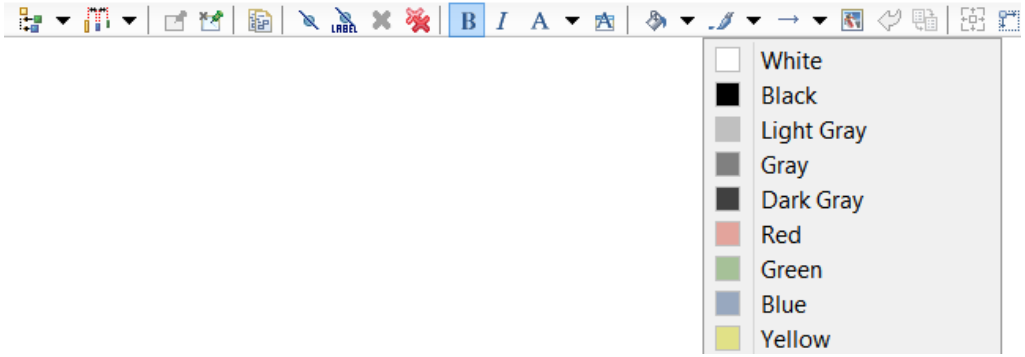
Generic Editing Features

- Graphical modeling editors provide **generic editing features** for changing the **appearance** of diagrams and **interacting** with diagrams

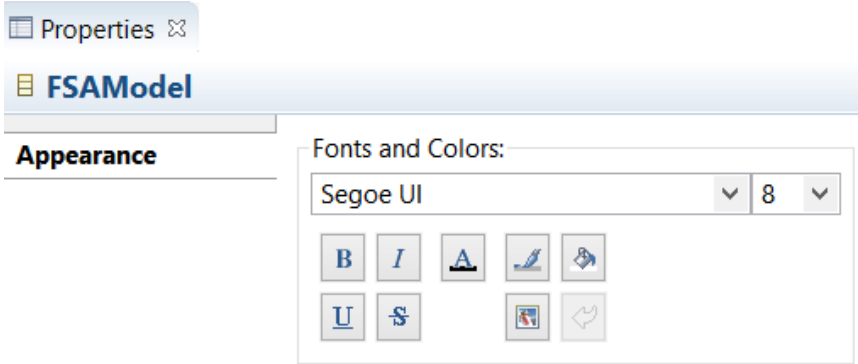
Actions Menu



Actions Tool Bar

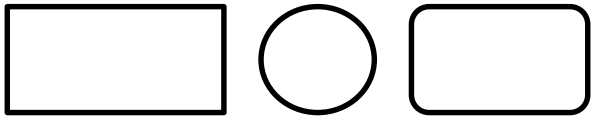


Appearance Properties



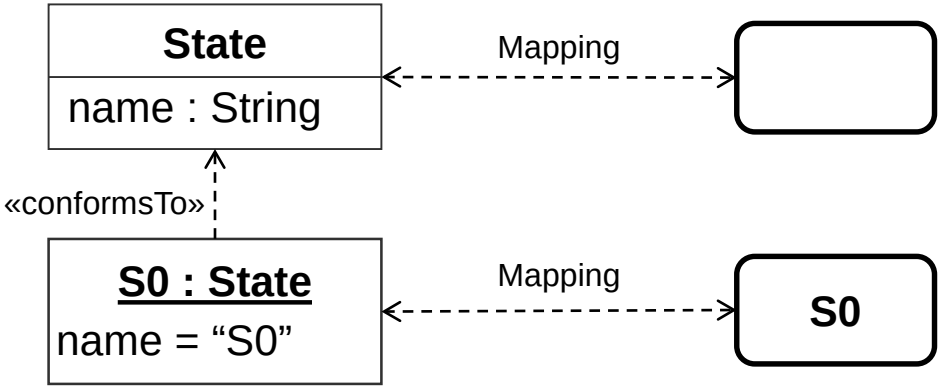
Graphical Concrete Syntax

- A Graphical Concrete Syntax (GCS) consists of
  - **Graphical symbols**
    - e.g., rectangles, circles, rounded rectangles

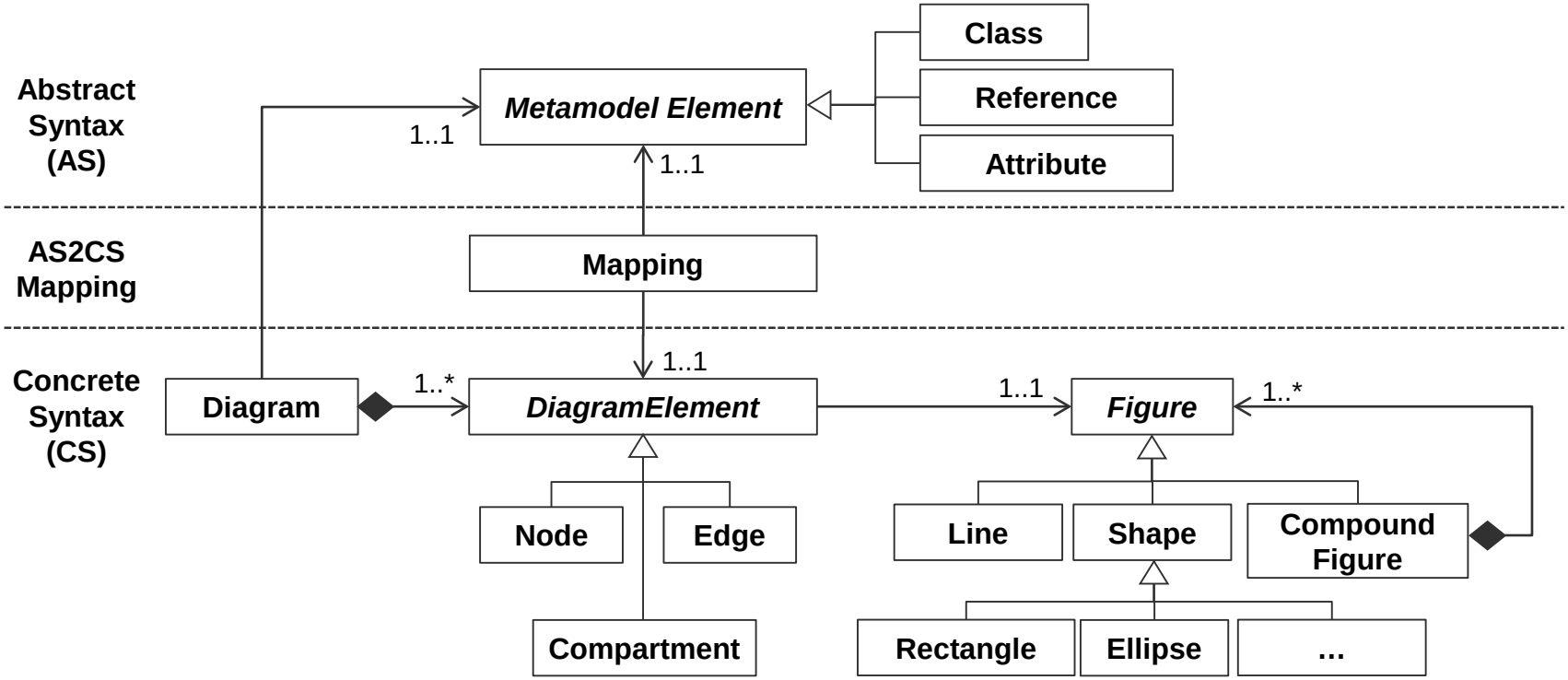


  - **Compositional rules**
    - e.g., nesting of elements in compartments, border nodes

  - **Mapping** between **graphical symbols** and **abstract syntax elements**
    - e.g., instances of a metaclass “State” are visualized by rounded rectangles



Generic Metamodel for GCS



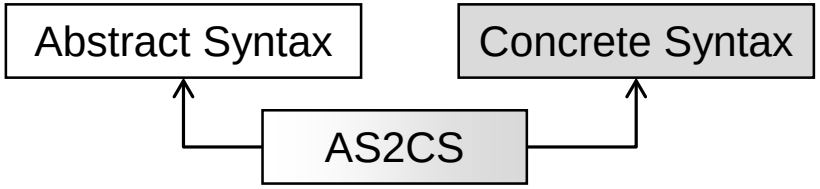
M7: Graphical Modeling Languages

Contents

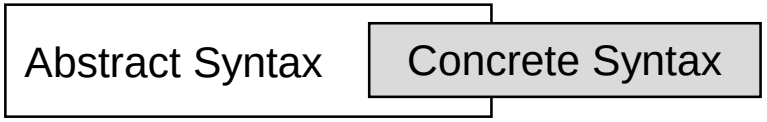
- Anatomy of Graphical Modeling Languages
- **Graphical Concrete Syntax Approaches**
  - Mapping-based Approach
  - Annotation-based Approach
  - API-based Approach
- Eclipse Sirius Framework

GCS Approaches

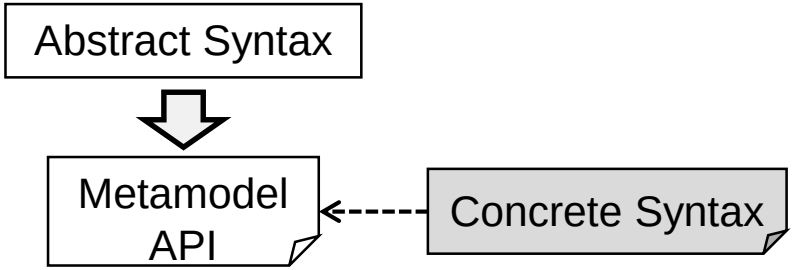
- **Mapping-based**
  - Explicit mapping model between abstract syntax, i.e., the metamodel, and concrete syntax
  - Example: GMF, Sirius



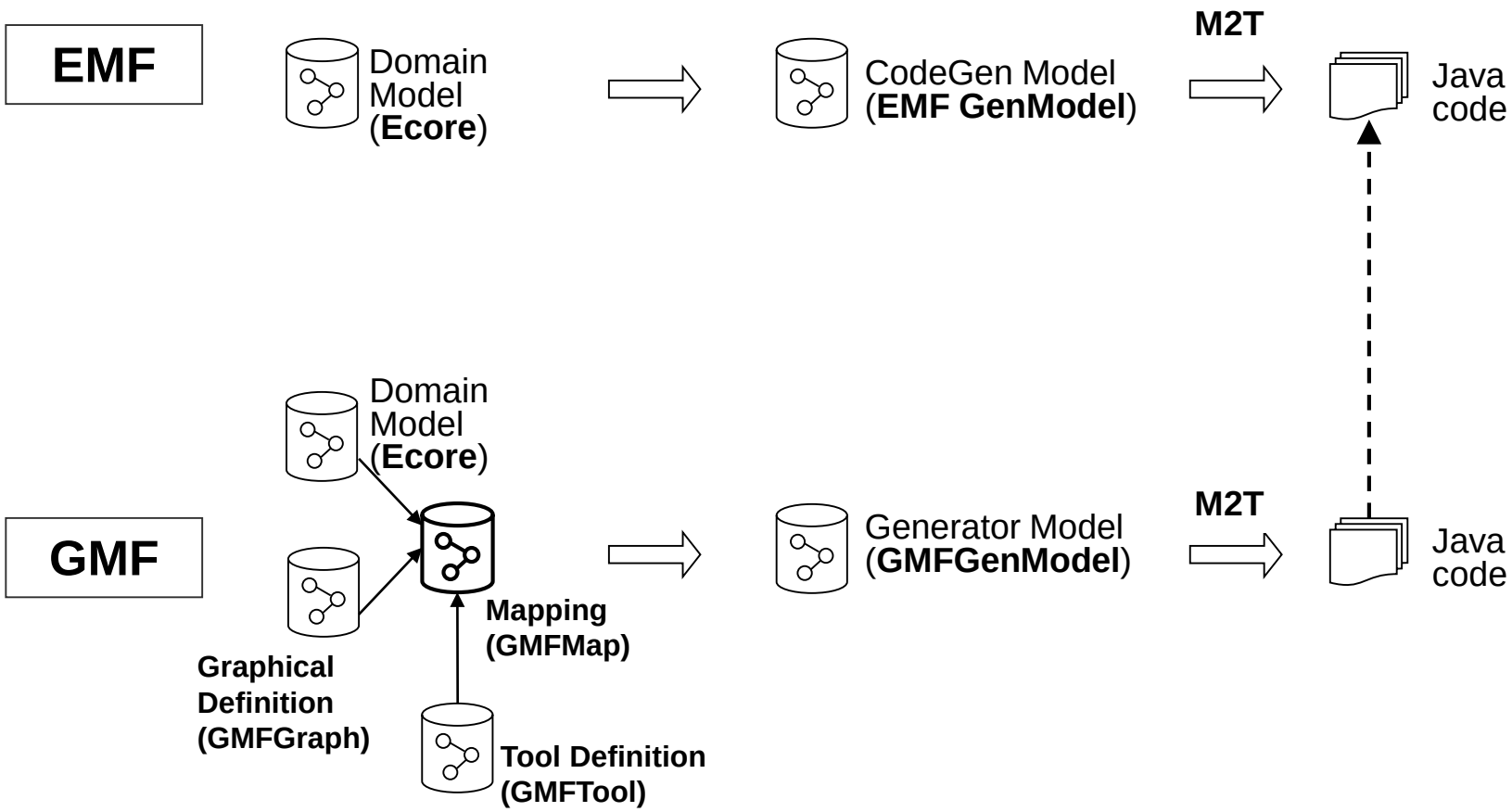
- **Annotation-based**
  - The metamodel is annotated with concrete syntax information
  - Example: Eugenia



- **API-based**
  - Concrete syntax is described by a programming language using a dedicated API for graphical modeling editors
  - Example: Graphiti



Mapping-based Approach: GMF



http://www.eclipse.org/modeling/gmp/

Annotation-based Approach: Eugenia

- **Ecore** metamodels are **annotated** with GCS information
- From the annotated metamodels, a **generator** produces GMF models
- GMF generators are reused to produce the actual modeling editors
- http://www.eclipse.org/epsilon/doc/eugenia/

Be aware:
Application of MDE techniques for developing MDE tools!

Annotation-based Approach: Eugenia

Eugenia Annotations (Excerpt)

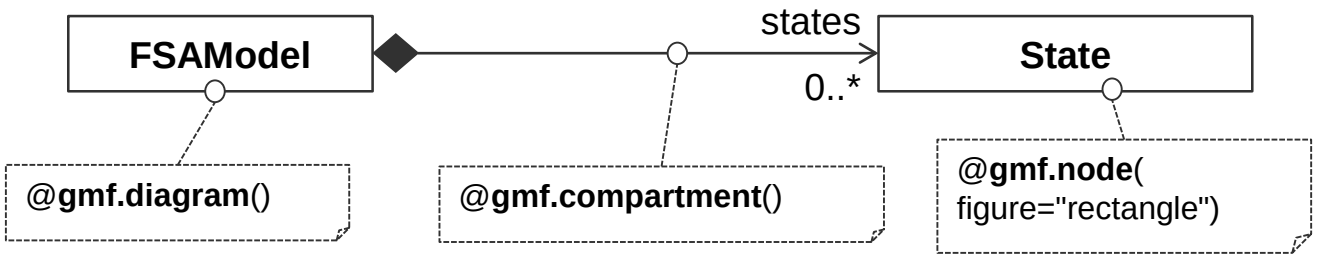
- **Diagram**
  - For marking the root class of the metamodel that directly or transitively contains all other classes
  - Represents the modeling canvas
- **Node**
  - For marking classes that should be represented by nodes such as rectangles, circles, …
- **Link**
  - For marking references or classes that should be visualized as lines between two nodes
- **Compartment**
  - For marking elements that may be nested in their containers directly
- **Label**
  - For marking attributes that should be shown in the diagram representation of the models

Annotation-based Approach: Eugenia

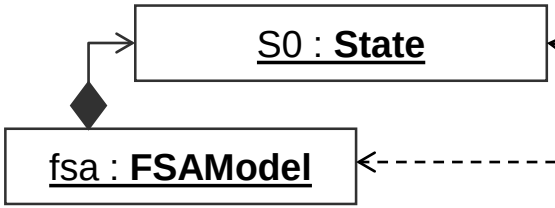
Eugenia Example #1

- *HypertextLayer* elements should be **directly embeddable** in the **modeling canvas** that represents *WebModels*

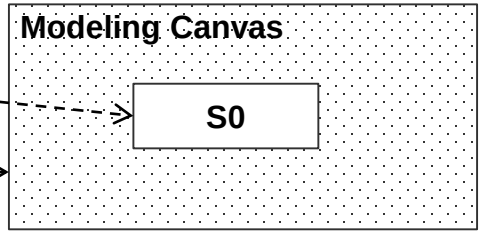
Metamodel with EuGENia annotations



Model fragment in AS



Model fragment in GCS



API-based Approach: Graphiti

- Powerful **programming framework** for developing graphical modeling editors
  - API hides complexity of GEF and Draw2D
  - API follows a programming model that is based on features

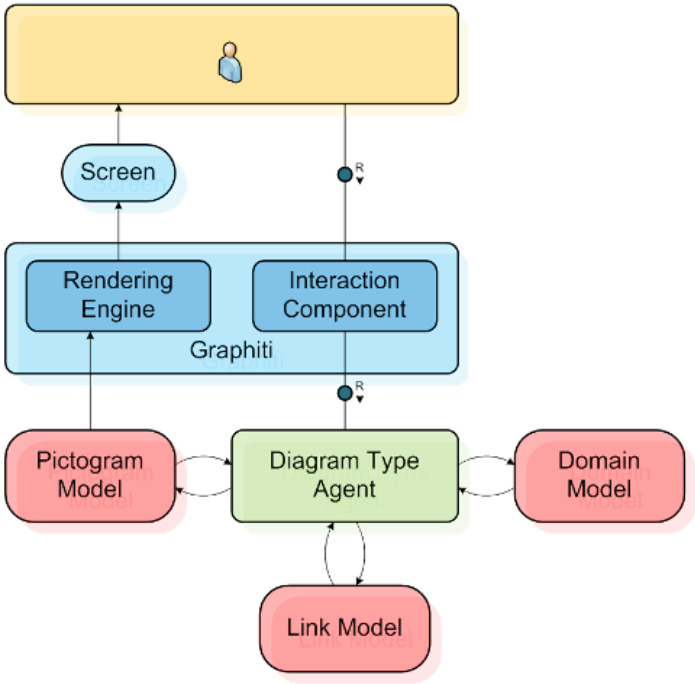
- Incremental development using default implementations
  - Get first results fast using default implementations:
    Move, resize, delete, ... etc. work immediately for added elements
  - Add feature by feature through adding custom implementations

- https://eclipse.org/graphiti

API-based Approach: Graphiti

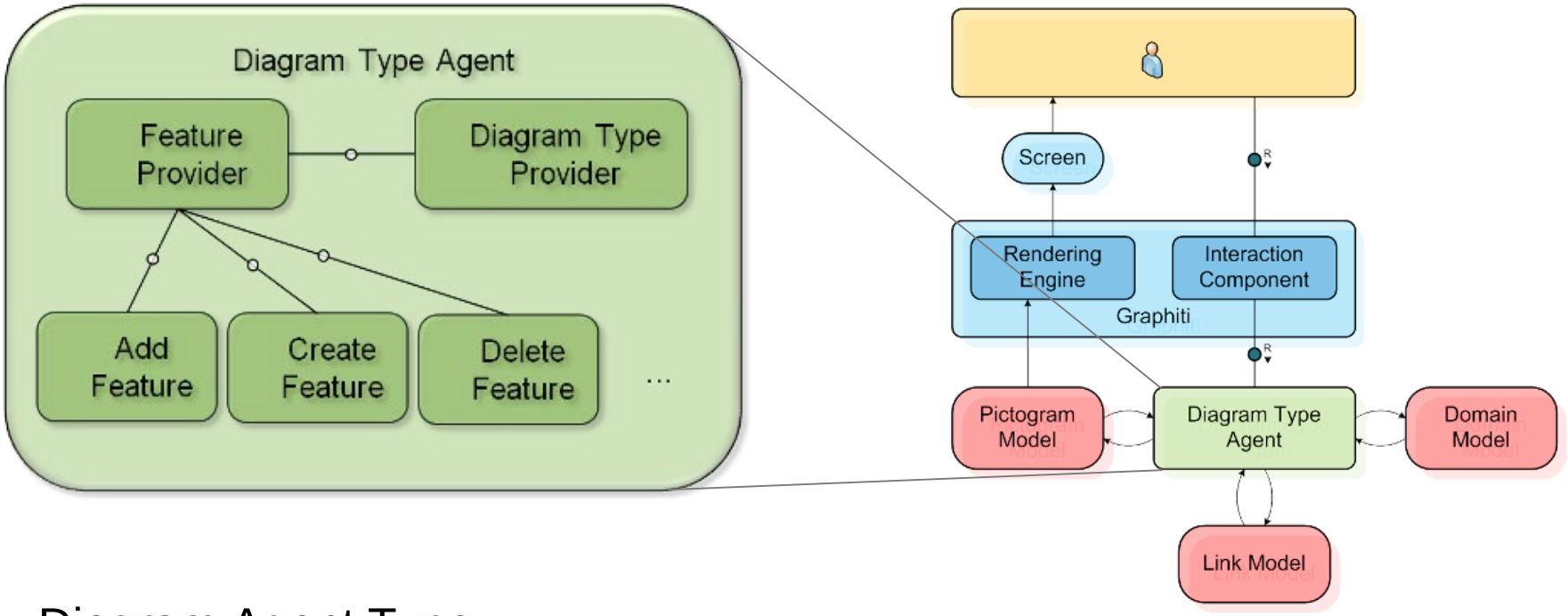
Architecture

- Domain Model
  - Contains the data to be visualized (i.e., the EMF-based model)
- Pictogram Model
  - Describes the visualization and the hierarchy of concrete syntax elements
- Link Model
  - Connects the domain model and the pictogram model



API-based Approach: Graphiti

Diagram Type Agent



- Diagram Agent Type
  - Features manage (create, delete, ...) pictogram, link, and domain model elements
  - Features have to be implemented (by sub-classing base features) and added to the feature provider

API-based Approach: Graphiti

Graphiti Example

● **Graphical representation of initial node**

- Example: Add feature for initial nodes of activity diagrams
  - Adds the graphical representation of an initial node to an activity diagram

```
public class AddInitialNodeFeature
      extends AbstractAddFeature {

  @Override
  public PictogramElement add(IAddContext context) {
    Object initialNode = context.getNewObject();
    ContainerShape targetContainer = (ContainerShape) context.getTargetContainer();

    Shape initialNodeShape = getPeCreateService().createShape(targetContainer, true);
    Ellipse initialNodeEllipse = getGaService().createEllipse(initialNodeShape);
    getGaService().setLocationAndSize(initialNodeEllipse, context.getX(),
       context.getY(), INITIAL_NODE_SIZE, INITIAL_NODE_SIZE);

    link(initialNodeShape, initialNode);

    return initialNodeShape;
  }

  ...

}
```

Retrieve **model element** for which a graphical representation should be added

Create the **graphical representation**

Map the graphical representation to the model element

M7: Graphical Modeling Languages

Contents

- Anatomy of Graphical Modeling Languages
- Graphical Concrete Syntax Approaches
  - Mapping-based Approach
  - Annotation-based Approach
  - API-based Approach
- **Eclipse Sirius Framework**

Eclipse Sirius

Introduction

- **Mapping-based approach** for defining the graphical concrete syntax of EMF-based modeling languages
- **Model-based approach**, i.e., the graphical concrete syntax of a modeling language is defined in a model (*Viewpoint Specification Model*)
  - *Viewpoint Specification Model* defines
    - Graphical symbols (*Styles*)
    - Composition rules (*Viewpoints*, *Representations*, *Container Mappings*)
    - Mappings (*Viewpoints*, *Representations*, *Container Mappings*)
    - Editing behavior (*Tools*)
  - *Viewpoint Specification Model* is **dynamically interpreted** by the Sirius runtime
    - No code generation
    - Graphical representations are automatically updated in response to changes of the graphical concrete syntax
- http://www.eclipse.org/sirius/

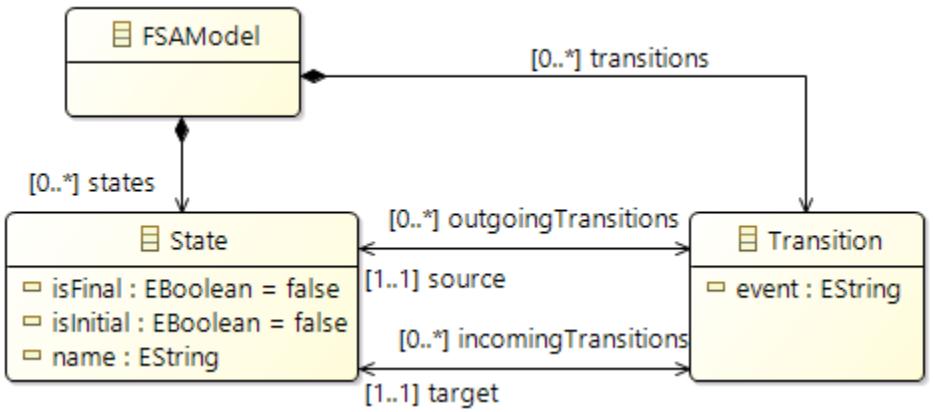
Example

Finite State Automata

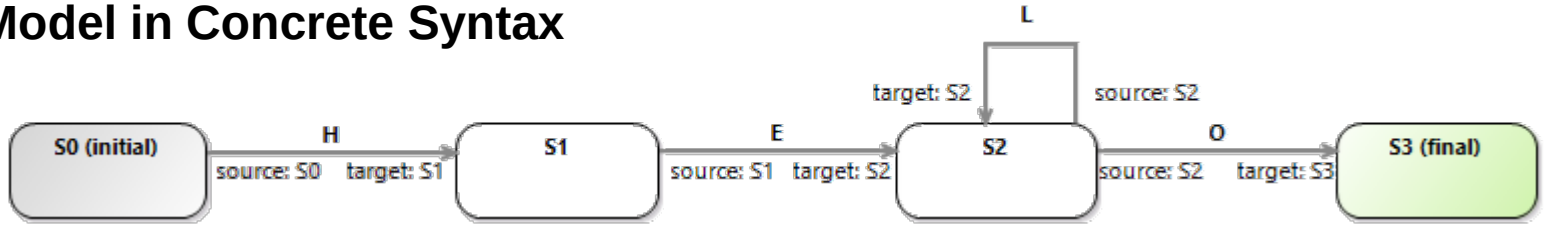
Concrete Syntax

Concept	Notation
State	<<name>>
Initial State	<<name>> (initial)
Final State	<<name>> (final)
Transition	<<event>> source: <<name>> target: <<name>>

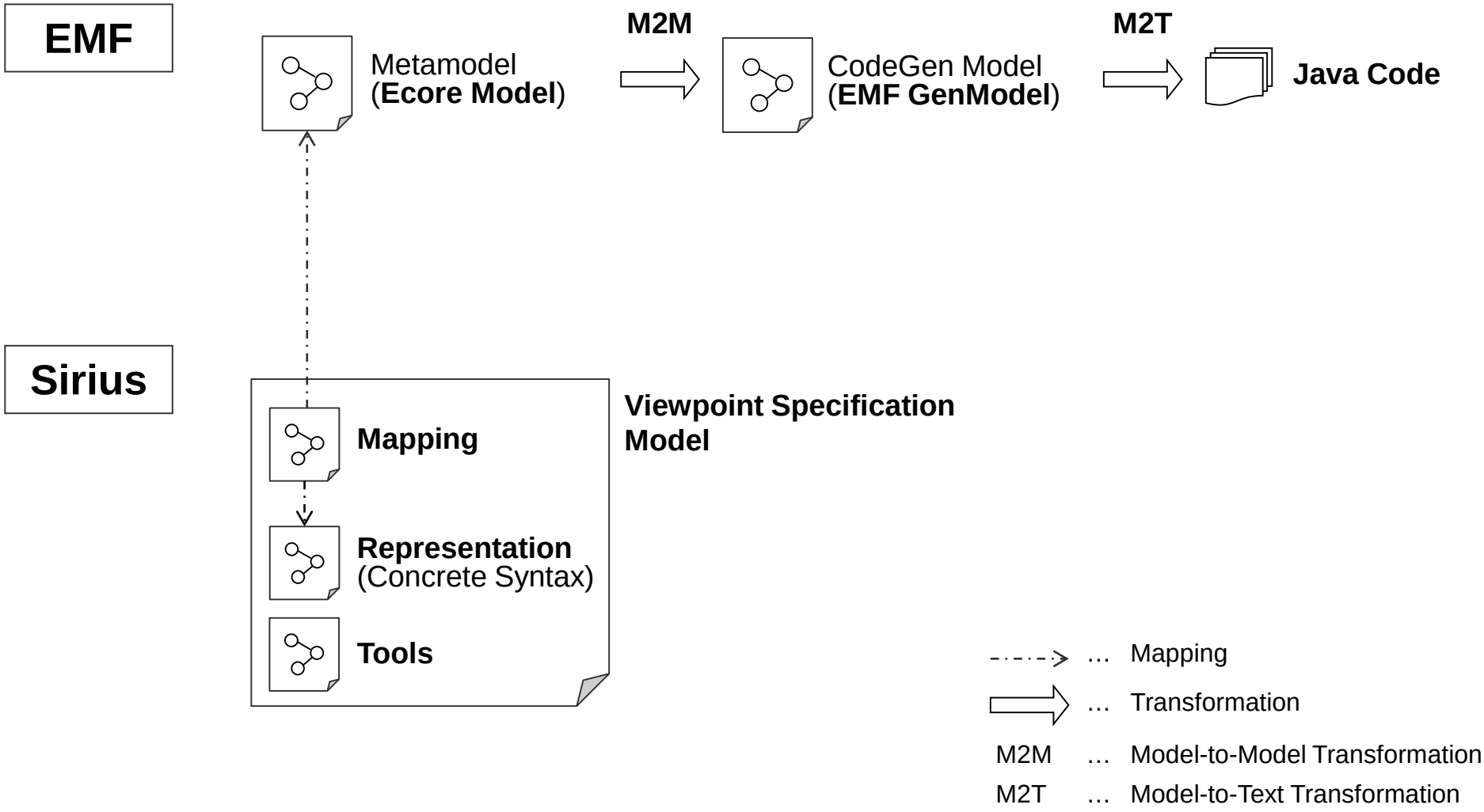
Abstract Syntax



Model in Concrete Syntax

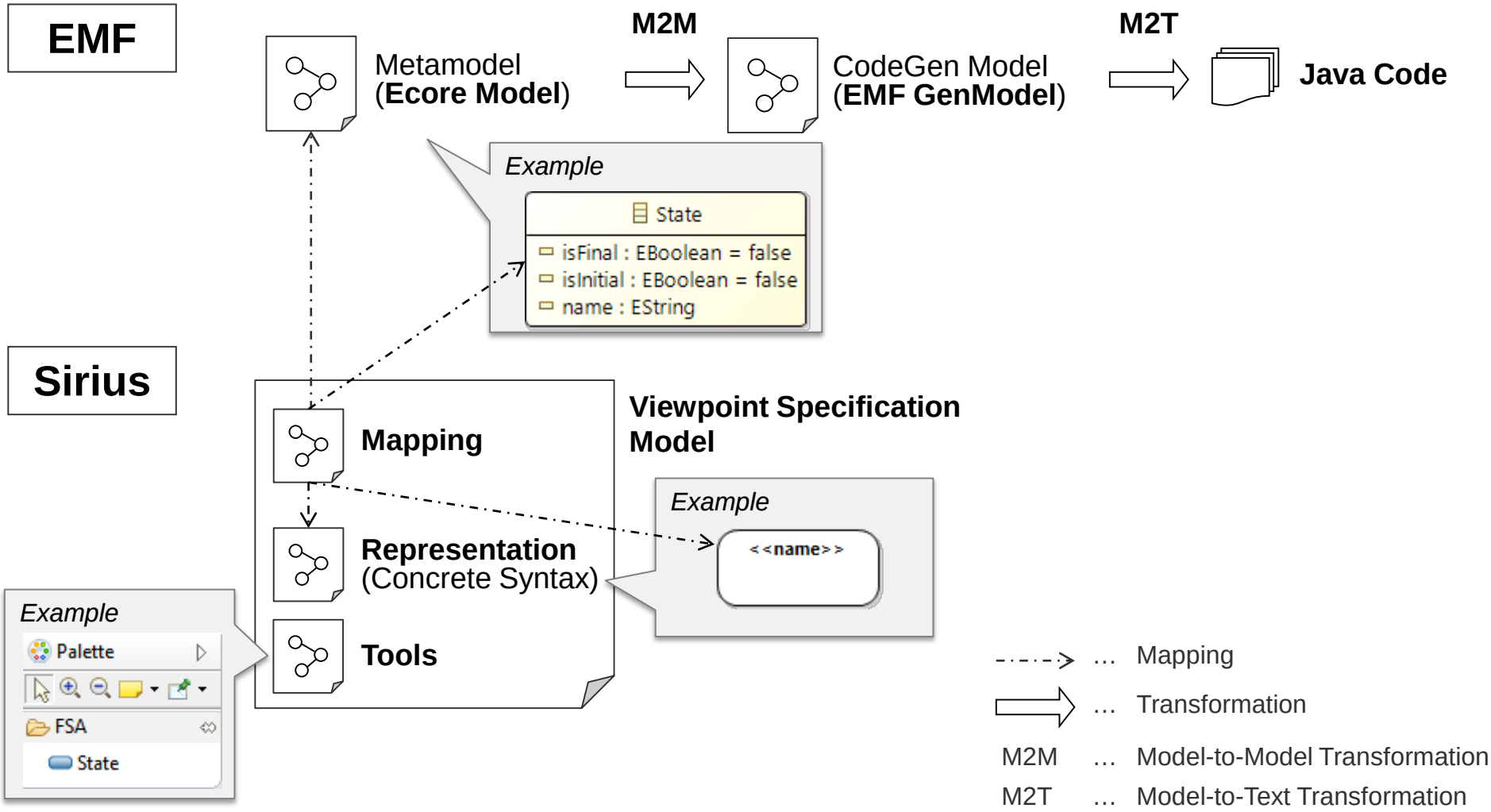


Overview

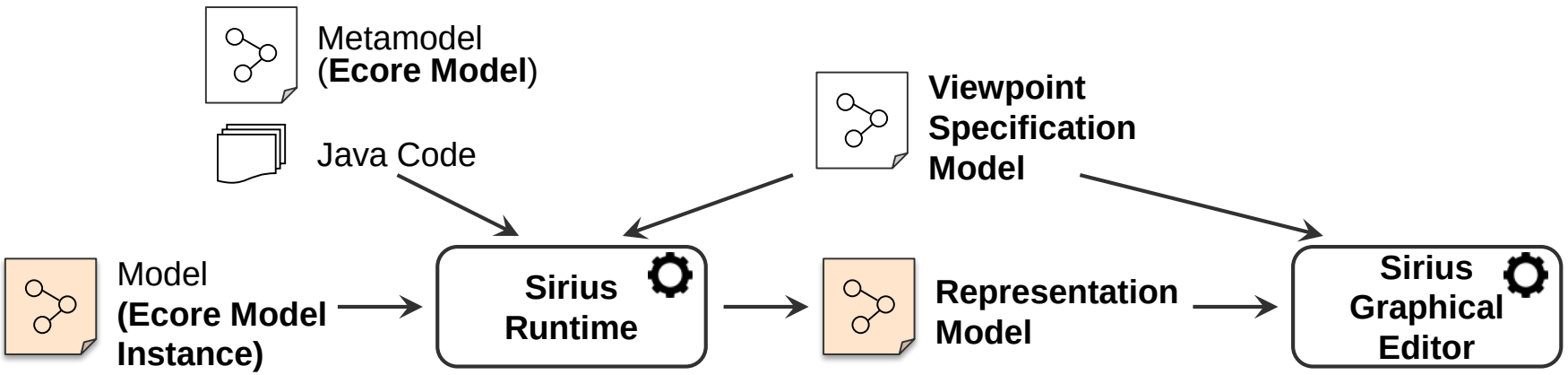

EMF
Metamodel
(Ecore Model)
M2M
CodeGen Model
(EMF GenModel)
M2T
Java Code
Sirius
Mapping
Representation
(Concrete Syntax)
Tools
Viewpoint Specification
Model
… Mapping
… Transformation
M2M … Model-to-Model Transformation
M2T … Model-to-Text Transformation

Overview

Example



Architecture



- **Sirius runtime:** Interprets viewpoint specification model and produces representation model for a user model
- **Representation model:** Defines the concrete representation of a user model according to the definitions in the viewpoint specification model (representation)
- **Graphical editor:** Shows the representation model
  - Enables **interactions** with user model as defined in viewpoint specification model (tools)
  - Interactions may trigger **modifications** of user model and representation model
  - Graphical editor is **incrementally refreshed** when user model, representation model, or viewpoint specification model is modified

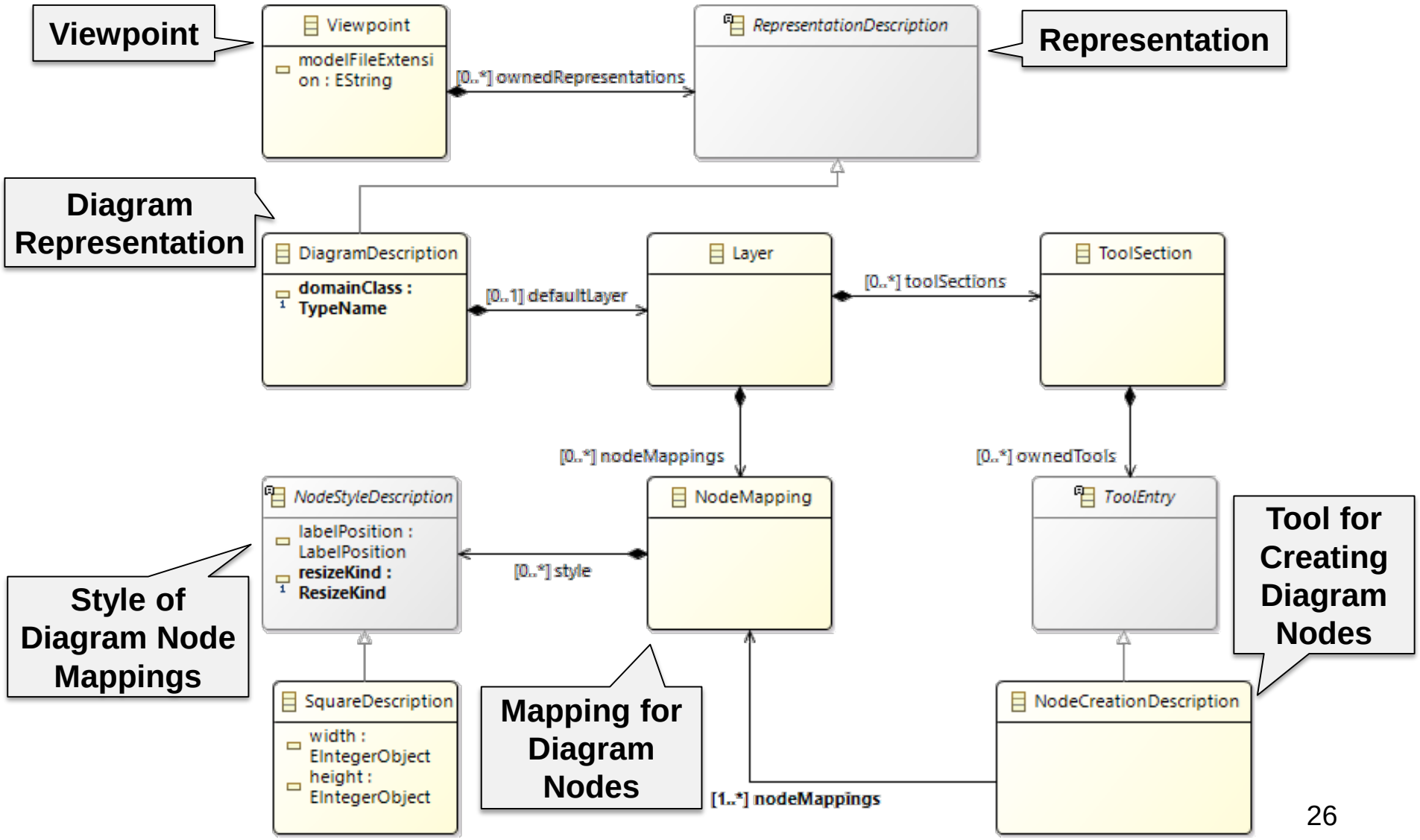
Viewpoint Specification Model

Definitions in a viewpoint specification model

- **Viewpoint**: Set of related representations intended for a certain use case (e.g., different view points for different stakeholders)
- **Representation**: Concrete representation of a model defining the structure and appearance of the model, and the possible interactions with the model (e.g., diagram representations)
- **Mapping:** Selection of model elements that should be shown in a representation (e.g., mapping of model elements to diagram nodes)
- **Style:** Visual appearance of model elements selected by a mapping (e.g., visualization of diagram nodes as rounded rectangles)
- **Tool:** Defines interactions with the model that are possible in a representation (e.g., creation tools available in tool palette for creating new model elements via diagrams)

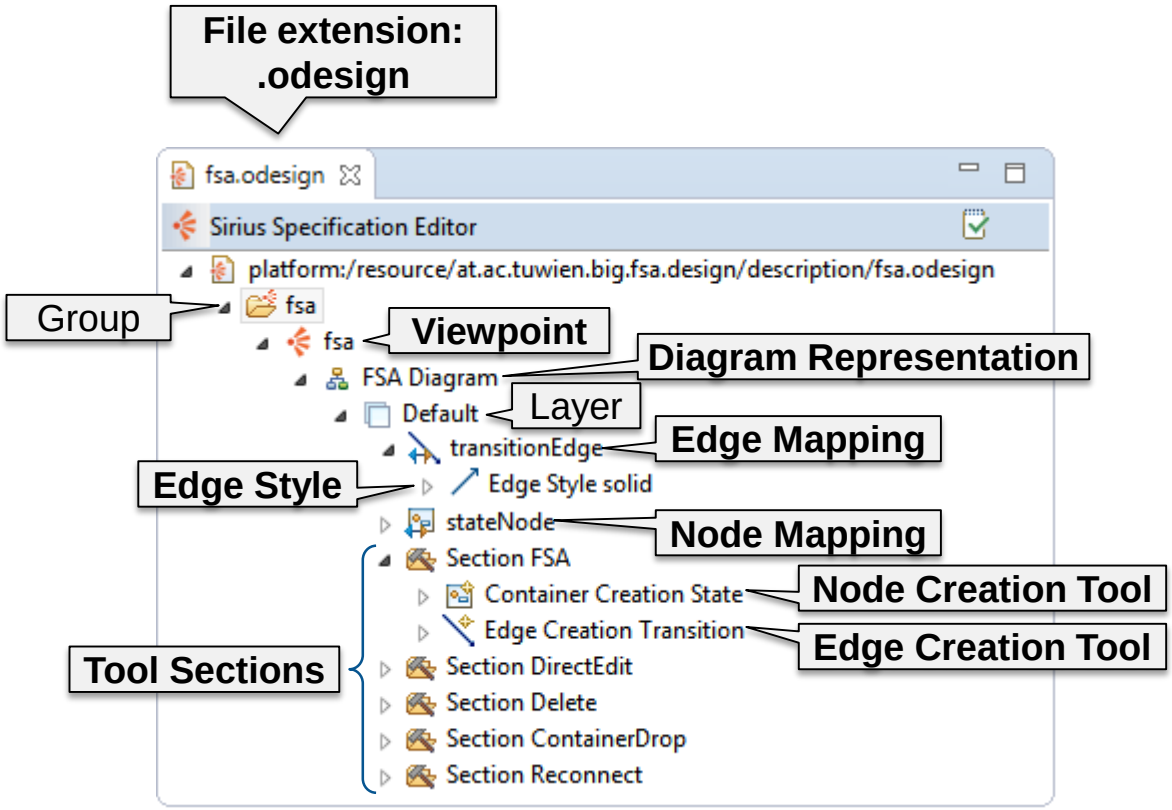
Viewpoint Specification Model

Metamodel



Viewpoint Specification Model

Example



Groups and Viewpoints

Group

- Top-level element serving as container
- Features:
  - **Name**: Never change the name because it is part of the identification of all contained elements (representation models will break)

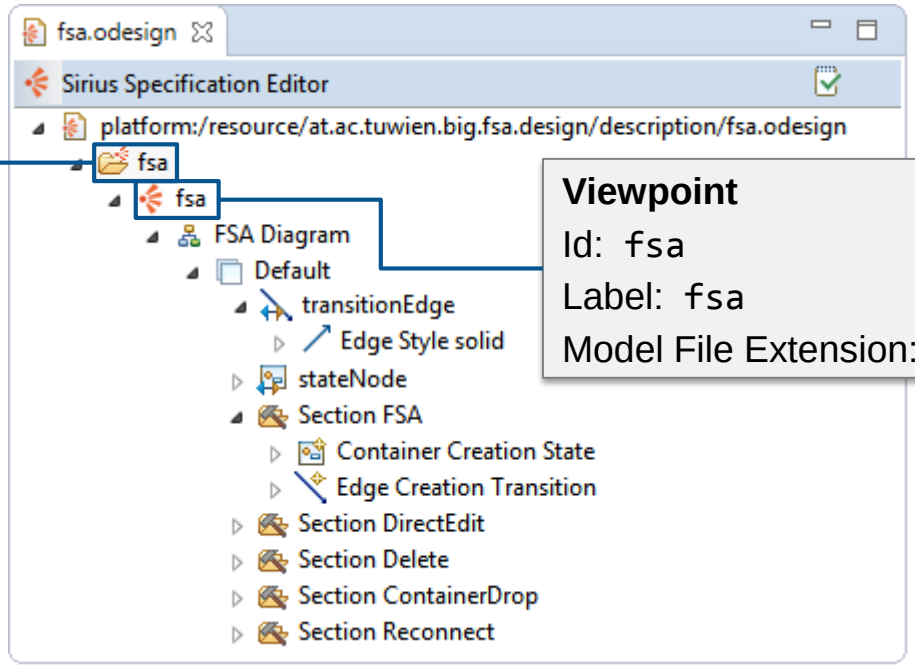
Viewpoint

- Defines a particular viewpoint on a model
- A viewpoint is associated with a particular type of model, i.e., a model conforming to a particular metamodel
- Features:
  - **Model File Extension**: Associates the viewpoint with model files having a particular file extension and, thus, with models conforming to a certain metamodel ("*" for any)
- Common features for viewpoints and contained elements:
  - **Id**: Mandatory identifier of the element; never change the identifier because it breaks existing representation models
  - **Label**: Label used to display the element in the editor

Groups and Viewpoints

Example

Group
Name: `fsa`



Viewpoint
Id: `fsa`
Label: `fsa`
Model File Extension: `fsa`

Representations

Representation Description and Diagram Description

Representation Description

- Defines a concrete representation of a model
- Supported representations: diagram, table, cross-table, tree, sequence diagram

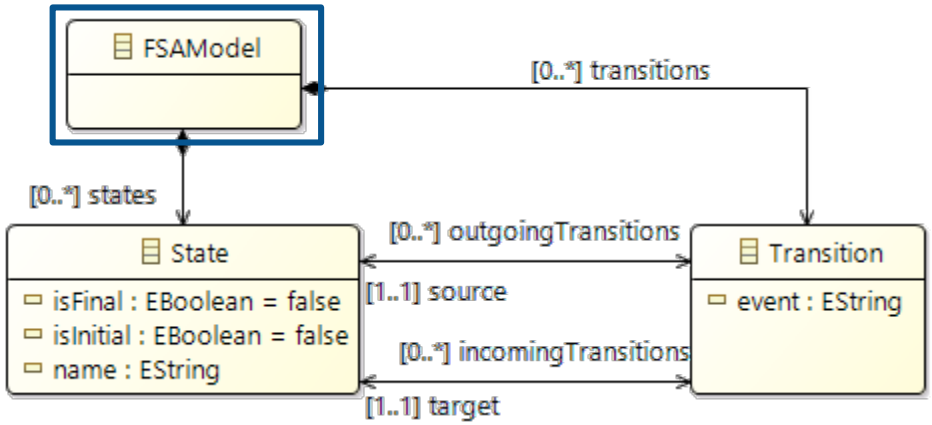
Emphasis of this lecture

Diagram Description

- Defines a diagrammatic representation of a model
- A diagram is always associated with a certain model element, which is usually the top-level element of your model serving as root container
- Features:
  - **Domain Class**: Metaclass that is the type of the model element to be associated with the diagram
- Components of a diagram description:
  - Layers defining diagram elements that should be shown
  - Tool sections
  - Validation rules and quick fixes
  - Filters of model elements to be shown on the diagram
  - Diagram style customizations

Representations

Diagram Description: Example



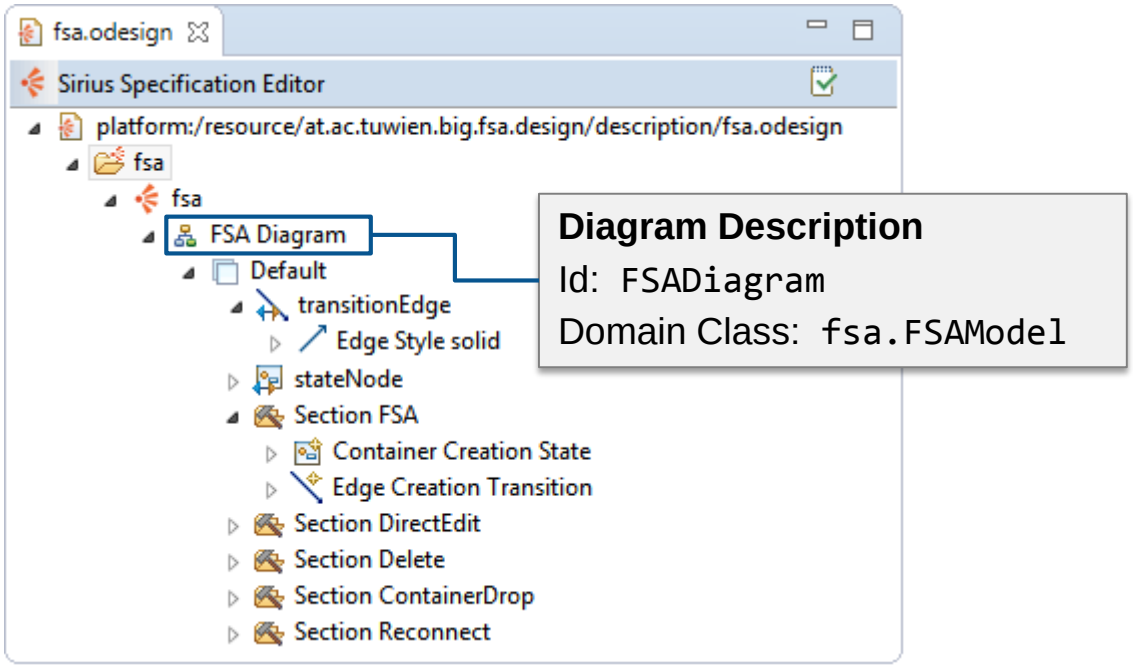


Diagram Descriptions

Layers and Mappings

Layer

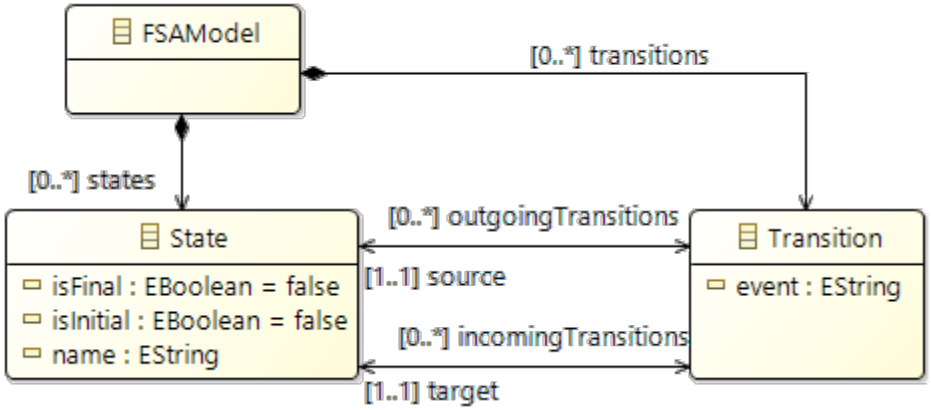
- All diagram elements and tools are part of a layer
- A diagram always has a default layer, which is always shown in the editor
- Additional layers can be enabled and disabled in the editor by the user

Mapping

- Selects the elements to be shown on the diagram (“diagram element”) and their graphical notation
- Supported mappings:
  - Diagram nodes
    - Node: Simple node
    - Container: Node that may contain sub nodes, bordered nodes or other containers
  - Diagram edges
    - Relation based edge: Edge for representing an EReference
    - Element based edge: Edge for representing a relationship expressed by a metaclass (EClass)

Diagram Descriptions

Layers and Mappings: Example



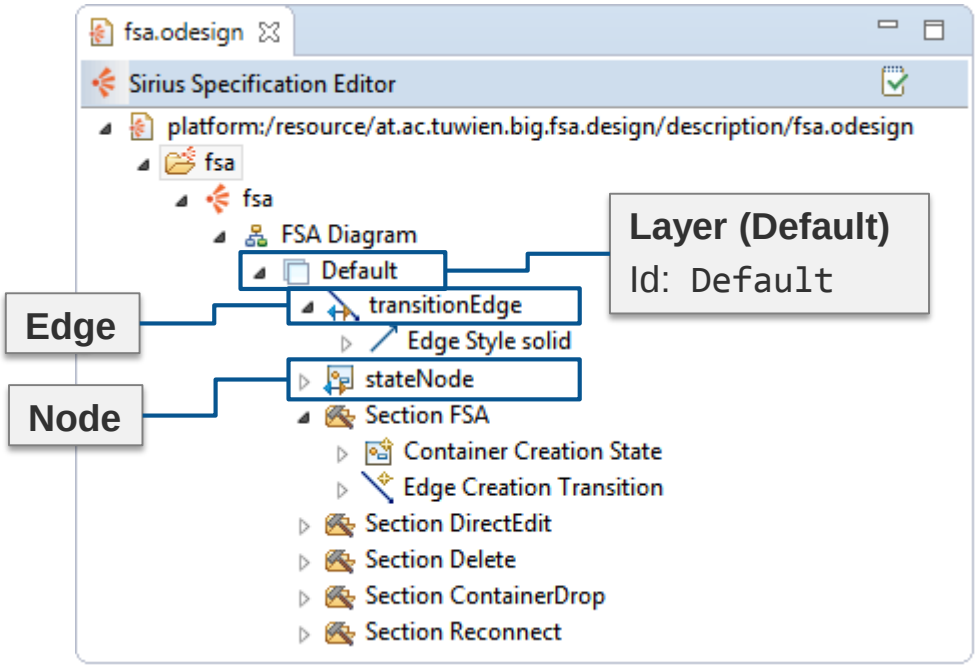


Diagram Elements

Nodes

Features of Nodes

- **Domain Class**: Type of the model elements (metaclass) for which the node defines the graphical notation (maps the concrete syntax to the abstract syntax)
- **Semantic Candidate Expression**: Further restricts the model elements that should be associated with the node
  - Example: Only the states of one particular FSA model should be displayed on the diagram

Nodes Styles

- **(Simple) Node**
  - **Basic Shape:**

Square Stroke Triangle Dot Ring

  - **Square:**

  - **Dot:**

  - **Note:**

  - **Diamond:**

  - **Ellipse:**

  - **Gauge**
  - **Image**
  - **Custom Style (Java implementation)**

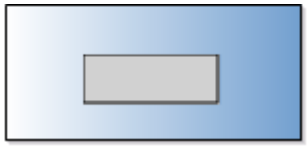
Node Styles

- **Container**
  - **Gradient**

  - **Parallelogram**

  - **Image**

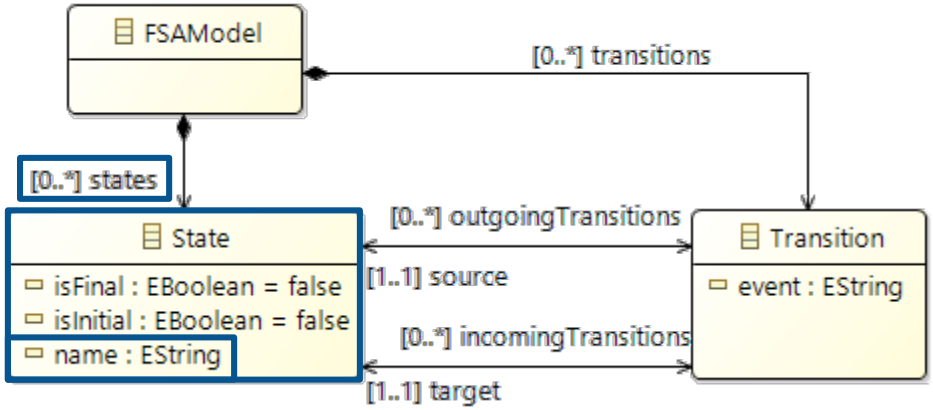
- **Sub Node**
  - Is a (simple) node within a container
  - Arrangement of nodes: free form, list

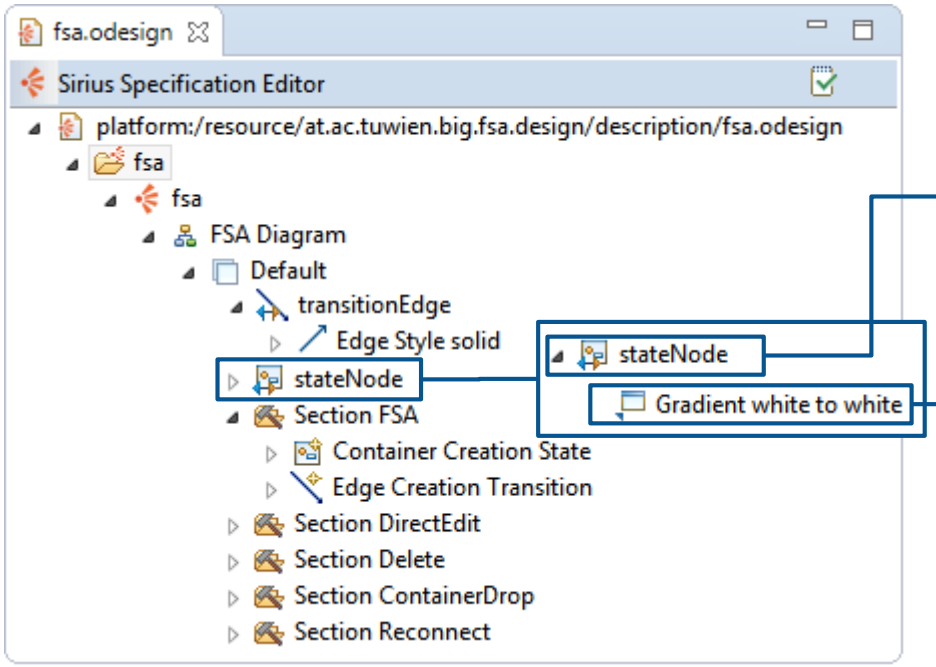


- **Bordered Node**
  - Is a (simple) node shown on the borders of a (simple) node or container

Diagram Elements

Nodes: Example





Container

Id: `stateNode`

Domain Class: `fsa.State`

Semantic Candidate Expression:

`feature:states`

Node is used for displaying only states of the FSA model shown in the diagram

Gradient Style

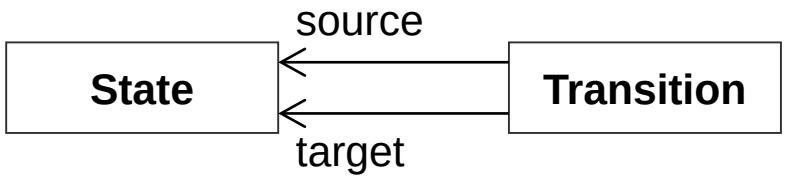
Background Color: `white`

Foreground Color: `white`

`...`

Element Based Edge

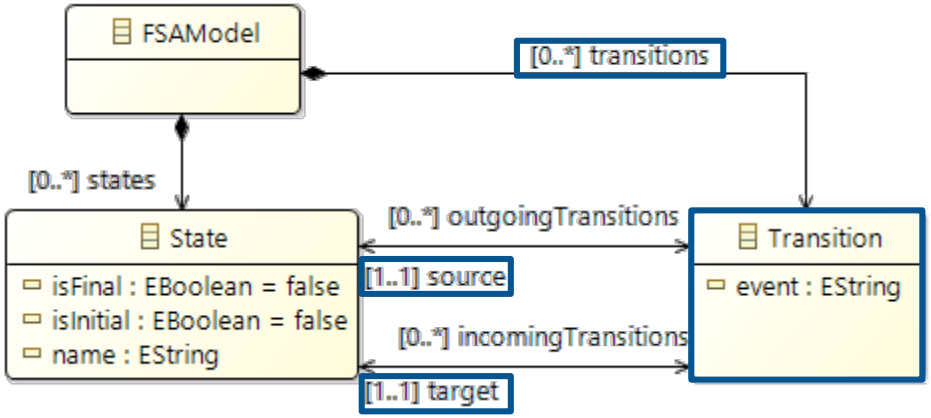
- Used when a model element should be displayed as an edge, i.e., if a metaclass in the metamodel represents a connection between model elements
- Example:

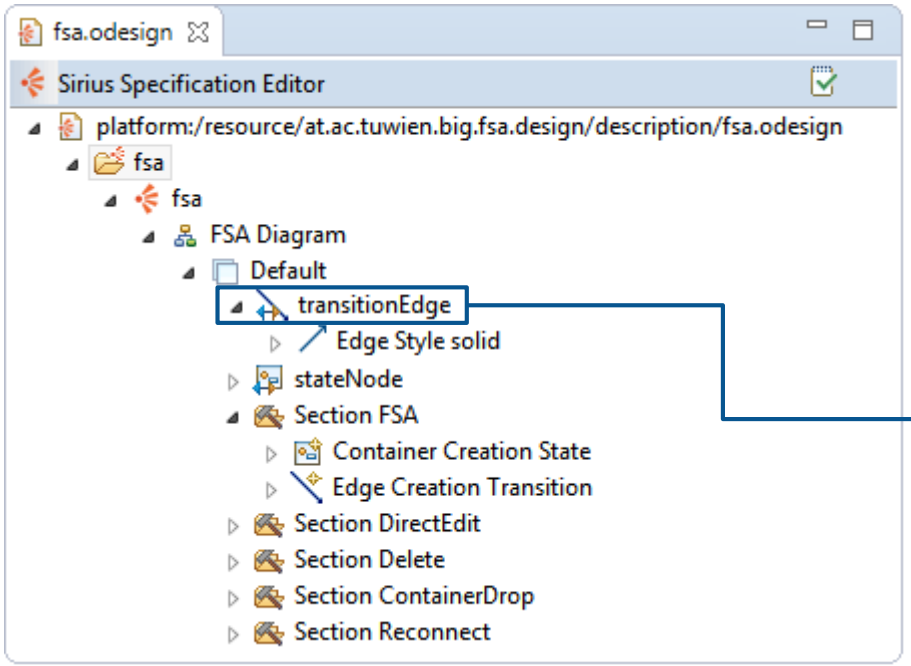


- Features:
  - **Source Mapping, Target Mapping**: Node mappings for the source node and the target node to be connected by the edge
  - **Domain Class**: Type of the model element (metaclass) which defines a relationship to be represented by the edge
  - **Semantic Candidate Expression:** Further restricts the model elements associated with the edge
  - **Source Finder Expression, Target Finder Expression**: Expression for retrieving the model elements being the source element and target element to be connected by the edge

Diagram Elements

Element Based Edges: Example





Element Based Edge

Id: `transitionEdge`

Source Mapping: `stateNode`

Target Mapping: `stateNode`

Domain Class: `fsa.Transition`

Semantic Candidate Expression:
`feature:transitions`

Edge is used for displaying only transitions of the FSA model shown in the diagram

Source Finder Expression:
`feature:source`

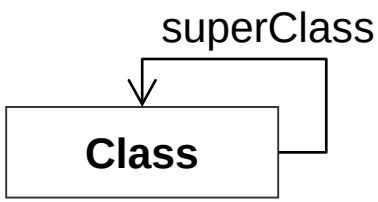
EReference Transition.source

Target Finder Expression:
`feature:target`

EReference Transition.source

Relation Based Edge

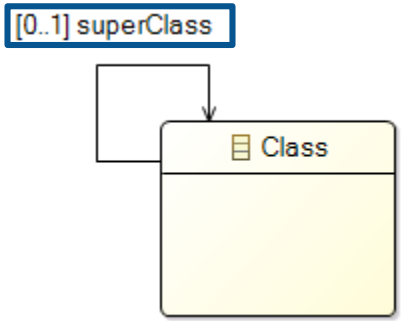
- Used when a reference between two model element should be displayed as an edge, i.e., if a reference (EReference) in the metamodel represents a connection between model elements
- Example:

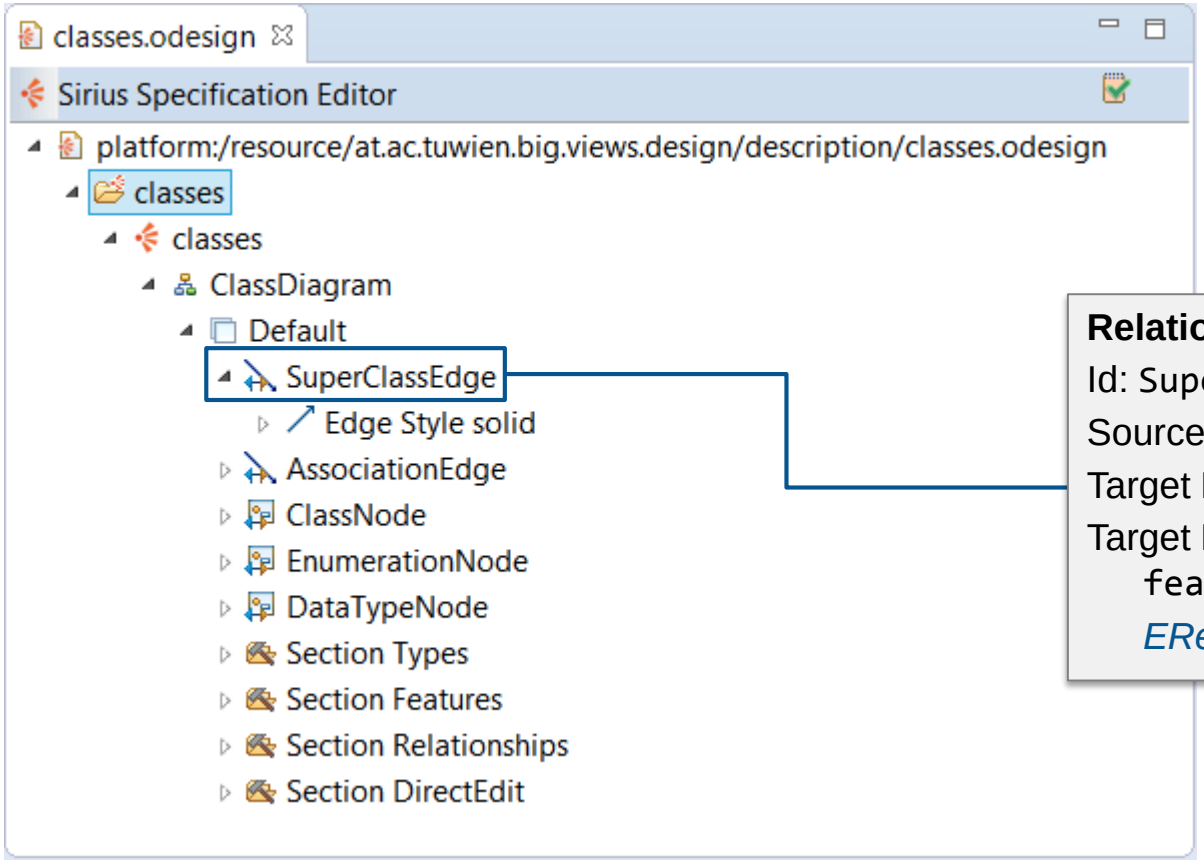


- Features:
  - **Source Mapping, Target Mapping**: Node mappings for the source node and the target node to be connected by the edge
  - **Target Finder Expression**: Reference (EReference) represented by the edge

Diagram Elements

Relation Based Edges: Example





Relation Based Edge

Id: `SuperClassEdge`

Source Mapping: `ClassNode`

Target Mapping: `ClassNode`

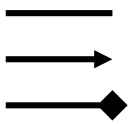
Target Finder Expression:

`feature:superClass`

EReference Class.superClass

Edge Style

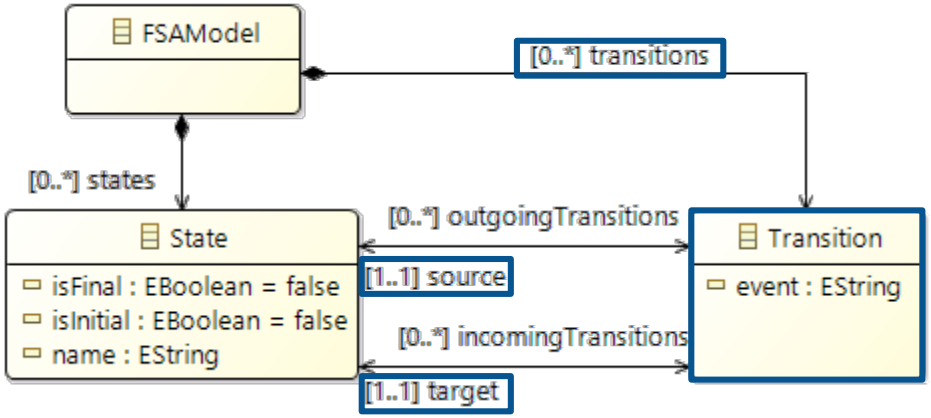
- Features:
  - **Line style**: solid, dash, dot, dash_dot
  - **Routing style**: straight, manhattan (angles), tree (joins edges with same target)
  - **Decorators**: source arrow and target arrow

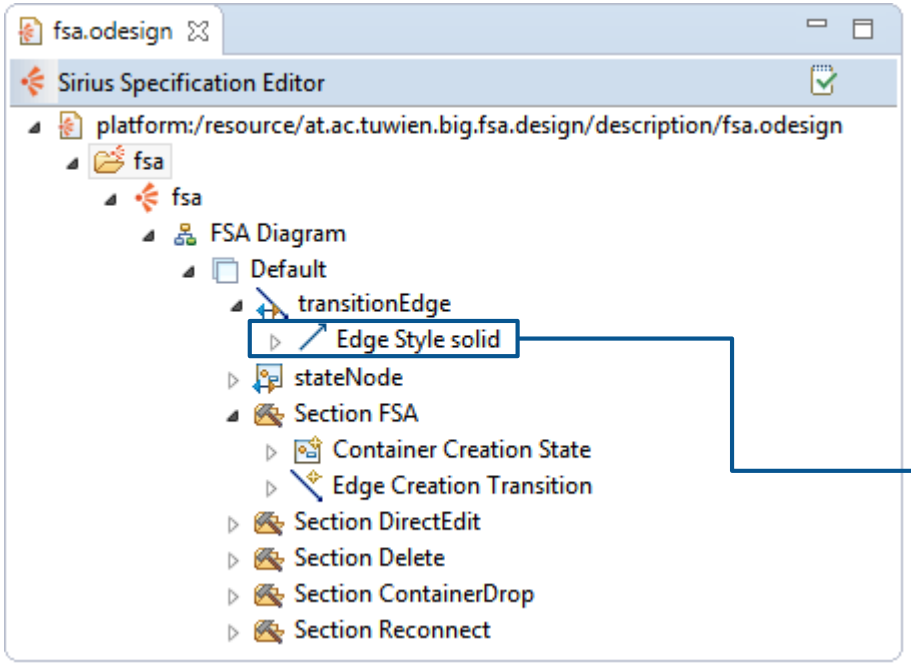


...

Diagram Elements

Edges: Example





Edge Style

Line Style: `solid`

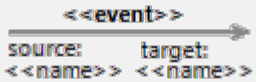
Source Arrow: `NoDecoration`

Target Arrow: `InputArrow`

Stroke Color: `gray`

Routing style: `Straigth`

`...`

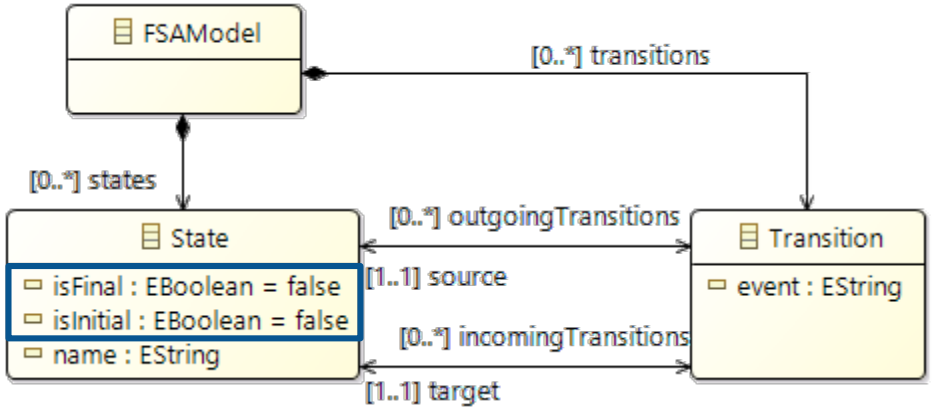


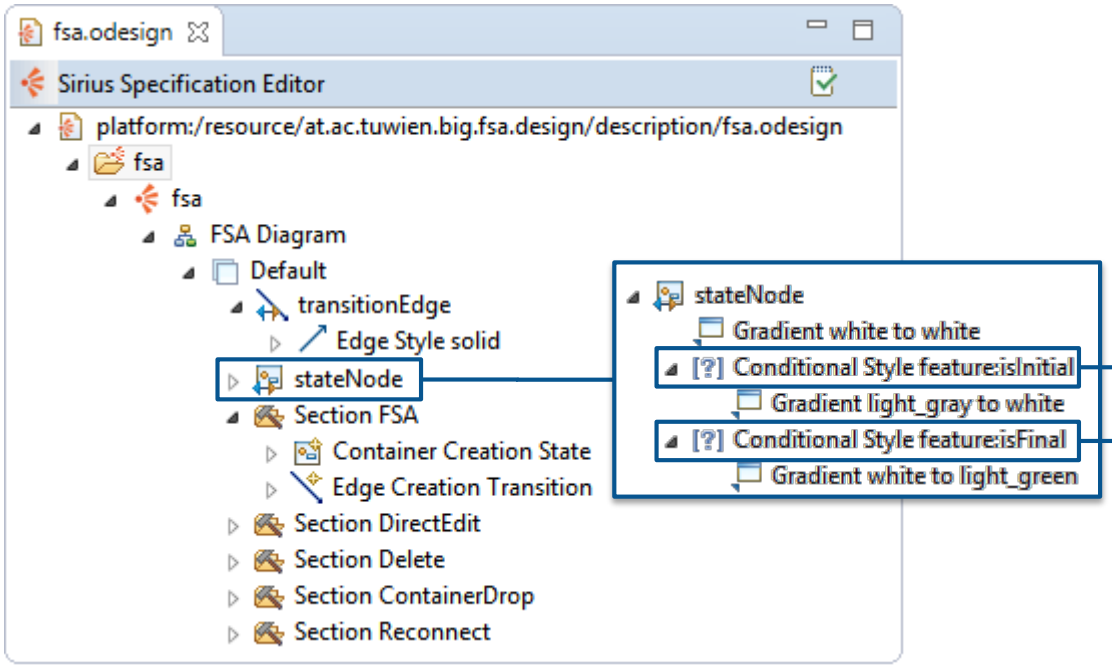
Conditional Styles

- Styles for nodes and edges that are applied if a certain condition is fulfilled for the represented model element
- Features:
  - **Predicate Expression**: Condition on the model element under which the conditional style is applied

Diagram Elements

Conditional Style: Example

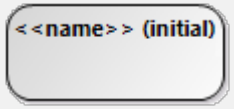




Conditional Style

Predicate Expression:

`feature:isInital`



Conditional Style

Predicate Expression:

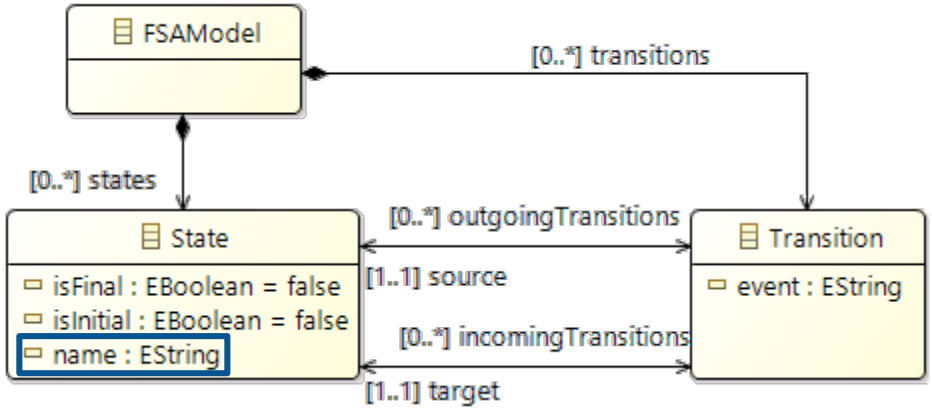
`feature:isFinal`

Labels

- Node styles and edge styles define labels that display attribute values of the represented model elements
- Features:
  - **Label Expression**: Retrieves the String to be displayed as label
  - **Label Size**
  - **Label Format**: bold italic, underline, strike through
  - **Label Alignment**: left, center, right
  - **Icon Path** (in Advanced properties): Path to the icon to be displayed next to the label
  - **Show Icon**: Whether or not an icon should be shown
  - …

Diagram Elements

Node Labels: Example



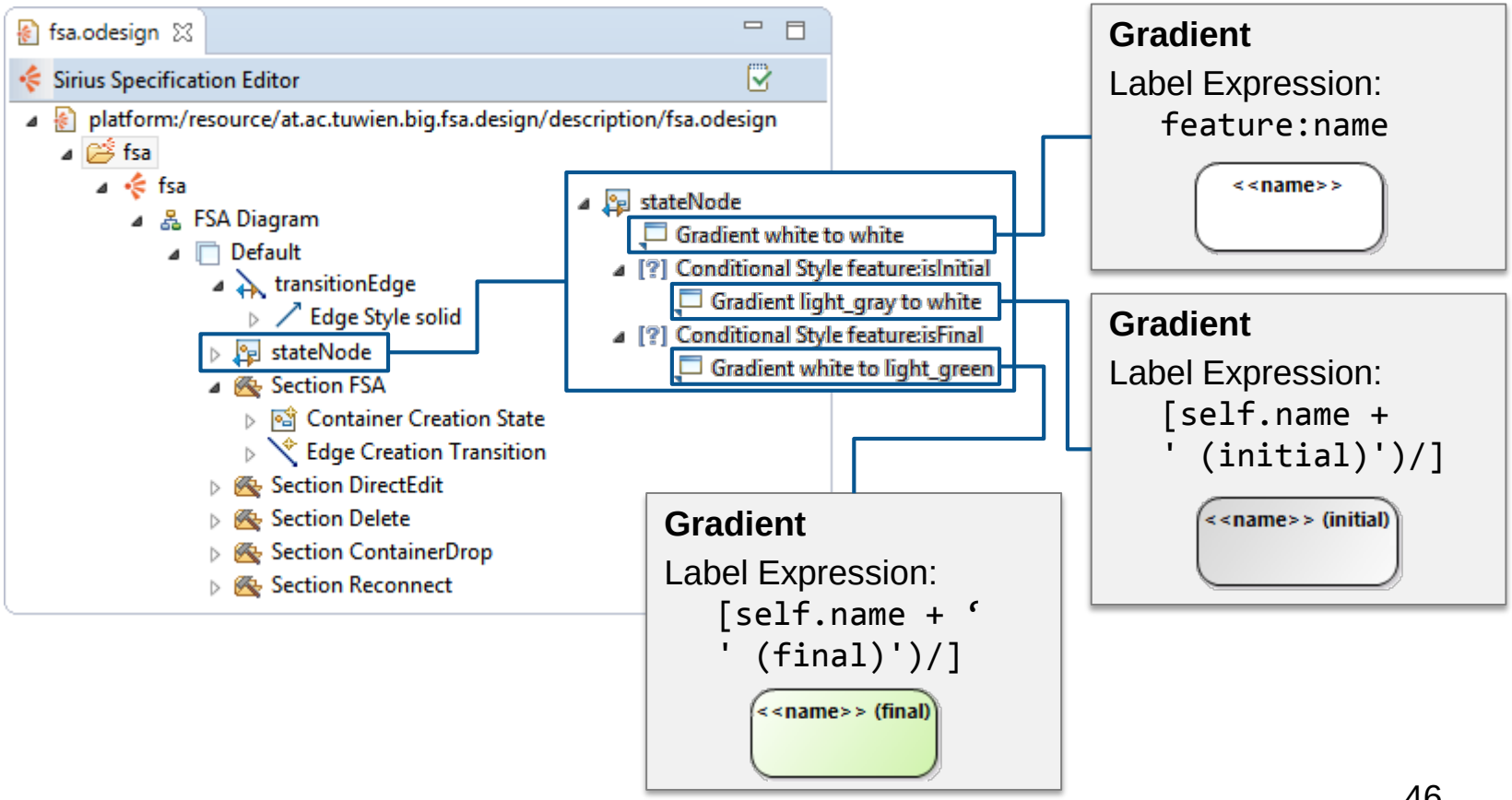
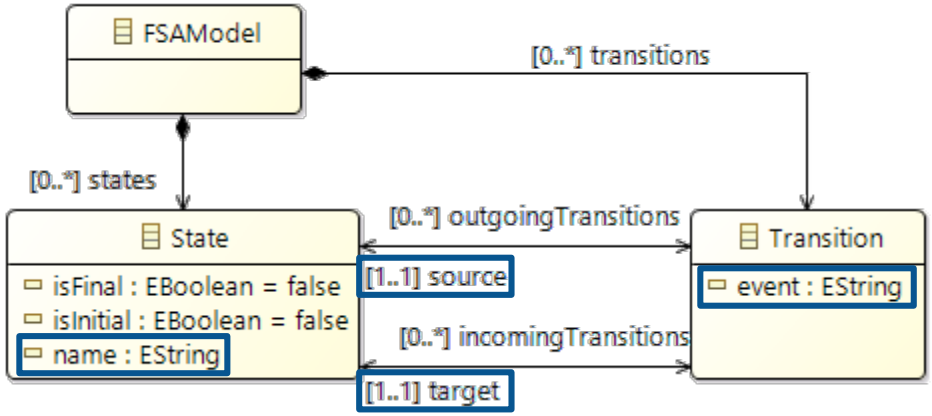
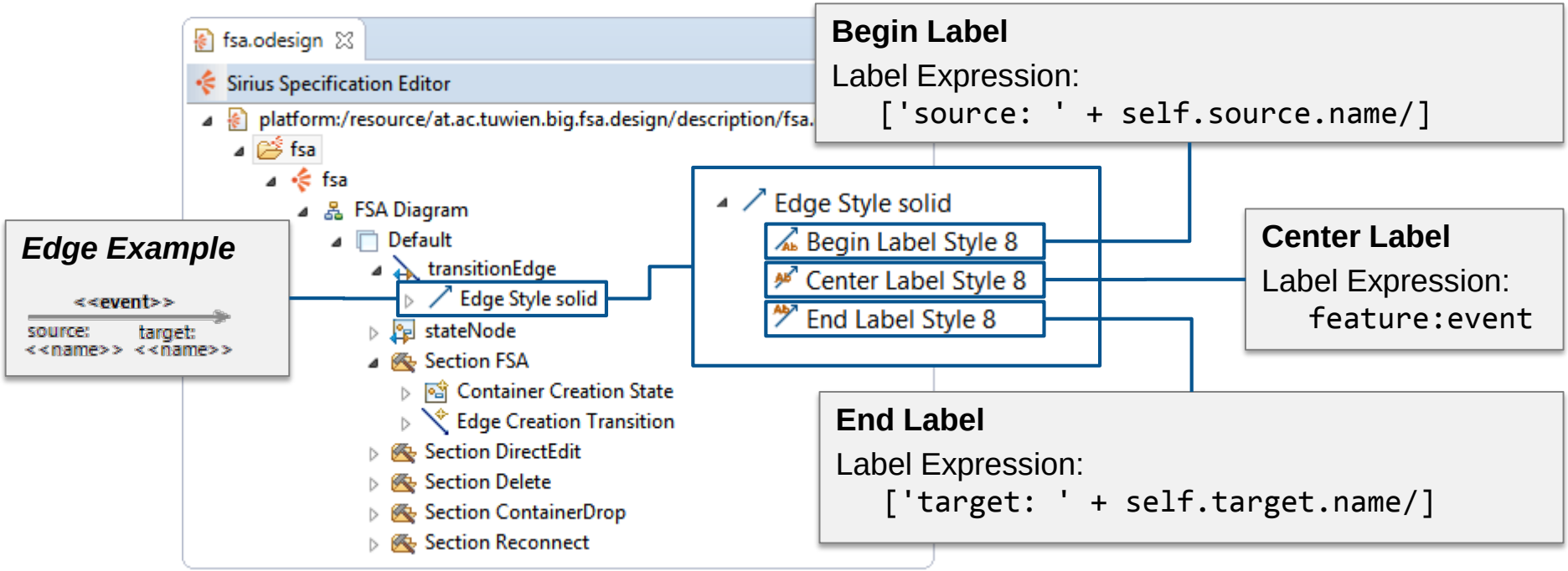


Diagram Elements

Edge Labels: Example





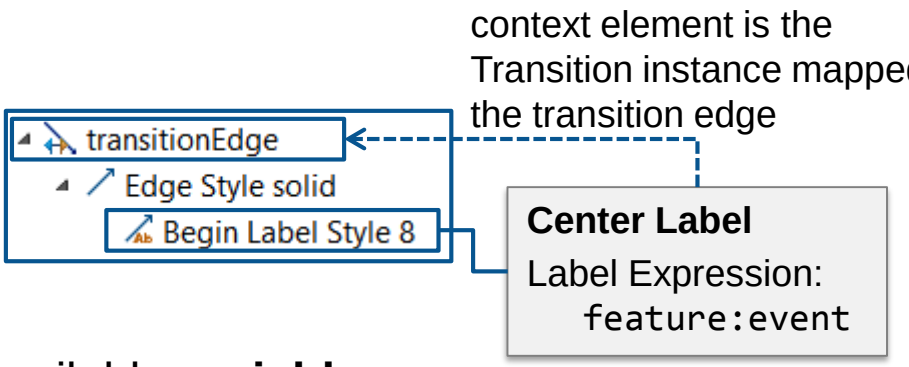
Interpreted Expressions

- Many parts of a viewpoint specification model require the definition of **interpreted expressions**
  - Examples:
    - Node mappings: Semantic Candidate Expression
    - Edge mappings: Source / Target Finder Expression, Semantic Candidate Expression
    - Conditional style: Predicate expression
    - Label: Label expression
- Sirius provides **three specialized interpreters** for handling simple expressions very efficiently
- For complex expressions the **Acceleo Expression Language** can be used

Interpreted Expressions

Simple Expressions

- **`feature:`**
  - Provides direct access to **named features** of the context element
  - Example
    - `feature:name`



- **`var:`**
  - Provides direct access to available **variables**
  - Example
    - `var:self` returns the current context element

- **`service:`**
  - Enables direct invocations of **service methods** implemented with Java

Interpreted Expressions

Acceleo Expression Language

- Acceleo expressions are enclosed with `[ /]`
- Available expressions
  - **OCL expressions**
    - *Example*: Retrieve the outgoing transition of a state (context) that processes the event ‘H’

```
[self.outgoingTransitions -> select(t | t.event = 'H') ->
asSequence() -> first().event /]
```

  - **EMF operations**
    - *Example*: Retrieve the FSAModel that contains a transitions (context)

```
[self.eContainer()/]
```

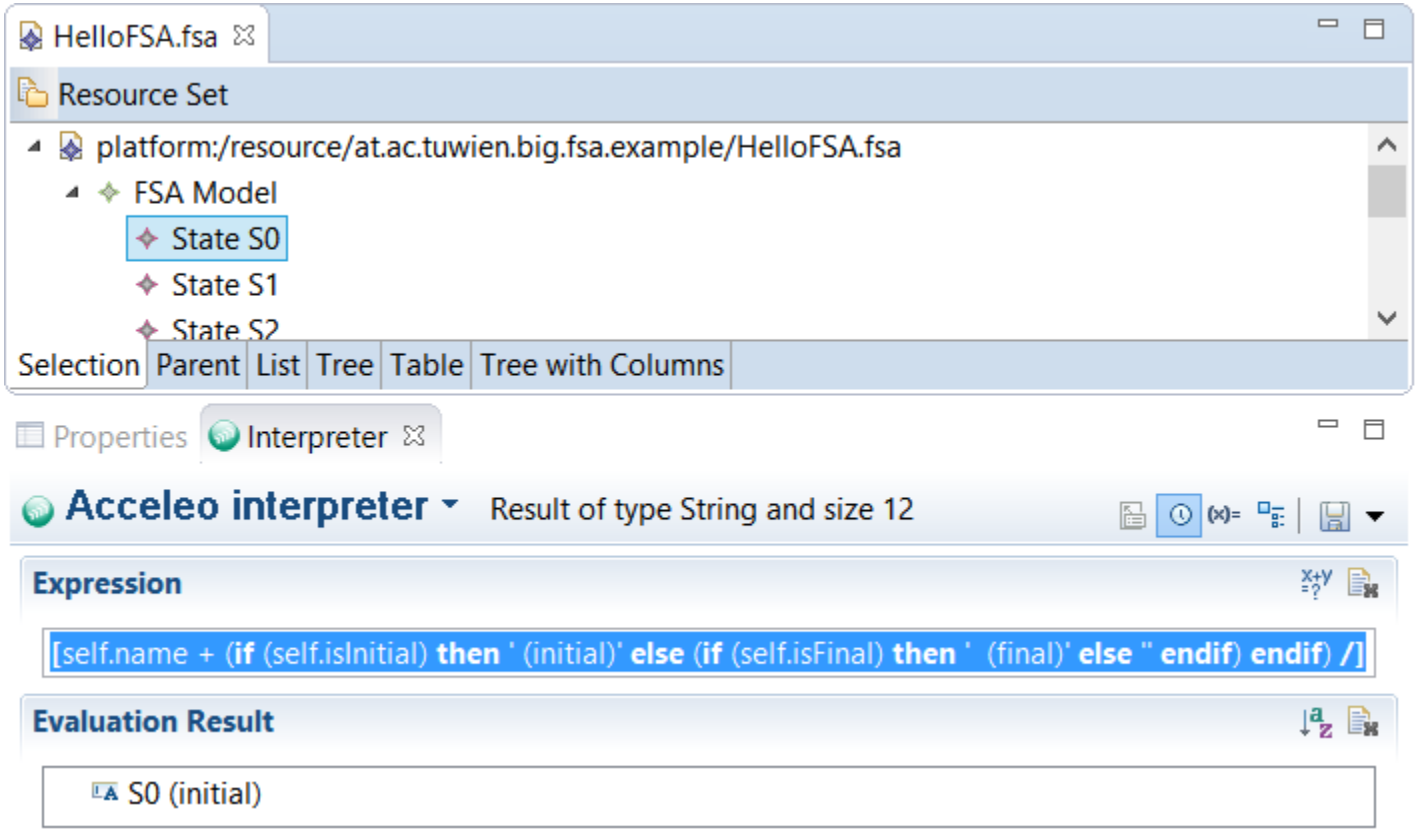
- **Control flow statements**
  - If: `[(if (conditionExp) then Exp else Exp endif) /]`
    - Example: Compute the label of a state (context)

```
[self.name + (if (self.isInitial) then ' (initial)' else
(if (self.isFinal) then '  (final)'
else '' endif) endif) /]
```

Interpreted Expressions

Acceleo Interpreter

- The **Acceleo interpreter** lets you test Acceleo expressions
- **TIP**: Test your expressions with the Acceleo interpreter before you add them to your viewpoint specification model



Tools

Tool

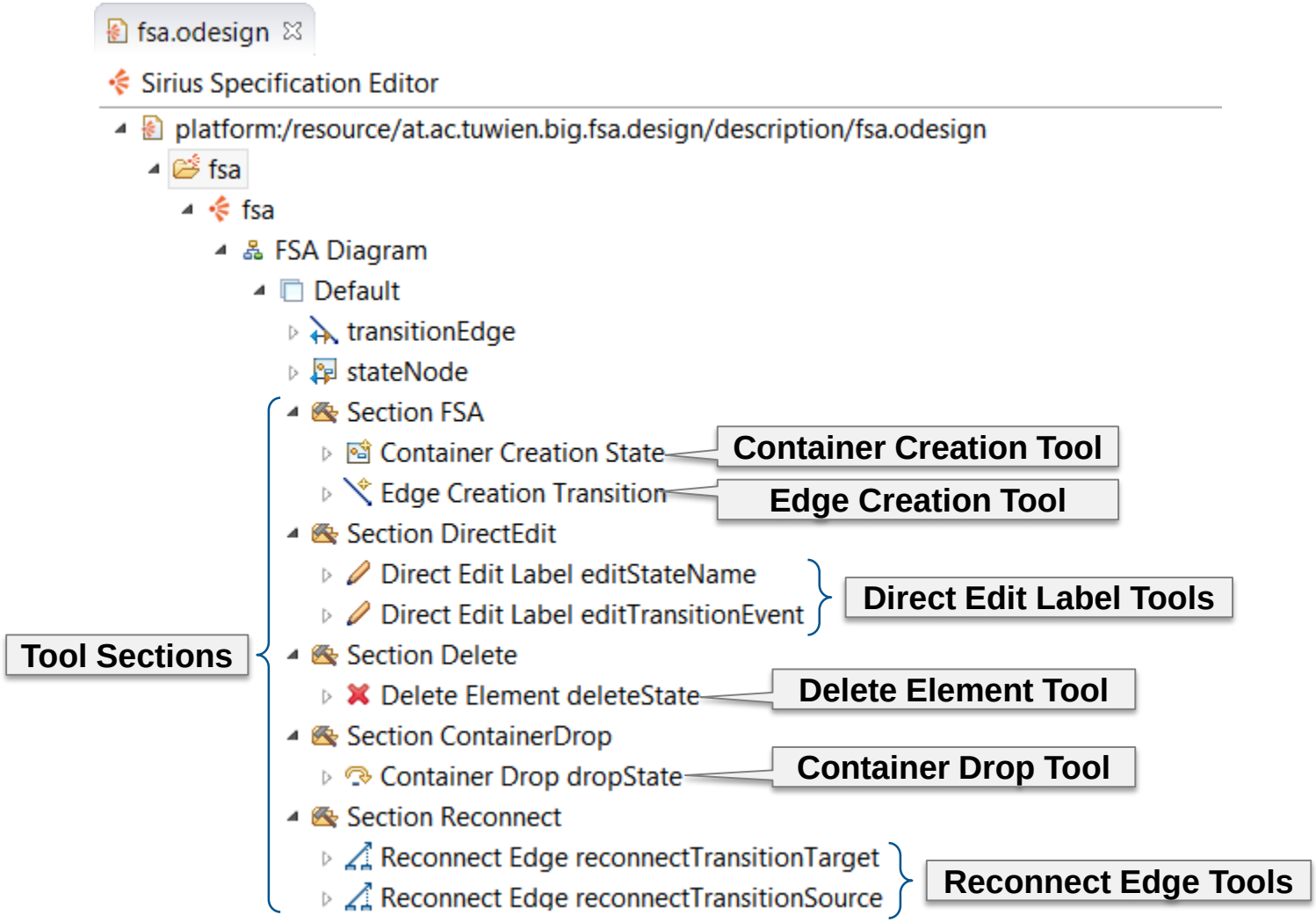
- Defines interactions with the model that are possible in a representation
- Tools are organized in **tool sections**
- For diagrams, tool sections are defined within a layer

Tool Types

- **Element creation**: Creation of nodes, containers, and edges
- **Direct edit label**: Direct editing of labels
- **Delete element**: Deletion of elements
- **Reconnect edge**: Reconnection of edges
- **Container drop**: Drag’n’drop operations
- **Double click**: Behavior of double clicking on elements
- **Selection wizard**: Wizards for selecting elements and applying operations on them
- **Paste**: Behavior for pasting elements onto the representation
- **Representation creation:** Creation of diagrams, trees, and tables

Tools

Example



Tool Definition Contents

- **Features**: Tool-specific features
  - Example: Node Creation Tool
    - Node Mappings: Node to be created
- **Variables**: Tool-specific variables for accessing elements of the user model or elements of the representation model
  - Example: Node Creation Tool
    - Node Creation Variable `container`: The element of the user model mapped to the container to which the new node shall be added
    - Container View Variable `containerView`: The graphical element representing the container to which the new node shall be added
- **Model Operations**: Operations to be applied on the user model or the representation model
  - Flow control operations
  - Representation operations
  - Model change operations
  - Java extension: Invocation of Java code

Flow Control Operations

- **Begin**: Entry point to a tool's behavior
- **Change Context**:
  - Every model operation is executed in the context of a specific element
  - The default context is the element of the user model on which the tool is applied
  - With the change context operation, the context can be changed to an element selected with an interpreted expression
  - Features
    - **Browse Expression**: Selects the new context element
- **If, Switch/Case, For**: Control flow statements

Model Change Operations

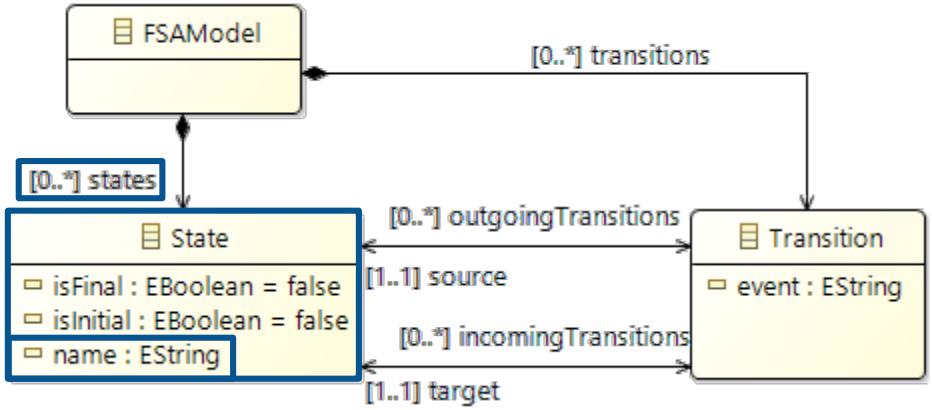
- **Create Instance**: Create elements in the user model
  - Features
    - **Type Name**: Type (metaclass) to be instantiated
    - **Reference Name**: Name of the containment reference used for adding the new element to the current context element (container)
- **Set, Unset**: Set / unset a feature value of the context element
  - Features
    - **Feature Name**: Name of the attribute or reference to be set / unset
    - **Value Expression**: Calculates the value to be set / unset
- **Remove**: Remove the context element
- **Move**: Move the context element into a new container
  - Features
    - **New Container Expression**: Retrieves the new container element
    - **Feature Name**: Name of the containment reference used for adding the element to the new container element

Representation Operations

- **Create View**: Create a node for the context element
  - Features
    - **Mapping**: Node mapping to be created
    - **Container View Expression**: Retrieves the container view to which the new node shall be added
- **Create Edge View**: Create an edge for the context element
  - Features
    - **Mapping, Container View Expression**
    - **Source Expression, Target Expression**: Elements of the user model that are the source and target elements connected by the new edge
- **Delete View**: Remove a node or edge from the diagram without deleting the mapped user model element
- **Navigation**: Navigate to another diagram that represent the context element

Tools

Model Operations: Example 1



- Section FSA
  - Container Creation State
    - Node Creation Variable container
    - Container View Variable containerView
    - Begin
      - Change Context var:container
        - Create Instance fsa.State
          - Set name

Container Creation Tool

Container Mappings: `stateNode`

Change Context Operation

Browse Expression: `var:container`

Switches to the container in which the diagram elements was created (FSAModel instance)

Create Instance Operation

Type Name: `fsa.State`

Reference Name: `states`

Creates a new State instance and adds it to the container FSAModel using the containment reference FSAModel.states

Set Operation

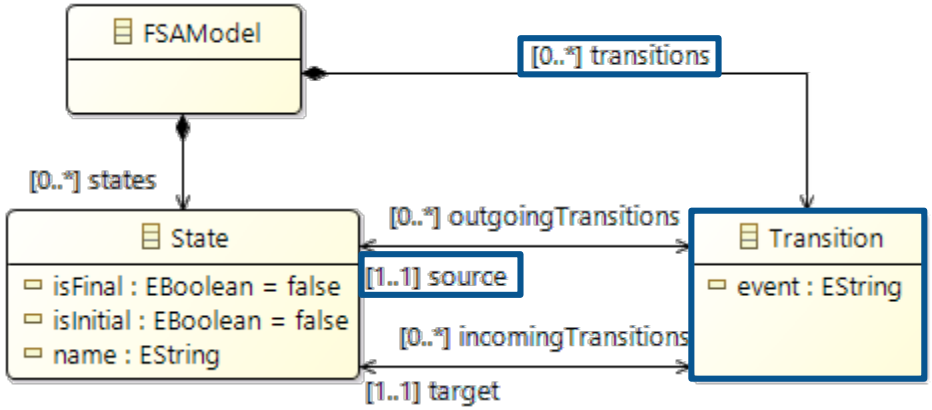
Feature Name: `name`

Value Expression: `['state'/]`

Initializes name of new State instance

Tools

Model Operations: Example 2



- Edge Creation Transition
  - Source Edge Creation Variable source
  - Target Edge Creation Variable target
  - Source Edge View Creation Variable sourceView
  - Target Edge View Creation Variable targetView
  - Begin
    - Change Context var:source
      - Change Context [self.eContainer()/]
        - Create Instance fsa.Transition
          - Set source
          - Set target
          - Set event

Edge Creation Tool

Edge Mappings: `transitionEdge`

Change Context Operation

Browse Expression: `var:source`

Switches to source of drawn edge, i.e., source State of Transition

Change Context Operation

Browse Expression: `[self.eContainer()]`

Switches to container of source State (FSAModel instance)

Create Instance Operation

Type Name: `fsa.Transition`

Reference Name: `transitions`

Set Operation

Feature Name: `source`

Value Expression: `var:source`

Further Features

- **Further representations**:
  - Sequence diagrams
  - Tables
  - Trees
- **Validation rules and quick fixes**: Specify validation rules for user models and quick fix operations that can be applied if rules are violated
- **Filters**: Filter shown model elements that fulfill certain conditions
- **Properties views**: Specify properties views customized to modeling languages (generic properties view is provided out-of-the-box)

Demo

Step 1: Prepare your workspace

- Projects required in your Eclipse workspace:
  - Model project
  - Edit project (.edit)
  - Editor project (.editor)

Step 2: Start a new Eclipse instance (Runtime Eclipse)

- Switch into the *Sirius* perspective

Step 3: Create or import an example model

- Create a new project: *File → New → Sirius / Modeling Project*
- Create a new model: *File → New → Other → Example EMF Model Creation Wizards / Fsa Model*

Demo

Step 4: Create a viewpoint specification project

- *File → New → Sirius / Viewpoint Specification Project*
- Contains viewpoint specification model (*.odesign)

Step 5: Define your graphical editor

- Add a viewpoint, diagram, mappings, and tools to the viewpoint specification model

Demo

Step 6: Test your graphical editor

- Select the defined viewpoint: Right click on the example modeling project → *Viewpoints Selection* → *Select viewpoint fsa*
- Create a diagram:
  - Expand the example model in the *Model Explorer*
  - Right-click on *FSAModel* element → *New representation* → *new FSA diagram*
  - Nodes and edges corresponding to your defined mappings will be automatically created for existing model elements

TIP: Iterate between step 5 and 6, i.e., build your graphical editor incrementally and test it iteratively

Literature

- Sirius
  - Project Website: http://www.eclipse.org/sirius/
  - Documentation: http://www.eclipse.org/sirius/doc/
  - Tutorials:
    https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial
    https://wiki.eclipse.org/Sirius/Tutorials/AdvancedTutorial
  - Examples:
    https://www.eclipse.org/sirius/gallery.html

- Acceleo Expression Language
  - http://help.eclipse.org/oxygen/topic/org.eclipse.acceleo.doc/pages/reference/operations.html?cp=5_3_2