Sirius Workshop Guide

Let's create a graphical modeling editor for a robot !

Sirius Concepts

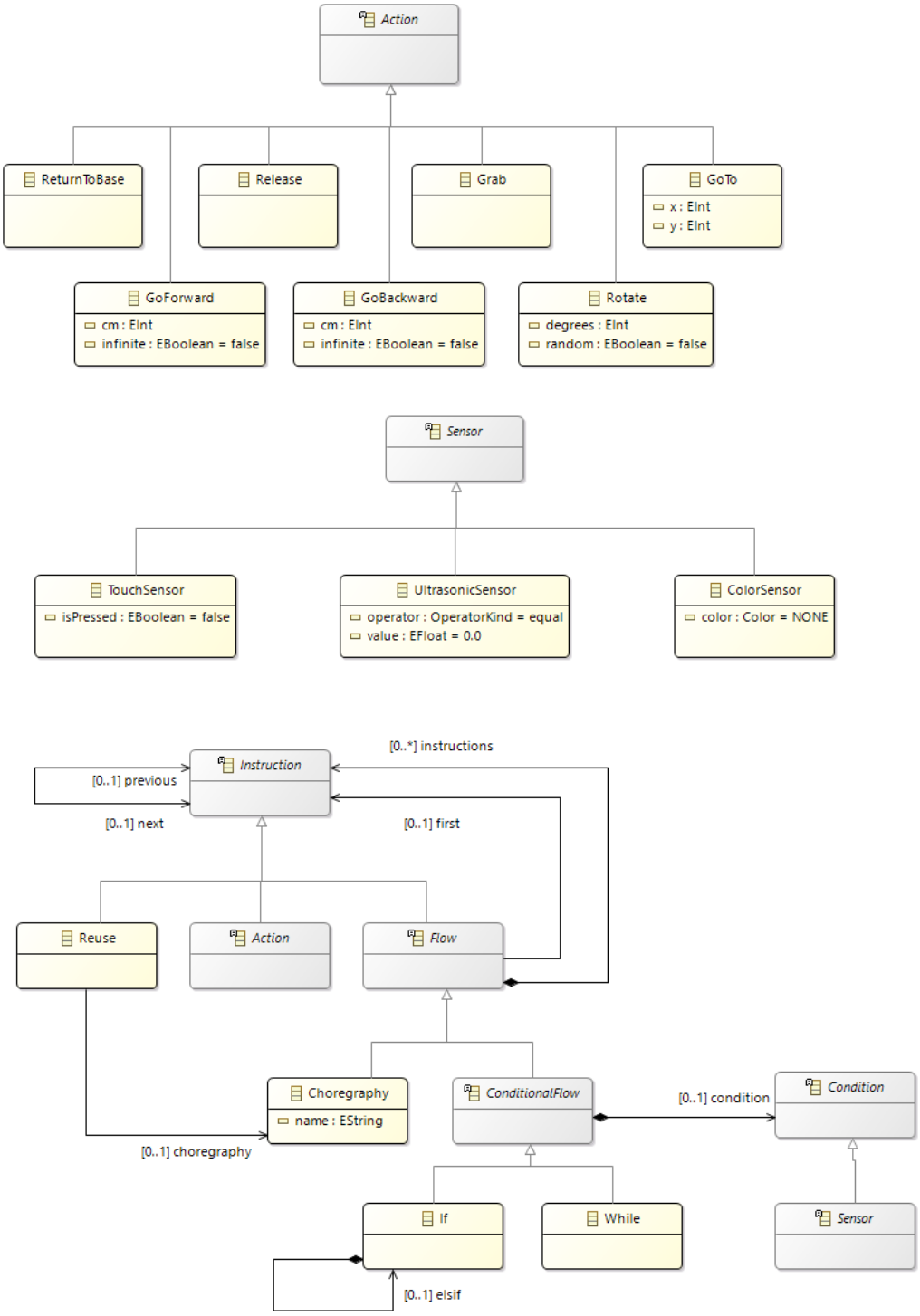
During this workshop, you will mainly use these Sirius concepts:

- **Viewpoint Specification Project**
  - The Eclipse project that defines a Sirius modeling tool
  - Contains a **odesign** file that describes the representations and Java services used by the tool
- **Viewpoint**
  - A viewpoint defines Sirius representations (diagrams, tables, matrices, trees) dedicated to a specific need
- **Diagram Description**
  - Describes a kind of graphical representation for your model
  - It defines which elements to display on the diagram, how (style) and the tools to edit them
- **Node**
  - Describes model elements displayed via an image or a simple shape
  - It defines how to find the model elements to display
  - It defines a graphical style (shape, label, color, ...)
- **Viewpoint Specification Project**
  - The Eclipse project that defines a Sirius modeling tool
  - Contains a odesign files that describes the representations and Java services used by the tool
- **Viewpoint**
  - A viewpoint defines Sirius representations (diagrams, tables, matrices, trees) dedicated to a specific need
- **Diagram Description**
  - Describes a kind of graphical representation for your model
  - It defines which elements to display on the diagramm, how (style) and the tools to edit them
- **Node**
  - Describes model elements displayed via an image or a simple shape
  - It defines how to find the model elements to display
  - It defines a graphical style (shape, label, color, ...)
- **User Colors Palette**
  - Defines specific colors that you can use for the diagram elements (background, foreground, border, ...)
- **Container**
  - Describes graphical model elements that can contain other elements (as free form, lists or compartments)
  - Defines how to find the container elements to display and their sub-elements (nodes, containers, ports)
  - Defines a graphical style (shape, label, color, ...)
- **Select Model Element Variable**
  - Can be used by tools to interactively propose a list of model elements to select by the user

- **Edge Creation tool**
  - Describes the tool in the palette that allows the user to create a link between two graphical elements
- **Double-Click tool**
  - Defines which actions to execute when the user double-click on a graphical element
- **Direct Edit Label tool**
  - Defines how to interpret the label value changes that are made directly on the diagram
- **Reconnect Edge tool**
  - Defines how to interpret the modification of edges ends (origin or destination) made directly from the diagram
  - Provides three variables :
    - element (the origin of the edge)
    - source (the destination of the edge before the execution of the tool)
    - target (the destination of the edge after the execution of the tool)
- **Stack option for Container's children presentation**
  - Sub-containers are displayed as Vertical or Horizontal stacks
- **Properties Views Description**
  - Describes how model element properties are edited in the Eclipse Properties Views

Step 0: Preparation

- Launch **Obeo Designer**
- Import all existing projects from archive **fr.obeo.dsl.tuto.mindstorms.step0.initial.zip**



- Create and launch a new **Eclipse Launch Configuration**

Note: all the next actions have to be performed in this new runtime

Step 1: Basic Modeling Tool

Objective

Create a basic Diagram to display and edit a simple flow:

- **Nodes and Edges**
  - Start of the flow (the flow itself)
  - ReturnToBase actions
  - Rotate to left actions (the rotate instances with degrees > 0)
  - Link from the start and the first node of the flow (the reference 'first' of the flow)
  - Link between an action and its next action (the reference 'next' of each action)
- **Node creation tools**
  - Create a ReturnToBase action
  - Create a Rotate action

Instructions

- Import the project containing a sample model from **fr.obeo.dsl.tuto.mindstorms.step1.initial.zip**
- Create a Viewpoint Specification Project named `fr.obeo.dsl.tuto.mindstorms.design`
  - Create a folder named `icons`
- Import files from **icons.zip** into the `icons` folders

Note: all the next actions have to be performed in the .odesign file (except the creation of Java services)

- Create a **Viewpoint** named `Choregraphy`
- Create a **Representation** of type **Diagram Description**
  - *Name* = `Flow Diagram`
  - *Domain Class* = `mindstorms.Flow`
  - Reference the **mindstorms** metamodel
    - in *metamodels* tab, add the **mindstorms package** from the registry

Note: from now, you should be able to create a blank Flow Diagram from the sample model:
- Activate the Choregraphy viewpoint from the sample project
- Right-click on the main choregraphy of the model (in the model explorer) and select "New representation"

- Create a **Composite Layout** to force linked objects to be displayed from left to right
  - *Direction* = `Left To Right`
- In the **Default Layer** Create a **Node** representing the current flow
  - *Name* = `Start`
  - *Domain class* = `mindstorms.Flow`
  - Semantic Candidate Expression = `var:self`
  - Create a **Workspace Image** for this Node

        - *Image Path* = `Begin.svg` (prefixed by its path)
        - *Label Size* = `12`
        - *Label Format* = `Bold`
        - *Show Icon* = `false`
        - *Label Expression* = `Start`
        - *Label Position* = `border`

- Create a **Node** representing the actions of the current flow (default image is ReturnToBase)
    - *Name* = `Action`
    - *Domain class* = `mindstorms.Action`
    - Semantic Candidate Expression = `feature:instructions`
    - Create a **Workspace Image** for this Node
        - *Image Path* = `ReturnToBase.svg` (prefixed by its path)
        - *Label Size* = `12`
        - *Label Format* = `Bold`
        - *Show Icon* = `false`
        - Remove the value of *Label Expression*
        - *Label Position* = `border`

- Declare the EMF **mindstoms metamodel** in the **MANIFEST.MF**
    - In the **Dependencies** add `fr.obeo.dsl.tuto.mindstorms`

- Create a **Java Package** named `fr.obeo.dsl.tuto.mindstorms.design.services`
    - Create a **Java Class** named `LabelServices`
        - Create a **Method** named `computeLabel`

```
public String computeLabel(Rotate block) {
	if (block.isRandom()) {
		return "?";
	}
	int degrees = block.getDegrees();
	return "" + Math.abs(degrees) + "°";
}
```

- On the Viewpoint, create a **Java Extension** to declare the Java class
    - *Qualified Class Name* = `fr.obeo.dsl.tuto.mindstorms.design.services.LabelServices`

- Create a **Style Customization** for **ActionNode** to represent **Rotate** instances with rotation to left (degrees >=0)
    - *Predicate Expression* = `aql:self.oclIsKindOf(mindstorms::Rotate) and self.degrees >= 0`
    - Create a Property Customization Expression (by expression) to change the *workspacePath* property of the previous Workspace Image
        - *New image* = `Rotate_Left.svg` (prefixed by its path)
    - Create a Property Customization Expression (by expression) to change the *labelExpression* property of the previous Workspace Image
        - *New label* = `service:computeLabel()`

- Create a **Relation Based Edge** named `First` that links the Start node to the **first** Action node
  - *Target Finder Expression* = `feature:first`

- Create a **Relation Based Edge** named `Next` that links any Action node to its **next** Action node
  - *Target Finder Expression* = `feature:next`

- Create a **Section** named `Actions` for the palette section dedicated to actions
  - Create a **Node Creation** to create instances of **ReturnToBase**
    - *Icon Path* = `ReturnToBase_16px.png` (prefixed by its path)
    - Create a **Change Context** to set on which objects the instructions will be executed
      - *Browse Expression* = `var:container` (this is the current Flow)
    - Add a **Create Instance**
      - *Feature name* = `instructions`
      - *Type Name* = `mindstorms.ReturnToBase`
  - Proceed similarly to create a **Rotate** to the left creation tool
    - *Icon Path* = `Rotate_Left_16px.png` (prefixed by its path)
    - Add a **Set** operation
      - *Feature Name* = `degrees` and *Value Expression* = `90`

Bonus: if you are ahead of time, you can try to complete this step with:

- Nodes for other kinds of actions (GoTo, GoForward, GoBackward, RotateRight, Delay, Grab and Release
- Corresponding Node Creation tools
- A Container for Choregraphy instances containing the list of their instructions

Step 2: Advanced Modeling Tool

Objective

Complete previous diagram with more advanced concepts:

- **Declare a custom color** for Choregraphy containers
- **Display Reuse instances** with the instructions of the reused Choregraphy
- **Create other edition tools**
  - A dialog-box to select a Choregraphy when creating a Reuse
  - A tool in the palette to create links
  - Double-click on a Choregraphy to create/open a new diagram
  - The possibility to edit labels on graphical elements
  - The possibility to change the destination of a link

Instructions

- Import in a clean workspace the projects from **fr.obeo.dsl.tuto.mindstorms.step2.initial.zip**

- At the modeler root, create a **User Colors Palette**
  - Create a **Used Fixed Color** named `MindstormColor1` (R=`186`, G=`223`, B=`225`)
  - Create a **Used Fixed Color** named `MindstormColor2` (R=`0`, G=`119`, B=`136`)

- Modify the Style of the **Choregraphy Container** to use these new colors
  - *Background Color* = `MindstormColor1`
  - *Foreground Color* = `MindstormColor1`
  - *Border Color* = `MindstormColor2`

- In the **Flow Diagram**, create a **Container** to represent the instances of **Reuse**
  - *Domain Class* = `mindstorms.Reuse`
  - *Semantic Candidate Expression* = `feature:instructions`
  - *Children Presentation* = `List`
  - Create a **Style** of type **Gradient**
    - *Label Expression* = `service:computeLabel()`
    - *Label Size* = `12`
    - *Label Format* = `Bold`
    - *Background Color* = `MindstormColor1`
    - *Foreground Color* = `MindstormColor1`
    - *Border Color* = `MindstormColor2`
  - Create a **Sub Node** to represent the instructions of the reused **Choregraphy**
    - *Domain Class* = `mindstorms.Instruction`
    - *Semantic Candidate Expressions* = `aql:self.choregraphy.getOrderedInstructions()`
    - Create a **Dot** as **Style** (or any other style with a **Label**)
      - *Label Expression* = `service:computeLabel()`
      - *Label Size* = `12`
      - *Label Format* = `Bold`

- Create a **Section** for the tools related to **Choregraphy** and **Reuse** elements
  - Create a **Container Creation** to create **Choregraphy** instances
    - For the operations after the Begin, proceed like the creation of actions (initialized with the name `NewChoregraphy`)
  - Create a **Container Creation** to create **Reuse** instances
    - Create a **Select Model Element Variable** named `selectedChoregraphy` for selecting an existing **Choregraphy**
      - *Candidates Expression* = `aql:container.instructions ->select(x|x.oclIsTypeOf(mindstorms::Choregraphy))`
      - For the operations after the Begin, proceed like the creation of actions (initialized with *choregraphy* = `var:selectedChoregraphy`)

- Create a **Section** for the tools related to Links
  - Create an **Edge Creation** to allow the user to create **First** and **Next** edges
    - After the Begin, create a **Change Context** to invoke a service that manages the different possibilities (depending of the kinds of source and target)
      - *Browse Expression* = `service:source.linkInstructions(target)`

- Create a **Section** for the other edition tools not visible from the palette

  - Create a **Double-Click** for **Choregraphy** to navigate to its detailed diagram
    - After the Begin, create a Navigation to **Flow Diagram**
      - *Create if not Existent* = `true`

  - Create a **Double-Click** for **Reuse** to allow the user to navigate to the detailed diagram of the related **Choregraphy**
    - After the **Begin**, change the context to the choregraphy then create a **Navigation** to **Flow Diagram**

  - Create a **Direct Edit Label** to allow the user to edit the label of the instructions
    - *Mappings*: any mapping which has a label that can be changed by the user
    - *Input Label Expression* = `service:getEditLabel`
    - After the **Begin** create a **Change Context** to execute this expression : `aql:self.editElement(arg0)`

  - Create a **Reconnect Edge** to allow the user to change the first instruction of the current flow by graphically changing the destination of the **Start** node
    - Change the context to the variable **element** which refers to the source of the edge (the **Start** node)
    - Set its **first** feature to `var:target` (the new destination of the edge)

  - Create a **Reconnect Edge** to allow the user to change the next instruction of an existing instruction by graphically changing the destination of an existing edge
    - Change the context to the variable **element** which refers to the source of the edge
    - Set its **next** feature to `var:target` (the new destination of the edge)

Bonus: if you are ahead of time, you can try to complete this step with:

- Containers for While and If
- A validation rule to verify that a Flow has at least one instruction
- A Diagram Creation tool to provide a menu on flows for creating new Flow Diagrams
- Deletion tools for Action node, First edge, Next edge … and Start node (to prevent the user to delete this node)
- A layer to display only on-demand the instructions of a reused choregraphy

Step3: Compartments and Properties Views

Objective

Complete previous modeler with Compartments and custom Properties Views:

- **Complete the If container**
  - Display the instructions of the main If within a first compartment
  - Display elsifs instructions in distinct containers
- **Provide specific properties views**
  - Previous and Next
    - On the same row
    - Filled only with instructions of the current flow, without current instruction
  - Degrees (for Rotate instances)
    - Disabled if random is checked
    - Yellow background if value is not between -180 and 180
    - Warning if value is null and random is not checked

Instructions

- Import in a clean workspace the projects from **fr.obeo.dsl.tuto.mindstorms.step3.initial.zip**
- Modify the **Container If** to set its **Children Presentation** feature to `Vertical Stack`

  Note: when defining or modifying compartments in a Diagram Description, the diagrams that already exist can't take the modification dynamically: they have to be created again.

- In the **Container If** create a first **Container** to represent a compartment which lists the instructions of the main **If**
  - *Domain Class* = `mindstorms.If`
  - *Semantic candidate Expression* = `var:self`
  - *Children Presentation* = `List`
  - Create the same **Style** as the **Container If**, just change this property:
    - *Hide Label By Default* = `true`
  - Create a **Sub Node** to represent the instructions
    - *Domain Class* = `mindstorms.Instruction`
    - *Semantic Candidate Expression* = `aql:self.getOrderedInstructions()`
    - Create the same **Style** as the **Choregraphy Instructions** node (in the **Choregraphy Container)**
- In the **Container If** create a second **Container** to display a compartment for each **elsif** instruction
  - *Domain Class* = `mindstorms.If`
  - *Semantic candidate Expression* = `aql:self.getAllElsifs()`
  - *Children Presentation* = `List`
  - Create the same **Style** as the first compartment
    - Just change *Hide Label By Default* = `false`
  - Create a similar **Sub Node** as the first compartment
- In the **Section Conditional Flows**, create a tool to allow the user to add an **elsif** compartment
  - Add a precondition to prevent the user to create a **elsif** on a **If** which has already a **elsif**
    - *Precondition* = `aql:self.elsif=null`

- In the *Extra Mappings*, add **Container Elsif Instructions** to allow the user to also create an elsif by clicking on an Elsif compartment
- On the **Create Instance** use the reference `elsif` and the type `mindtsorms.If`

- At the root of the modeler definition create a **Properties View Description** to define custom property views for the instructions

  - Modify the first **Page**
    - *Domain Class* = `mindstorms.Instruction`
    - *Label Expression* = `General`
  - Modify the first **Group** to display and edit **previous** and **next** references of an **Instruction**
    - Add this new **Group** to the **Page**
    - *Domain Class* = `mindstorms.Instruction`
    - *Label Expression* = `Links`
    - Add a **Container** to place the two references on the same row
      - Create a **Fill Layout**
        - *Orientation* = `HORIZONTAL`
      - Create a **Select** for **previous** reference
        - Add a **Begin** and a **Set** operation
          - *Feature name* = `previous`
          - *Value Expression* = `var:newValue`
        - *Label Expression* = `Previous`
        - *Value Expression* = `aql:self.previous`
        - *Candidates Expression* = `aql:self.eContainer().instructions ->excluding(self)`
        - *Candidate Display Expression* = `aql:candidate.eClass().name+' '+candidate.computeLabel()`
      - Proceed similarly for the **next** reference
  - Create a **Group** to display and edit the **degrees** and the **random** properties of **Rotate** instructions
    - Add this new **Group** to the **Page**
    - *Domain Class* = `mindstorms.Rotate`
    - *Label Expression* = `Properties`
    - Add a **Text** for the **degrees** property
      - *Label Expression* = `Degrees`
      - *Value Expression* = `aql:self.degrees`
      - *Is Enabled Expression* = `aql:not self.random`
      - Add a **Begin** and a **Set** operation (set `var:NewValue` to `degrees`)
    - Add a **Checkbox** for the **random** property
      - *Label Expression* = `Random`
      - *Value Expression* = `aql:self.random`
      - Add a **Begin** and a **Set** operation (set `var:NewValue` to `random`)
    - Create a **Conditional Style** for the **degrees Text** in order to color the text background when **degrees** is not between -180 and 180
      - *Precondition Expression* = `aql:self.degrees>180 or self.degrees<-180`
    - Add a **Group Validations** to warn the user when **degrees** is null and **random** is false (useless rotate instruction).
      - Create a **Property Validation Rule**
        - *Id* = `Rotation degrees should not be null`
        - *Level* = `Warning`
        - *Message* = `This rotation is useless`
        - Create an **Audit**
          - *Audit Expression* = `aql:self.degrees<>0 or self.random`
        - Create a **Fix** that sets the **random** property to `true`