eTrice User Guide

Thomas Schuetz, Henrik Rentz-Reichert, Thomas Jung and contributors

Chapter 1. eTrice Overview

1.1. What is eTrice?

eTrice provides an implementation of the ROOM modeling language (Real Time Object Oriented Modeling) together with editors, code generators for Java, C++ and C code and exemplary target middleware.

The model is defined in textual form (Xtext) with graphical editors (Graphiti) for the structural and behavioral (i.e. state machine) parts.

1.2. Who should use eTrice?

Basically everyone who develops eventdriven realtime or embedded systems.

If you have other ideas how to use it, tell us!

1.3. How Does It Work?

TODO

1.4. Who is Behind eTrice?

TODO

Chapter 2. Introduction to the ROOM Language

2.1. Scope of ROOM

2.2. Basic Concepts

2.3. Benefits

2.4. Execution Models

2.4.1. Logical Machine

- run to completion

2.4.2. Queue Based Execution Model

2.4.3. Polled Execution Model not implemented yet

2.4.4. Hybrid Execution Model not implemented yet

Chapter 3. Tutorial HelloWorld

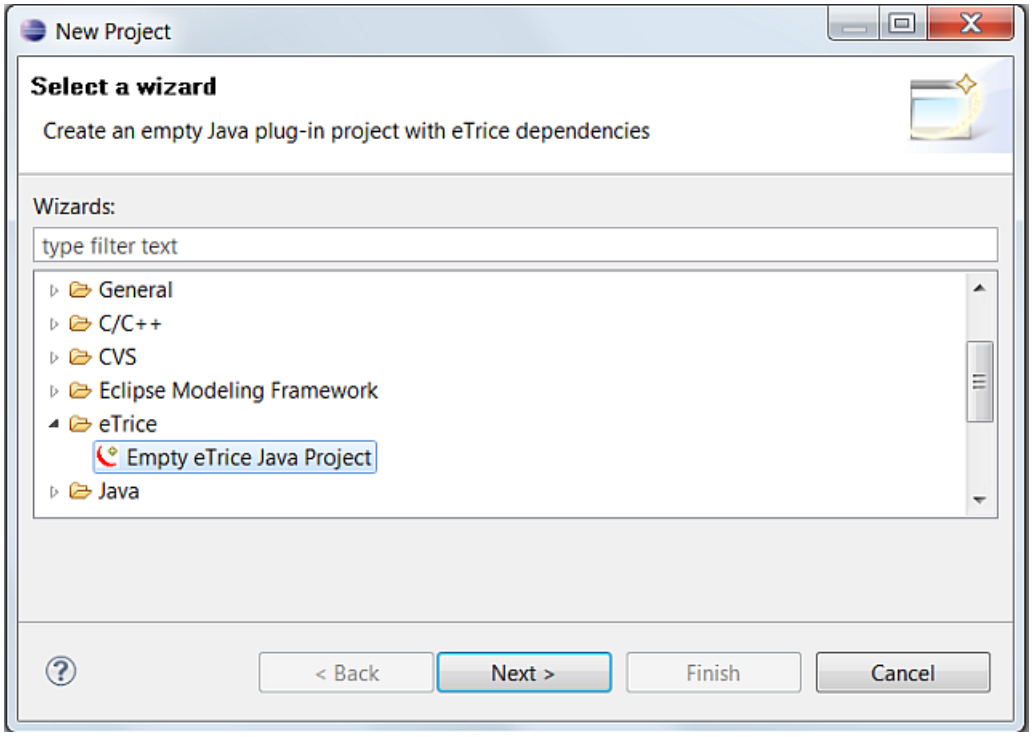
3.1. Scope

In this tutorial you will build your first very simple etrice model. The goal is to learn the work flow of eTrice and to understand a few basic features of ROOM. You will perform the following steps:

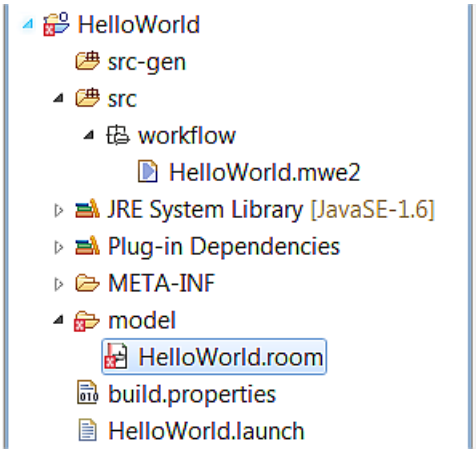
1. create a new model from scratch
2. add a very simple state machine to an actor
3. generate the source code
4. run the model
5. open the message sequence chart

3.2. Create a new model from scratch

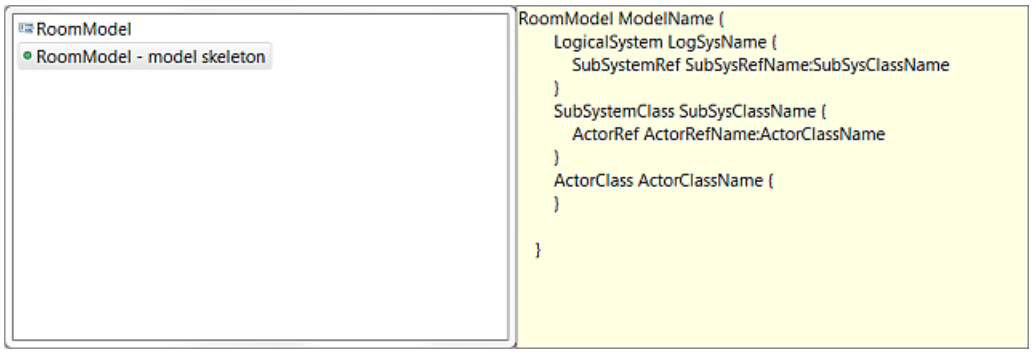
The easiest way to create a new eTrice Project is to use the eclipse project wizard. From the eclipse file menu select [File->New->Project] and create a new eTrice project and name it [HelloWorld]



The wizard creates everything that is needed to create, build and run a eTrice model. The resulting project should look like this:



Within the model directory the model file [HelloWorld.room] was created. Open the [HelloWorld.room] file and position the cursor at the very beginning of the file. Open the content assist with Ctrl+Space and select [model skeleton].



Edit the template variables and remove the artefacts from the wizard.

The resulting model code should look like this:

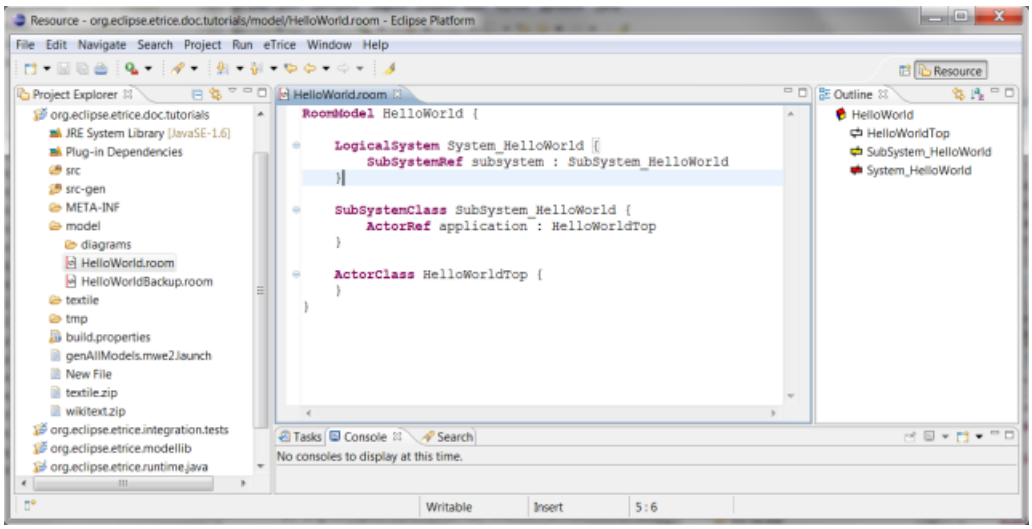
```
RoomModel HelloWorld {

 LogicalSystem System_HelloWorld {
  SubSystemRef subsystem : SubSystem_HelloWorld
 }

 SubSystemClass SubSystem_HelloWorld {
  ActorRef application : HelloWorldTop
 }

 ActorClass HelloWorldTop {
 }
}
```

The goal of eTrice is to describe distributed systems on a logical level. In the current version not all elements will be supported. But as prerequisite for further versions the following elements are mandatory for an eTrice model:

- the [LogicalSystem]
- at least one [SubSystemClass]
- at least one [ActorClass]

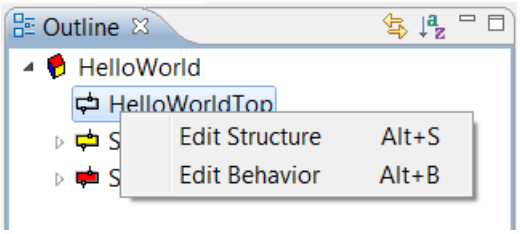
The [LogicalSystem] represents the complete distributed system and contains at least one [SubSystemRef]. The [SubSystemClass] represents an address space and contains at least one [ActorRef]. The [ActorClass] is the building block of which an application will be build of. It is a good idea to define a top level actor that can be used as reference within the subsystem.

Mention that a outline view was created that represents all currently existing model elements in a graphical way.



3.3. Create a state machine

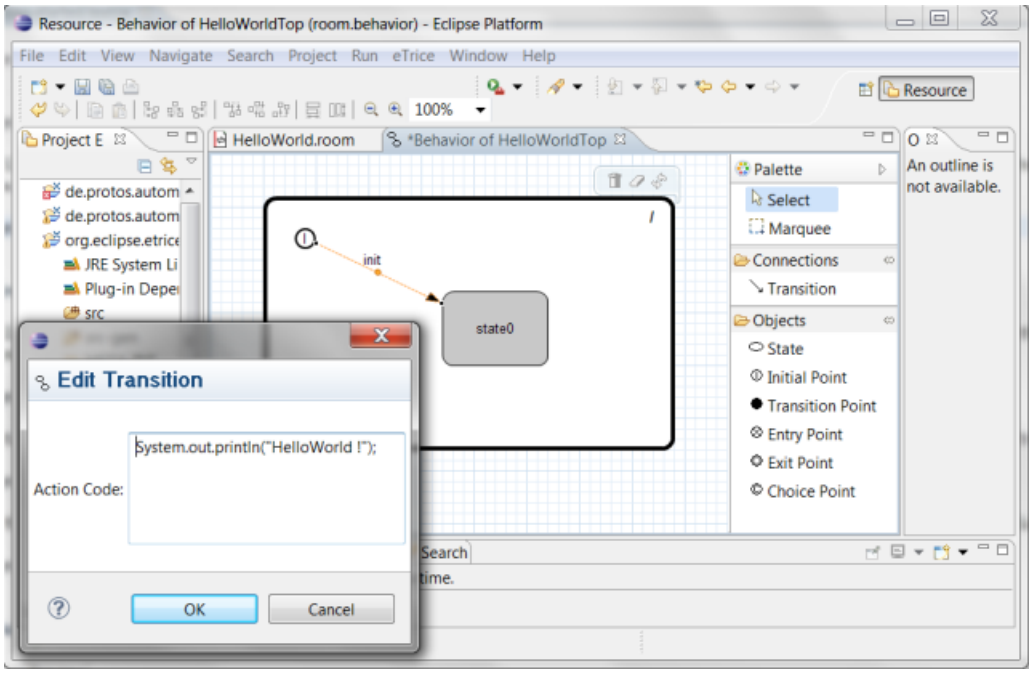
We will implement the Hello World code on the initial transition of the [HelloWorldTop] actor. Therefore open the state machine editor by right clicking the [HelloWorldTop] actor in the outline view and select [Edit Behavior].



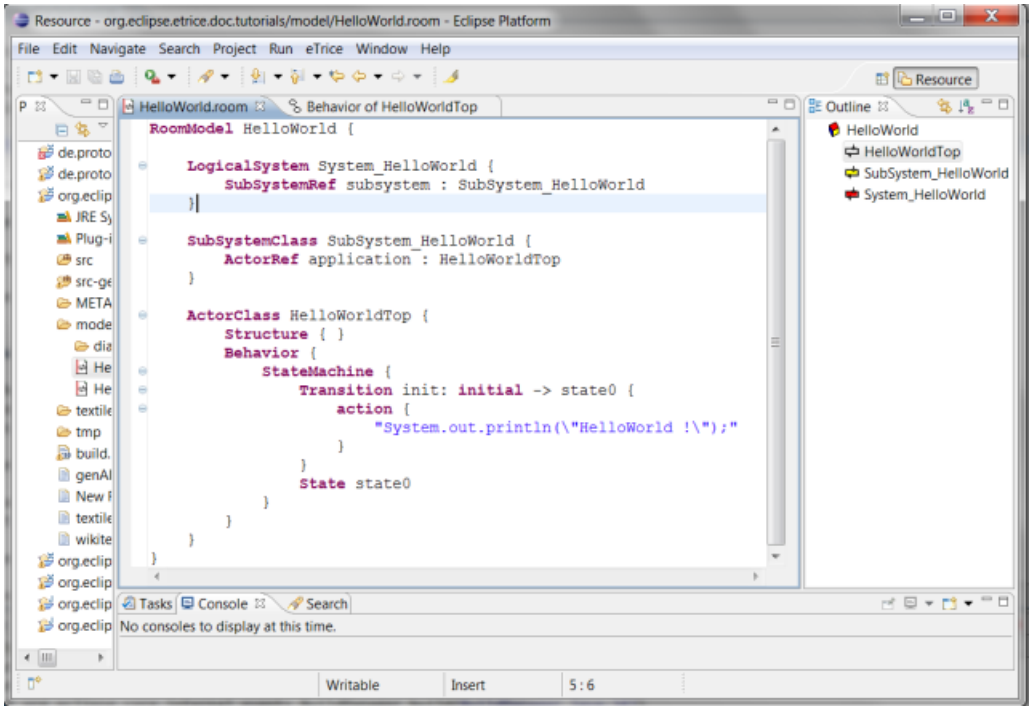
The state machine editor will be opened. Drag and drop an [Initial Point] from the tool box to the diagram into the top level state. Drag and drop a [State] from the tool box to the diagram. Confirm the dialogue with [ok]. Select the [Transition] in the tool box and draw the transition from the [Initial Point] to the State. Open the transition dialogue by double clicking the caption of the transition and fill in the action code.

```
System.out.println("Hello World !");
```

The result should look like this:

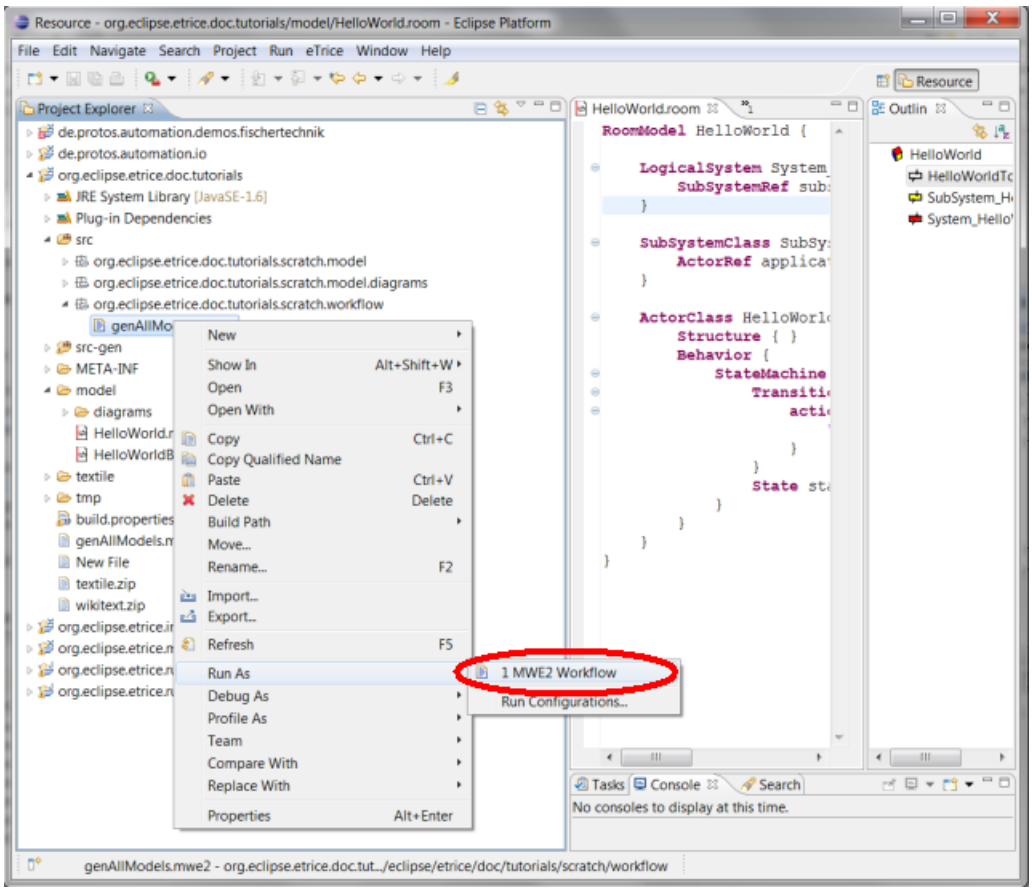


Save the diagram and inspect the model file. Note that the textual representation was created after saving the diagram.

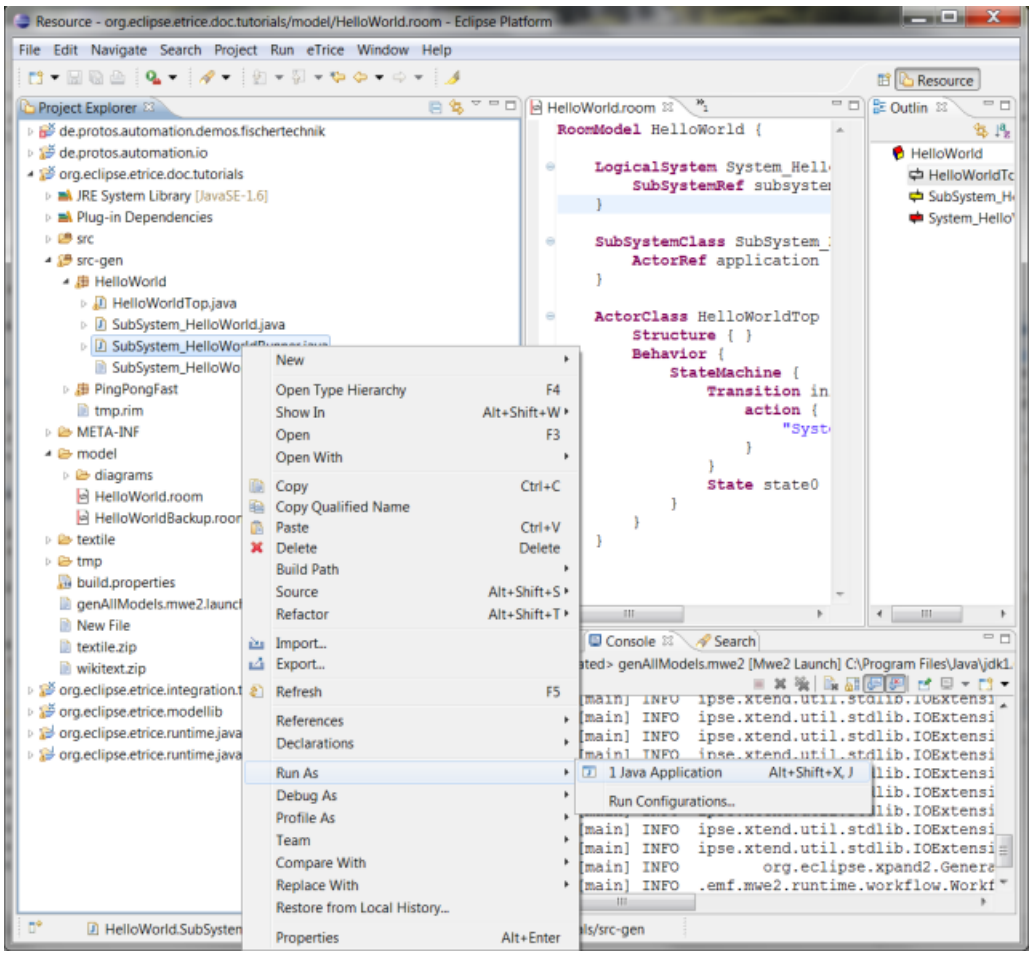


3.4. Build and run the model

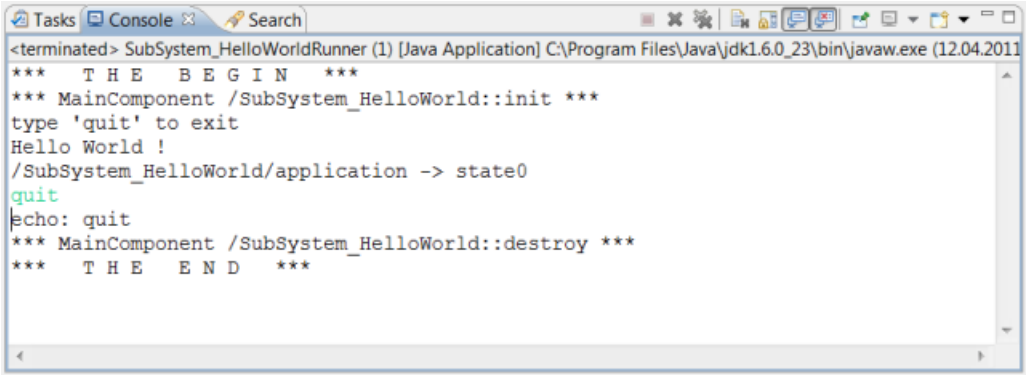
Now the model is finished and source code can be generated. The project wizard has created a workflow that is responsible to generate the source code. From [HelloWorld/src/workflow] right click [HelloWorld.mwe2] and run it as MWE2Workflow. All model files in the model directory will be generated.



The code will be generated to the src-gen directory. The main class will be contained in [SubSystem_HelloWorldRunner.java]. Select this file and run it as Java application.

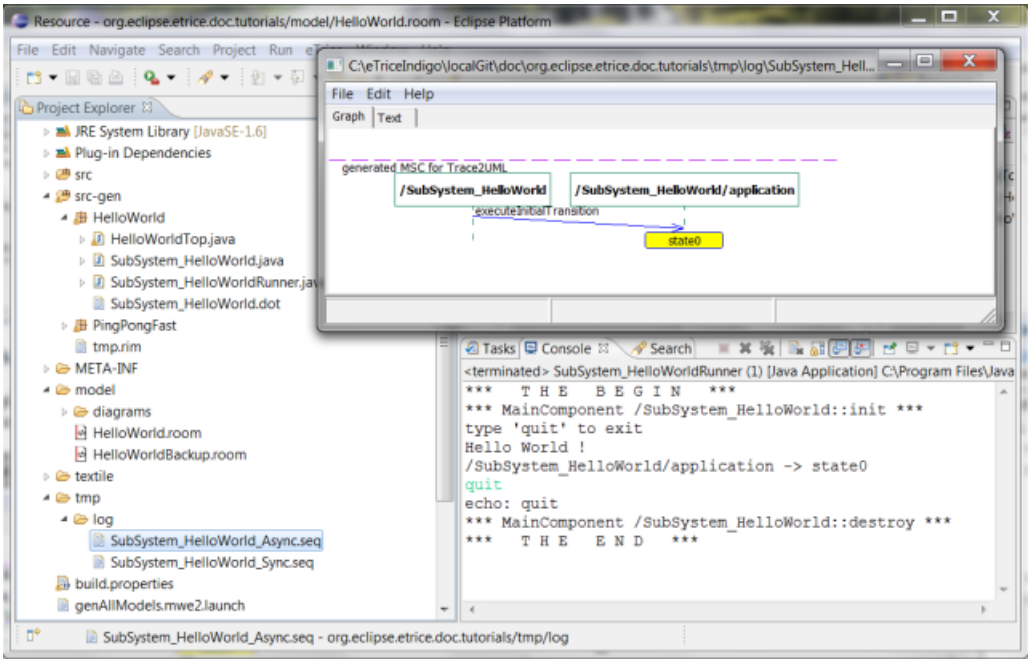


The Hello World application starts and the string will be printed on the console window. To stop the application the user must type [quit] in the console window.



3.5. Open the Message Sequence Chart

During runtime the application produces a MSC and wrote it to a file. Open / org.eclipse.etrice.doc.tutorials/tmp/log/SubSystem_HelloWorld_Async.seq. You should see something like this:



3.6. Summary

Now you have generated your first eTrice model from scratch. You can switch between diagram editor and model (.room file) and you can see what will be generated during editing and saving the diagram files. You should take a look at the generated source files to understand how the state machine is generated and the life cycle of the application. The next tutorials deals with more complex state machines hierarchies in structure and behavior.

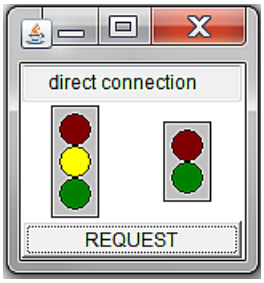
Chapter 4. Tutorial Blinky

4.1. Scope

This tutorial describes how to use the [TimingService], combine a generated model with manual code and how to modeling a hierarchical state machine. The idea of the tutorial is, to switch a LED on and off. The behavior of the LED should be: blinking in a one second interval for 5 seconds, stop blinking for 5 seconds, blinking, stop,... For this exercise we will use a little GUI class that will be used in more sophisticated tutorials too. The GUI simulates a pedestrian traffic crossing. For now, just a simple LED simulation will be used from the GUI.

To use the GUI please copy the package [de.protos.PedLightGUI] to your [src] directory. The package contains four java classes which implements a little window with a 3-light traffic light which simulates the signals for the car traffic and a 2-light traffic light which simulates the pedestrian signals.

The GUI looks like this:



Within this tutorial we just will switching on and off the yellow light.

You will perform the following steps:

1. create a new model from scratch
2. define a protocol
3. create an actor structure
4. create a hierarchical state machine
5. use the predefined [TimingService]
6. combine manual code with generated code
7. build and run the model
8. open the message sequence chart

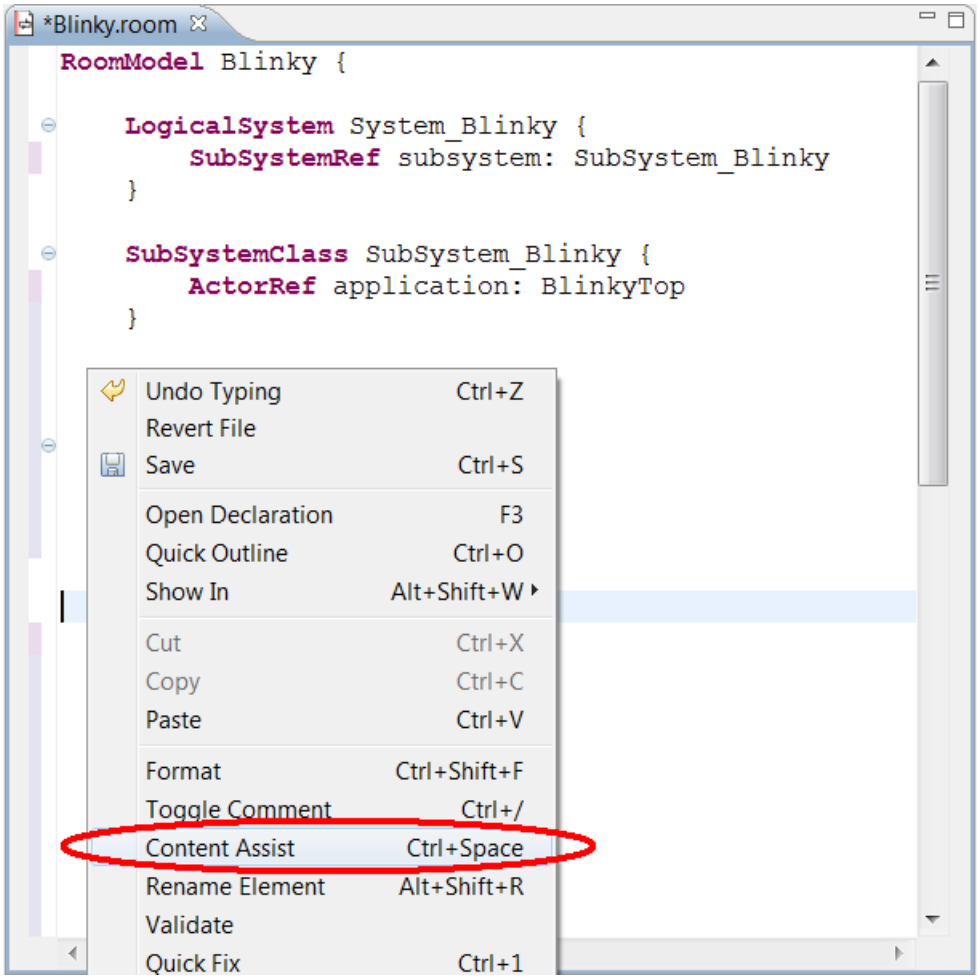
4.2. Create a new model from scratch

Remember exercise [HelloWorld]. Create a new eTrice project and name it [Blinky] Open the [Blinky.room] file and copy the following code into the file or use content assist to create the model.

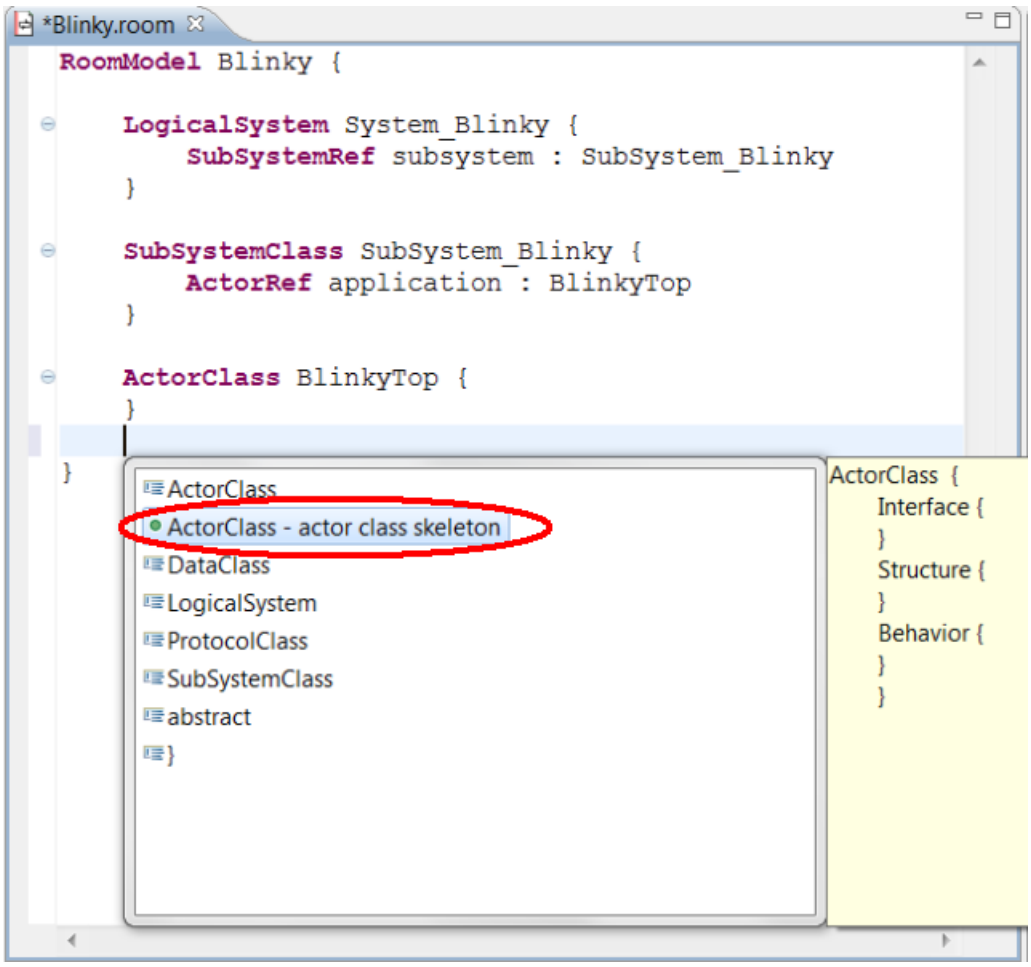
```
RoomModel Blinky {

 LogicalSystem System_Blinky {
  SubSystemRef subsystem : SubSystem_Blinky
 }

 SubSystemClass SubSystem_Blinky {
  ActorRef application : BlinkyTop
 }

 ActorClass BlinkyTop {
 }
}
```

4.3. Add two additional actor classes

Position the cursor outside any class definition and right click the mouse within the editor window. From the context menu select [Content Assist]



Select [ActorClass – actor class skeleton] and name it [Blinky].



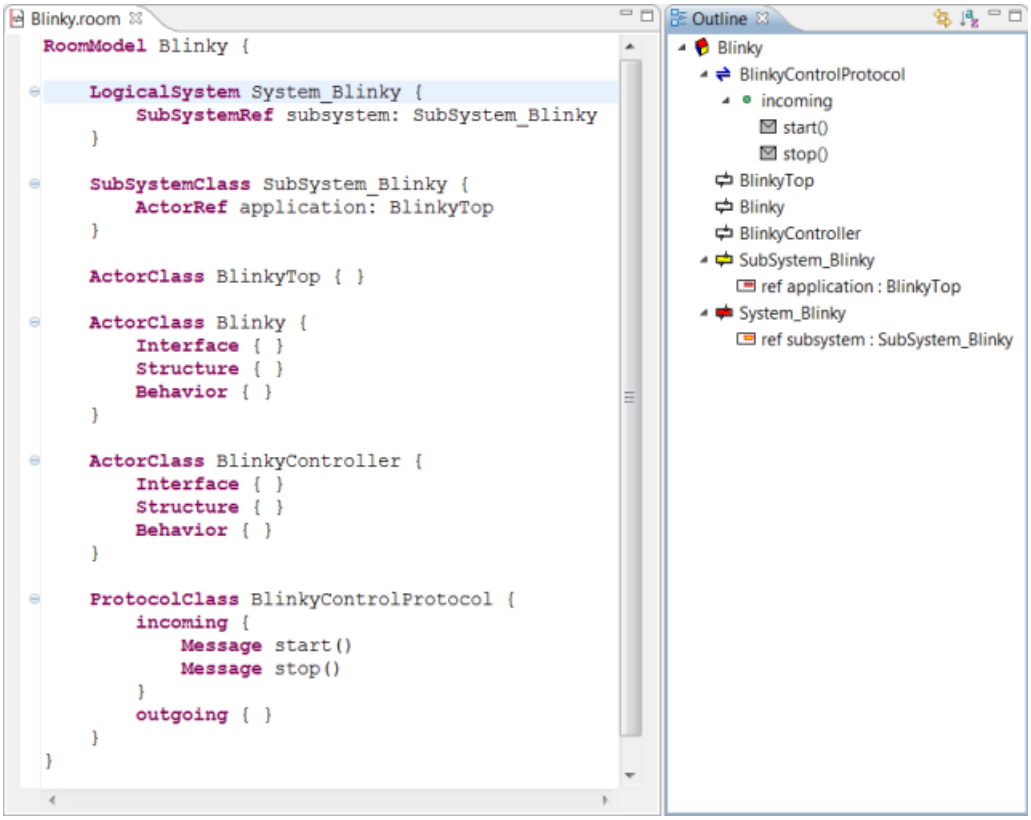
Repeat the described procedure and name the new actor [BlinkyController].

Save the model and visit the outline view.

4.4. Create a new protocol

With the help of [Content Assist] create a [ProtocolClass] and name it [BlinkyControlProtocol]. Inside the brackets use the [Content Assist] (CTRL+Space) to create two incomming messages called [start] and [stop].

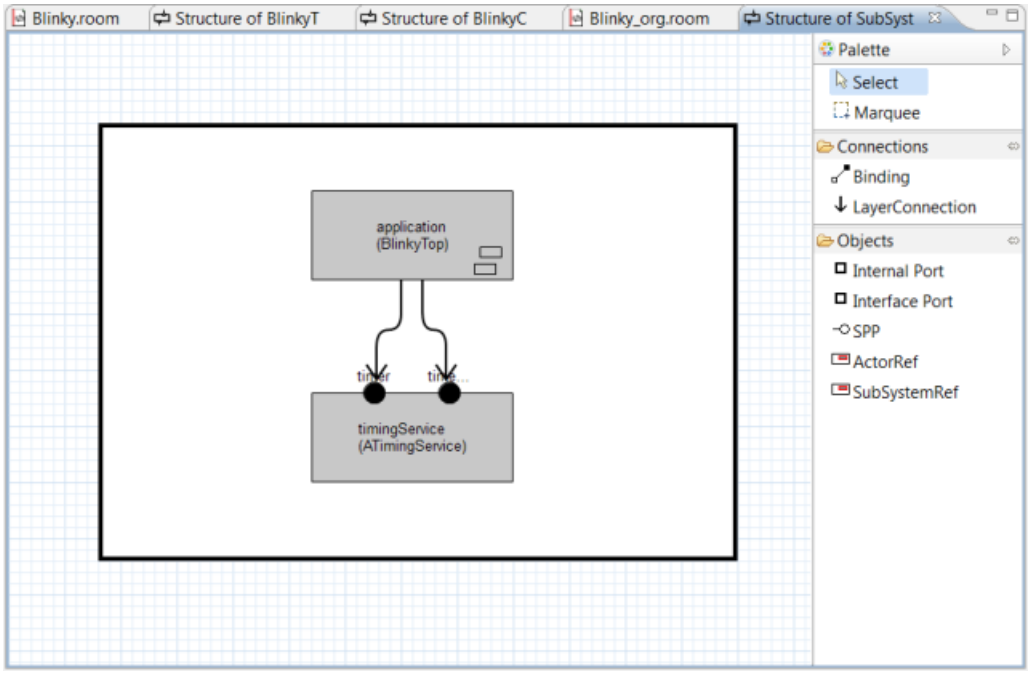
The resulting code should look like this:



With Ctrl-Shift+F or selecting [Format] from the context menu you can format the text. Mention that all elements are displayed in the outline view.

4.5. Import the Timing Service

Switching on and off the LED is timing controlled. Therefore a timing service is needed. To import the timing service in the outline view right click to [SubSystem_Blinky]. Select [Edit Structure]. Drag and Drop an [ActorRef] to the [SubSystem_Blinky] and name it [application]. From the actor class drop down list select [BlinkyTop]. Do the same clicks for the timing service. Name it [timingService] and from the drop down list select [room.basic.service.timing.ATimingService]. Draw a [LayerConnection] from [application] to each service provision point (SPP) of the [timingService]. The resulting structure should look like this:

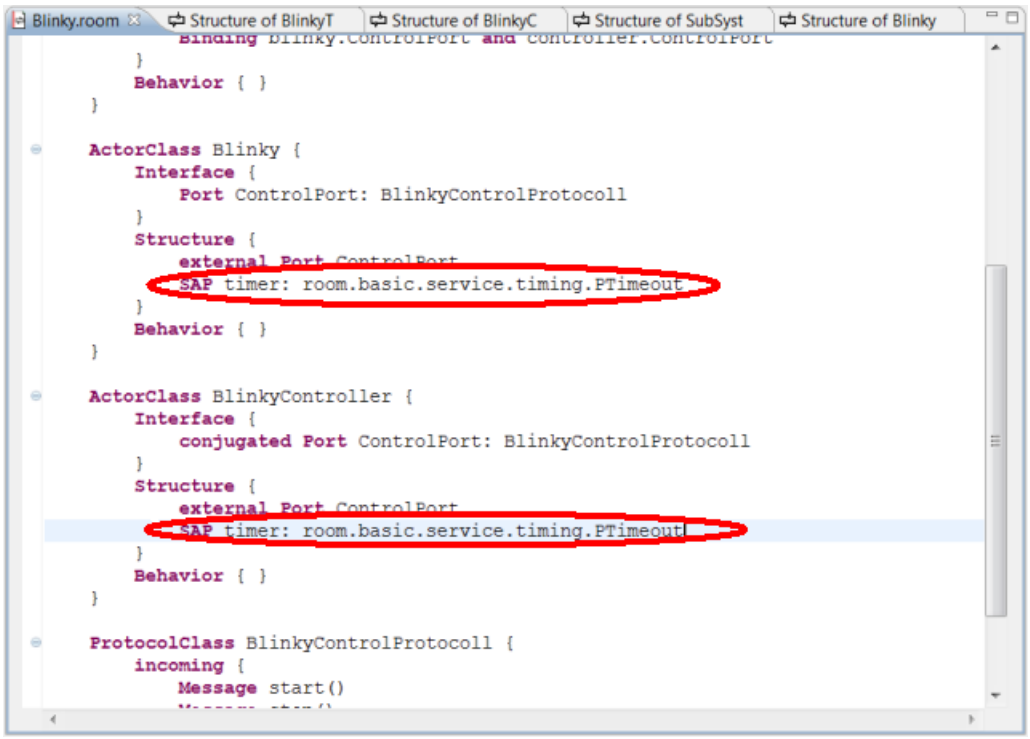


The current version of eTrice does not provide a graphical element for a service access point (SAP). Therefore the SAPs to access the timing service must be added in the .room file. Open the [Blinky.room] file and navigate to the [Blinky] actor. Add the following line to the structure of the actor:

```
SAP timer: room.basic.service.timing.PTimeout
```

Do the same thing for [BlinkyController].

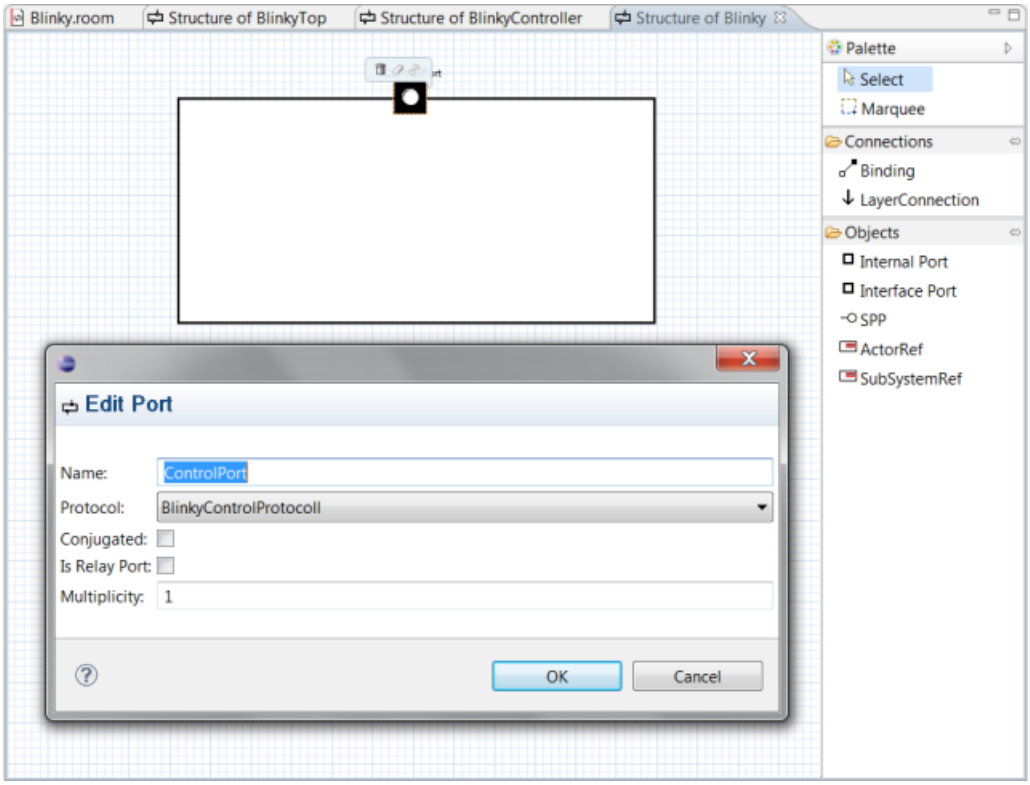
The resulting code should look like this:



4.6. Finish the model structure

From the outline view right click to [Blinky] and select [Edit Structure]. Drag and Drop an [Interface Port] to the boarder of the [Blinky] actor. Note that an interface port is not possible inside the the actor. Name

the port [ControlPort] and select [BlinkyControlProtocol] from the drop down list. Uncheck [Conjugated] and [Is Relay Port]. Klick [ok]. The resulting structure should look like this:

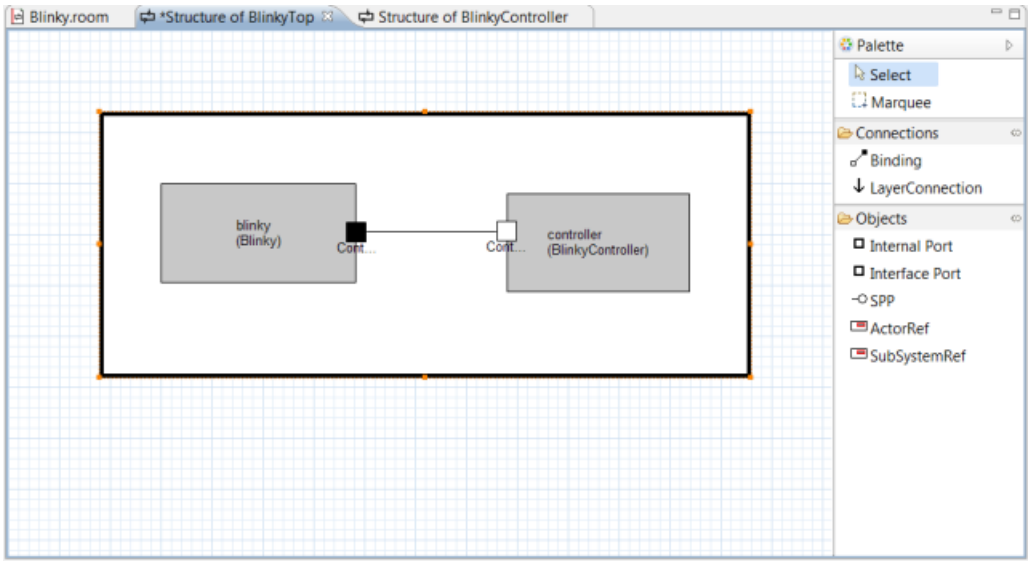


Repeat the above steps for the [BlinkyController]. Make the port [Conjugated]

Keep in mind that the protocol defines [start] and [stop] as incoming messages. [Blinky] receives this messages and therefore [Blinky]'s [ControlPort] must be a base port and [BlinkyController]'s [ControlPort] must be a conjugated port.

From the outline view right click [BlinkyTop] and select [Edit Structure].

Drag and Drop an [ActorRef] inside the [BlinkyTop] actor. Name it [blinky]. From the actor class drop down list select [Blinky]. Do the same for [controller]. Connect the ports via the binding tool. The resulting structure should look like this:



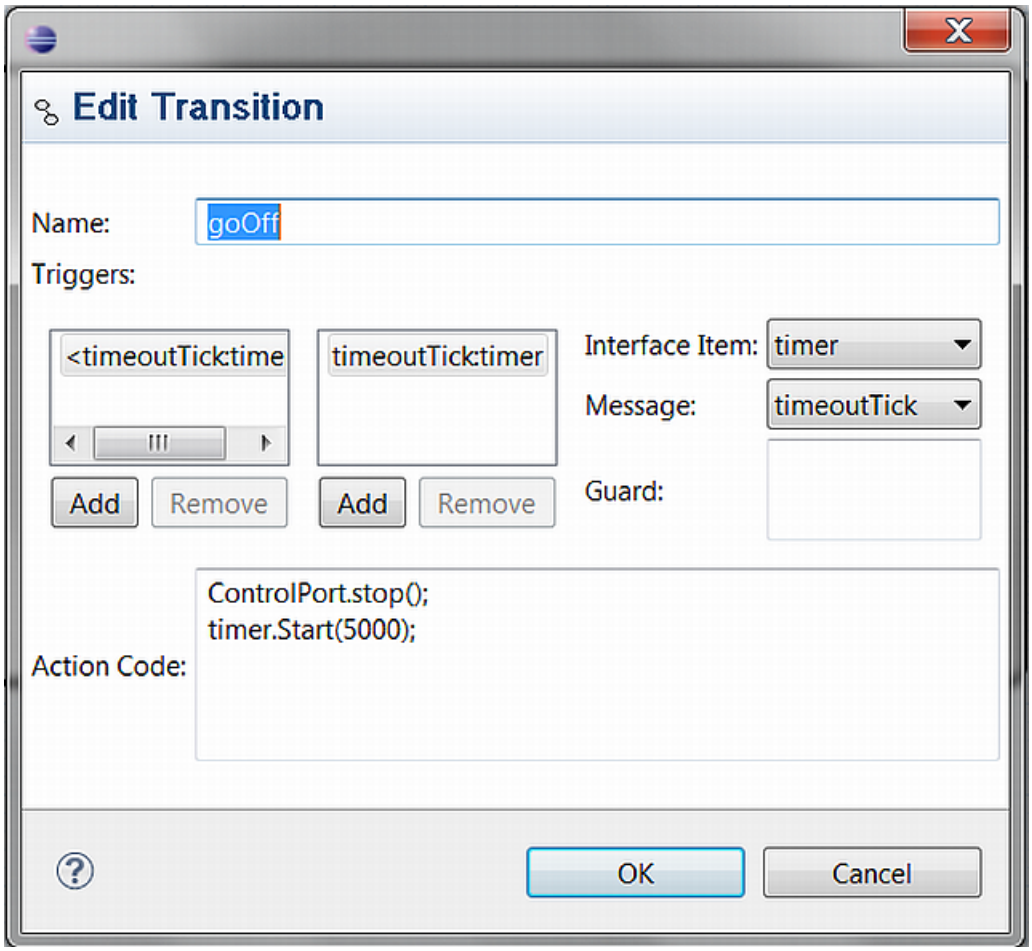
4.7. Implement the Behavior

The application should switch on and off the LED for 5 seconds in a 1 second interval, than stop blinking for 5 seconds and start again. To implement this behavior we will implement two FSMs. One for the 1 second interval and one for the 5 second interval. The 1 second blinking should be implemented in [Blinky]. The 5 second interval should be implemented in [BlinkyController]. First implement the Controller.

Right click to [BlinkyController] and select [Edit Behavior]. Drag and Drop the [Initial Point] and two [States] into the top state. Name the states [on] and [off]. Use the [Transition] tool to draw transitions from [init] to [off] from [on] to [off] and from [off] to [on].

Open the transition dialog by double click the arrow to specify the trigger event and the action code of each transition. Note that the initial transition does not have a trigger event.

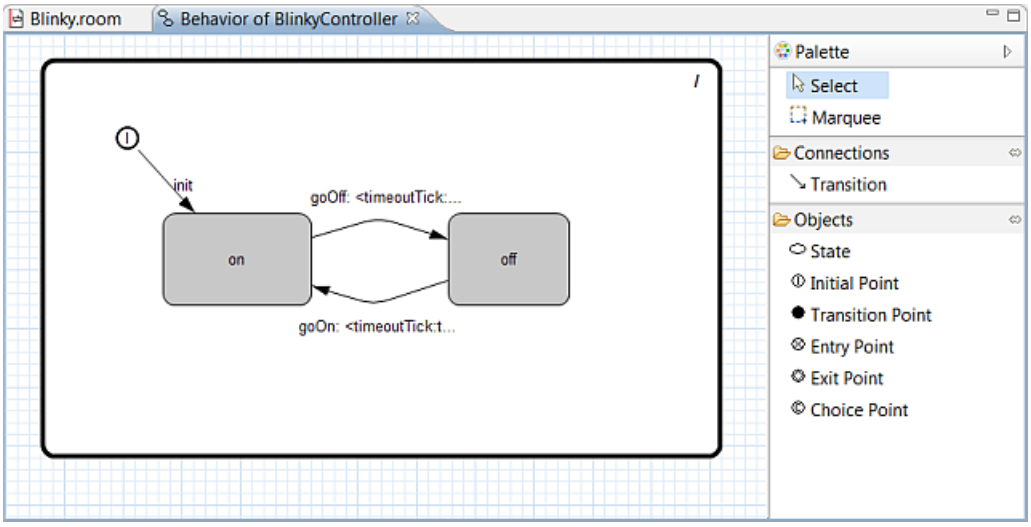
The dialog should look like this:



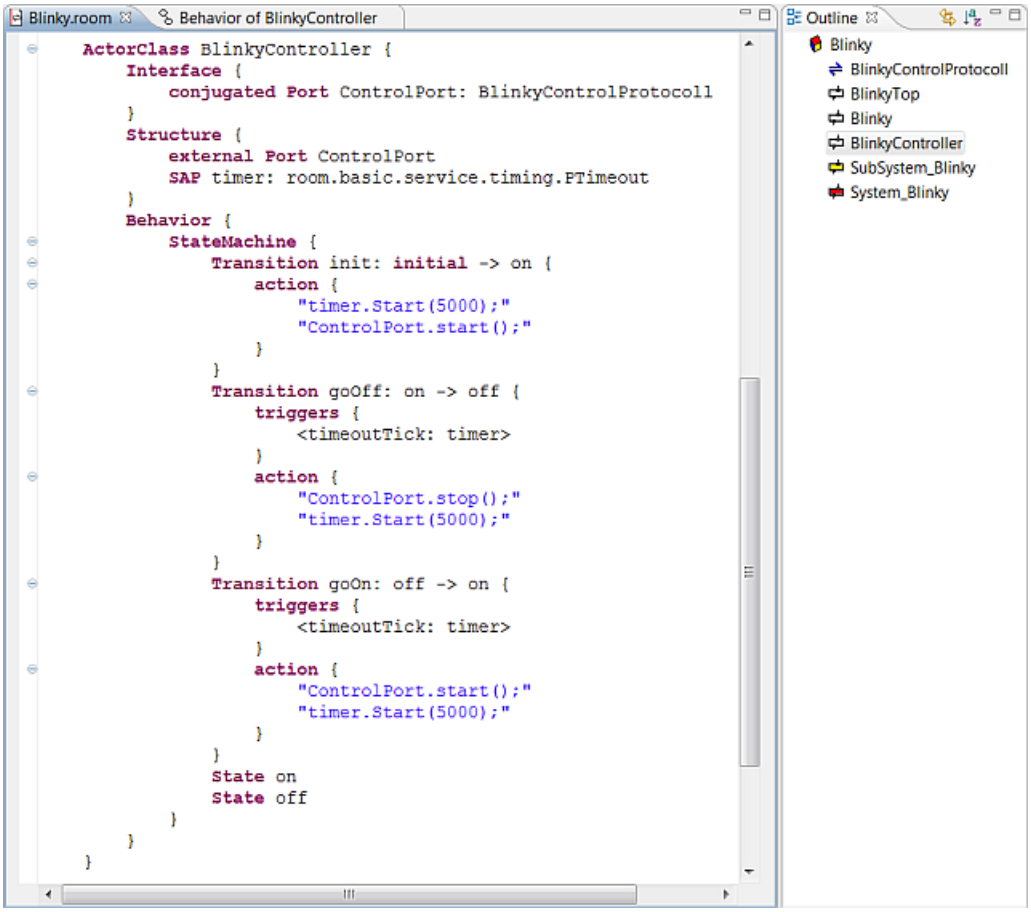
The defined ports will be generated as a member attribute of the actor class from type of the attached protocol. So, to send e message you must state [port.message(p1,p2);]. In this example [ControlPort.start()] sends the [start] message via the [ControlPort] to the outside world. Assuming that [Blinky] is connected to this port, the message will start the one second blinking FSM. It is the same thing with the [timer]. The SAP is also a port and follows the same rules. So it is clear that [timer.Start(5000);] will send the [Start] message to the timing service. The timing service will send a [timeoutTick] message back after 5000ms.

Within each transition the timer will be restarted and the appropriate message will be sent via the [ControlPort].

The resulting state machine should look like this:



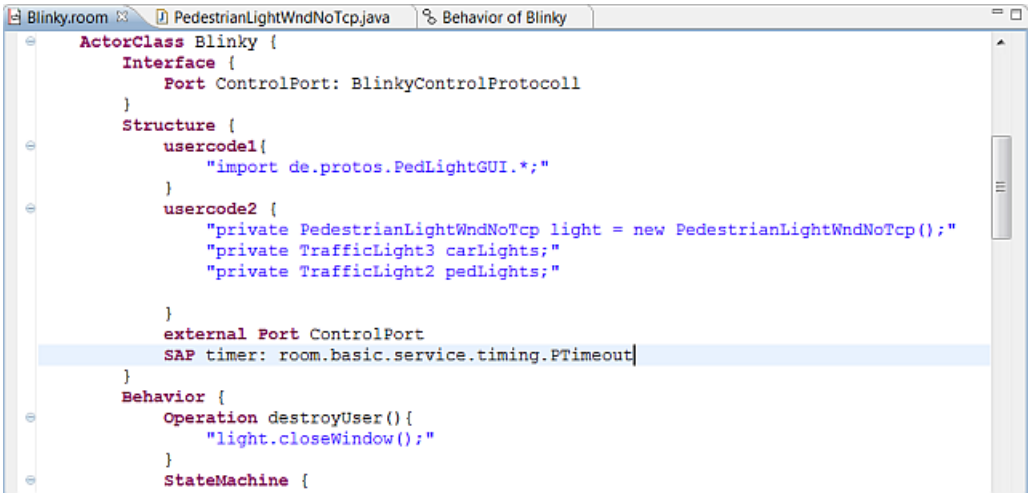
Save the diagram and inspect the [Blinky.room] file. The [BlinkyController] should look like this:




```
ActorClass BlinkyController {
    Interface {
        conjugated Port ControlPort: BlinkyControlProtocoll
    }
    Structure {
        external Port ControlPort
        SAP timer: room.basic.service.timing.PTimeout
    }
    Behavior {
        StateMachine {
            Transition init: initial -> on {
                action {
                    "timer.Start(5000);"
                    "ControlPort.start();"
                }
            }
            Transition goOff: on -> off {
                triggers {
                    <timeoutTick: timer>
                }
                action {
                    "ControlPort.stop();"
                    "timer.Start(5000);"
                }
            }
            Transition goOn: off -> on {
                triggers {
                    <timeoutTick: timer>
                }
                action {
                    "ControlPort.start();"
                    "timer.Start(5000);"
                }
            }
            State on
            State off
        }
    }
}
```

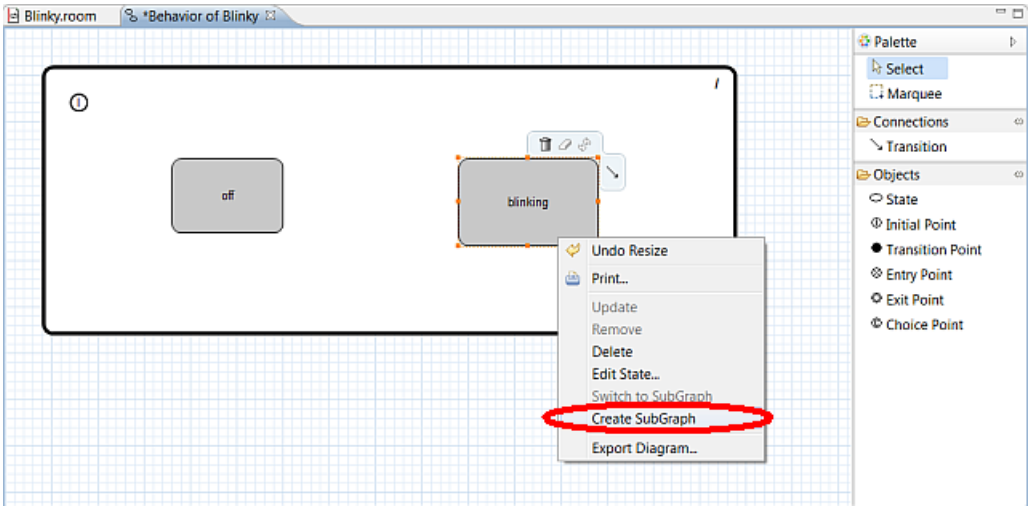
Now we will implement [Blinky]. Due to the fact that [Blinky] interacts with the GUI class a view things must to be done in the model file.

Double click [Blinky] in the outline view to navigate to [Blinky] within the model file. Add the following code:

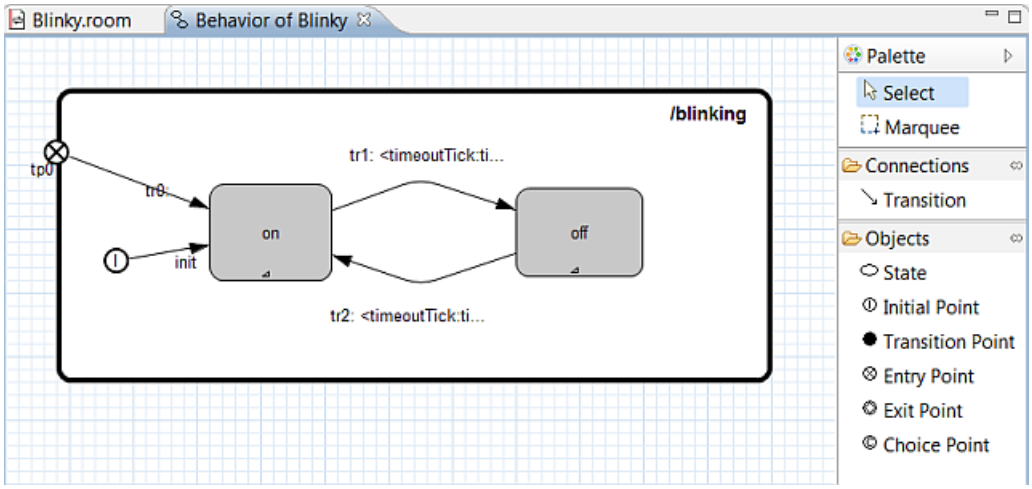


[usercode1] will be generated at the beginning of the file, outside the class definition. [usercode2] will be generated within the class definition. The code imports the GUI class and instantiates the window class. Attributes for the carLights and pedLights will be declared to easily access the lights in the state machine. The Operation [destroyUser()] is a predefined operation that will be called during shutdown of the application. Within this operation, cleanup of manual coded classes can be done.

Now design the FSM of [Blinky]. Open the behavior diagram of [Blinky] by right clicking the [Blinky] actor in the outline view. Create two states named [blinking] and [off]. Right click to [blinking] and create a subgraph.



Create the following state machine. The trigger events between [on] and [off] are the [timeoutTick] from the [timer] port.



Create entry code for both states by right clicking the state and select [Edit State...]

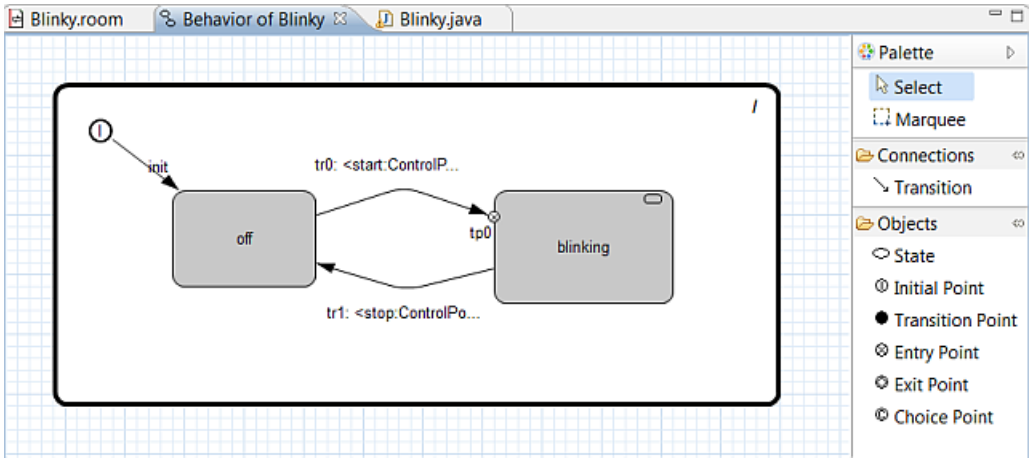
Entry code of [on] is:

```
timer.Start(1000);
carLights.setState(TrafficLight3.YELLOW);
```

Entry code of [off] is:

```
timer.Start(1000);
carLights.setState(TrafficLight3.OFF);
```

Navigate to the Top level state by double clicking the [/blinking] state. Create the following state machine:



The trigger event from [off] to [blinking] is the [start] event from the [ControlPort].The trigger event from [blinking] to [off] is the [stop] event from the [ControlPort].

Action code of the init transition is:

```
carLights = light.getCarLights();
pedLights = light.getPedLights();
carLights.setState(TrafficLight3.OFF);
pedLights.setState(TrafficLight2.OFF);
```

Action code from [blinking] to [off] is:

```
timer.Kill();
carLights.setState(TrafficLight3.OFF);
```

The complete resulting model looks like this:

```
RoomModel Blinky {

 LogicalSystem System_Blinky {
  SubSystemRef subsystem: SubSystem_Blinky
 }

 SubSystemClass SubSystem_Blinky {
  ActorRef application: BlinkyTop
  ActorRef timingService: room.basic.service.timing.ATimingService
  LayerConnection ref application satisfied_by timingService.timer
  LayerConnection ref application satisfied_by timingService.timeout
 }

 ActorClass BlinkyTop {
  Structure {
   ActorRef blinky: Blinky
   ActorRef controller: BlinkyController
   Binding blinky.ControlPort and controller.ControlPort
  }
  Behavior { }
 }

 ActorClass Blinky {
  Interface {
   Port ControlPort: BlinkyControlProtocoll
  }
  Structure {
   usercode1{
    "import de.protos.PedLightGUI.*;"
   }
   usercode2 {
    "private PedestrianLightWndNoTcp light = new PedestrianLightWndNoTcp();"
    "private TrafficLight3 carLights;"
    "private TrafficLight2 pedLights;"

   }
   external Port ControlPort
   SAP timer: room.basic.service.timing.PTimeout
  }
  Behavior {
   Operation destroyUser(){
    "light.closeWindow();"
   }
   StateMachine {
    Transition init: initial -> off {
     action {
      "carLights = light.getCarLights();"
      "pedLights = light.getPedLights();"
      "carLights.setState(TrafficLight3.OFF);"
      "pedLights.setState(TrafficLight2.OFF);"
     }
    }
    Transition tr0: off -> tp0 of blinking {
     triggers {
      <start: ControlPort>
     }
    }
    Transition tr1: blinking -> off {
     triggers {
      <stop: ControlPort>
     }
     action {
      "timer.Kill();"
      "carLights.setState(TrafficLight3.OFF);"
     }
    }
    State off
    State blinking {
     subgraph {
      Transition tr0: my tp0 -> on
      Transition tr1: on -> off {
```

The model is complete now. You can run and debug the model as described in getting started. Have fun.

4.8. Summary

Run the model and take look at the generated MSCs. Inspect the generated code to understand the runtime model of eTrice. Within this tutorial you have learned how to create a hierarchical FSM with group transitions and history transitions and you have used entry code. You are now familiar with the basic features of eTrice. The further tutorials will take this knowledge as a precondition.

Chapter 5. Tutorial Sending Data

5.1. Scope

This tutorial shows how data will be sent in a eTrice model. Within the example you will create two actors (MrPing and MrPong). MrPong will simply loop back every data it received. MrPing will sent data and verify the result.

You will perform the following steps:

1. create a new model from scratch
2. create a data class
3. define a protocol with attached data
4. create an actor structure
5. create two simple state machines
6. build and run the model

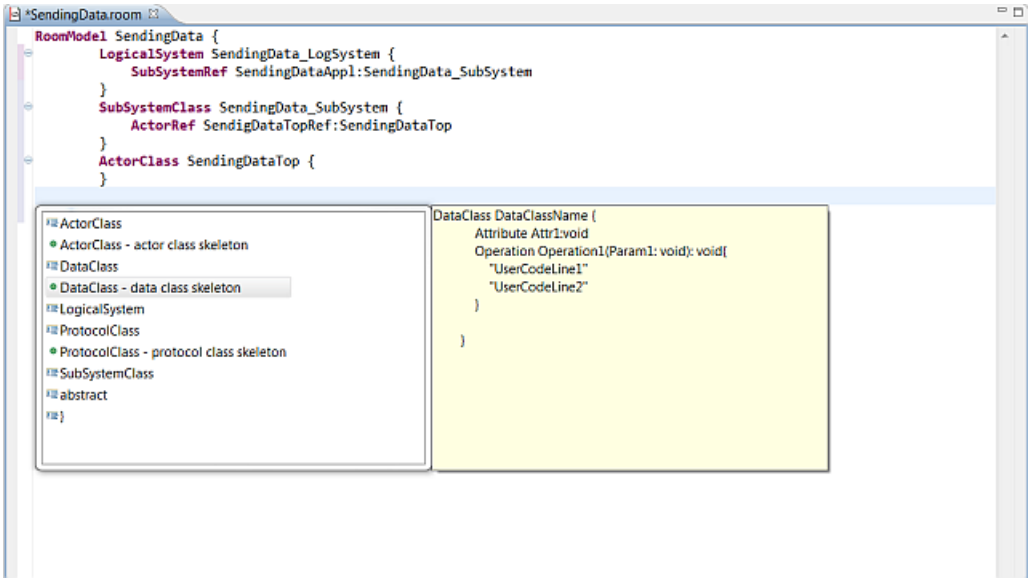
5.2. Create a new model from scratch

Remember exercise [HelloWorld]. Create a new eTrice project and name it [SendingData] Open the [SendingData.room] file and copy the following code into the file or use content assist to create the model.

```
RoomModel SendingData {
  LogicalSystem SendingData_LogSystem {
   SubSystemRef SendingDataAppl:SendingData_SubSystem
  }
  SubSystemClass SendingData_SubSystem {
   ActorRef SendigDataTopRef:SendingDataTop
  }
  ActorClass SendingDataTop {
  }
 }
```

5.3. Add a data class

Position the cursor outside any class definition and right click the mouse within the editor window. From the context menu select [Content Assist] (or Ctrl+Space).



Select [DataClass – data class skeleton] and name it [DemoData]. Remove the operations and ass the following Attributes:

```
DataClass DemoData {
 Attribute int32Val: int32 = "4711"
 Attribute int8Array [ 10 ]: int8 = "{1,2,3,4,5,6,7,8,9,10}"
 Attribute float64Val: float64 = "0.0"
 Attribute stringVal: string = "\"empty\""
}
```

Save the model and visit the outline view. Note that the outline view contains all data elements as defined in the model.

5.4. Create a new protocol

With the help of [Content Assist] create a [ProtocolClass] and name it [PingPongProtocol]. Create the following messages:

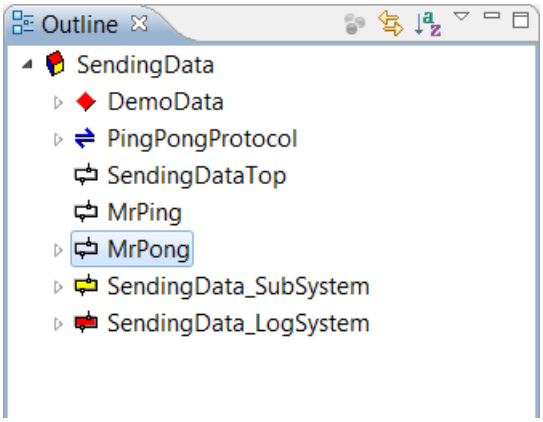
```
ProtocolClass PingPongProtocol {
     incoming {
      Message ping(data: DemoData)
      Message pingSimple(data:int32)
     }
     outgoing {
      Message pong(data: DemoData)
      Message pongSimple(data:int32)
     }
    }
```

5.5. Create MrPing and MrPong Actors

With the help of [Content Assist] create two new actor classes and name them [MrPing] and [MrPong]. The resulting model should look like this:

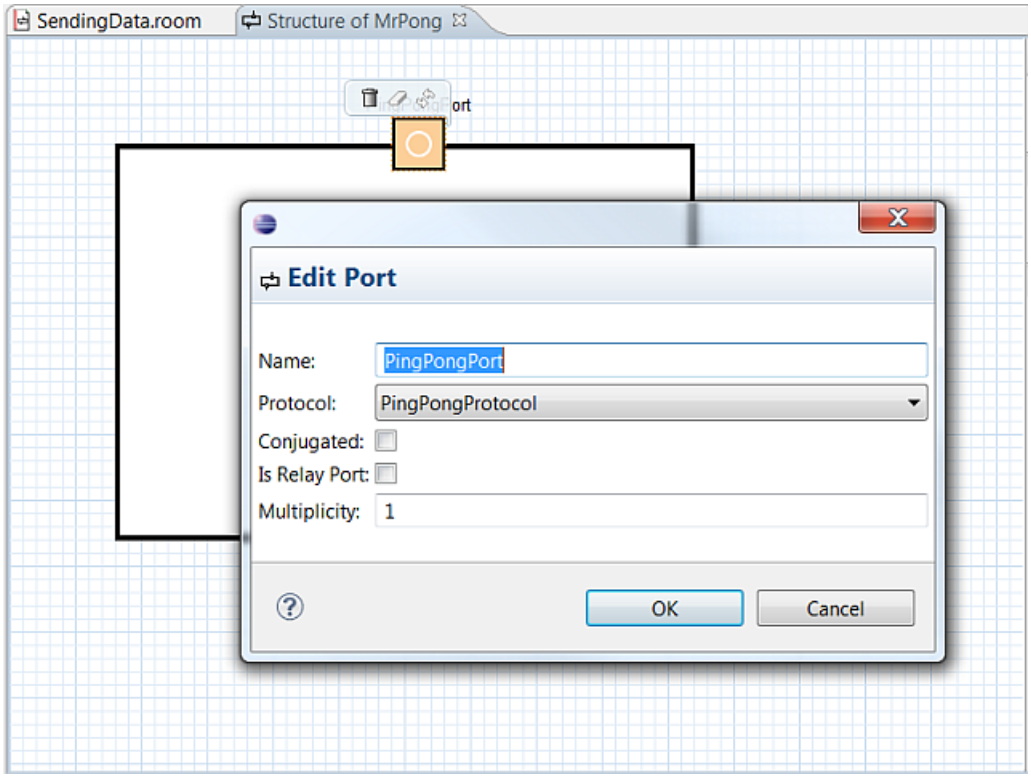
```
RoomModel SendingData {

 LogicalSystem SendingData_LogSystem {
  SubSystemRef SendingDataAppl: SendingData_SubSystem
 }

 SubSystemClass SendingData_SubSystem {
  ActorRef SendigDataTopRef: SendingDataTop
 }

 ActorClass SendingDataTop { }

 DataClass DemoData {
  Attribute int32Val: int32 = "4711"
  Attribute int8Array [ 10 ]: int8 = "{1,2,3,4,5,6,7,8,9,10}"
  Attribute float64Val: float64 = "0.0"
  Attribute stringVal: string = "\"empty\""
 }

 ProtocolClass PingPongProtocol {
  incoming {
   Message ping(data: DemoData)
   Message pingSimple(data: int32)
  }
  outgoing {
   Message pong(data: DemoData)
   Message pongSimple(data: int32)
  }
 }

 ActorClass MrPing {
  Interface { }
  Structure { }
  Behavior { }
 }

 ActorClass MrPong {
  Interface { }
  Structure { }
  Behavior { }
 }
}
```

The outline view should look like this:



5.6. Define the Actors Structure and Behavior

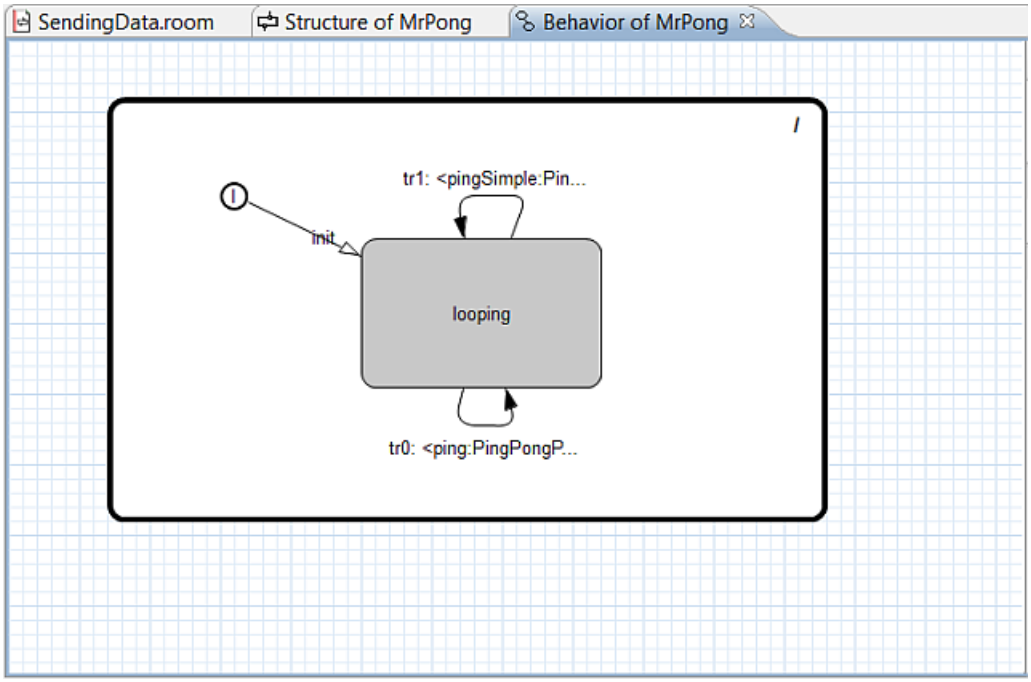
Save the model and visit the outline view. Within the outline view, right click on the [MrPong] actor and select [Edit Structure]. Select an [Interface Port] from the toolbox and add it to MrPong. Name the Port [PingPongPort] and select the [PingPongProtocol]



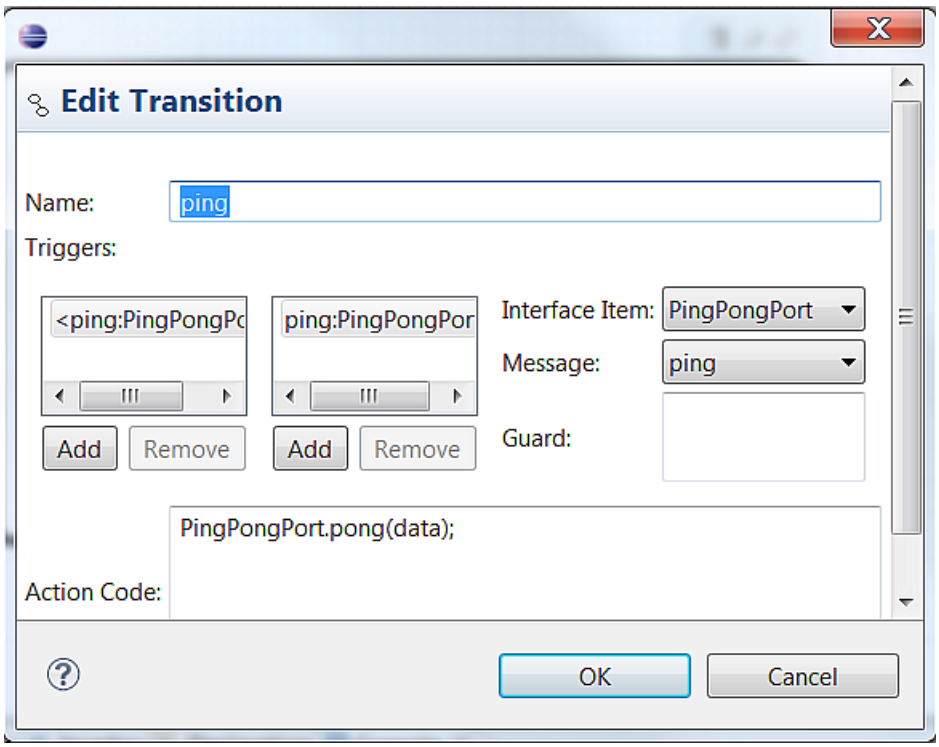
Do the same with MrPing but mark the port as [conjugated]

5.6.1. Define MrPongs behavior

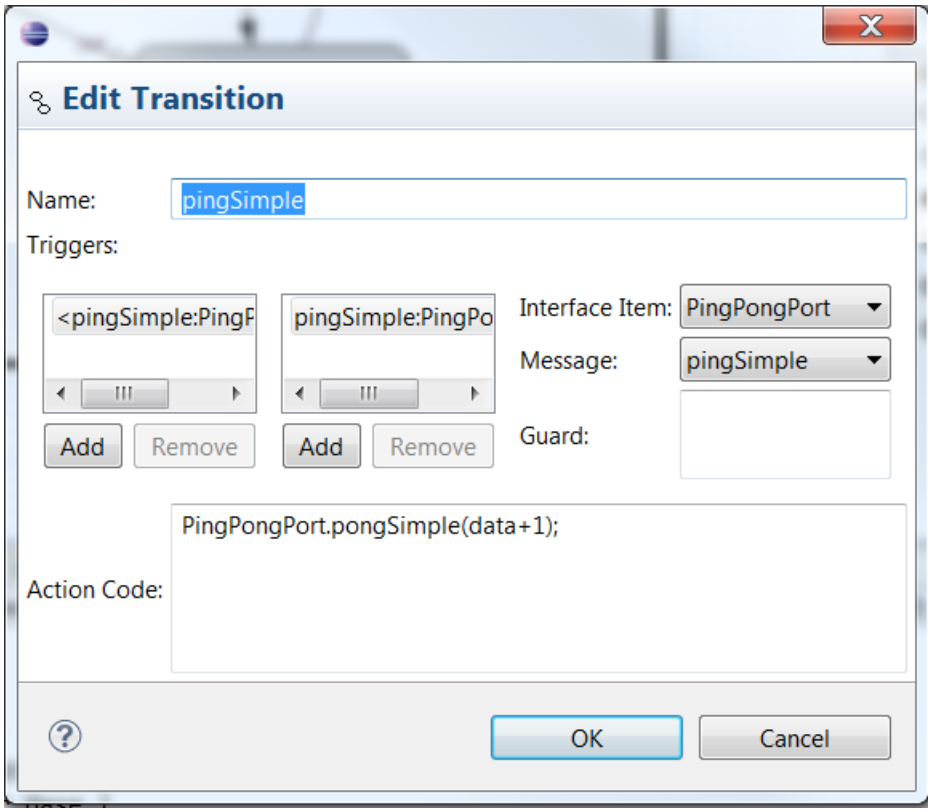
Within the outline view, right click MrPong and select [Edit Behavior]. Create the following state machine:



The transition dialogues should look like this: For [ping]:

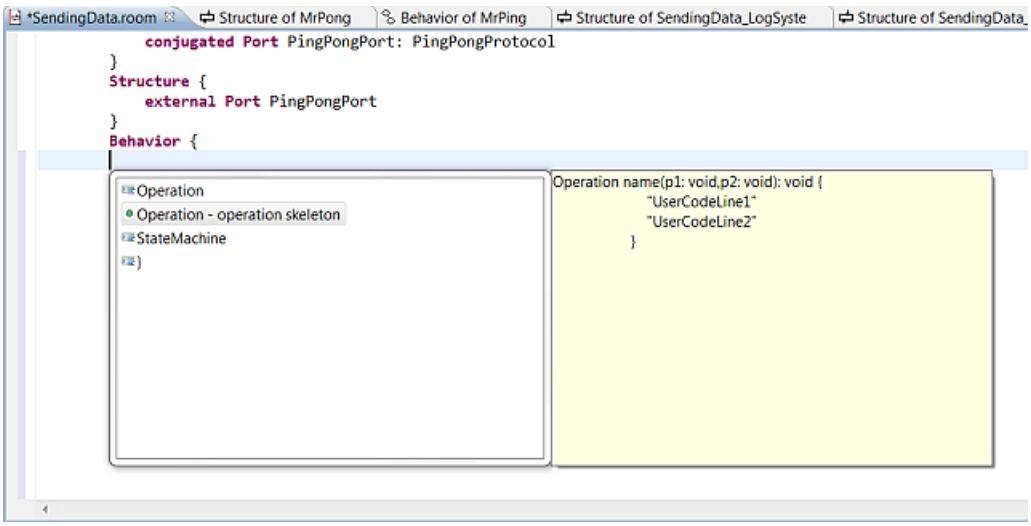


For [pingSimple]:



5.6.2. Define MrPing behavior

Within the outline view double click MrPing. Navigate the cursor to the behavior of MrPing. With the help of content assist create a new operation.

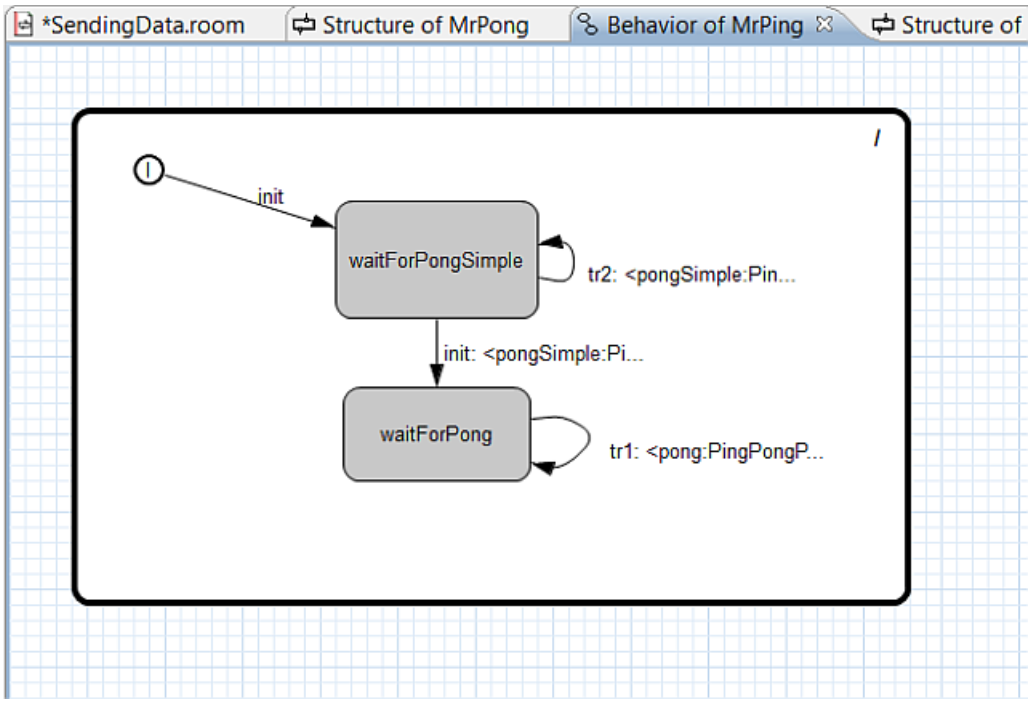


Name the operation [printData] and define the DemoData as a parameter.

Fill in the following code:

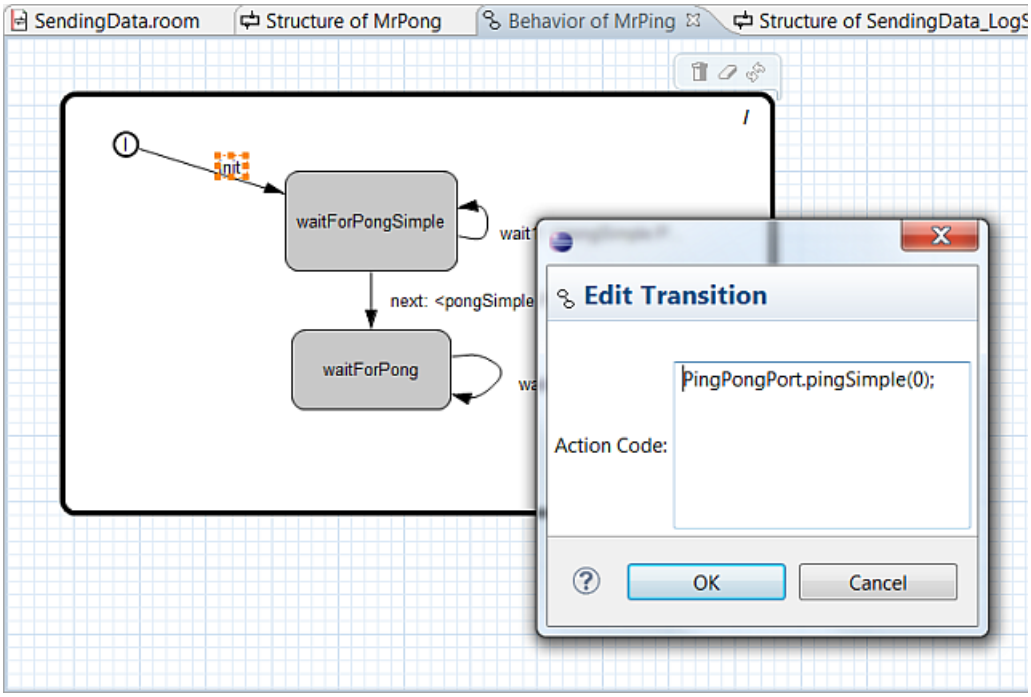
```
Operation printData(d: DemoData) : void {
   "System.out.printf(\"d.int32Val: %d\\n\",d.int32Val);"
   "System.out.printf(\"d.float64Val: %f\\n\",d.float64Val);"
   "System.out.printf(\"d.int8Array: \");"
   "for(int i = 0; i<d.int8Array.length; i++) {"
   "System.out.printf(\"%d \",d.int8Array[i]);}"
   "System.out.printf(\"\\nd.stringVal: %s\\n\",d.stringVal);"
  }
```

For MrPing create the following state machine:

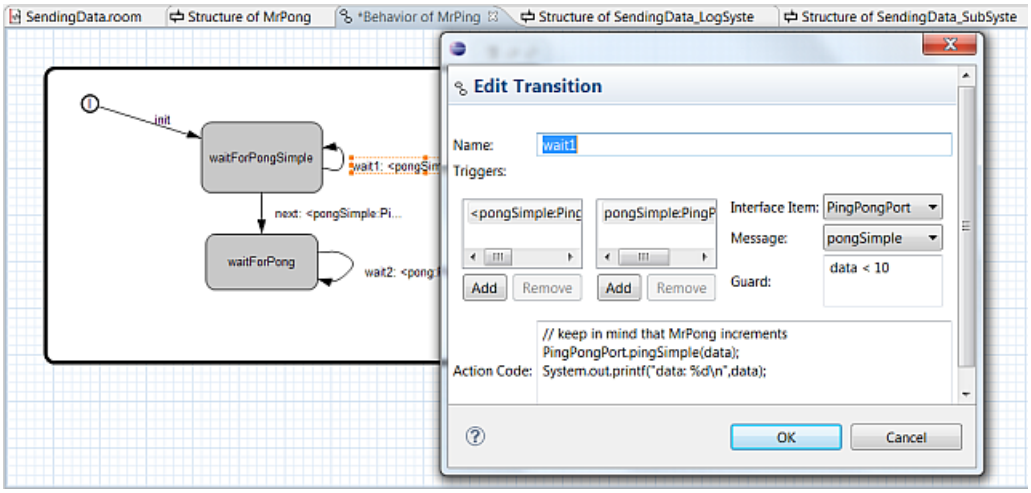


The transition dialogues should look like this:

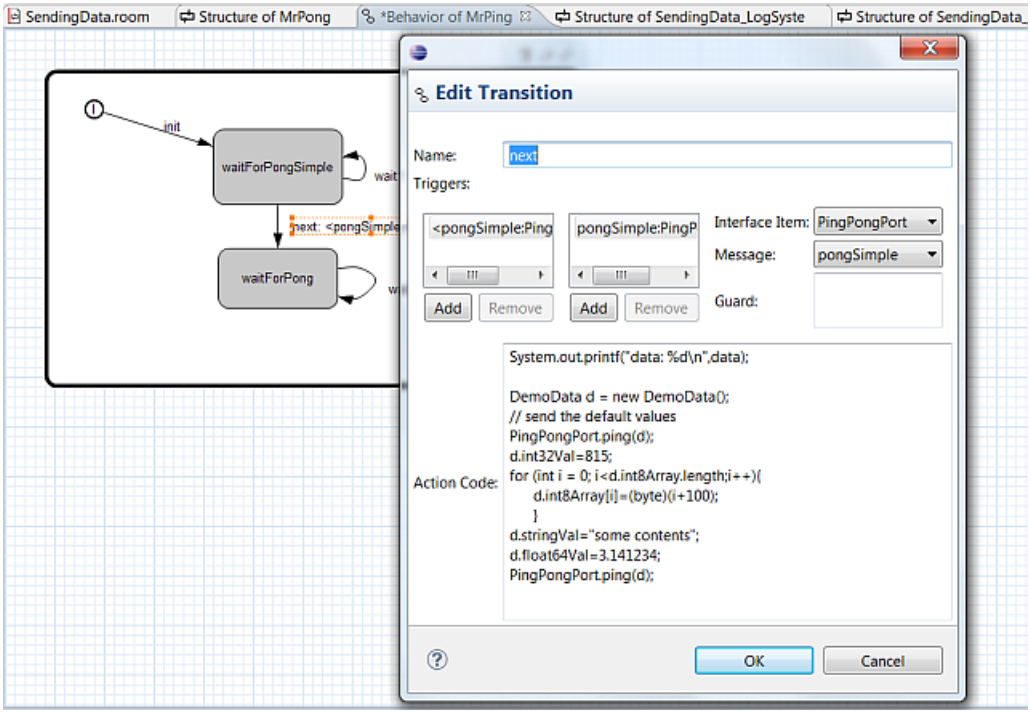
For [init]:



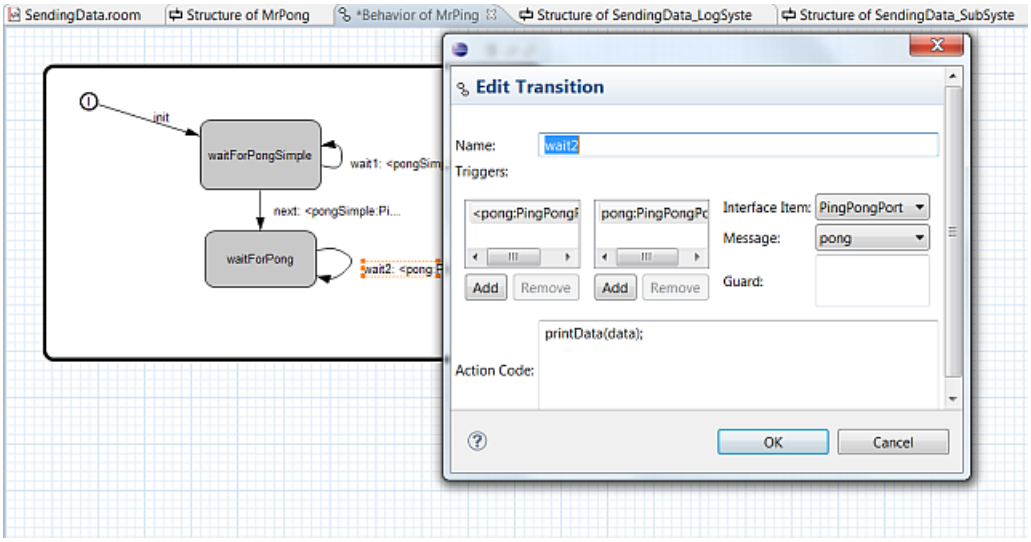
For [wait1]:



For [next]:

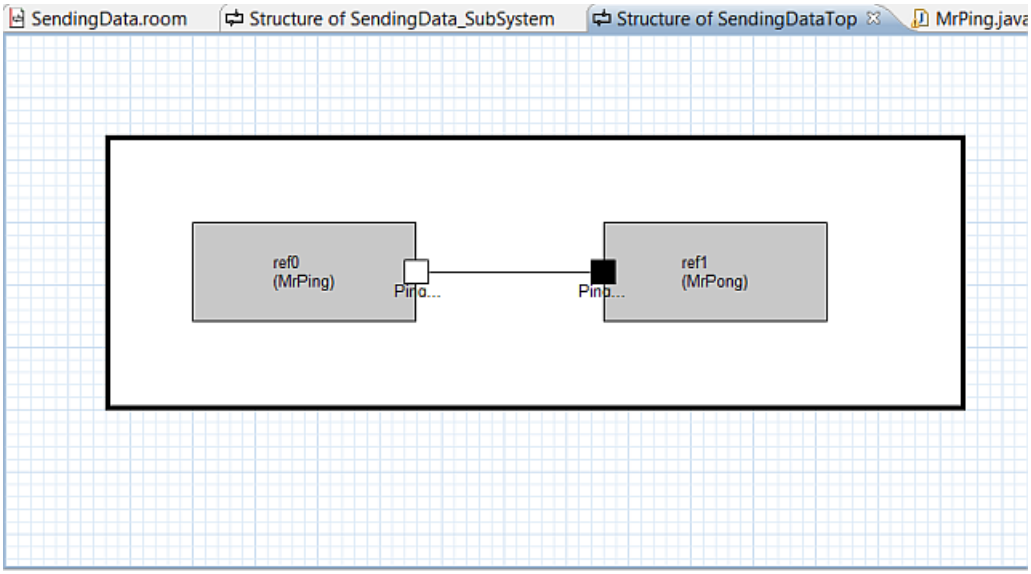


For [wait2]:



5.7. Define the top level

Open the Structure from SendingDataTop and add MrPing and MrPong as a reference. Connect the ports.



The model is finished now and the model file should look like this:

```
RoomModel SendingData {

 LogicalSystem SendingData_LogSystem {
  SubSystemRef SendingDataAppl: SendingData_SubSystem
 }

 SubSystemClass SendingData_SubSystem {
  ActorRef SendigDataTopRef: SendingDataTop
 }

 ActorClass SendingDataTop {
  Structure {
   ActorRef ref0: MrPing
   ActorRef ref1: MrPong
   Binding ref0.PingPongPort and ref1.PingPongPort
  }
  Behavior { }
 }

 ActorClass MrPing {
  Interface {
   conjugated Port PingPongPort: PingPongProtocol
  }
  Structure {
   external Port PingPongPort
  }
  Behavior {

   Operation printData(d: DemoData) : void {
      "System.out.printf(\"d.int32Val: %d\\n\",d.int32Val);"
      "System.out.printf(\"d.float64Val: %f\\n\",d.float64Val);"
      "System.out.printf(\"d.int8Array: \");"
      "for(int i = 0; i<d.int8Array.length; i++) {"
      "System.out.printf(\"%d \",d.int8Array[i]);}"
      "System.out.printf(\"\\nd.stringVal: %s\\n\",d.stringVal);"
     }

   StateMachine {
    Transition wait2: waitForPong -> waitForPong {
     triggers {
      <pong: PingPongPort>
     }
     action {
      "printData(data);"
     }
    }
    Transition wait1: waitForPongSimple -> waitForPongSimple {
     triggers {
      <pongSimple: PingPongPort guard {
       "data < 10"
      }>
     }
     action {
      "// keep in mind that MrPong increments"
      "PingPongPort.pingSimple(data);"
      "System.out.printf(\"data: %d\\n\",data);"
     }
    }
    Transition next: waitForPongSimple -> waitForPong {
     triggers {
      <pongSimple: PingPongPort>
     }
     action {
      "System.out.printf(\"data: %d\\n\",data);"
      ""
      "DemoData d = new DemoData();"
      "// send the default values"
      "PingPongPort.ping(d);"
      "d.int32Val=815;"
      "for (int i = 0; i<d.int8Array.length;i++){"
      "\td.int8Array[i]=(byte)(i+100);"
      "\t}"
```

5.8. Generate and run the model

With the MWe2 workflow generate the code and run the model. The output should look like this:

type ‚quit' to exit /SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple / SendingData_SubSystem/SendigDataTopRef/ref1 -> looping /SendingData_SubSystem/ SendigDataTopRef/ref1 -> looping data: 1 /SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/SendigDataTopRef/ref1 -> looping data: 2 / SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/ SendigDataTopRef/ref1 -> looping data: 3 /SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/SendigDataTopRef/ref1 -> looping data: 4 / SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/ SendigDataTopRef/ref1 -> looping data: 5 /SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/SendigDataTopRef/ref1 -> looping data: 6 / SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/ SendigDataTopRef/ref1 -> looping data: 7 /SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/SendigDataTopRef/ref1 -> looping data: 8 / SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/ SendigDataTopRef/ref1 -> looping data: 9 /SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPongSimple /SendingData_SubSystem/SendigDataTopRef/ref1 -> looping data: 10 /SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPong /SendingData_SubSystem/ SendigDataTopRef/ref1 -> looping /SendingData_SubSystem/SendigDataTopRef/ref1 -> looping d.int32Val: 4711 d.float64Val: 0,000000 d.int8Array: 1 2 3 4 5 6 7 8 9 10 d.stringVal: empty /SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPong d.int32Val: 815 d.float64Val: 3,141234 d.int8Array: 100 101 102 103 104 105 106 107 108 109 d.stringVal: some contents / SendingData_SubSystem/SendigDataTopRef/ref0 -> waitForPong quit echo: quit

5.9. Summary

Within the first loop a integer value will be incremented from [MrPong] and sent back to [MrPing]. Is long as the guard is true [MrPing] sends back the value.

Within the [next] transition, [MrPing] creates a data class and sends the default values. Than [MrPing] changes the values and sends the class again. At this point you should note that during the send operation, a copy of the data class will be created and sent. Otherwise it would not be possible to send the same object two times, even more it would not be possible to send a stack object at all. In later versions of eTrice a additional mechanism to send references will be implemented. However, keep in mind that sending references takes the responsibility of the life cycle of the sent object to the user. It looks simple but is a very common source of failures.

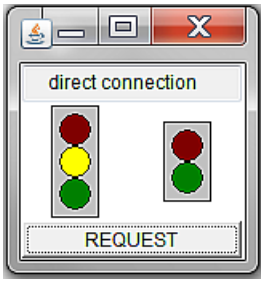
Chapter 6. Tutorial Pedestrian Lights

6.1. Scope

The scope of this tutorial is to demonstrate how to receive model messages from outside the model. Calling methods which are not part of the model is simple and you have already done this within the blinky tutorial (this is the other way round model => external code). Receiving events from outside the model is a very common problem and a very frequently asked question. Therefore this tutorial shows how an external event (outside the model) can be received from the model.

This tutorial is not like hello world or blinky. Be familiar with the basic tool features is a precondition for this tutorial. The goal is to understand the mechanism not to learn the tool features.

The idea behind the exercise is, to control a Pedestrian crossing light. We will use the same GUI as for the blinky tutorial but now we will use the [REQUEST] button to start a FSM, which controls the traffic lights.



The [REQUEST] must lead to a model message which starts the activity of the lights.

There are a view possibilities to receive external events (e.g. TCP/UDP Socket, using OS messaging mechanism), but the easiest way is, to make a port usable from outside the model. To do that a view steps are necessary:

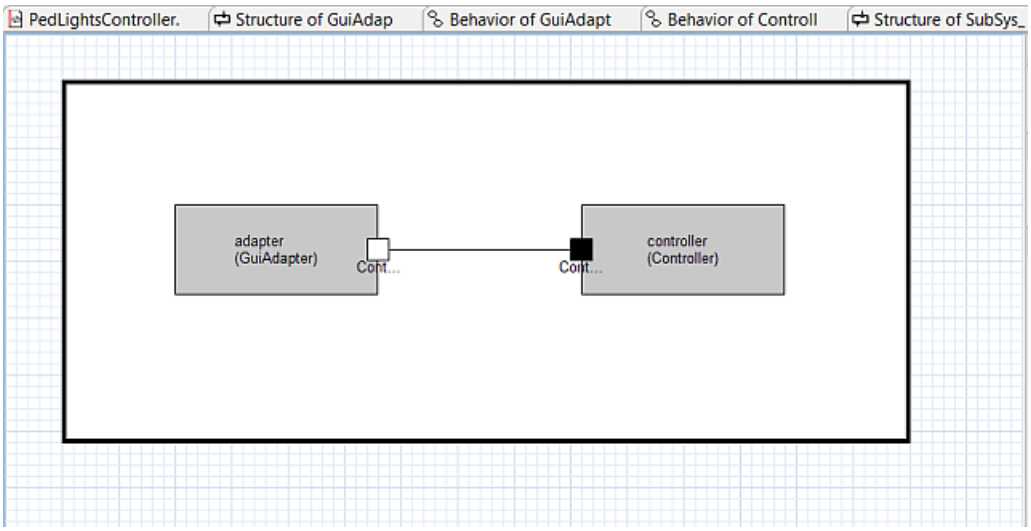
1. specify the messages (within a protocol) which should be sent into the model
2. model an actor with a port (which uses the specified protocol) and connect the port to the receiver
3. the external code should know the port (import of the port class)
4. the external code should provide a registration method, so that the actor is able to allow access to this port
5. the port can be used from the external code

6.2. Setup the model

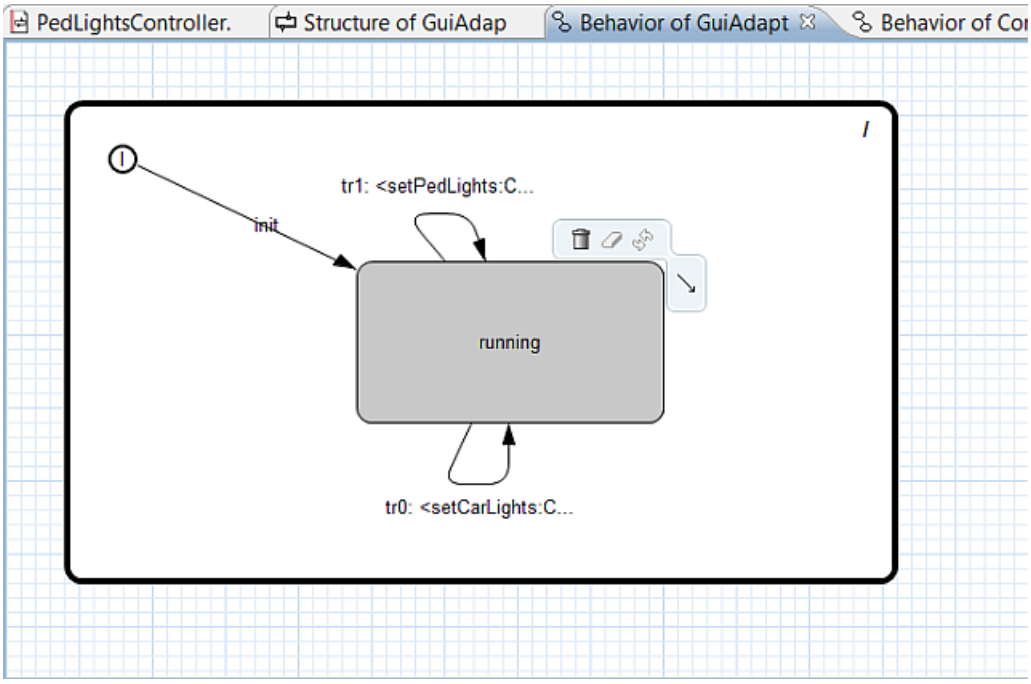
- Use the [New Model Wizzard] to create a new eTrice project and name it [PedLightsController].
- Copy the package [de.protos.PedLightGUI] to your [src] directory (see blinky tutorial).
- Uncoment line 15 (import), 36, 122 (usage) and 132-134 (registration).
- Copy the following model to your model file:

```
RoomModel PedLightsController {

 LogicalSystem LogSys_PedLights {
  SubSystemRef application: SubSys_PedLights
 }

 SubSystemClass SubSys_PedLights {
  ActorRef PedLightsTopRef: PedLightsTop
  ActorRef timingService: room.basic.service.timing.ATimingService
  LayerConnection ref PedLightsTopRef satisfied_by timingService.timer
  LayerConnection ref PedLightsTopRef satisfied_by timingService.timeout
 }

 ActorClass PedLightsTop {
  Structure {
   ActorRef adapter: GuiAdapter
   ActorRef controller: Controller
   Binding adapter.ControlPort and controller.ControlPort
  }
  Behavior { }
 }

 ActorClass GuiAdapter {
  Interface {
   conjugated Port ControlPort: PedControlProtocol
  }
  Structure {
   usercode1 {
    "import PedLightGUI.*;"
   }
   usercode2 {
    "private PedestrianLightWndNoTcp lights = new PedestrianLightWndNoTcp(\"Pedestrian Ligh
    "private TrafficLight3 carLights;"
    "private TrafficLight2 pedLights;"
   }
   external Port ControlPort
  }
  Behavior {
   Operation destroyUser() {
    "lights.closeWindow();"
   }
   StateMachine {
    Transition init: initial -> running {
     action {
      "carLights=lights.getCarLights();"
      "pedLights=lights.getPedLights();"
      "carLights.setState(TrafficLight3.OFF);"
      "pedLights.setState(TrafficLight2.OFF);"
      "lights.setPort(ControlPort);"
     }
    }
    Transition tr0: running -> running {
     triggers {
      <setCarLights: ControlPort>
     }
     action {
      "carLights.setState(state);"
     }
    }
    Transition tr1: running -> running {
     triggers {
      <setPedLights: ControlPort>
     }
     action {
      "pedLights.setState(state);"
     }
    }
    State running
   }
  }
 }
```

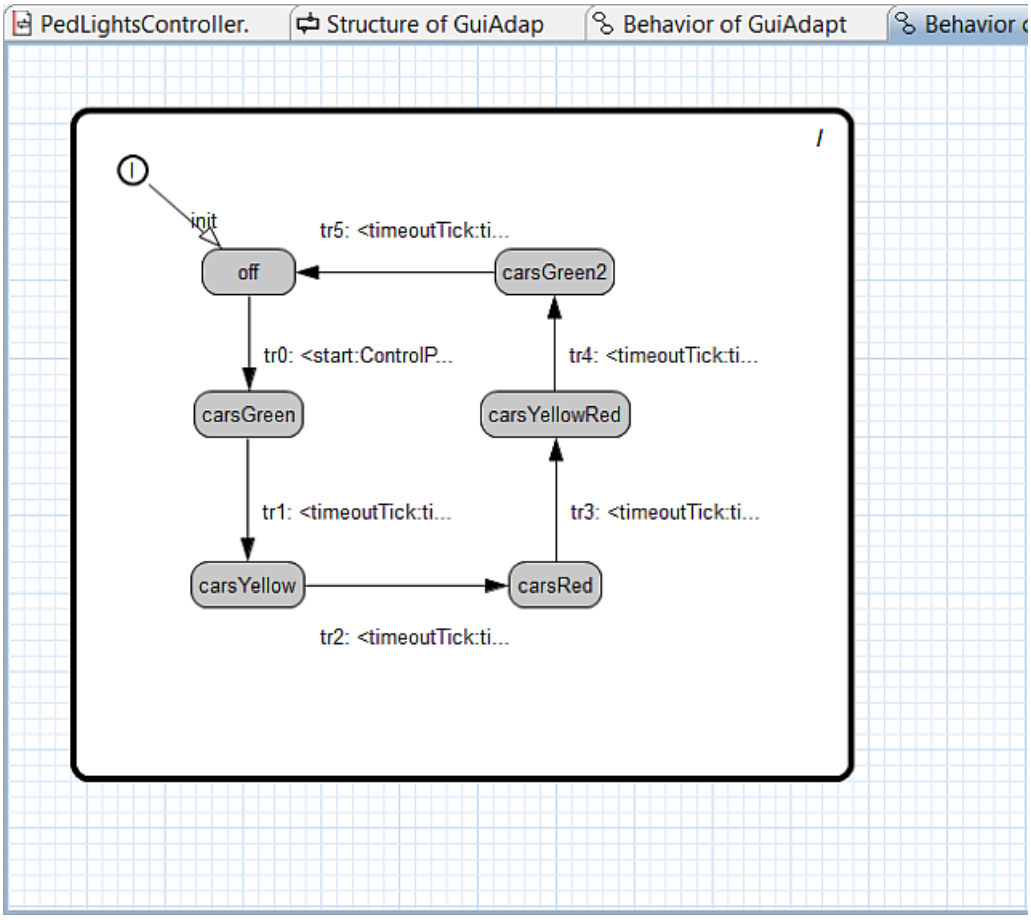
- Arrange the Structure and the Statemachines to understand the model



The [GuiAdapter] represents the interface to the external code. It registers its [ControlPort] by the external code.



Visit the initial transition to understand the registration. The actor handles the incoming messages as usual and controls the traffic lights as known from blinky.

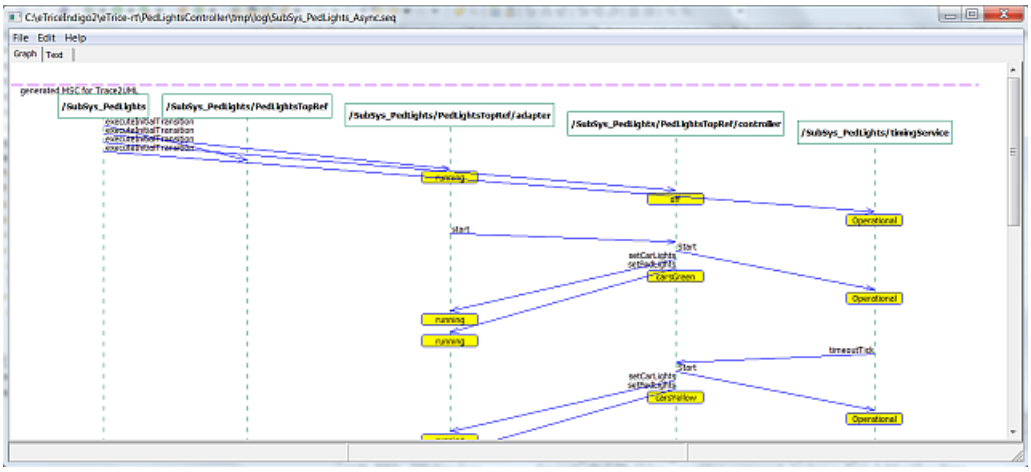


The [Controller] receives the [start] message and controls the timing of the lights. Note that the [start] message will be sent from the external code whenever the [REQUEST] button is pressed.

- Visit the model and take a closer look to the following elements:

1. PedControlProtocol => notice that the start message is defined as usual
2. Initial transition of the [GuiAdapter] => see the registration
3. The [Controller] => notice that the [Controller] receives the external message (not the [GuiAdapter]). The [GuiAdapter] just provides its port and handles the incoming messages.
4. Visit the hand written code => see the import statement of the protocol class and the usage of the port.

- Generate and test the model
- Take a look at the generated MSC => notice that the start message will shown as if the [GuiAdapter] had sent it.



6.3. Why does it work and why is it save?

The tutorial shows that it is generally possible to use every port from outside the model as long as the port knows its peer. This is guaranteed by describing protocol and the complete structure (especially the bindings) within the model. The only remaining question is: Why is it save and does not hurt the „run to completion” semantic. To answer this question, take a look at the [MessageService.java] from the runtime environment. There you will find the receive method which puts each message into the queue.

```
@Override
public synchronized void receive(Message msg) {
 if (msg!=null) {
  messageQueue.push(msg);
  notifyAll(); // wake up thread to compute message
 }
}
```

This method is synchronized. That means, regardless who sends the message, the queue is secured. If we later on (e.g. for performance reasons in C/C++) distinguish between internal and external senders (same thread or not), care must be taken to use the external (secure) queue.

Chapter 7. ROOM Concepts

7.1. Main Concepts

7.1.1. ActorClass

7.1.2. Port

7.1.3. Protocol