Tutorial

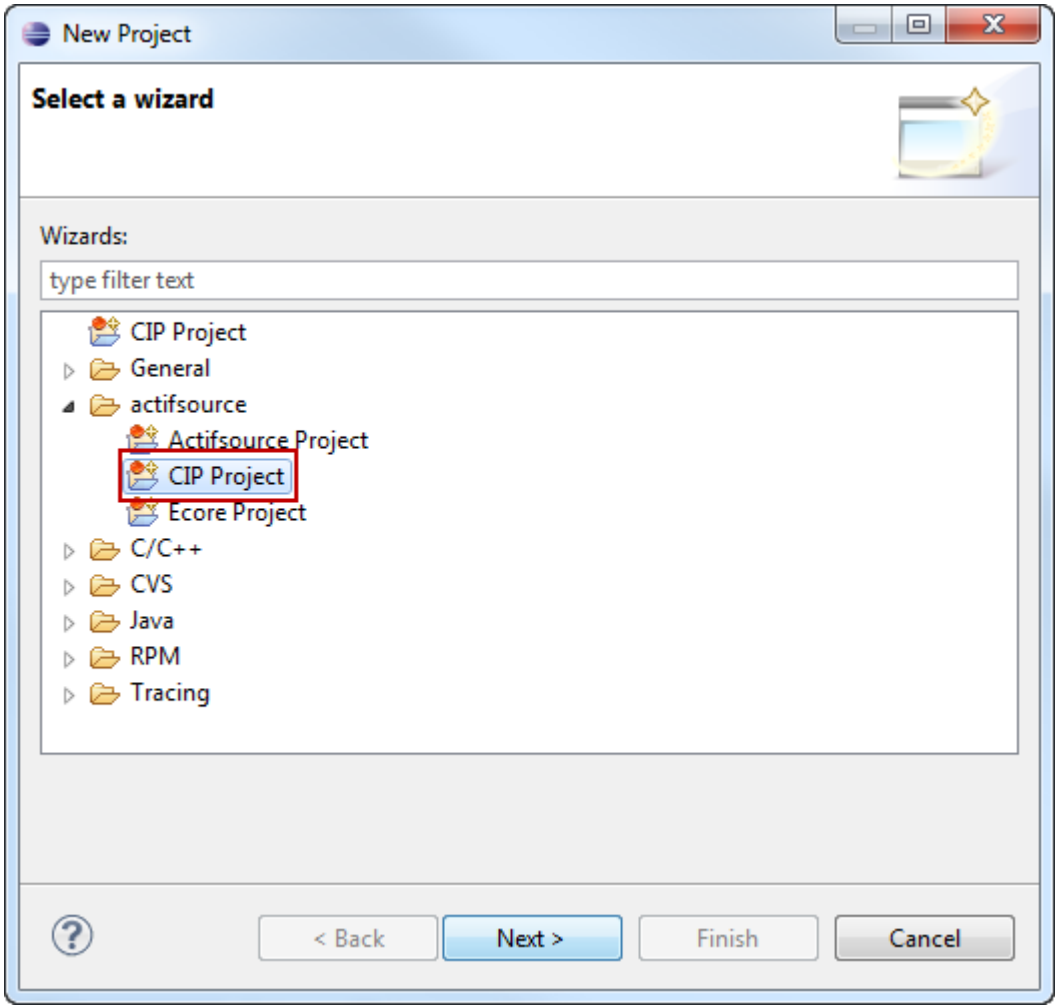
CIP Statemachine - Lamp

Tutorial	Actifsource Tutorial – CIP Statemachine - Lamp
Required Time	• 60 Minutes
Prerequisites	• Actifsource Tutorial – Installing Actifsource • Actifsource Tutorial – Simple Service
Goal	• Creating a state machine using the CIP method • Generating real time C code for any embedded system
Topics covered	• Setting up a new CIP Project • Communicating with the Outer World • Specify the State Machine • Generating State Machine Code
Notation	To do ⓘ Information • **Bold**: Terms from actifsource or other technologies and tools • **Bold underlined**: actifsource Resources • Underlined: User Resources • *UnderlinedItalics*: Resource Functions • `Monospaced`: User input • *Italics*: Important terms in current situation
Disclaimer	The authors do not accept any liability arising out of the application or use of any information or equipment described herein. The information contained within this document is by its very nature incomplete. Therefore the authors accept no responsibility for the precise accuracy of the documentation contained herein. It should be used rather as a guide and starting point.
Contact	**actifsource GmbH** Täfernstrasse 37 5405 Baden-Dättwil Switzerland www.actifsource.com
Trademark	**actifsource** is a registered trademark of **actifsource GmbH** in Switzerland, the EU, USA, and China. Other names appearing on the site may be trademarks of their respective owners.
Compatibility	Created with **actifsource** Version 5.8.7

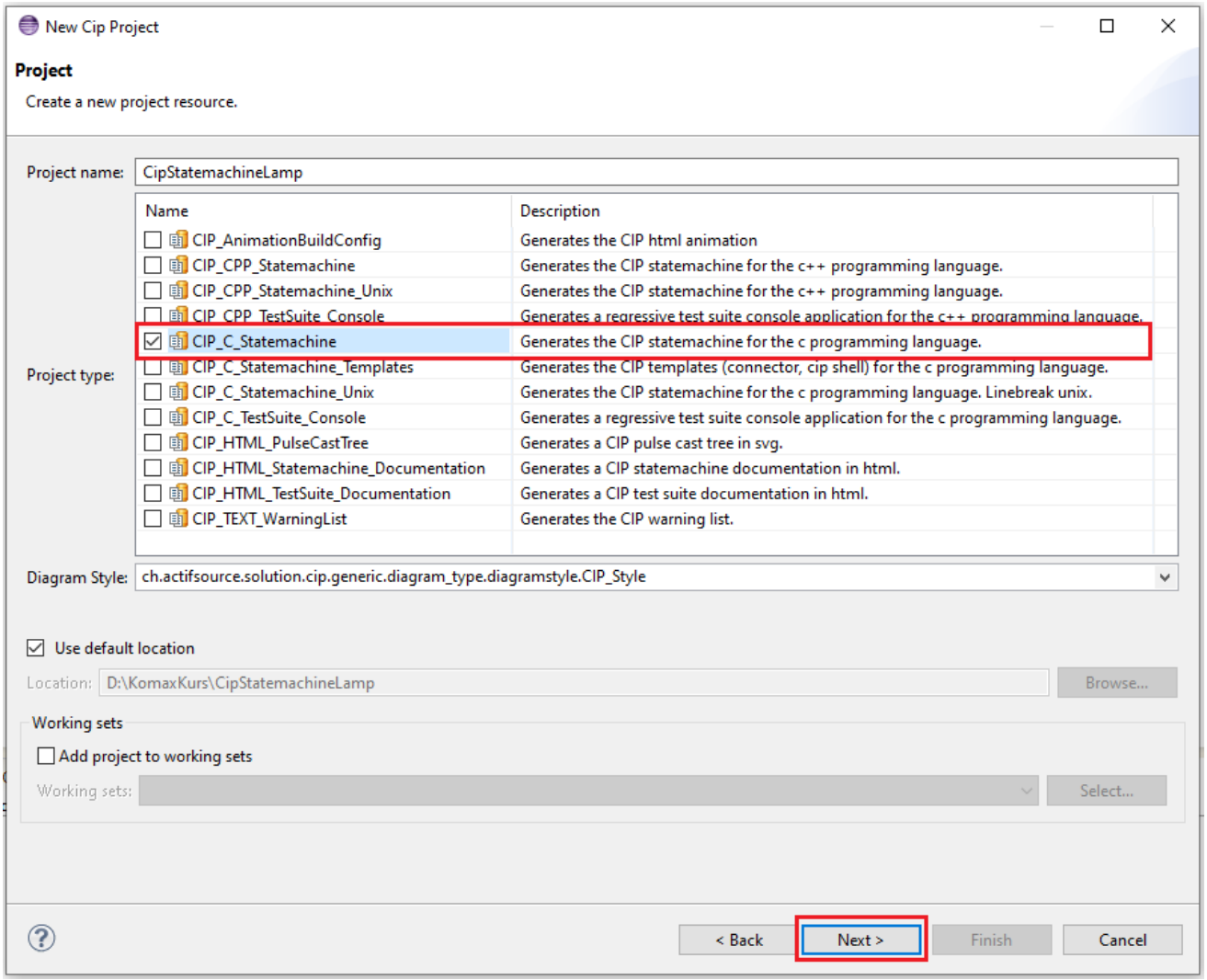
- Learn how to specify a simple state machine
- Example
  - Button to turn on and off a lamp
  - Turning off the lamp shall be delayed
- CIP Method
  - The CIP System is the root element
  - The CIP System consists of Clusters
    - Clusters are used to model distributed state machines
  - The CIP Cluster consists of Processes
    - The process declares the state of the state machine
  - The CIP Process consists of Modes
    - Modes are used for different situations like normal, error, run-in, run-out
    - The mode declares the transitions between the states

Part I: 4
Setting up a new CIP Project

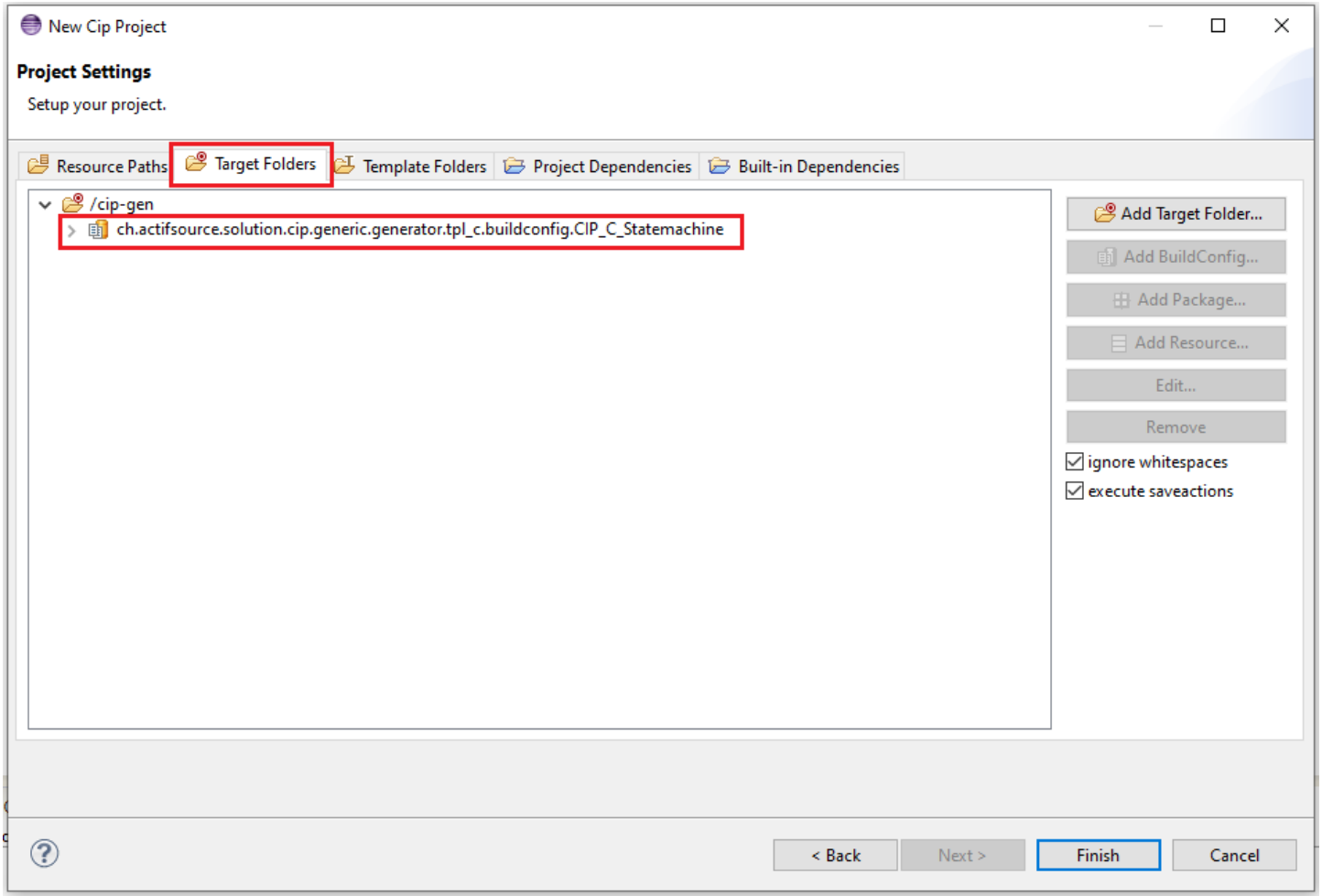
- Create a new CIP Project with the CIP Project wizard



- Prepare a new **Actifsource/CIP Project** using the Actifsource CIP wizard
  - File/new/other
  - Actifsource/CIP Project
- Click *Next*



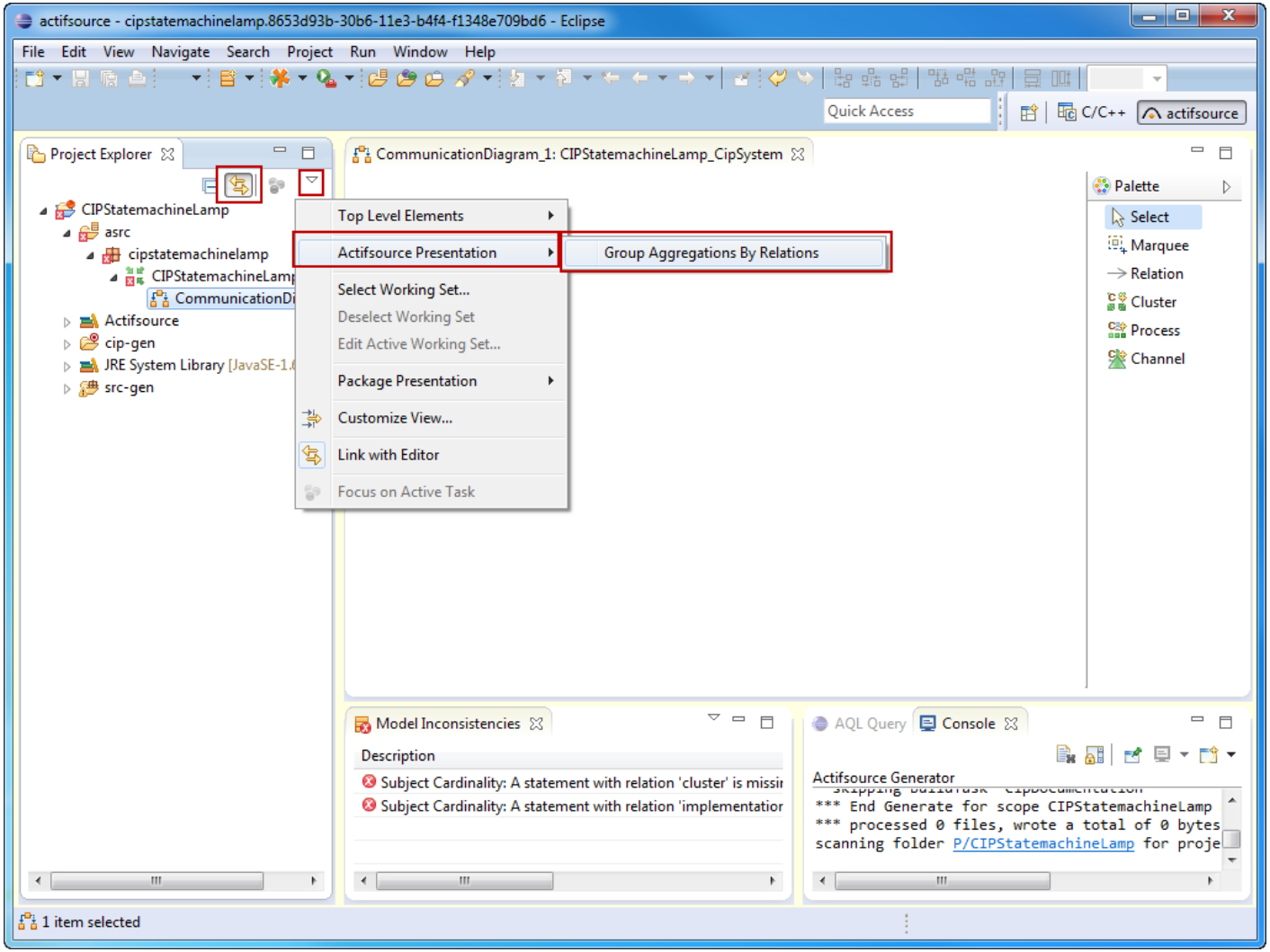
- Specifiy *Project name*
- Specify *Project type*
- Click *Next*



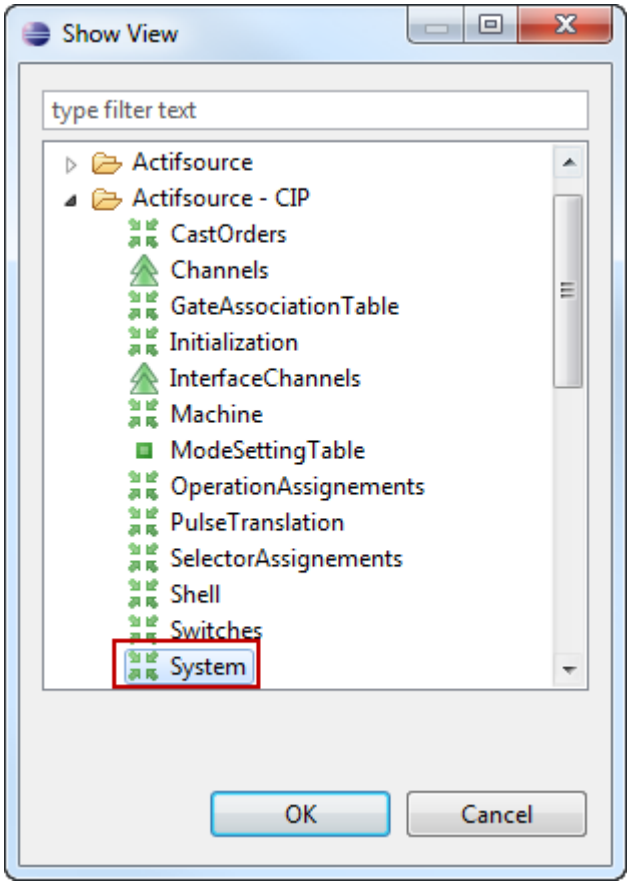
ⓘ Note that the BuildConfigs in TargetFolder are equivalent to the previously selected project type

ⓘ You may add other build configs (i.e. test suites) as needed

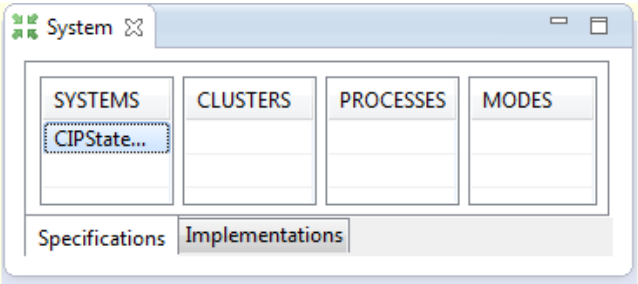
↳ Click *Finish*



- *Link with Editor*
- Enable *Actifsource Presentation* Flag Group Aggregations By Relations

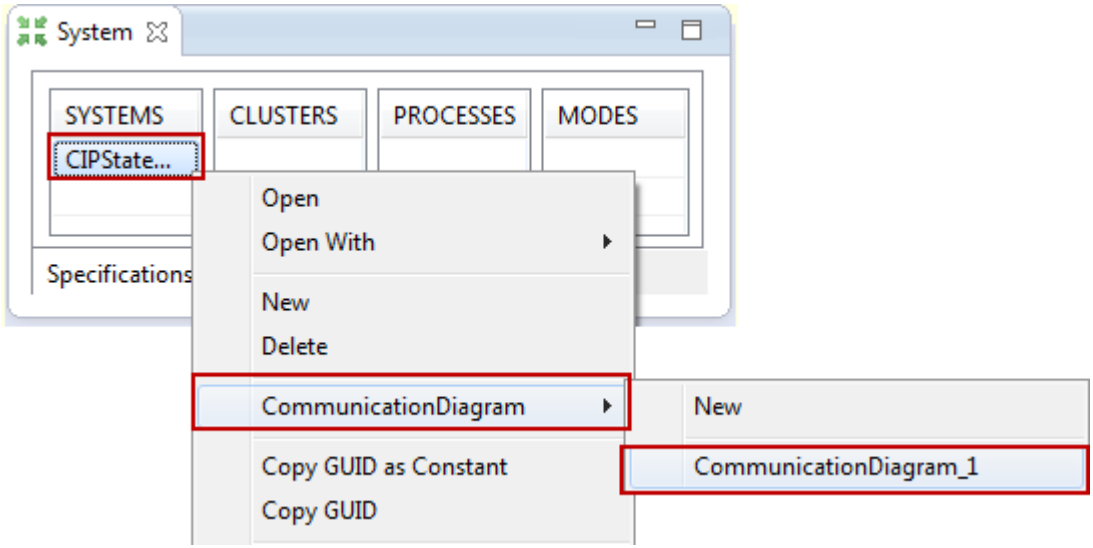


- Open *CIP System View*
  - Window/Show View/Other…
  - Actifsource – CIP / System

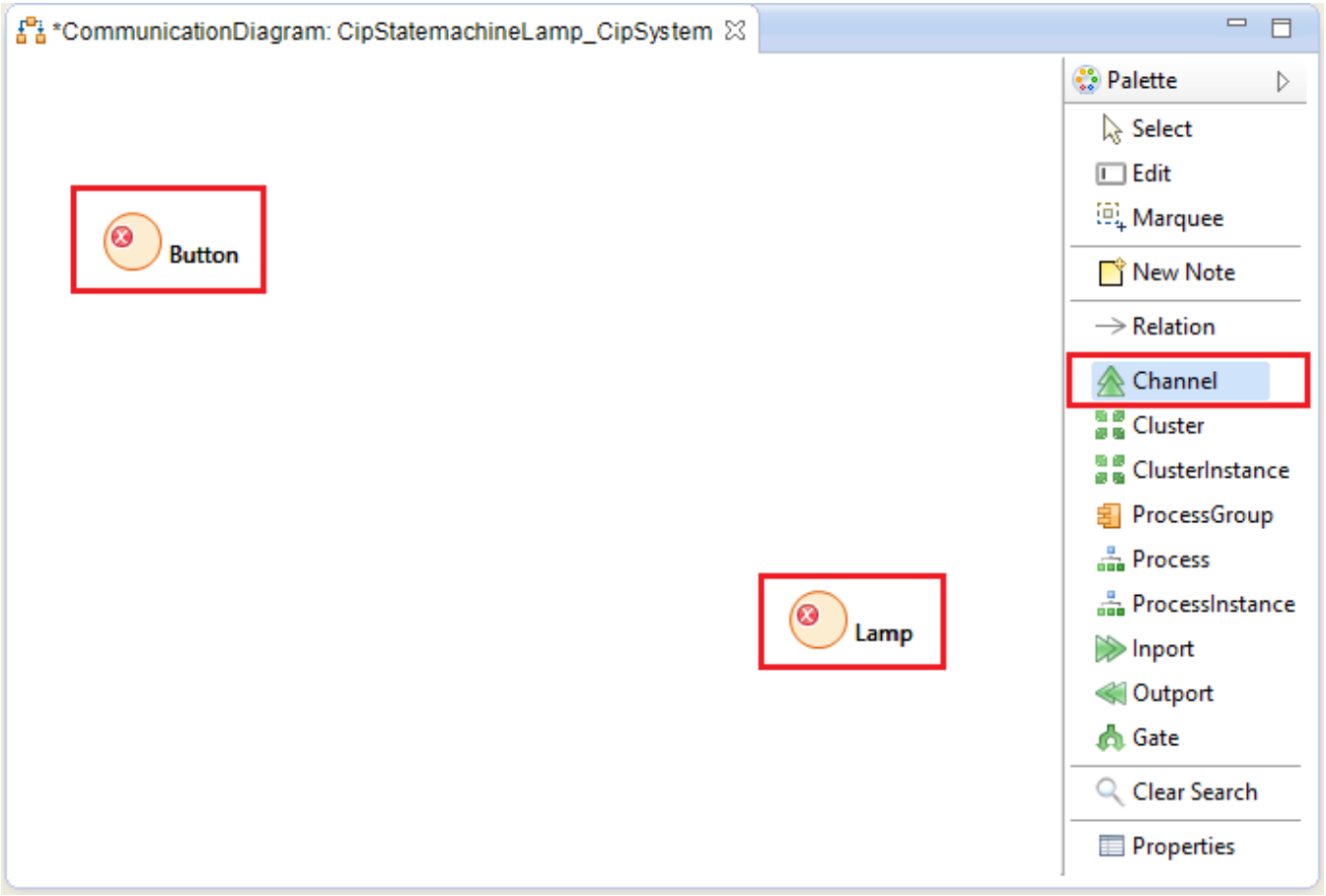


Part II: 10
Communicating with the Outer World

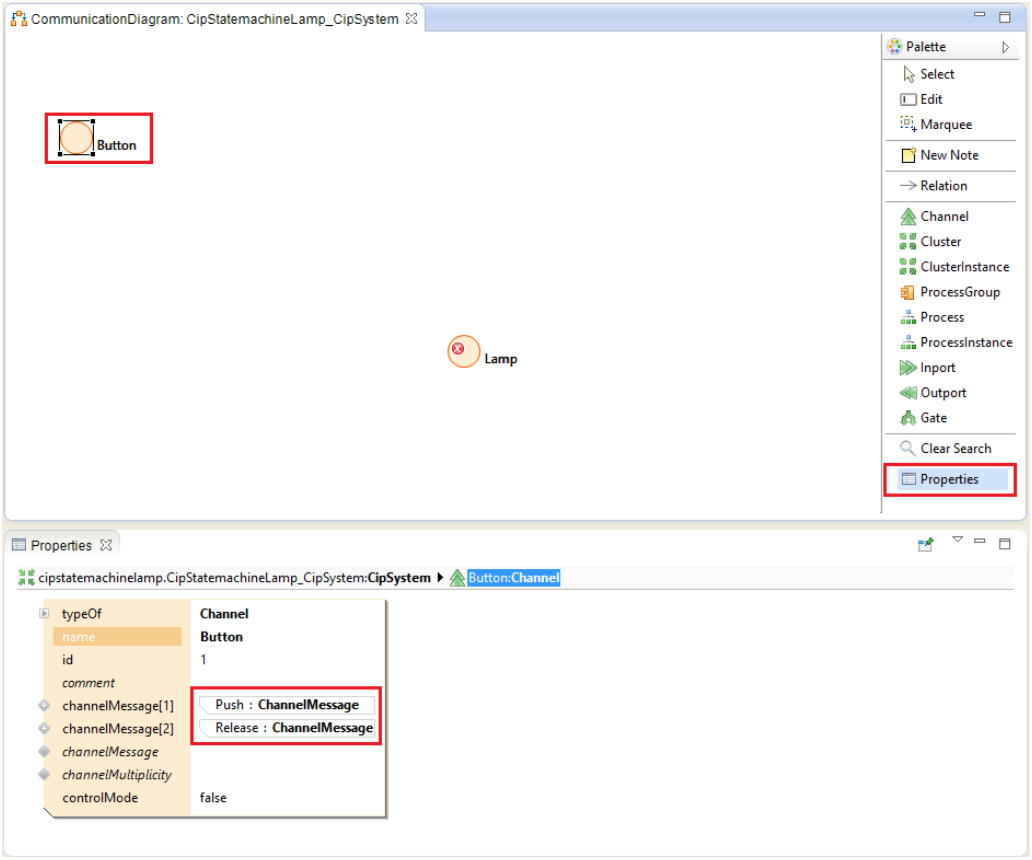
- Let’s communicate with the outer world
- Channels are providing messages from physical device



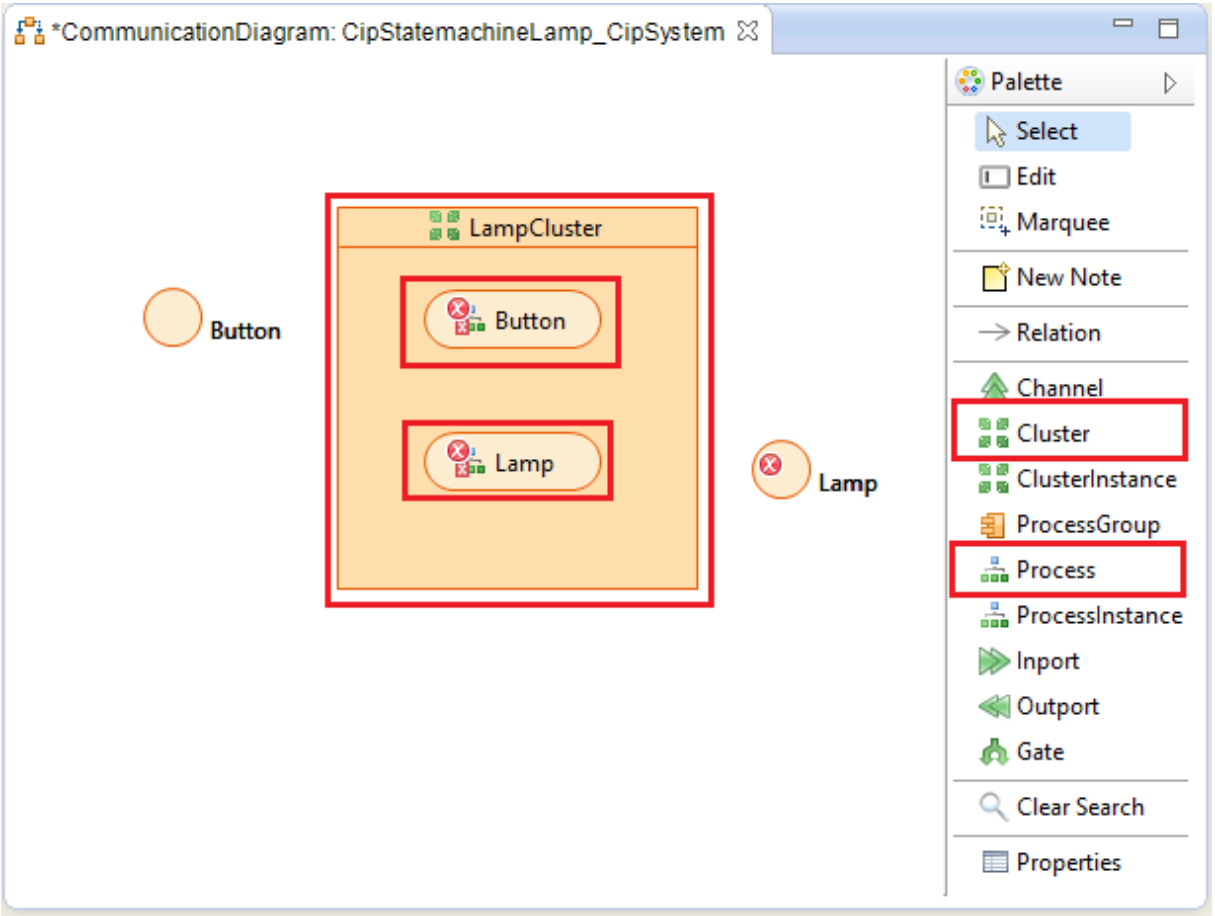
- Open the Communication Diagram (if not already open)
  - Right Click on System1 in the *System View*
  - Select *CommunicationDiagram* from the context menu
  - Open CommunicationDiagram_1



↳ Create two new **Channels** named Button and Lamp using the *Channel* tool from the *Palette*



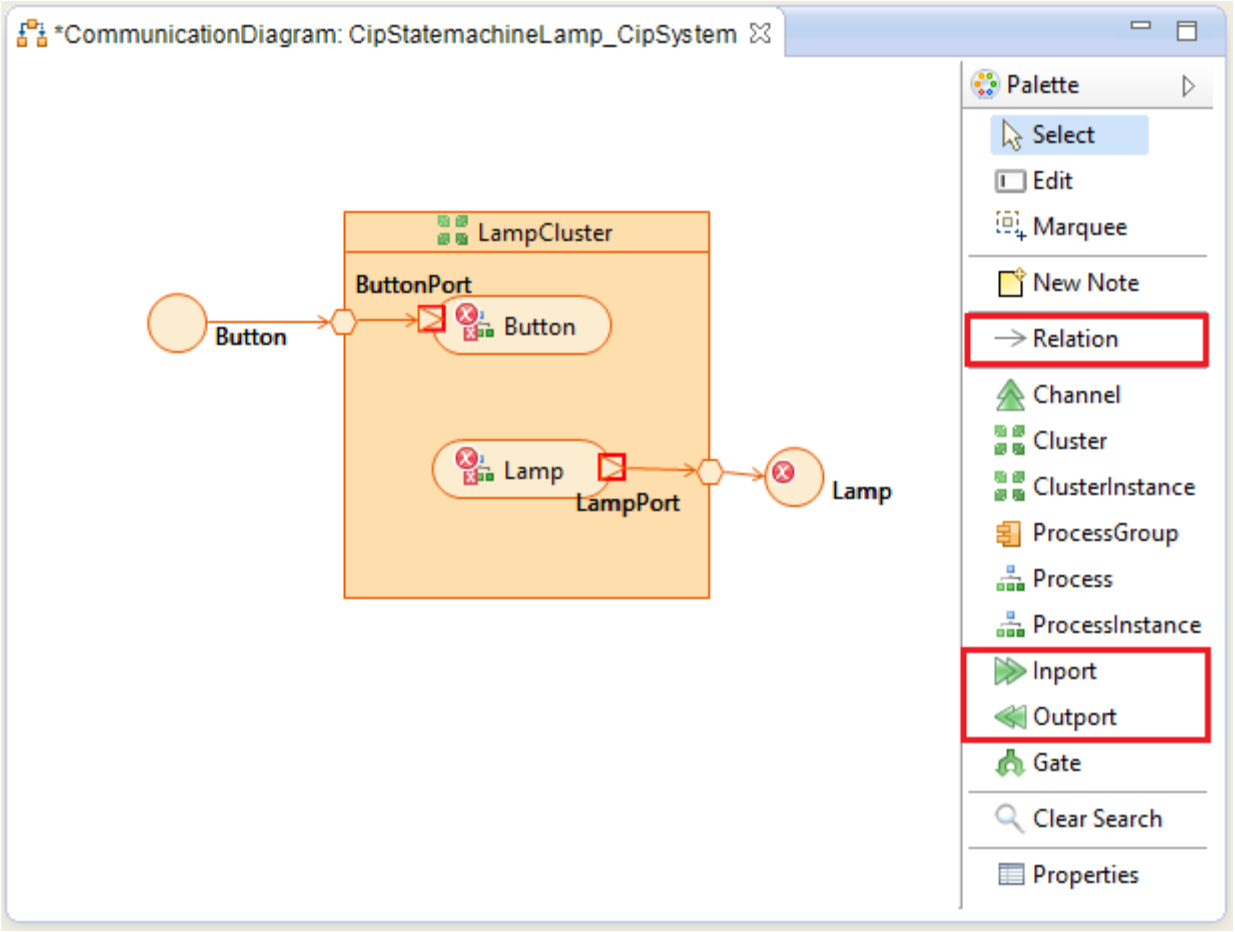
- ⓘ Arrange the Graphical Editor and the Resource Editor together on the same screen as shown above
- ↳ Add the two **ChannelMessages** Push and Release to the **Channel** Button
  - Ctrl+Click on the Button label
  - Add **ChannelMessage** Push and **ChannelMessage** Release
- ↳ Add the two **ChannelMessages** Bright and Dark to the **Channel** Lamp
  - Ctrl+Click on the Lamp label
  - Add **ChannelMessage** Bright and **ChannelMessage** Dark
- ⓘ Messages are given as function calls from the other world to the state machine and vice versa



ⓘ The CIP Method specifies that every physical process needs a logical counterpart in the model

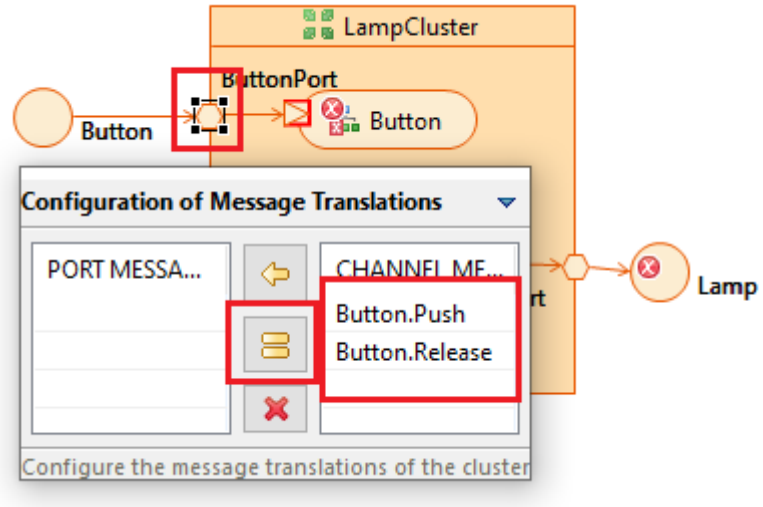
ⓘ Note that different Clusters may run on different processors. This allows you to design distributed state machines using the CIP Method.

↳ Add the two **Processes** Button and Lamp to the **Cluster** LampCluster



ⓘ Channels are Delivering Messages to a Process via Port. Doing so allows you to consider the Process to a self-consistent component.

- Create an **Inport** ButtonPort and a relation from the **Channel** Button to the **Port** ButtonPort
- Create an **Outport** LampPort and a relation from the **Port** LampPort to the **Channel** Lamp

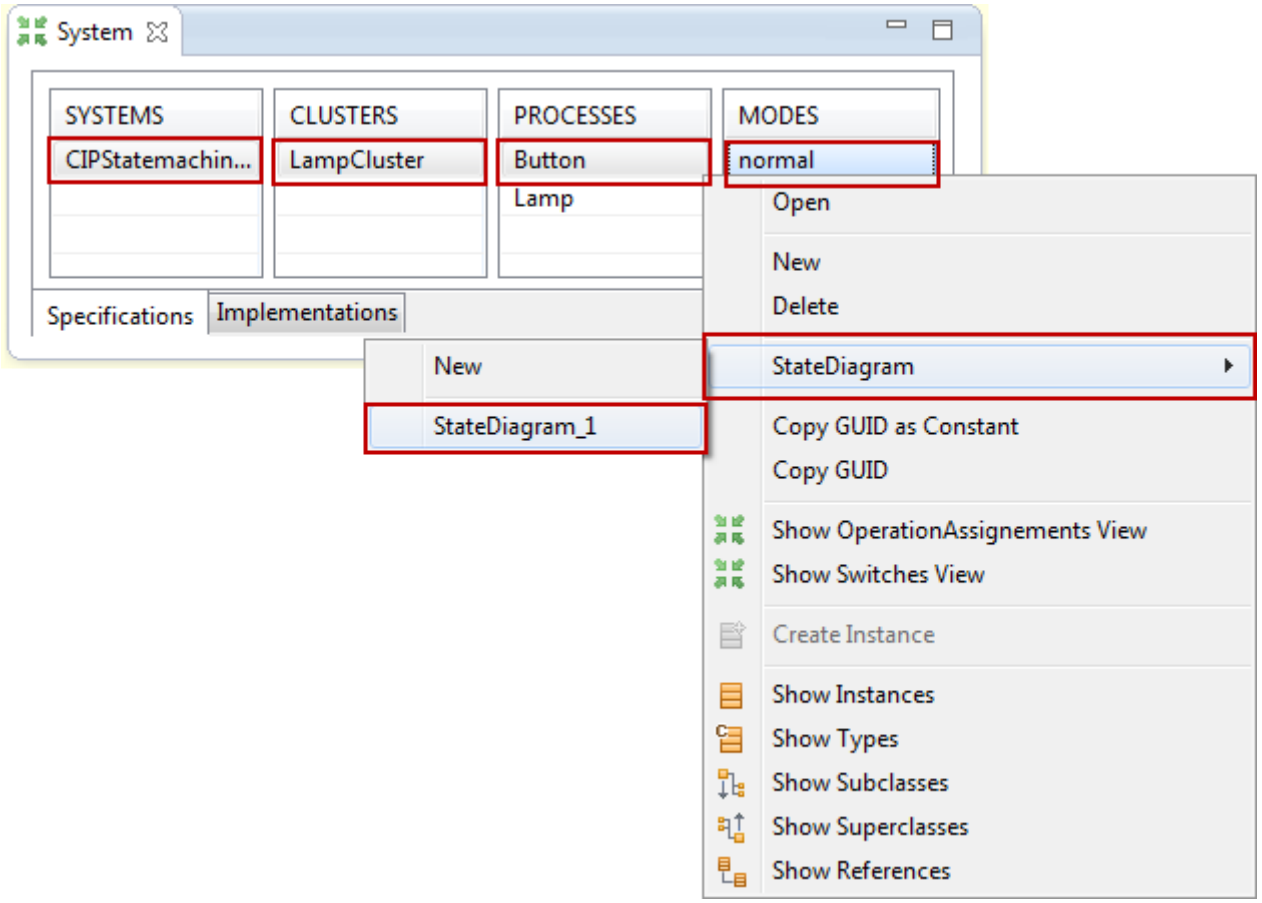


ⓘ Since every **Process** is a self-consistent component we have to specify a **Message** interface on the port to. The **=** tool in the *Message Translation* helps us to create the same messages as found on the **Channel** also on the corresponding **Port**.

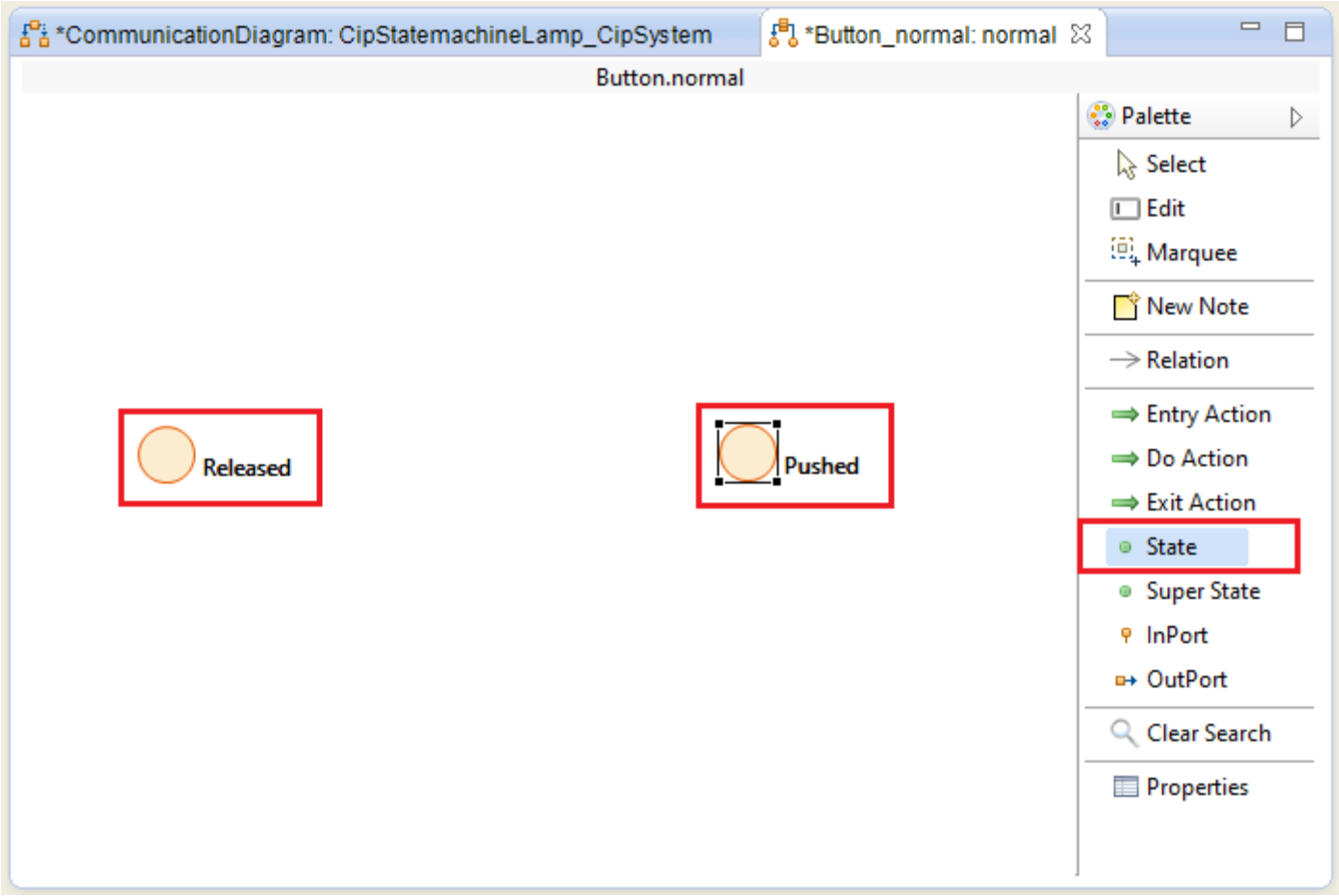
- Configure the *Message Translation* between the **Channel** Button and the **Process** Button
  - Double-Click on the hexagon of the relation between the **Channel** Button and the **Process** Button
  - Select the **ChannelMessage**s Button.Push and Button.Release
  - Use the =-Tool to create the corresponding **Messages** on the **Port** ButtonPort of the **Process** Button
- Configure the *Message Translation* between the **Process** Lamp and the **Channel** Lamp
  - Double-Click on the hexagon of the relation between the the **Process** Lamp and the **Channel** Lamp
  - Select the **ChannelMessage**s Lamp.Bright and Lamp.Dark
  - Use the =-Tool to create the corresponding **Messages** on the **Port** LampPort of the **Process** Lamp

Part III: Specify the State Machine

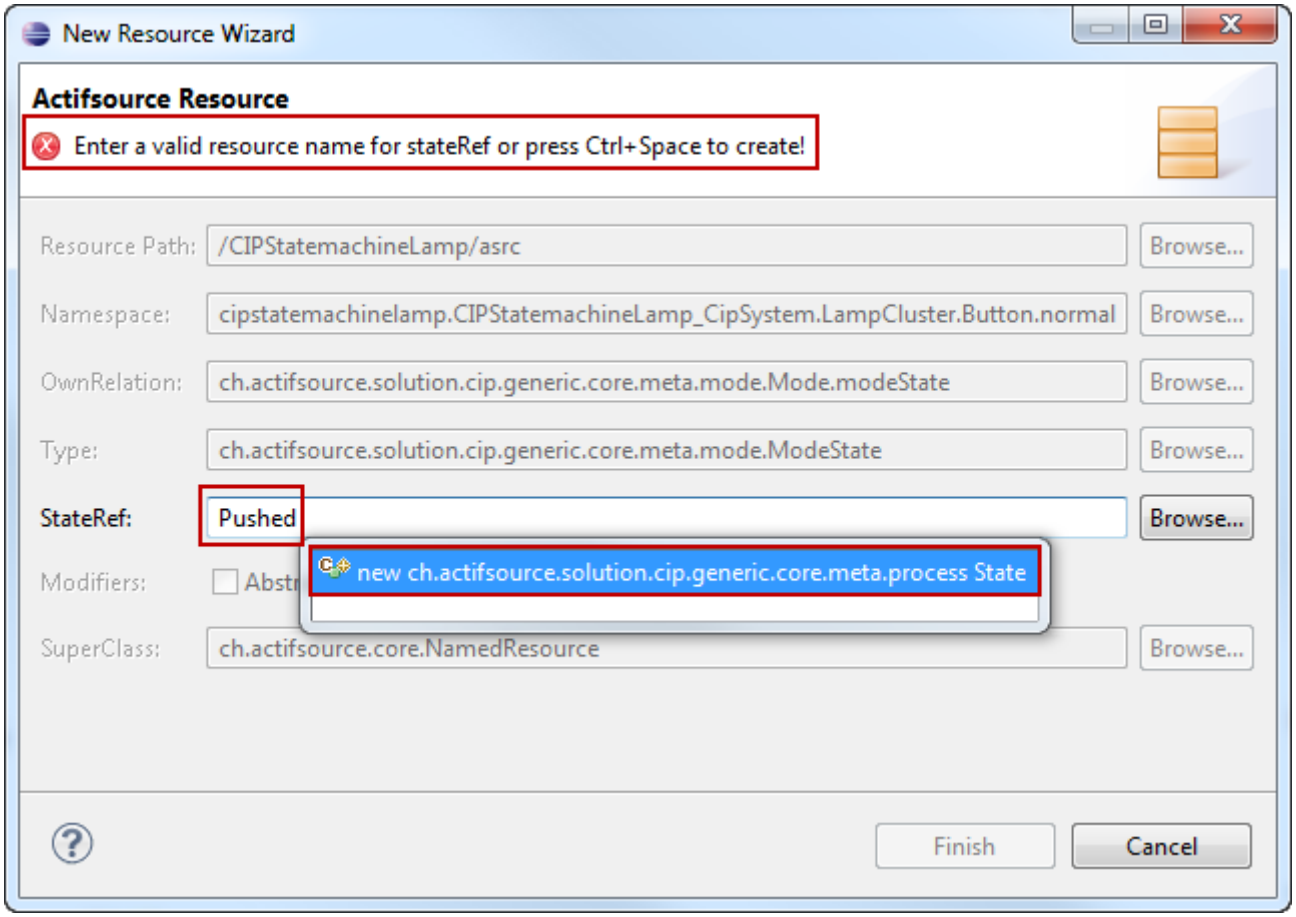
- We are now set to specify the state machines for each process
- Note that the CIP Method knows so called **Modes**
  - A Process can have one or more Modes
  - States are declared by the Process
  - Transitions are declared by the Mode
- Making this difference it becomes possible to provide several Modes for several situations
  - Normal Mode
  - Error Mode
  - Run-In Mode
  - Run-Out Mode
- There are the following rules
  - **States** are defined in the **Process**
  - **States** are shared for every **Mode**
  - **Transitions** are defined in the **Mode**



- Open the *StateDiagram* for **Process** Button
  - On the *System View* Right-Click on System1.Lamp.Button.normal
  - Select *StateDiagram*
  - Select StateDiagram_1



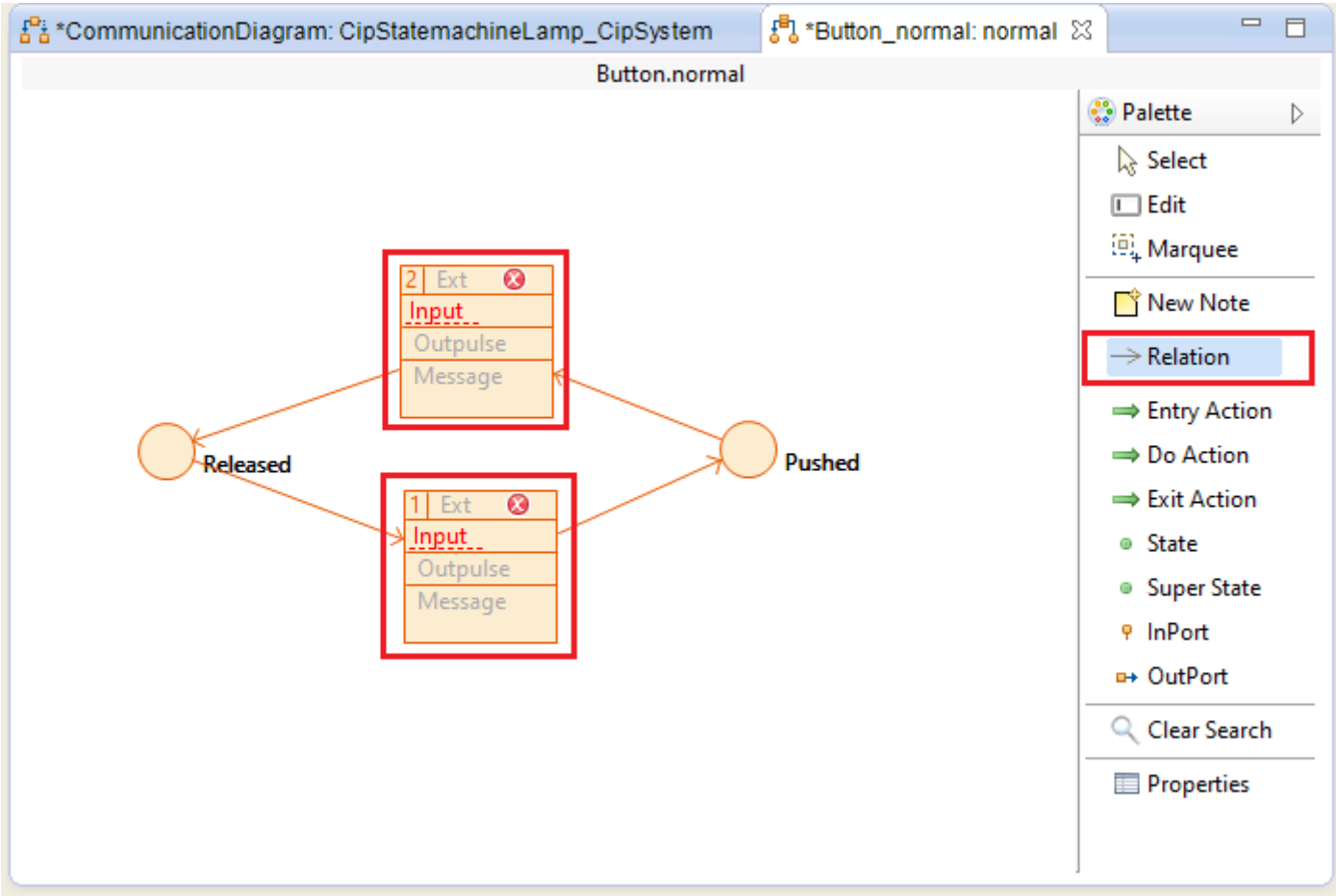
↳ Enter two new **States** Released and Pushed using the *Palette*



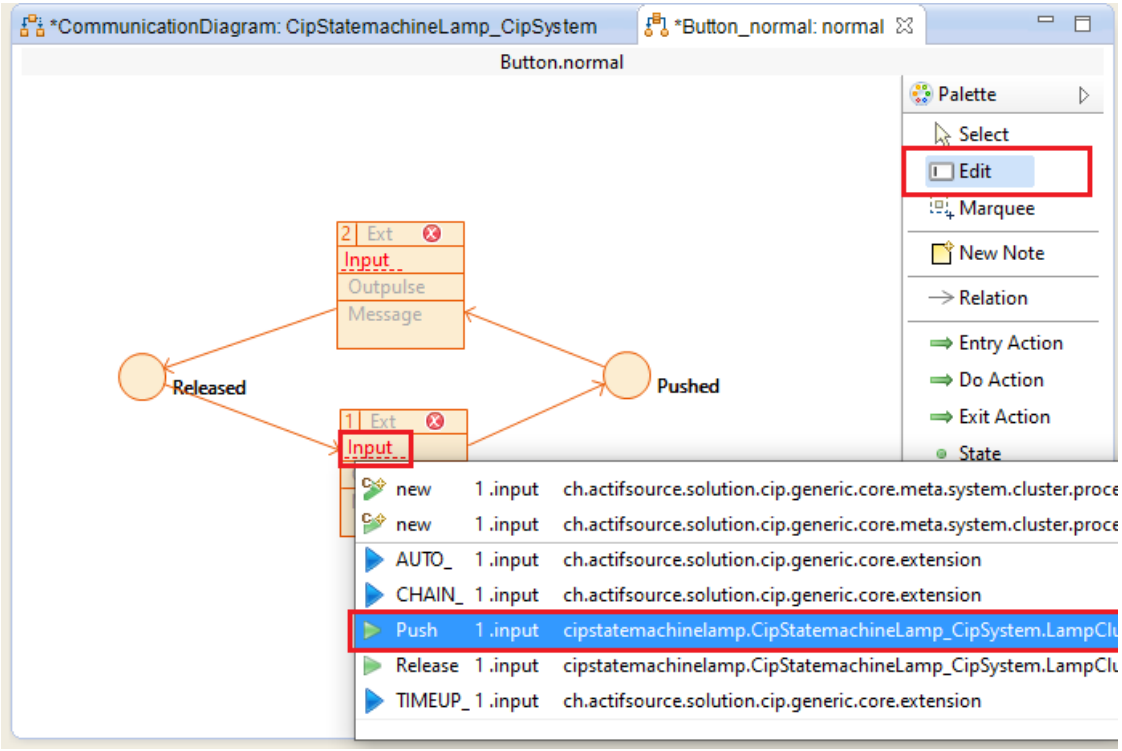
ⓘ State are shared for every mode

↳ Enter the desired **State** name Released and press Ctrl-Space

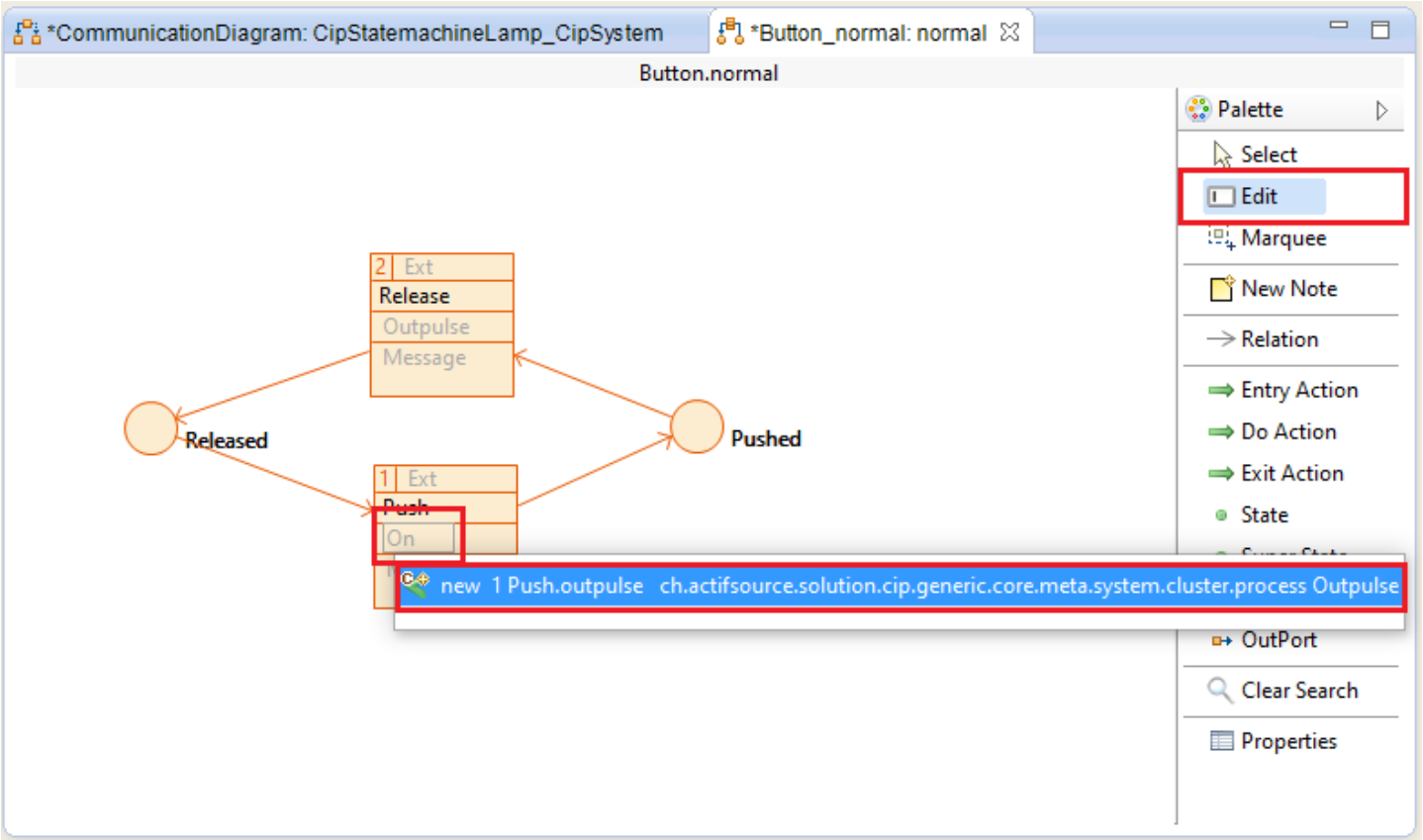
- Select an existing **State**
  or
- Create a new **State** (as we do in our example)



- Create a new **Transition** from **State** Released to **State** Pushed using the *Relation* Tool from the *Palette*
- Create a new **Transition** from **State** Pushed to **State** Released using the *Relation* Tool from the *Palette*

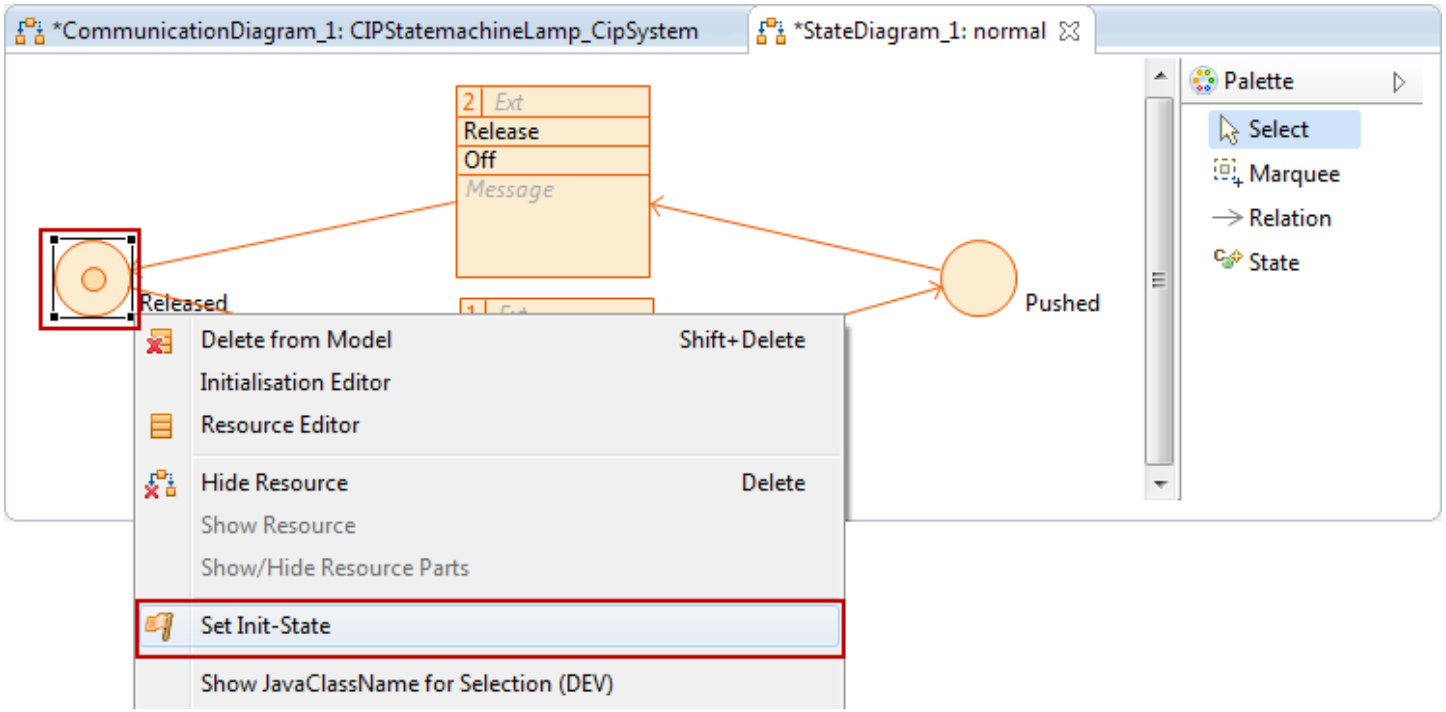


- ⓘ Every Transition needs an input to be triggered
- ⓘ Note that the Actifsource Validator marks all resources that are incomplete
- ↳ Add the input message to **Transition** from **State** Released to **State** Pushed
  - Ctrl-Klick on the italics label *Input*
  - Select the **InportMessage** Push
- ↳ Add the input message to **Transition** from **State** Pushed to **State** Released
  - Ctrl-Klick on the italics label *Input*
  - Select the **InportMessage** Release
- ⓘ Note that you now selecting the **Messages** that we have previously created on the **Port** Button using the =-Tool in the *Message Translation*

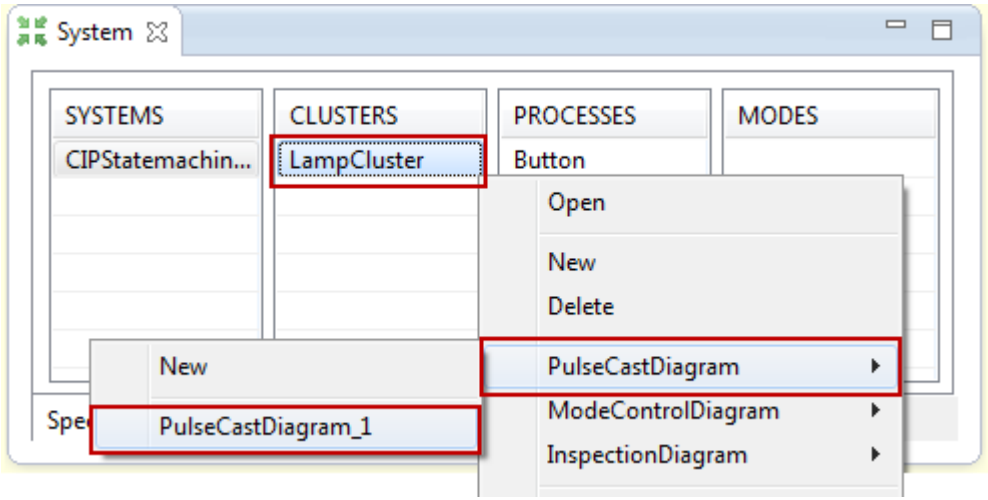


- ⓘ Use **Messages** to communicate with **Processes** from the outer world
- ⓘ Use **Pulses** to communicate between **Processes** within the same **Cluster**
- ↳ Add the **Outpulse** On to **Transition** from **State** Released to **State** Pushed
  - Ctrl-Klick on the italics label *Outpulse*
  - Enter the desired **Outpulse** named On and press Ctrl-Space
    - Select an existing **Outpulse**

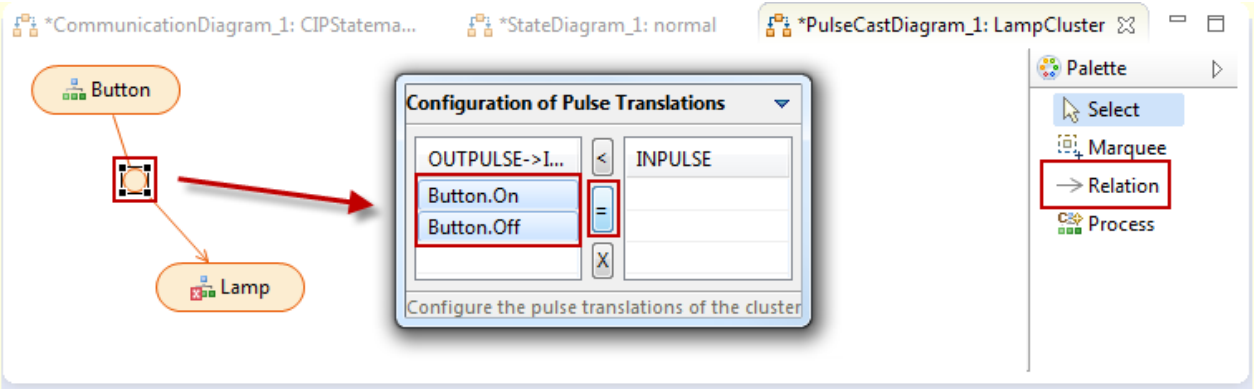
      or
    - Create a new **Outpulse** (as we do in our example)
- ↳ Add the **Outpulse** Off to **Transition** from **State** Pushed to **State** Released



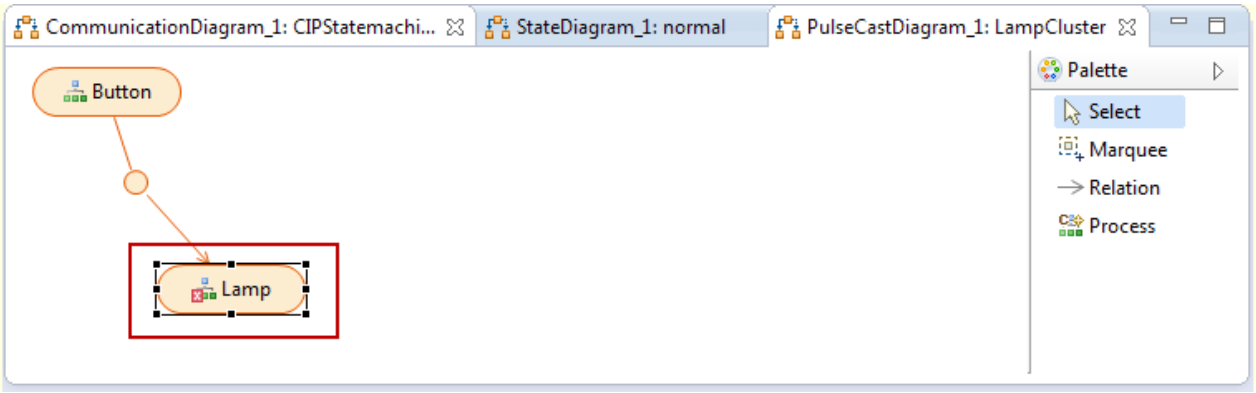
- Set the *Init-State* to specify the **State** to start with
  - Right-Click on **State** Released
  - Shape Actions/Set Init-State



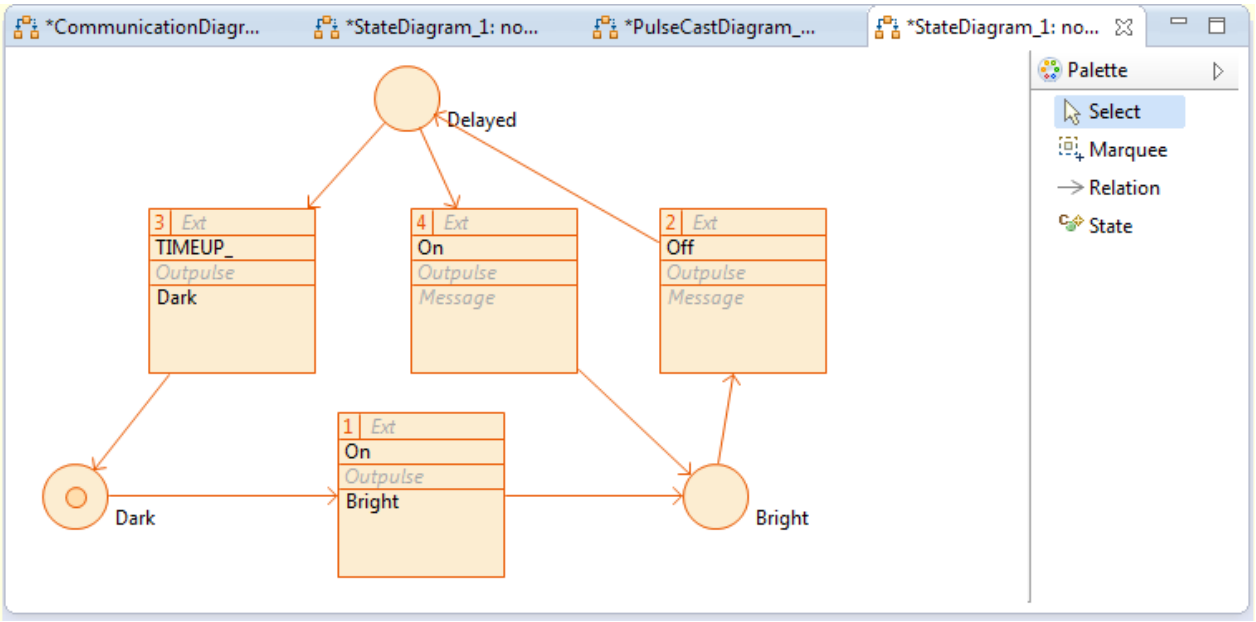
- ⓘ Use **Pulses** to communicate between **Processes** within the same **Cluster**
- ⓘ The *PulseCastDiagram* specifies which **Processes** sends **Pulses** to which other **Processes**
- ↳ Open the *PulseCastDiagram* for **Cluster** Lamp
  - On the *System View* Right-Click on System1.LampCluster
  - Select *PulseCastDiagram*
  - Select *PulseCastDiagram_1 (or create a new one if necessary)*



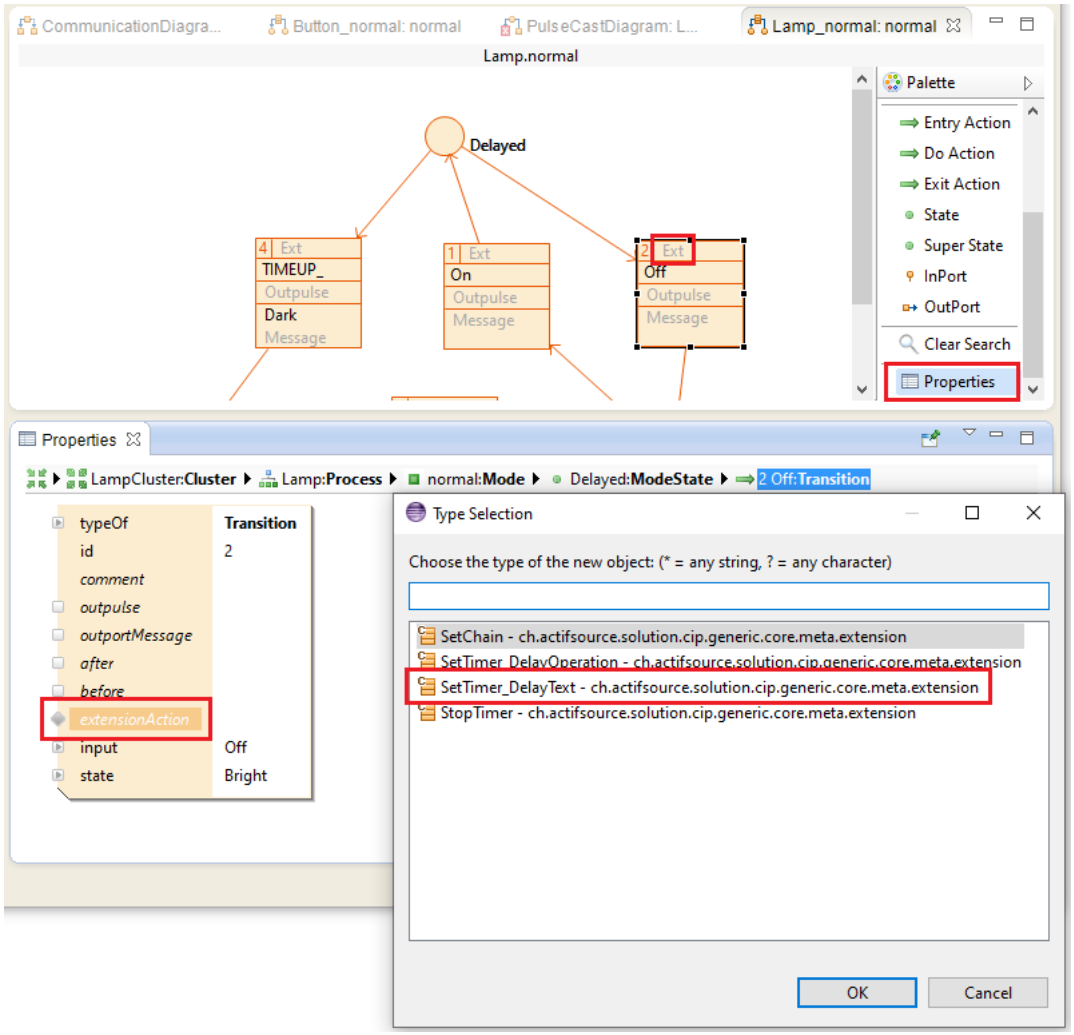
- Add the **Process** Button and the **Process** Lamp to your *PulseCastDiagram*
  - Right-Click on the Diagram
  - Show Resource
  - Select Button and Lamp
- Connect the **Process** Button and the **Process** Lamp using the *Relation* Tool from the *Palette*
- Configure the *Pulse Translation* between the **Process** Button and the **Process** Lamp
  - Double-Click on the circle of the relation between the the **Process** Button and the **Process** Lamp
  - Select the **Outpulses** Button.On and Button.Off
  - Use the **=**-Tool to create the corresponding **Inpulses** on the **Process** Lamp



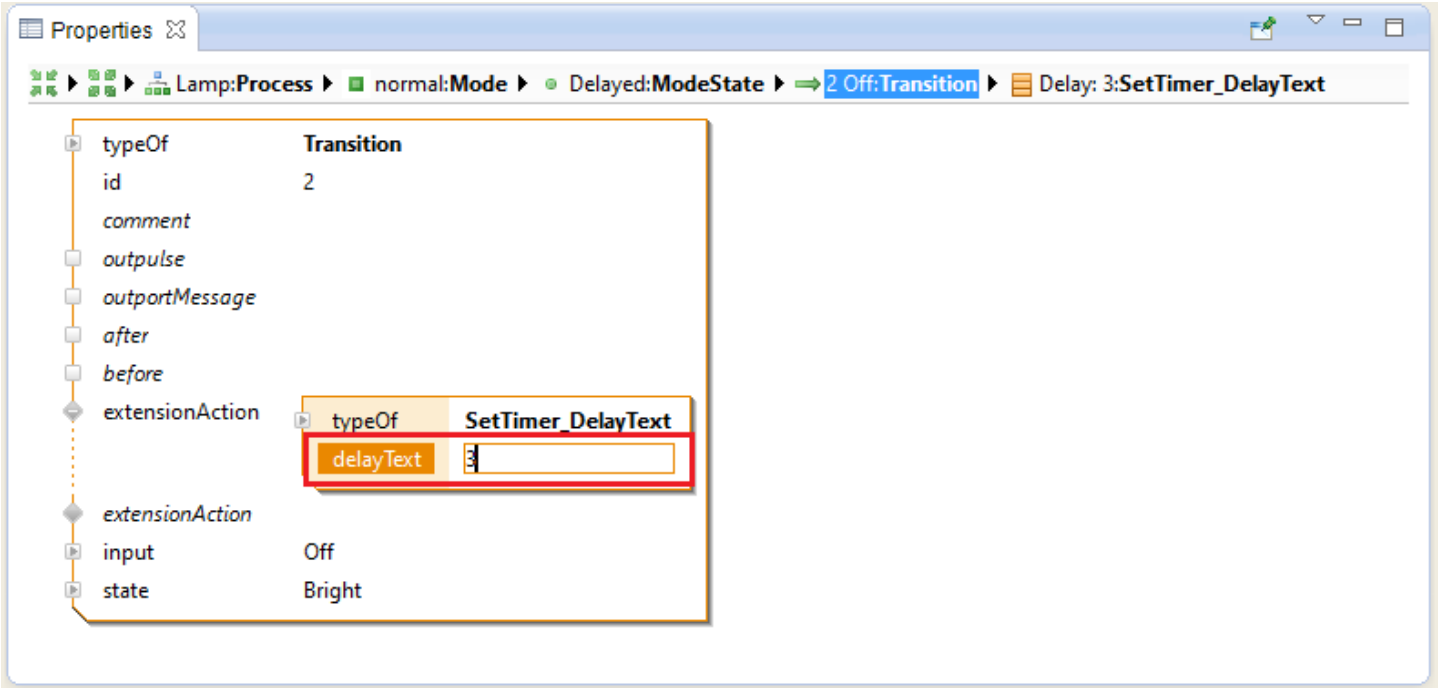
↳ Double-Click on **Process** Lamp to open the *StateDiagram*



- Open the *StateDiagram* for **Process** Lamp (if not already done)
  - On the *System View* Right-Click on System1.Lamp.Button.normal
  - Select *StateDiagram*
  - Select StateDiagram_1
- Create **States** Dark, Bright and Delayed
- Make **State** Dark the *Init-State*
- Create Transition as shown above
  - Select **Pulses** which have been created in the *Pulse Translation*
  - _TIMEUP is a special pulses emitted if timer expires
  - Select Lamp **Messages** to the outer world



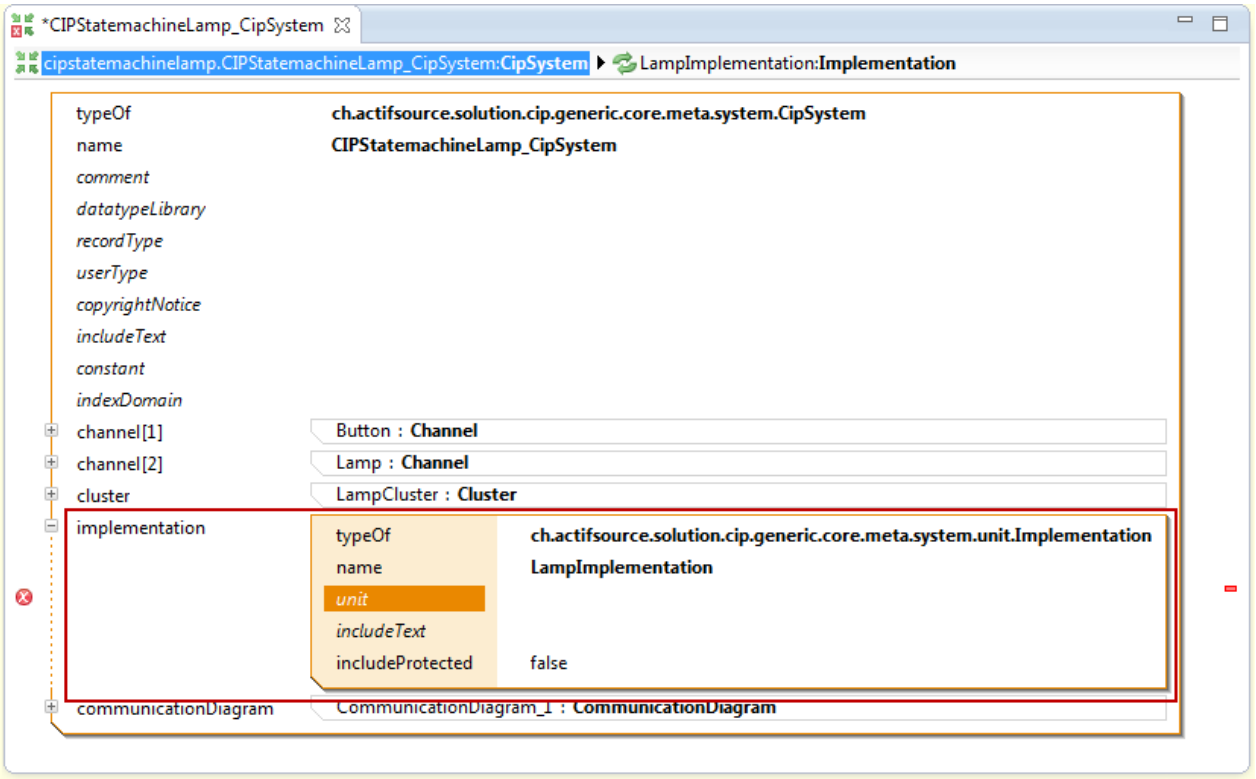
- Start Timer in Transition #2
  - Ctrl-Click in label *Ext*
  - Select SET_TIMER
  - Specify DelayOperation (see next page)
- Stop Timer in Transition #4
  - Ctrl-Click in label *Ext*
  - Select STOP_TIMER



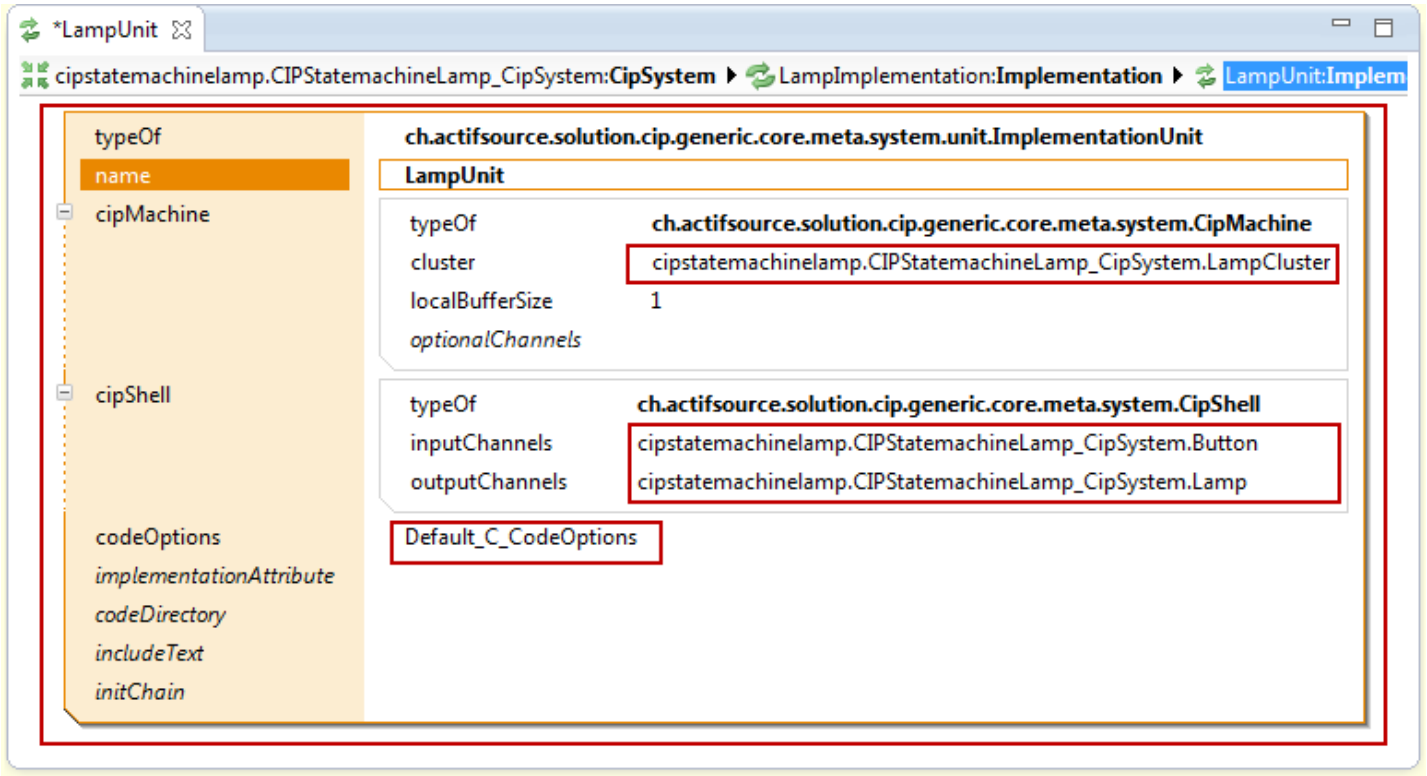
- Create a SetTimer_DelayText
- The DelayText shall return the delay in system ticks

Part IV: Generating State Machine Code

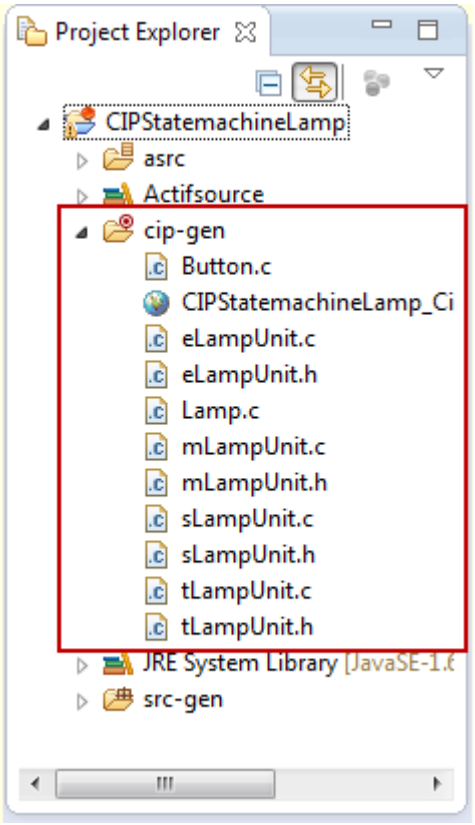
- We have to define the details for the state machine implementation to generate the code



- Create a new *Implementation* for **System** System1
  - On the *System View* Right-Click on System1
  - Create a new Implementation



- Create a new *Implementation* as shown above



- ⓘ Find the generated code in the folder *cip-gen*
- ⓘ Find the documentation (html) code in the folder *cip-gen*