Tutorial Hands-On

Four steps

This guide explains step-by-step how to proceed to create the Mindstorms modeling tool.

It is organized in 4 main steps:

- Metamodel
- Visualization tool
- Container and edition tools
- Properties views

We provide a git repository or the zip files containing the solution for each step.

Sirius expressions syntaxes

Dynamic parts of a modeling tool created with Sirius require you to write expressions that will be evaluated at runtime. Some of these expressions return model elements while others simply produce text.

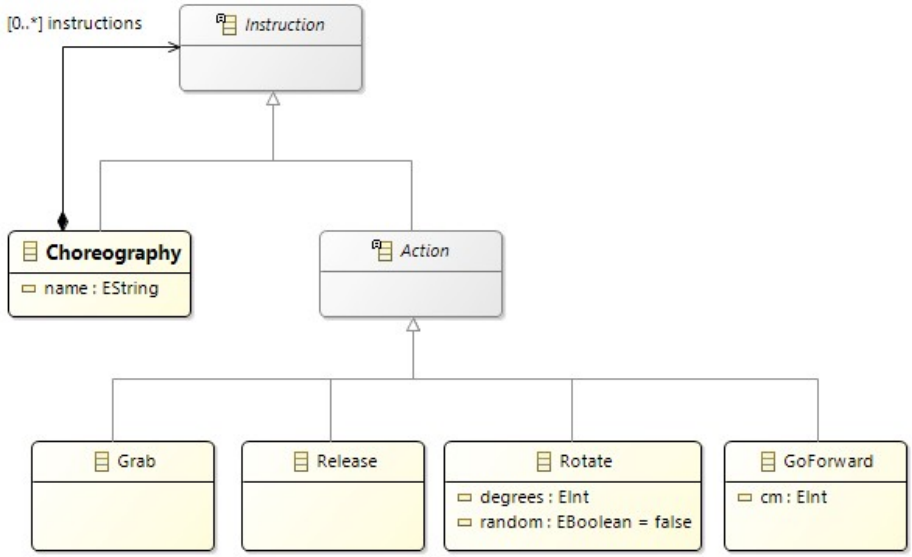
Sirius proposes four main syntaxes to write these queries:

- **var:**
  - allows Sirius to evaluate a variable
  - examples:
    - `var:self`
    - `var:container`
- **feature:**
  - allows Sirius to evaluate an EMF feature (property or reference) on the current context
  - examples:
    - `feature:name`
    - `feature:instructions`
- **service:**
  - allows Sirius to evaluate a Java method defined in a Class declared as an extension
  - examples:
    - `service:getNextInstruction()`
    - `service:setNextInstruction(i)`
- **aql:**
  - allows Sirius to evaluate an expression written in AQL (Acceleo Query Language)
    - https://www.eclipse.org/acceleo/documentation/aql.html
  - examples:
    - `aql:self.instructions->at(1)`
    - `aql:self.oclIsKindOf(mindstorms::Rotate) and self.degrees >= 0`

1. Metamodel

1.1. Objectives

Define the concepts used by the Mindstorms modeling tool



1.2. Instructions

- Launch **Obeo Designer**
- Create an **Ecore Modeling Project** named `fr.obeo.dsl.tuto.mindstorms`
  - Use the Palette to create the **EClasses**: `Choreography`, `Instruction`, `Action`, `Grab`, `Release`, `Rotate` and `GoForward`
    - Set `Instruction` and `Action` as abstract
  - Use the Palette to create **SuperType** relations:
    - from `Grab`, `Release`, `Rotate` and `GoForward` to `Action`
    - from `Action` and `Choreography` to `Instruction`
  - Use the Palette to create a **Composition** relation named `instructions` between `Choreography` and `Instruction`
  - Use the Palette to create EAttributes:
    - Choreography
      - `name: EString`
    - Rotate
      - `degrees: EInt`
      - `random: EBoolean`
    - GoForward
      - `cm: Eint`
  - Select the diagram, go to the properties set the **Ns URI** to `http://www.obeo.fr/dsl/mindstorms/1.0.0`
  - Right-click on the diagram and select the menu **Generate**
    - Generate the **Model** code
    - Generate the **Edit** code
    - Generate the **Editor** code

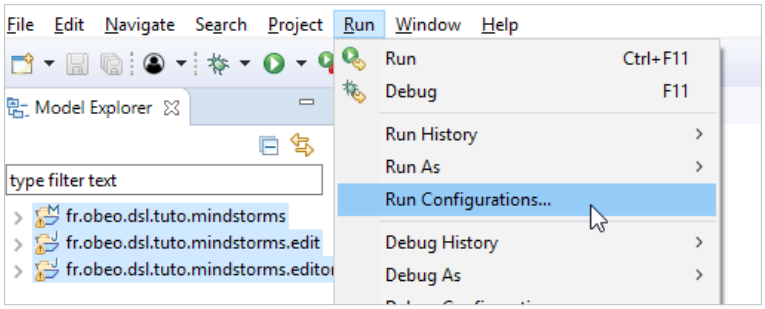
- Edit the **`fr.obeo.dsl.tuto.mindstorms.edit`** generated project to improve the default labels and icons
  - Replace the icons contained in **`icons/full/obj16`** by those from **icons-metamodel**
  - Replace the **`getImage`** method of the class **RotateItemProvider.java** (in **fr.obeo.dsl.tuto.mindstorms.edit**) by the one defined in **methods.txt.**

1.3. Solution

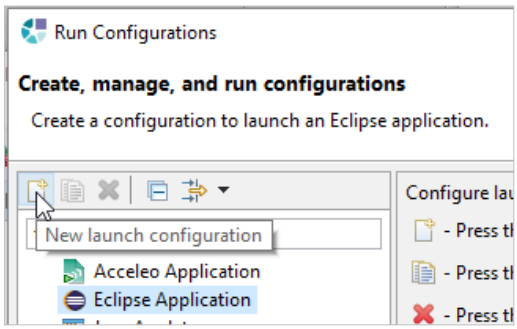
- Import the three existing Eclipse projects contained in the archive **solution1.zip or switch to branch step1 of the git repository**
  - They define the **Mindstorms** metamodel

- Create and launch a new **Eclipse Launch Configuration** (In this new runtime, the **Mindstorms** metamodel will be available for execution)

  - Click on **Run / Run Configuration…**



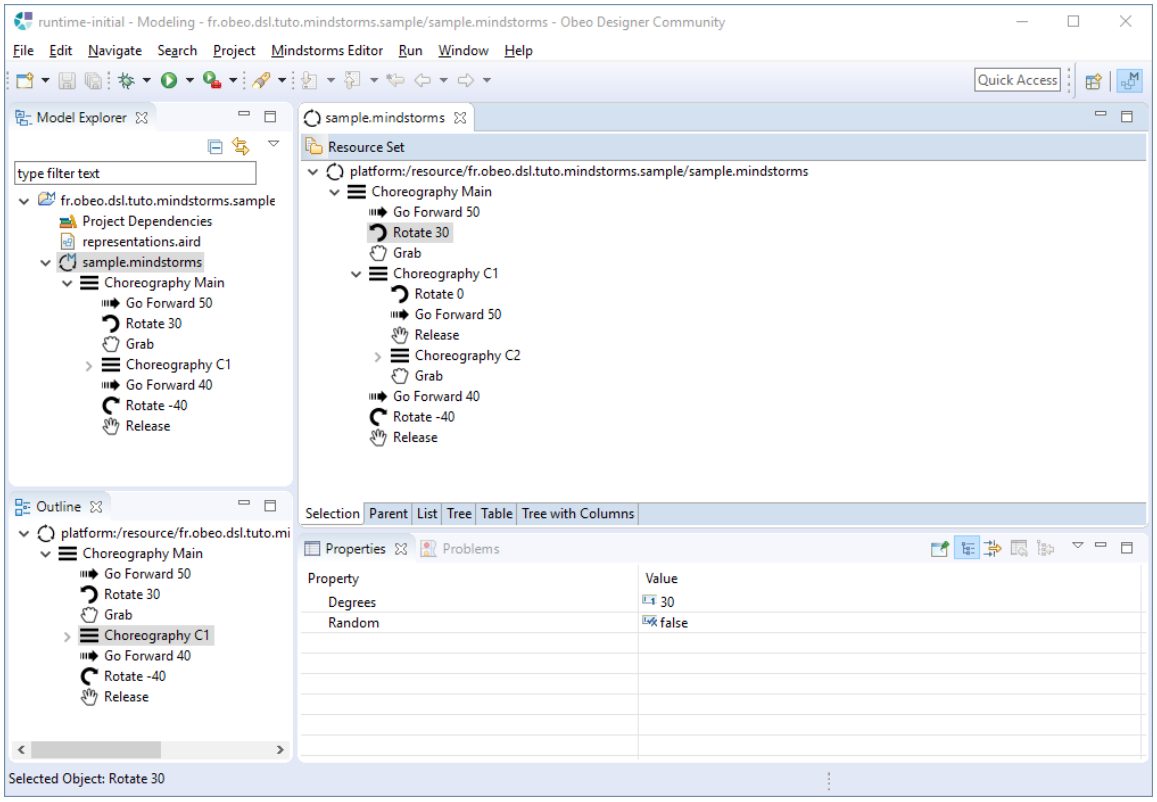
o Select **Eclipse Application**, click on **New**



  - Then click on **Run**

- In the new runtime, import the Eclipse project contained in the archive **sample.zip**
  - It contains a sample **Mindstorms** model that will be used to test your modeling tool
  - You can open this model with the default editor generated by EMF

- Your environment should look like this:



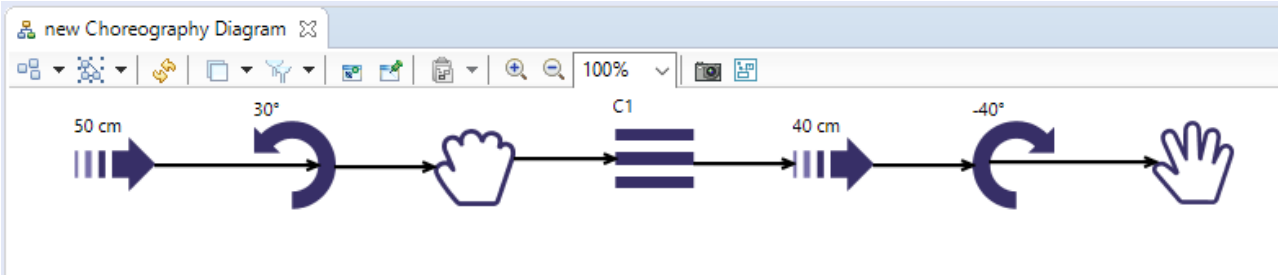
2. Visualization tool

Note: all the following actions have to be performed in the Eclipse **runtime** launched previously

2.1. Objectives

Create a basic Diagram to display the instructions of a Mindstorms choreography and provide tools to allow the user to create new instructions and sub-diagrams.

- **Nodes**
  - Grab actions
  - Release actions
  - GoForward actions
  - Rotate actions
- **Edges**
  - Link between an instruction and its next instruction



2.2. Sirius Concepts

During this step, 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, ...)
- **Relation Based Edge**
  - Describes the relation between two objects
  - The relation can be computed
  - It defines graphical style (color, line style, size, routing style, ...)
- **Section**
  - Describes a category of tools in the palette

- **Node Creation Tool**
  - Describes the tool in the palette that allows the user to create a new node
- **Container Creation Tool**
  - Describes the tool in the palette that allows the user to create a new container
- **Double-click Tool**
  - Describes the action to perform when the user double-clicks on a diagram element

2.3. Instructions

2.3.1. Create a Sirius project and a first diagram definition

- Select the **Sirius** perspective (button on the top right)
- Create a **Viewpoint Specification Project** named `fr.obeo.dsl.tuto.mindstorms.design`
  - *Viewpoint Specification Model name*: `mindstorms.odesign`
- Import the Archive File **icons-designer.zip** into this project

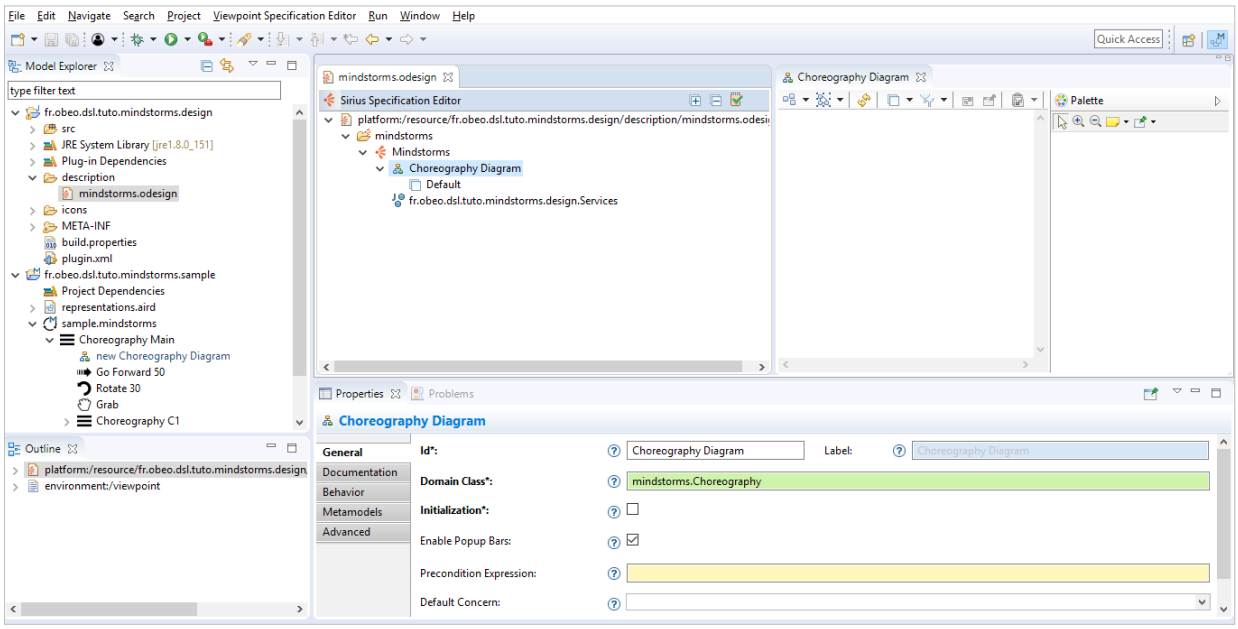
Note: all the next actions have to be performed in the **.odesign** file (except the creation of Java services)

- Update the **Viewpoint** created by default (**MyViewpoint**)
  - *Id* = `Mindstorms`
  - *Model File Extension* = `mindstorms`
- In this viewpoint, create a **Representation** of type **Diagram Description**
  - *Id* = `Choreography Diagram`
  - *Domain Class* = `mindstorms::Choreography`
  - Reference the **mindstorms** metamodel
    - in *metamodels* tab, add the **mindstorms package** from the registry (enter `*mindstorms` in the selection field)

Note: from now, if you save the .odesign file, you should be able to create a blank **Choreography Diagram** on the sample model:

- Open the sample project and activate the **Mindstorms** viewpoint (right-click on the project + menu **Viewpoint Selection**)
- Create a diagram (right-click on the root Choreography in the Model Explorer and select "**New representation**") : this diagram is still empty because we didn't define its structure yet

- Your environment should look like this:
  - Split the editors zone to display both the **mindstorms.odesign** file and the newly created **Choreography Diagram** (drag to the right the diagram's tab title)



2.3.2. Display each kind of instruction with a dedicated icon

- In the **Default Layer,** create a **Diagram Element** of type **Node** that displays the **Grab** instructions of the current choreography
  - *Id* = **`CD_Grab`**
  - *Domain class* = **`mindstorms::Grab`**
  - *Semantic Candidate Expression* = **`feature:instructions`**

  - Create a **Style** of type **Workspace Image** for this Node
    - *Image Path* = **`Grab.svg`** (prefixed by its path)
    - In the *Label* tab : *Show Icon* = **`false`**
    - Remove the value of *Label Expression*
    - *Label Position* = **`border`**

- Copy & Paste **CD_Grab** to create a **Node** that displays **Release** instructions
  - Change these values on the Node:
    - *Id* = **`CD_Release`**
    - *Domain class* = **`mindstorms::Release`**
  - Change these values on the Workspace Image
    - *Image Path* = **`Release.svg`** (prefixed by its path)

- Copy & Paste **CD_Grab** to create a **Node** that displays **GoForward** instructions
  - Change these values on the Node:
    - *Id* = **`CD_GoForward`**
    - *Domain class* = **`mindstorms::GoForward`**
  - Change these values on the Workspace Image
    - *Image Path* = **`GoForward.svg`** (prefixed by its path)
    - *Label Expression* = **`aql:self.cm+' cm'`**

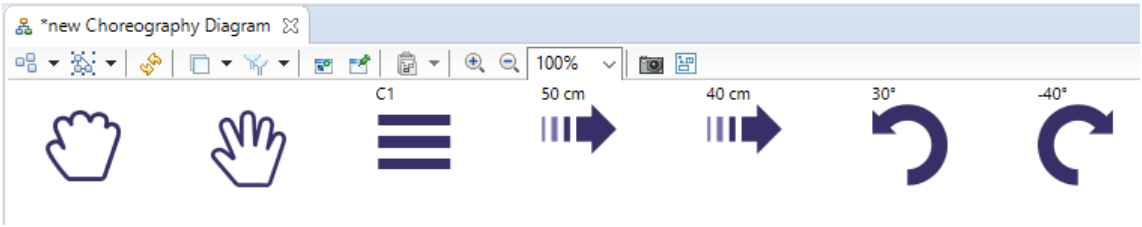
- Copy & Paste **CD_Grab** to create a **Node** that displays **Rotate** to left instructions
  - Change these values on the Node:
    - *Id* = **`CD_RotateLeft`**

        - *Domain class* = **`mindstorms::Rotate`**
        - In the *Advanced* tab : *Precondition Expression* = **`aql:self.degrees>=0`**
    - Change these values on the Workspace Image
        - *Image Path* = **`Rotate_Left.svg`** (prefixed by its path)
        - *Label Expression* = **`aql:if self.random then '?' else self.degrees+'°' endif`**

- Copy & Paste **CD_RotateLeft** to create a **Node** that displays **Rotate** to right instructions
    - Change these values on the Node:
        - *Id* = **`CD_RotateRight`**
        - *Precondition Expression* = **`aql:self.degrees<0`**
    - Change these values on the Workspace Image
        - *Image Path* = **`Rotate_Right.svg`** (prefixed by its path)

- Copy & Paste **CD_Grab** to create a **Node** that displays **Choreography** instructions
    - Change these values on the Node:
        - *Id* = **`CD_SubChoreography`**
        - *Domain class* = **`mindstorms::Choreography`**
    - Change these values on the Workspace Image
        - *Image Path* = **`Choreography.svg`** (prefixed by its path)
        - *Label Expression* = **`feature:name`**

- The diagram on the sample model should look like this:
    - click on the **Arrange All** button if necessary (the first one on the left in the tabbar)



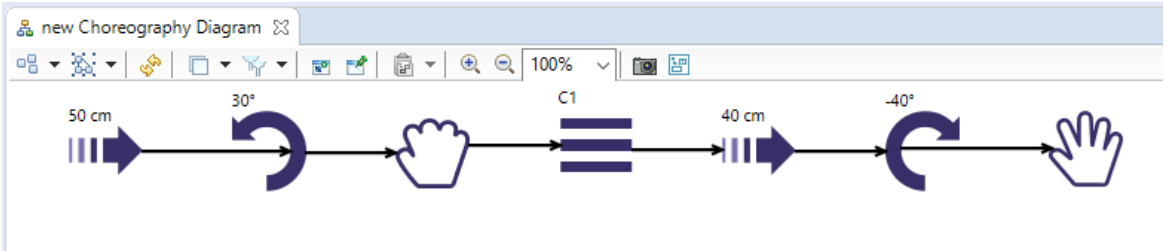
2.3.3. Display the relations between the instructions

- Define a service that computes the next instruction of a given instruction.
    - To be able to write Java code that references the Mindstorms concepts, declare the EMF implementation of the metamodel in the project
        - Edit the **META-INF/MANIFEST.MF** file
        - In the **Dependencies** tab, add **`fr.obeo.dsl.tuto.mindstorms`** to the *Required Plugins*
    - Copy the source code of the Method named **getNextInstruction** from the file **methods.txt** into the class **Services.java** (in the **src** folder)

- In the **Default** layer, create a **Relation Based Edge** that displays the links between an Instruction and its next Instruction
  - *Id* = `CD_Next`
  - *Source mapping* = `CD_GoForward, CD_Grab, CD_Release, CD_RotateLeft, CD_RotateRight, CD_SubChoreography`
  - *Target mapping* = `CD_GoForward, CD_Grab, CD_Release, CD_RotateLeft, CD_RotateRight, CD_SubChoreography`
  - *Target Finder Expression* = `service:getNextInstruction()`

- In the Diagram Definition, Create a **Composite Layout** to force linked objects to be displayed from left to right (*Direction* = `Left to Right`

- The diagram on the sample model should look like this:



2.4. Solution

The solution of this step is provided in **solution2.zip or switch to branch step2 of the git repository**

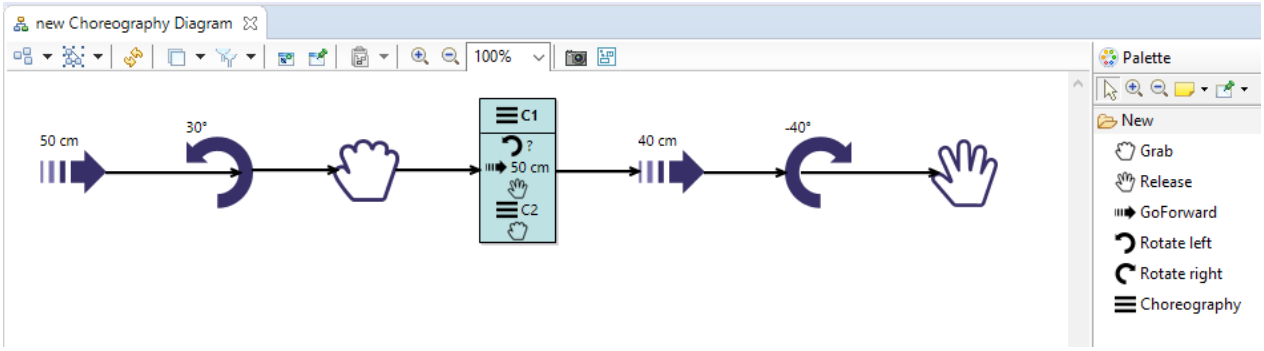
Warning: If you delete your current design project and load the solution, you should close the sample project and reopen it, in order the new version to be taken into account

3. Container and edition tools

3.1. Objectives

Enhance a basic Diagram to display the sub-choreographies with a container showing the sub-instructions and provide tools to allow the user to create new instructions and sub-diagrams.

- **Container**
  - Sub-choregraphies with the list of their sub-instructions
- **Creation Tools**
  - A button for each kind of instruction
  - Two buttons for Rotate (left and right)
- **Double-click Tool**
  - Create/Open a diagram by double-clicking on a Choreography



3.2. Sirius Concepts

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

- **Container**
  - Describes model elements displayed via a box which can show sub-elements
- **Color Palette**
  - Defines custom colors
- **Section**
  - Describes a category of tools in the palette
- **Node Creation Tool**
  - Describes the tool in the palette that allows the user to create a new node
- **Container Creation Tool**
  - Describes the tool in the palette that allows the user to create a new container
- **Double-click Tool**
  - Describes the action to perform when the user double-clicks on a diagram element

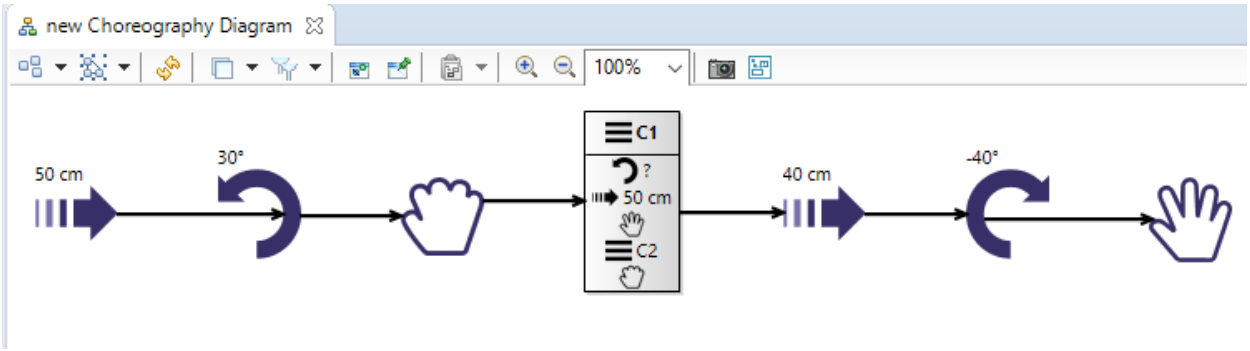
3.3. Instructions

3.3.1. Use a container to display the sub-choreographies with their instructions

- In the **Default Layer,** delete the Node **CD_SubChoreography** and replace it by a **Container** that will display the **Choreography** with its sub-instructions
    - *Id* = **`CD_SubChoreography`**
    - *Domain Class* = **`mindstorms::Choreography`**
    - *Semantic Candidate Expression* = **`feature:instructions`**
    - *Children Presentation* = **`List`**
    - Create a **Style** of type **Gradient**
        - *Label Format* = **`Bold`**
- Add this container to the *Source* and *Target* mappings of the edge **CD_Next**

Note: To display the sub-instructions, we will use a unique sub-node and compute the label with a **service** written in Java

- Define a service that computes the label of any kind of instruction.
    - Copy the source code of the **Method** named **`getLabel`** from the file **methods-tuto1.txt** into the class **Services.java**
- In **CD_SubChoreography**, Create a **Sub Node** named **`CD_SubInstruction`**
    - *Domain class* = **`mindstorms::Instruction`**
    - *Semantic Candidate Expression* = **`feature:instructions`**
    - Define a default **Style** (any kind of style with a label, for example **Square**)
        - *Show Icon* = **`true`**
        - *Label Expression* = **`service:getLabel()`**
- Now, the diagram on the sample model should look like this:



- Define a specific color for the container's background
  - Create a **Users Colors Palette**
    - Add a **User Fixed Color** named `MindstormsColor1`
      - `Red = 186`
      - `Green = 223`
      - `Blue = 225`
  - Update the style of the **CD_SubChoreography** container
    - Background Color = `MindstormsColor1`
    - ForegroundColor = `MindstormsColor1`

3.3.2. Add creation tools

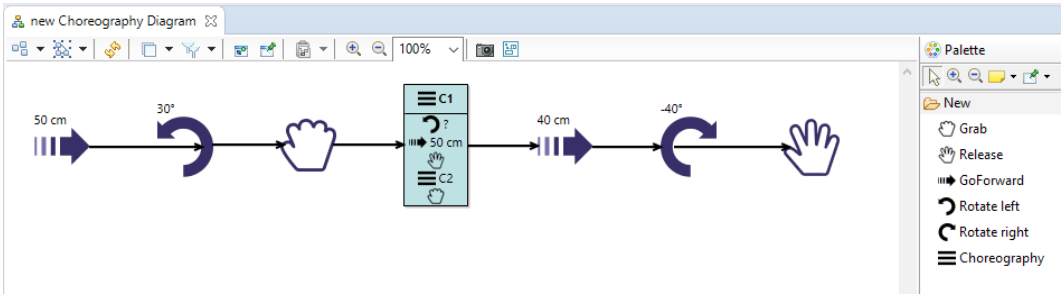
- In the Default layer, create a **Section** to provide a palette for the creation of objects

- In this section, create a **Node Creation** tool to create instances of **Grab**
  - *Id* = `Grab`
  - *Node Mapping* = `CD_Grab`
  - To also allow the users to create a Grab directly into a sub-choreography add `CD_SubChoreography` in the *Extra Mappings* property
  - Under the **Begin** node, create an **Operation** of type **Change Context** in order to define on which object the next operations will be executed
    - *Browse Expression* = `var:container` (this is the current Choreography)
  - Add an **Operation** of type **Create Instance**
    - *Reference Name* = `instructions`
    - *Type Name* = `mindstorms::Grab`

- Copy/Paste and adapt the previous **Node Creation** tool for the other kinds of **Action**:
  - **Release**
  - **GoForward**
    - After the creation of the instance, add a **Set**:
      - *Feature Name* = `cm`
      - *Value Expression* = `50`
  - **Rotate to Left**
    - After the creation of the instance, add a **Set**:
      - *Feature Name* = `degrees`
      - *Value Expression* = `90`
  - **Rotate to Right**
    - *Icon Path* = `Rotate_Right_16px.png` (prefixed with its path)
    - After the creation of the instance, add a **Set**:
      - *Feature Name* = `degrees`
      - *Value Expression* = `-90`

- To also provide a button in the palette that allows the user to create **Choreographies**, create a **Container Creation** tool and proceed like the Node Creation tools created previously
  - After the creation of the instance, add a **Set**:
    - *Feature Name* = `name`
    - *Value Expression* = `NewChoreography`

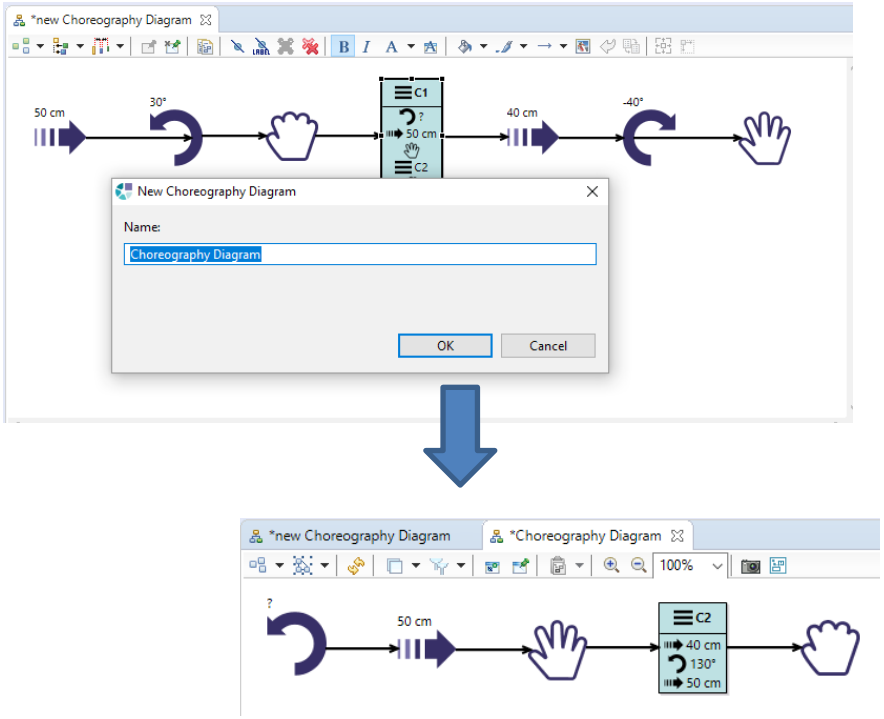
- Now, you should see some buttons in the palette that create new instructions within the current choreography:



3.3.3. Add a navigation tool

- In the previous **Section**, create a **Double-Click** tool to navigate from a **Choreography** to its own diagram
  - Mapping = `CD_SubChoreography`
  - After the **Begin**, create a **Navigation** to **Choreography Diagram**
    - *Create if not Existent* = `true`

- Close the diagram and reopen it: now, by double-clicking on the header of the container **C1**, a dialog box opens to create a new diagram:



3.4. Solution

The solution of this step is provided in **solution3.zip or switch to branch step3 of the git repository**

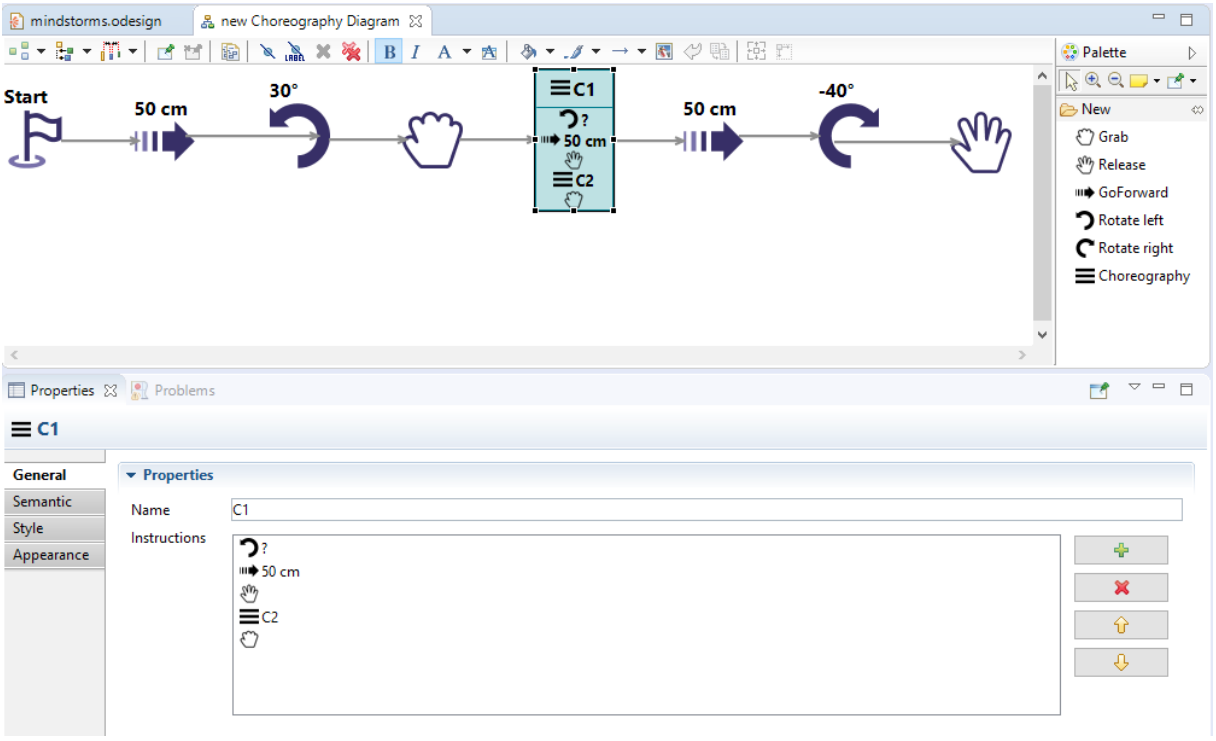
Warning: If you delete your current design project and load the solution, you should close the sample project and reopen it, in order the new version to be taken into account

4. Properties views

4.1. Objectives

Provide custom properties views for a modeling tool that allows the user to define the choreography of a Mindstorms robot.

- **GoForward's properties view**
  - Specific background color if *Cm* value is negative
  - Warning if *Cm* value is null
- **Rotate's properties view**
  - *Random* checkbox and *Degrees* field aligned on the same row
  - *Degrees* field disabled if *Random* is checked
  - Validation rule of type Warning, testing if *Degrees* is equal to 0 and *Random* is not checked
  - Two quick fixes on this rule to propose solutions
- **Choreography's properties view**
  - Editable list of *instructions*
  - Validation rule of type Error, testing if *Name* is already used by another Choreography at the same level



4.2. Sirius Concepts

- **Properties Views Description**
  - Describes how model element are shown and edited in the Eclipse Properties Views
- **Page**
  - Corresponds to a tab in the Properties View
- **Group**
  - Represents a group of widgets in a tab
- **Container**
  - Allows to specify alternate layouts
- **Fill Layout**
  - Organizes elements inside the container either horizontally or vertically

- **Text widget**
  - Represents a single line text
- **Checkbox widget**
  - Represents a checkbox
- **References widget**
  - Represents the value of a reference in the model

- **Property validation**
  - Defines a validation rule linked to a specific widget.
- **Semantic validation**
  - Define a validation rule linked to a group
- **Audit**
  - Evaluates if a validation rule has been broken

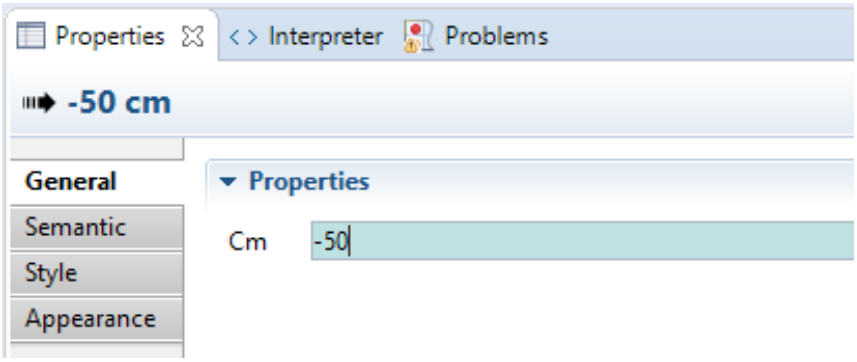
4.3. Instructions

4.3.1. Create a Properties View for GoForward instructions

- At the root of the modeler definition create a **New Properties View** to define custom properties views for the instructions
  - *Metamodels* = `mindstorms`
  - Update the **Page** named **Default Page** (already created by default)
    - *Domain Class* = `mindstorms::Instruction`
    - *Label Expression* = `General`

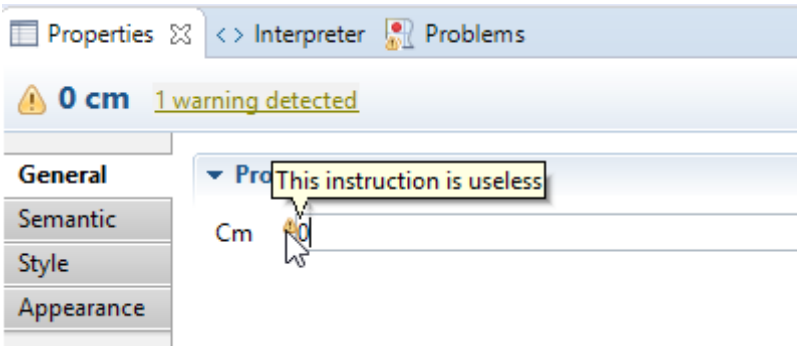
- Update the **Group** named **Default Group** to display and edit the **cm** property of **GoForward**
  - *Id* = `GoForward`
  - *Domain Class* = `mindstorms::GoForward`
  - *Label Expression* = `Properties`
  - Add a **Text** for the **name** property
    - *Id* = `CmText`
    - *Label Expression* = `Cm`
    - *Value Expression* = `feature:cm`
    - Under **Begin** add a **Set** operation (set `var:newValue` to `cm`)
  - Create a **Conditional Style** for the **cm** Text in order to color the text background when *cm* is lower than 0
    - *Precondition Expression* = `aql:self.cm<0`
    - Create a **Style** with *Background Color* = `MindstormsColor1`

- Now, if you select a **GoForward** instruction and enter a negative **Cm** value, you should see:



- Add a **Validation** to warn the user when *cm* is null (useless instruction).
  - Create a **Property Validation Rule**
    - *Targets* = `CmText`
    - *Id* = `UselessGoForward`
    - *Level* = `Warning`
    - *Message* = `This instruction is useless`
    - Create an **Audit**
      - *Audit Expression* = `aql:self.oclAsType(mindstorms::GoForward).cm<>0`

- Now, if you select a **GoForward** instruction and enter a null **Cm** value, you should see:



4.3.2. Create a Properties View for Rotate instructions

- Create a **Group** to display and edit the **degrees** and **random** properties of **Rotate**
  - *Id* = `Rotate`
  - *Domain Class* = `mindstorms::Rotate`
  - *Label Expression* = `Properties`

  - Add this new group to the **Default** page (*Groups* property on this page)

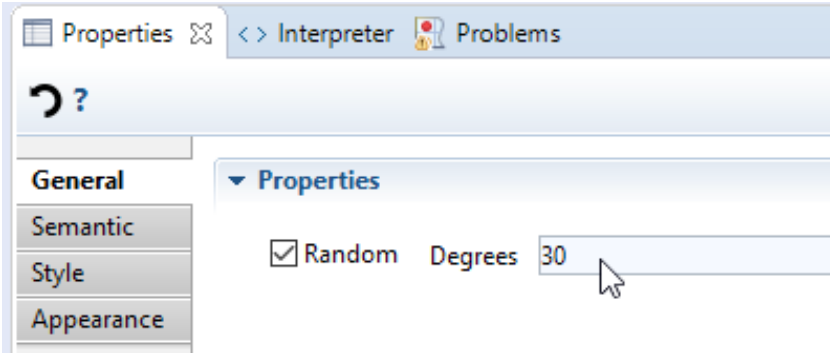
  - Create a Container to put the widgets in the same line
    - Create a Fill Layout with *Orientation* = `HORIZONTAL`

  - Add a **Checkbox** for the **random** property
    - *Id* = `RandomCheckbox`
    - *Label Expression* = `Random`
    - *Value Expression* = `feature:random`
    - Under **Begin** create a **Set** operation (set `var:newValue` to `random`)

  - Add a **Text** for the **degrees** property
    - *Id* = `DegreesText`
    - *Label Expression* = `Degrees`
    - *Is Enabled Expression* = `aql:not self.random`

        - *Value Expression* = `feature:degrees`
        - Under **Begin** create a **Set** operation (set `var:newValue` to `degrees`)

- Now, if you select a **Rotate** instruction the checkbox and the field should be aligned, and the check of Random should disable the Degrees field:



- Add a **Validation** to warn the user when *degrees* is null and *random* is false (useless rotate instruction).

    - Create a **Semantic Validation Rule**
        - *Id* = `UselessRotation`
        - *Level* = `Warning`
        - *Message* = `This instruction is useless`

    - Create an **Audit**
        - *Audit Expression* = `aql:self.degrees<>0 or self.random`

4.3.3. Create a Properties View for Choreographies instructions

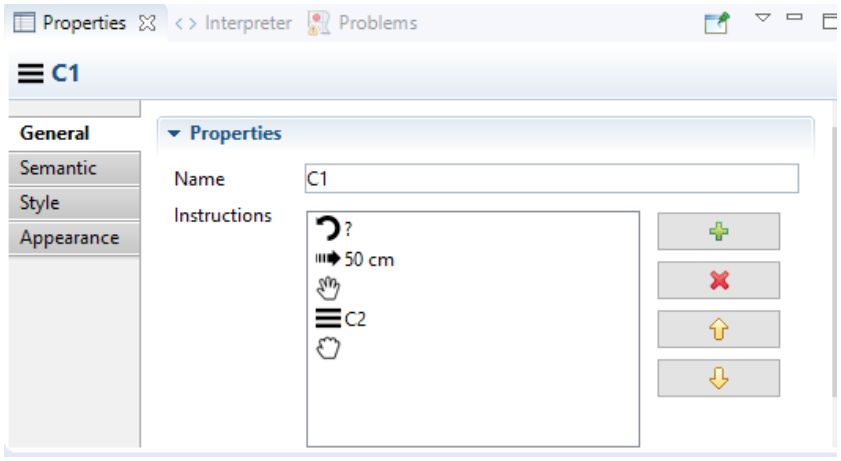
- Create a **Group** to display and edit the **name** and **instructions** properties of **Choreography**
    - *Id* = `Choreography`
    - *Domain Class* = `mindstorms::Choreography`
    - *Label Expression* = `Properties`

    - Add this new **Group** to the **Default** page (*Groups* property on this page)

    - Add a **Text** for the **name** property
        - *Id* = `NameText`
        - *Label Expression* = `Name`
        - *Value Expression* = `feature:name`
        - Under **Begin** create a **Set** operation (set `var:newValue` to `name`)

    - Add a **Reference** for the **instructions** property
        - *Id* = `InstructionsRef`
        - *Label Expression* = `Instructions`
        - *Reference Owner Expression* = `var:self`
        - *Reference Name Expression* = `instructions`

    - Now, if you select a **Choreography** instruction, the properties view contains a list with action buttons:



- Add a **Validation** to warn the user when the **name** is already used by a sibling choreography.
  - Create a **Property Validation Rule**
    - *Targets* = `NameText`
    - *Id* = `UniqueName`
    - *Level* = `Error`
    - *Message* = `Name must be unique`
  - Create an **Audit**
    - *Audit Expression* =
```
aql:not self.siblings()
->filter(mindstorms::Choreography)
->collect(i|i.name)
->includes(self.name)
```

4.4. Solution

The solution of this step is provided in **solution4.zip or switch to branch step4 of the git repository**