

eTrice

Henrik Renz-Reichert Thomas Schütz

February 18, 2013

Contents

1	eTrice Overview	5
1.1	What is eTrice?	5
1.2	Reduction of Complexity	5
2	Introduction to the ROOM Language	7
2.1	Scope of ROOM	7
2.1.1	Where does it come from?	7
2.1.2	Which kind of SW-Systems will be addressed?	8
2.1.3	What is the relation between OOP and ROOM?	8
2.1.4	What are the benefits of ROOM?	10
2.1.5	Which consequences must be taken into account?	10
2.2	Basic Concepts	11
2.2.1	Actor, Port, Protocol	11
2.2.2	Hierarchy in Structure and Behavior	12
2.2.3	Layering	13
2.2.4	Run to Completion	13
2.3	Execution Models	14
2.3.1	Communication Methods	14
2.3.2	Execution Methods	14
2.3.3	Execution Models	15
3	Working with the eTrice Tutorials	16
4	Setting up the Workspace for Java Projects	17
5	Tutorial HelloWorld for Java	22
5.1	Scope	22
5.2	Create a new model from scratch	22
5.3	Create a state machine	25

<i>CONTENTS</i>	2
5.4 Build and run the model	26
5.5 Open the Message Sequence Chart	29
5.6 Summary	29
6 Tutorial Blinky (Java)	30
6.1 Scope	30
6.2 Create a new model from scratch	31
6.3 Add two additional actor classes	31
6.4 Create a new protocol	33
6.5 Import the Timing Service	34
6.6 Finish the model structure	37
6.7 Implement the Behavior	39
6.8 Summary	43
7 Tutorial Sending Data (Java)	45
7.1 Scope	45
7.2 Create a new model from scratch	45
7.3 Add a data class	46
7.4 Create a new protocol	46
7.5 Create MrPing and MrPong Actors	47
7.6 Define Actor Structure and Behavior	48
7.6.1 Define MrPongs behavior	49
7.6.2 Define MrPing behavior	51
7.7 Define the top level	54
7.8 Generate and run the model	55
7.9 Summary	56
8 Tutorial Pedestrian Lights (Java)	57
8.1 Scope	57
8.2 Setup the model	58
8.3 Why does it work and why is it safe?	60
9 Tutorial Remove C-Comment (C)	61
9.1 Scope	61
9.2 Create a new model from scratch	61
9.3 Create your own data type	62
9.4 Create the model	63
9.5 Generate, build and run the model	65
9.6 Summary	66

10 ROOM Concepts	67
10.1 Actors	67
10.1.1 Description	67
10.1.2 Motivation	67
10.1.3 Notation	67
10.1.4 Details	67
10.2 Protocols	69
10.2.1 Description	69
10.2.2 Motivation	69
10.2.3 Notation	70
10.3 Ports	70
10.3.1 Description	70
10.3.2 Motivation	70
10.3.3 Notation	70
10.4 DataClass	72
10.4.1 Description	72
10.4.2 Notation	72
10.5 Layering	73
10.5.1 Description	73
10.5.2 Notation	73
10.6 Finite State Machines	73
10.6.1 Description	73
10.6.2 Motivation	75
10.6.3 Notation	75
10.6.4 Examples	76
11 eTrice Features	78
11.1 Codegenerators	78
11.1.1 Java Generator	78
11.1.2 C++ Generator	78
11.1.3 C Generator	78
12 Codegenerators	79
13 Runtimes	80
14 eTrice Models and Their Relations	81
14.1 The ROOM Model	82
14.2 The Config Model	84

<i>CONTENTS</i>	4
14.3 The Physical Model	84
14.4 The Mapping Model	86
15 eTrice Developer's Reference	88
15.1 Architecture	88
15.1.1 Editor and Generator Components	89
15.1.2 Runtimes	90
15.1.3 Unit Tests	90
15.2 Component Overview	90
15.2.1 Room Language Overview	90
15.2.2 Config Language Overview	92
15.2.3 Aggregation Layer Overview	92
15.2.4 Generator Overview	94

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 Reduction of Complexity

eTrice is all about the reduction of complexity:

- structural complexity
 - by explicit modeling of hierarchical Actor containment, layering and inheritance
- behavioral complexity
 - by hierarchical statemachines with inheritance
- teamwork complexity
 - because loosely coupled Actors provide a natural way to structure team work
 - since textual model notation allows simple branching and merging
- complexity of concurrent & distributed systems
 - because loosely coupled Actors are deployable to threads, processes, nodes
- complexity of variant handling and reuse (e.g. for product lines)
 - by composition of existing Actors to new structures

- since Protocols and Ports make Actors replaceable
- by inheritance for structure, behavior and Protocols
- by making use of model level libraries
- complexity of debugging
 - model level debugging: state machine animation, data inspection and manipulation, message injection, generated message sequence charts
 - model checking easier for model than for code (detect errors before they occur)

Chapter 2

Introduction to the ROOM Language

2.1 Scope of ROOM

This chapter will give a rough overview of what ROOM (R eal time O bject O riented M odeling) is and what it is good for. It will try to answer the following questions:

- Where does it come from?
- Which kind of SW-Systems will be addressed?
- What is the relation between OOP and ROOM?
- What are the benefits of ROOM?
- Which consequences must be taken into account?

2.1.1 Where does it come from?

Room was developed in the 1990th on the background of the upcoming mobile applications with the goal to manage the complexity of such huge SW-Systems. From the very beginning ROOM has focused on a certain type of SW-Systems and is, in contrast to the UML, well suited for this kind of systems. In this sense, ROOM is a DSL (Domain Specific Language) for distributed, event driven, real time systems.

Bran Selic, Garth Gullekson and Paul T. Ward have published the concepts 1994 in the book **Real-Time Object-Oriented Modeling**. The company *object time* TM developed a ROOM tool which was taken over by *Rational SW* TM and later on by *IBM* TM. The company *Protos Software Gmbh* TM also developed a ROOM tool called *Trice* TM for control software for production machines and automotive systems. *Trice* TM is the predecessor of eTrice (see Introduction to eTrice).

From our point of view ROOM provides still the clearest, simplest, most complete and best suited modeling concepts for the real time domain. All later proposals like the UML do not fit as well to this kind of problems.

2.1.2 Which kind of SW-Systems will be addressed?

As mentioned before ROOM addresses distributed, event driven, real time systems. But what is a *real time system*? ROOM defines a set of properties which are typical for a real time system. These properties are:

- Timeliness
- Dynamic internal structure
- Reactiveness
- Concurrency
- Distribution
- Reliability

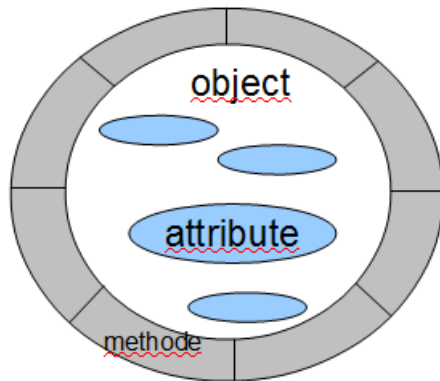
Each of these properties has potential to make SW development complex. If a given system can be characterized with a combination of or all of these properties, ROOM might be applied to such a system.

As an example take a look at a washing machine. The system has to react on user interactions, has to handle some error conditions like a closed water tap or a defective lye pump. It has to react simultaneously to all these inputs. It has to close the water valve in a certain time to avoid flooding the basement. So, the system can be characterized as timely, concurrent and reactive. As long as the washing machine does not transform to a laundry drier by itself, the system has no dynamic internal structure and as long as all functions are running on a single micro controller the (SW)-system is not distributed. ROOM fits perfect to such a system.

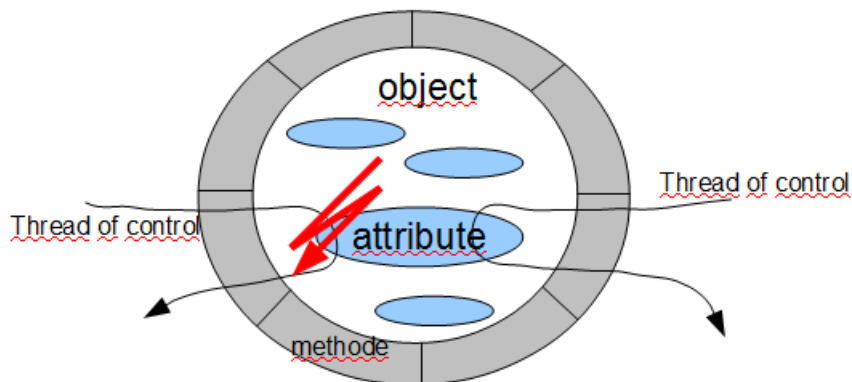
A SW system which mainly consists of data transformations like signal/image processing or a loop controller (e.g. a PID controller) cannot be characterized with any of the above mentioned properties. However, in the real world most of the SW systems will be a combination of both. ROOM can be combined with such systems, so that for example an actor provides a *run to completion* context for calculating an image processing algorithm or a PID controller.

2.1.3 What is the relation between OOP and ROOM?

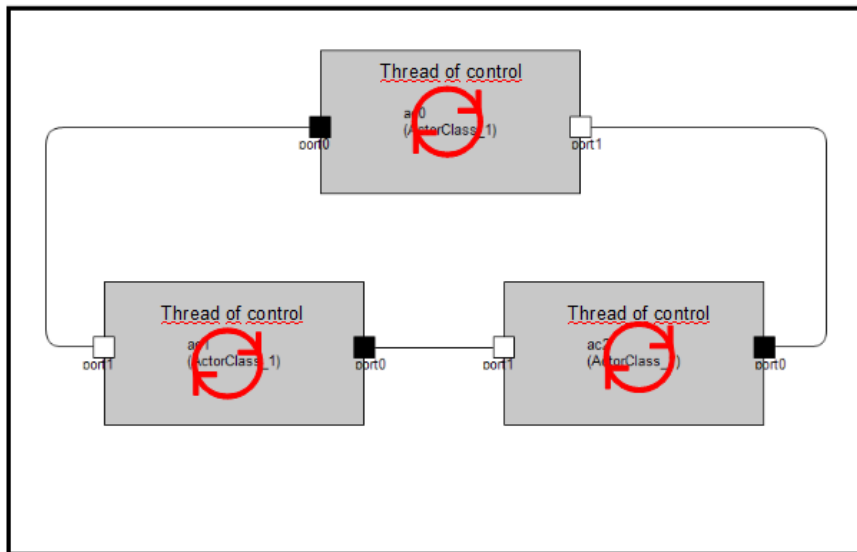
The relation between classical object oriented programming and ROOM is comparable to the relation between assembler programming and C programming. It provides a shift of the object paradigm. As the picture shows, the classic object paradigm provides some kind of information hiding. Attributes can be accessed via access methods. Logical higher level methods provide the requested behavior to the user.



As the figure illustrates, the classical object paradigm does not care about concurrency issues. The threads of control will be provided by the underlying operating system and the user is responsible to avoid access violations by using those operating system mechanisms directly (semaphore, mutex).



ROOM provides the concept of a logical machine (called actor) with its own thread of control. It provides some kind of cooperative communication infrastructure with **run to completion** semantic. That makes developing of business logic easy and safe (see basic concepts). The logical machine provides an encapsulation shell including concurrency issues (see chapter **Run to completion**).



This thinking of an object is much more general than the classic one.

2.1.4 What are the benefits of ROOM?

ROOM has a lot of benefits and it depends on the users point of view which is the most important one. From a general point of view the most important benefit is, that ROOM allows to create SW systems very efficient, robust and safe due to the fact that it provides some abstract, high level modeling concepts combined with code generation and a small efficient runtime environment.

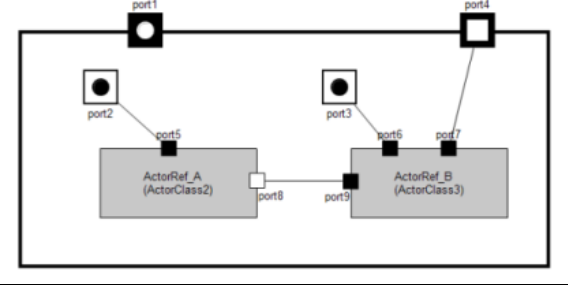
In detail:

- ROOM models contain well defined interfaces (protocols), which makes it easy to reuse components in different applications or e.g. in a test harness.
- Graphical modeling makes it easy to understand, maintain and share code with other developers
- Higher abstraction in combination with automated code generation provides very efficient mechanisms to the developer.
- ROOM provides graphical model execution, which makes it easy to understand the application or find defects in a very early phase.

2.1.5 Which consequences must be taken into account?

Generating code from models will introduce some overhead in terms of memory footprint as well as performance. For most systems the overhead will be negligible. However, the decision for using ROOM should be made explicitly and it is always a trade off between development costs, time to market and costs in

Table 2.1: Actor and Protocol Example

 <p>The diagram shows two actor references, ActorRef_A (ActorClass2) and ActorRef_B (ActorClass3), within a container. ActorRef_A has ports port2 and port5. ActorRef_B has ports port3, port6, and port7. A connection exists between port5 of ActorRef_A and port6 of ActorRef_B. Additionally, port8 of ActorRef_A and port9 of ActorRef_B are connected. External ports port1, port4, port3, and port7 are also shown as connection points.</p>	<pre> ProtocolClass ProtocolClass1 { incoming { Message m1(data: int32) Message m2() } outgoing { Message m3(data: DataClass1) Message m4() } } </pre>
Actor with Subactors	Protocol Definition

terms of a little bit more of memory and performance. Thanks to the powerful component model, ROOM is especially well suited for the development of software product lines with their need for reusable core assets.

Care must be taken during the introduction of the new methodology. Due to the fact that ROOM provides a shift of the object paradigm, developers and teams need a phase of adaption. Every benefit comes at a price.

2.2 Basic Concepts

2.2.1 Actor, Port, Protocol

The basic elements of ROOM are the actors with their ports and protocols. The protocol provides a formal interface description. The port is an interaction point where the actor interacts with its outside world. Each port has exactly one protocol attached. The sum of all ports builds up the complete interface of an actor. Each port can receive messages, with or without data, which are defined in the attached protocol. Each message will be handled by the actors behavior (state machine) or will be delegated to the actors internal structure.

The actor provides access protection for its own attributes (including complex types (classical objects)), including concurrency protection. An actor has neither public attributes nor public operations. The only interaction with the outside world takes place via interface ports. This ensures a high degree of reusability on actor level and provides an effective and safe programming model to the developer.

Receiving a message via a port will trigger the internal state machine. A transition will be executed depending on the message and the current state. Within this transition, detail level code will be executed and response messages can be sent.

With this model, a complex behavior can be divided into many relatively simple, linked actors. To put it the other way round: The complex behavior will be provided by a network of relatively simple components which are communicating with each other via well defined interfaces.

2.2.2 Hierarchy in Structure and Behavior

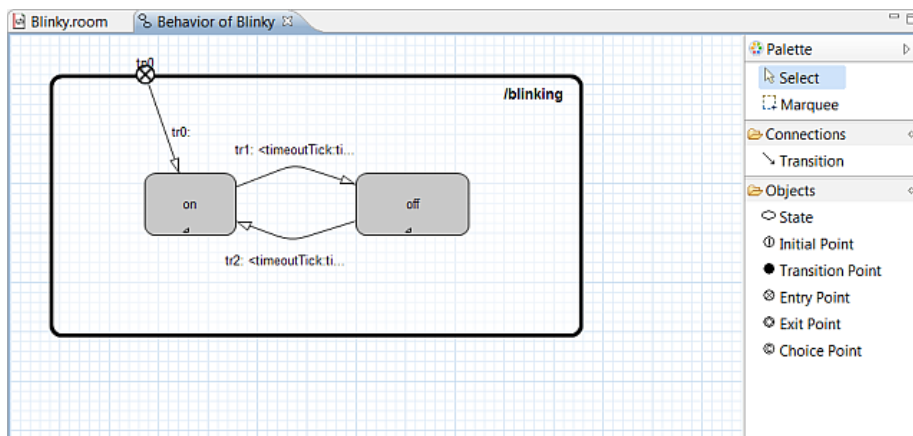
ROOM provides two types of hierarchy. Behavioral hierarchy and structural hierarchy. Structural hierarchy means that actors can be nested to arbitrary depth. Usually you will add more and more details to your application with each nesting level. That means you can focus yourself on any level of abstraction with always the same element, the actor. Structural hierarchy provides a powerful mechanism to divide your problem in smaller pieces, so that you can focus on the level of abstraction you want to work on.

The actor's behavior will be described with a state machine. A state in turn may contain sub states. This is another possibility to focus on an abstraction level. Take the simple FSM from the blinky actor from the blinky tutorial.

Top level:



blinking Sub machine:



From an abstract point of view there is a state *blinking*. But a simple LED is not able to blink autonomously. Therefore you have to add more details to your model to make a LED blinking, but for the current work it is not of interest how the blinking is realized. This will be done in the next lower level of the hierarchy.

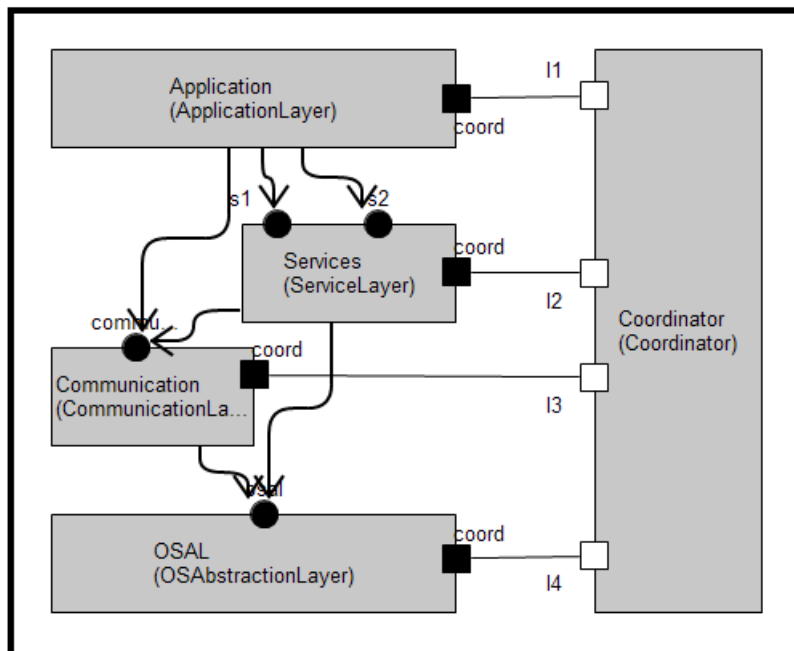
This simple example might give an idea how powerful this mechanisms is.

The hierarchical FSM provides a rich tool box to describe real world problems (see **room concepts**).

2.2.3 Layering

Layering is another well known form of abstraction to reduce complexity in the structure of systems. ROOM is probably the only language that supports Layering directly as language feature. Layering can be expressed in ROOM by Actors with specialized Ports, called Service Access Points (*SAP*) and Service Provision Points (*SPP*).

The Actor that provides a service implements an SPP and the client of that service implements an SAP. The Layer Connection connects all SAPs of a specific Protocol within an Actor hierarchy with an SPP that implements the service. From the Actors point of view, SAPs and SPPs behave almost like regular ports.



The Example shows a layered model. The Layer Connections define e.g. that the *ApplicationLayer* can only use the services of the *ServiceLayer* and the *CommunicationLayer*. Actors inside the *ApplicationLayer* that implement an SAP for those services are connected directly to the implementation of the services. Layering and actor hierarchies with port to port connections can be mixed on every level of granularity.

2.2.4 Run to Completion

Run to completion (RTC) is a very central concept of ROOM. It enables the developer to concentrate on the functional aspects of the system. The developer doesn't have to care about concurrency issues all the time. This

job is concentrated to the system designer in a very flexible way. What does **run to completion** mean: RTC means that an actor, which is processing a message, can not receive the next message as long as the processing of the current message has been finished. Receiving of the next message will be queued from the underlying run time system.

Note: It is very important not to confuse run to completion and preemption. Run to completion means that an actor will finish the processing of a message before he can receive a new one (regardless of its priority). That does not mean that an actor cannot be preempted from an higher priority thread of control. But even a message from this higher prior thread of control will be queued until the current processing has been finished.

With this mechanism all actor internal attributes and data structures are protected. Due to the fact that multiple actors share one thread of control, all objects are protected which are accessed from one thread of control but multiple actors. This provides the possibility to decompose complex functionality to several actors without the risk to produce access violations or dead locks.

2.3 Execution Models

Since from ROOM models executable code can be generated, it is important to define the way the actors are executed and communicate with each other. The combination of communication and execution is called the Execution Model. Currently the eTrice tooling only supports the **message driven** and parts of the **data driven** execution model. In future releases more execution models will be supported, depending on the requirements of the community.

2.3.1 Communication Methods

- **message driven** (asynchronous, non blocking, no return value): Usually the message driven communication is implemented with message queues. Message queues are inherently asynchronous and enable a very good decoupling of the communicating parties.
- **data driven** (asynchronous, non blocking, no return value): In data driven communication sender and receiver often have a shared block of data. The sender writes the data and the receiver polls the data.
- **function call** (synchronous, blocking, return value): Regular function call as known in most programming languages.

2.3.2 Execution Methods

- **execution by receive event**: The message queue or the event dispatcher calls a **receive event** function of the message receiver and thereby executes the processing of the event.
- **polled execution**: The objects are processed by a cyclic **execute** call

- **execution by function call:** The caller executes the called object via function call

2.3.3 Execution Models

In todays embedded systems in most cases one or several of the following execution models are used:

message driven

The message driven execution model is a combination of message driven communication and execution by receive event. This model allows for distributed systems with a very high throughput. It can be deterministic but the determinism is hard to proof. This execution model is often found in telecommunication systems and high performance automation control systems.

data driven

The data driven execution model is a combination of data driven communication and polled execution. This model is highly deterministic and very robust, but the polling creates a huge performance overhead. The determinism is easy to proof (simple mathematics). The execution model is also compatible with the execution model of control software generated by Tools like Matlab(TM) and LabView(TM). This model is usually used for systems with requirements for safety, such as automotive and avionic systems.

synchronous

The synchronous execution model could also be called **simple function calls**. This model is in general not very well suited to support the **run to completion** semantic typical for ROOM models, but could also be generated from ROOM models. With this execution model also lower levels of a software system, such as device drivers, could be generated from ROOM models.

Chapter 3

Working with the eTrice Tutorials

The eTrice Tutorials will help you to learn and understand the eTrice tool and concepts. ETrice supports several target languages. The concepts will not be explained for each language.

Most of the common concepts will be described for Java as target language. To start with a new language the first steps to setup the workspace and to generate and run the first model will be described also. Target language specific aspects will be described as well.

Therefore the best way to start with eTrice is to follow the Java Tutorials and after that switch to your target language.

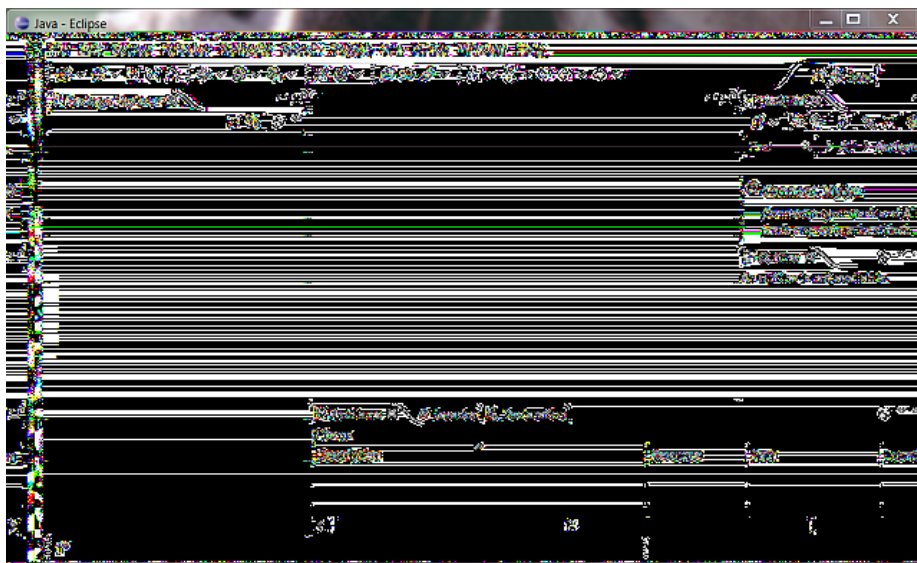
Chapter 4

Setting up the Workspace for Java Projects

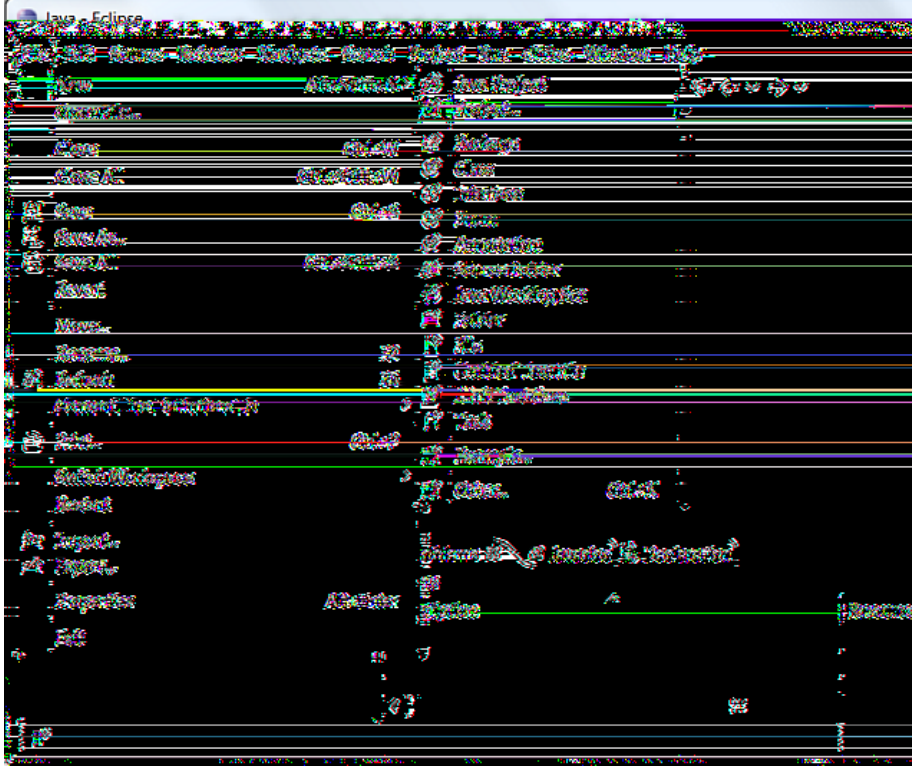
ETrice generates code out of ROOM models. The code generator and the generated code relies on a runtime framework and on some ready to use model parts. This parts provide services like:

- messaging
- logging
- timing

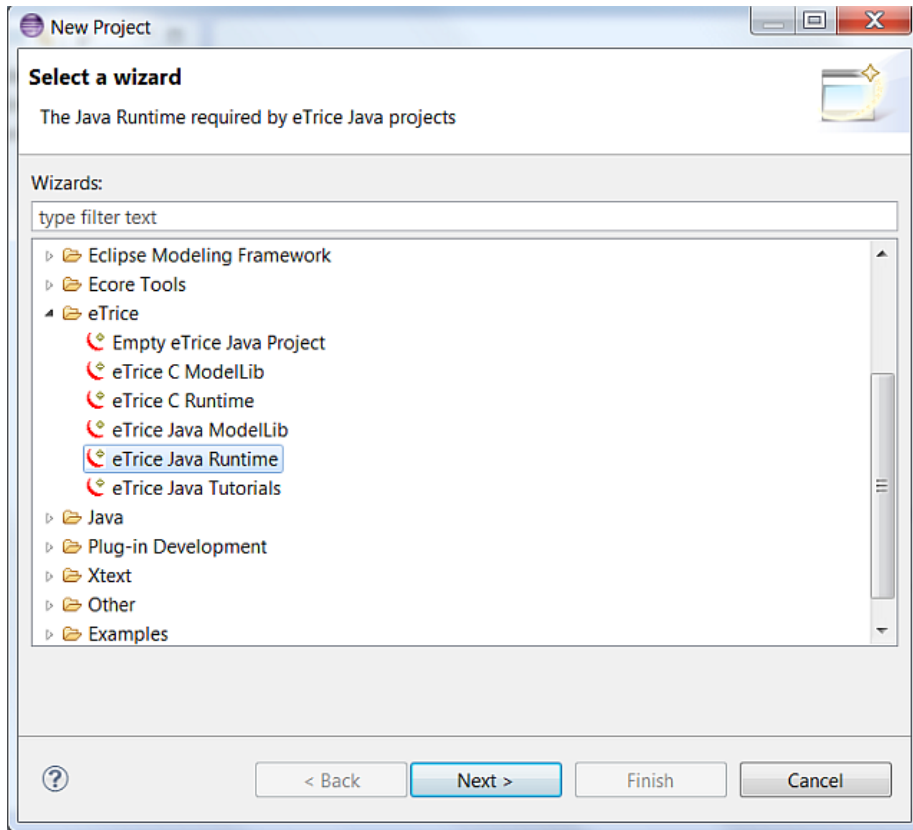
Additionally some tutorial models will be provided to make it easy to start with eTrice. All this parts must be available in our workspace before you can start working. After installation of eclipse (juno) and the eTrice plug in, your workspace should look like this:



Just the *eTrice* menu item is visible from the eTrice tool. From the *File* menu select **File->New->Project**



Open the *eTrice* tab and select *eTrice Java Runtime*
Press *Next* and *Finish* to install the Runtime into your workspace.

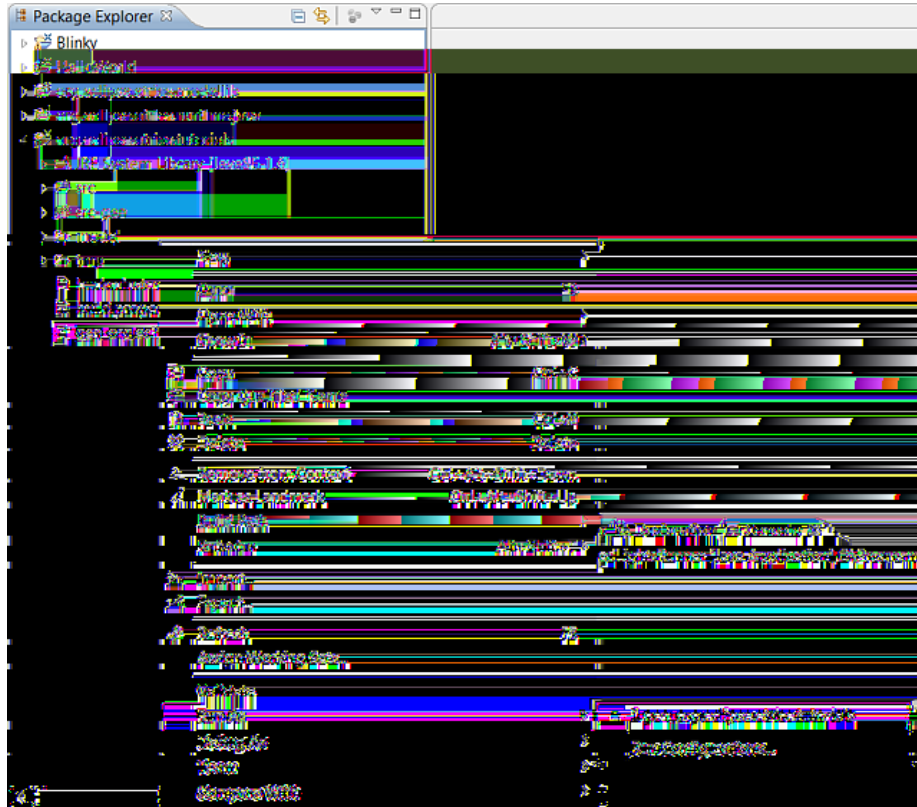


Do the same steps for *eTrice Java Modellib* and *eTrice Java Tutorials*. To avoid temporary error markers you should keep the proposed order of installation. The resulting workspace should look like this:

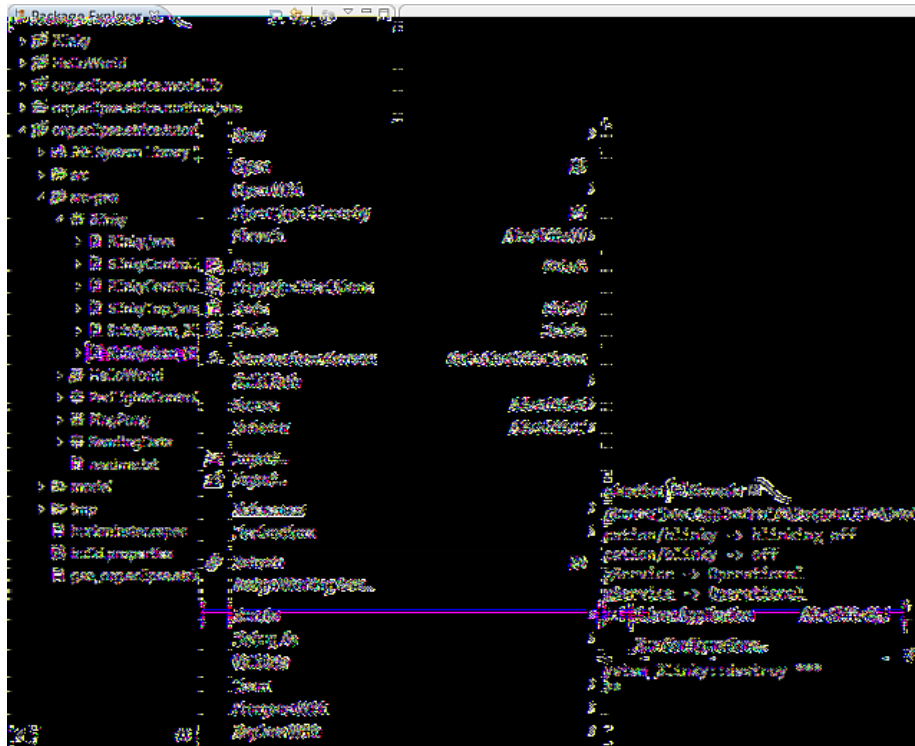


Now workspace is set up and you can perform the tutorials or start with your work.

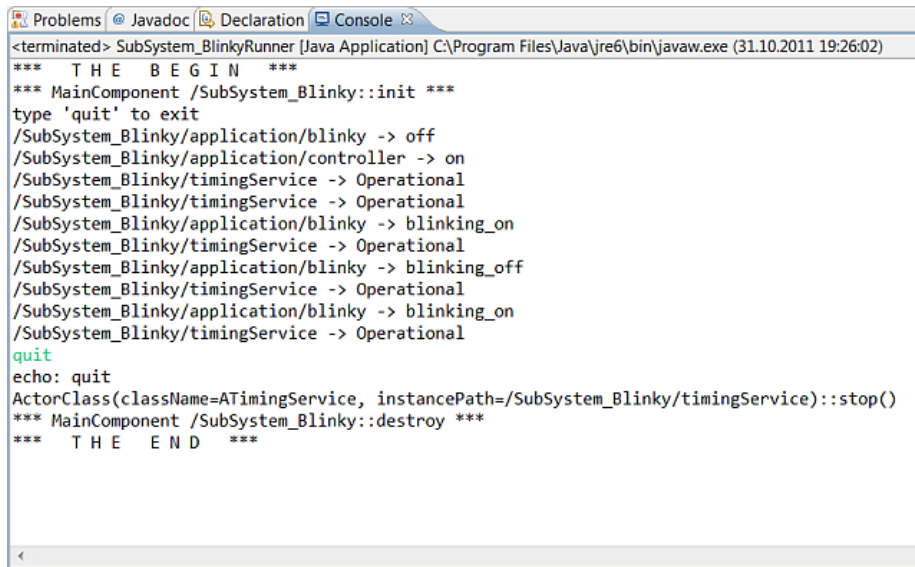
The tutorial models are available in the *org.eclipse.etrice.tutorials* project. All tutorials are ready to generate and run without any changes. To start the code generator simply run `gen_org.eclipse.etrice.tutorials.launch` as `gen_org.eclipse.etrice.tutorials.launch`:



After generation for each tutorial a java file called **SubSystem_ModelnameRunner.java** is generated. To run the model simply run this file as a java application:



To stop the application type *quit* in the console window.



Performing the tutorials will setup an dedicated project for each tutorial. Therefore there are some slight changes especially whenever a path must be set (e.g. to the model library) within your own projects. All this is described in the tutorials.

Chapter 5

Tutorial HelloWorld for Java

5.1 Scope

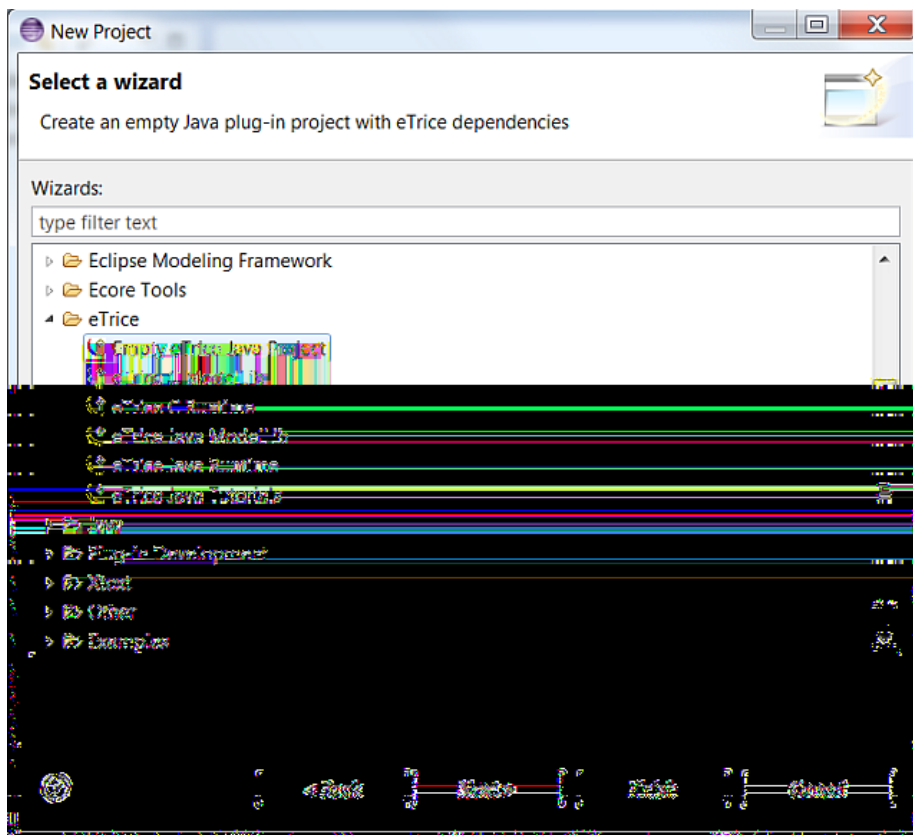
In this tutorial you will build your first very simple eTrice model. The goal is to learn the work flow of eTrice and to understand a few basic features of ROOM. You will perform the following steps:

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

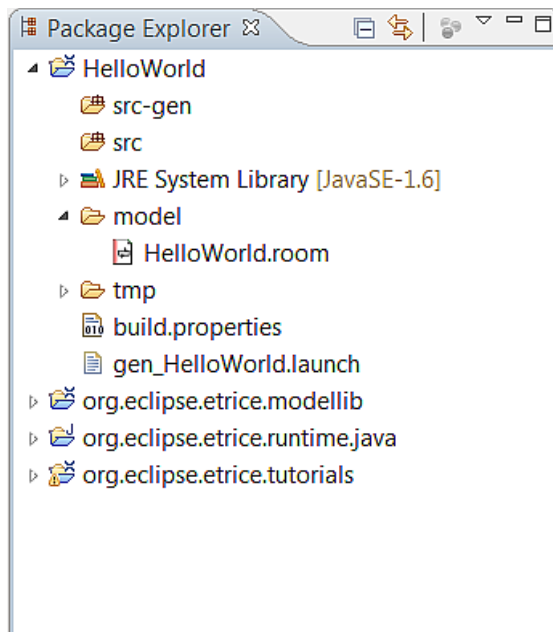
Make sure that you have set up the workspace as described in *Setting up the workspace*.

5.2 Create a new model from scratch

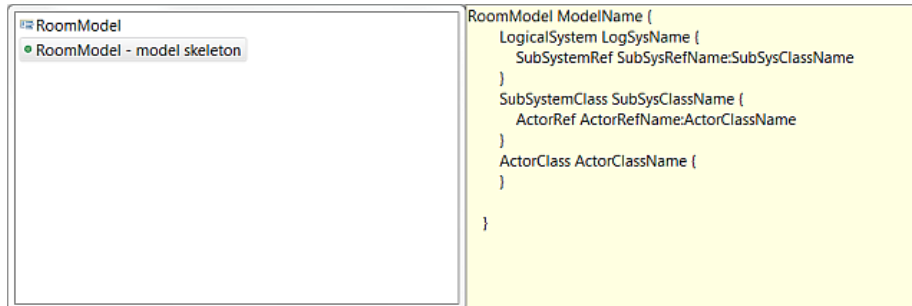
The easiest way to create a new eTrice Project is to use the eclipse project wizard. From the eclipse file menu select **File->New->Project** and create a new eTrice project and name it **HelloWorld**.



The wizard creates everything that is needed to create, build and run an 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 delete the contents of the file. Open the content assist with Ctrl+Space and select *model skeleton*.



Edit the template variables by typing the new names and jumping with Tab from name to name.

The resulting model code should look like this:

```
RoomModel HelloWorld {

    LogicalSystem System_HelloWorld {
        SubSystemRef subsystem : SubSystem_HelloWorld
    }

    SubSystemClass SubSystem_HelloWorld {
        ActorRef application : HelloWorldTop
    }

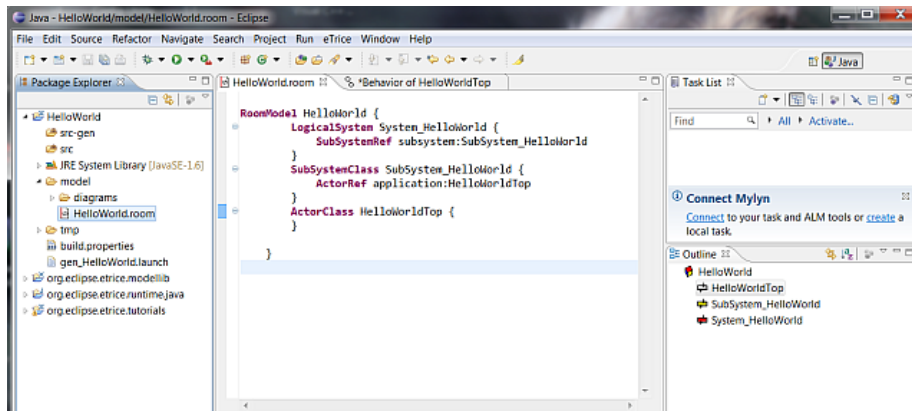
    ActorClass HelloWorldTop {
    }
}
```

The goal of eTrice is to describe distributed systems on a logical level. In the current version not all elements will be used. But as prerequisite for further versions the following elements can be defined:

- the *LogicalSystem* (currently optional)
- at least one *SubSystemClass* (mandatory)
- at least one *ActorClass* (mandatory)

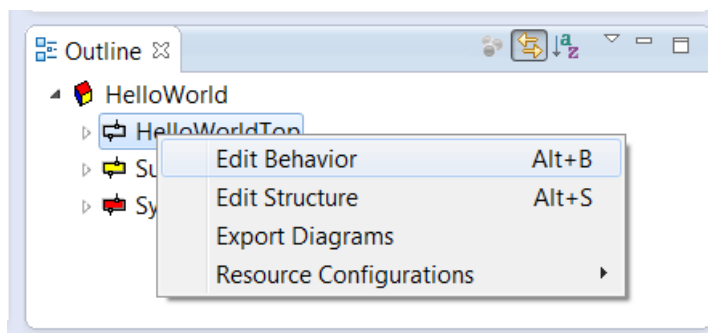
The *LogicalSystem* represents the complete distributed system and contains at least one *SubSystemRef*. The *SubSystemClass* represents an address space and contains at least one *ActorRef*. The *ActorClass* is the building block of which an application will be built of. It is in general a good idea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

The outline view of the textual ROOM editor shows the main modeling elements in an easy to navigate tree.



5.3 Create a state machine

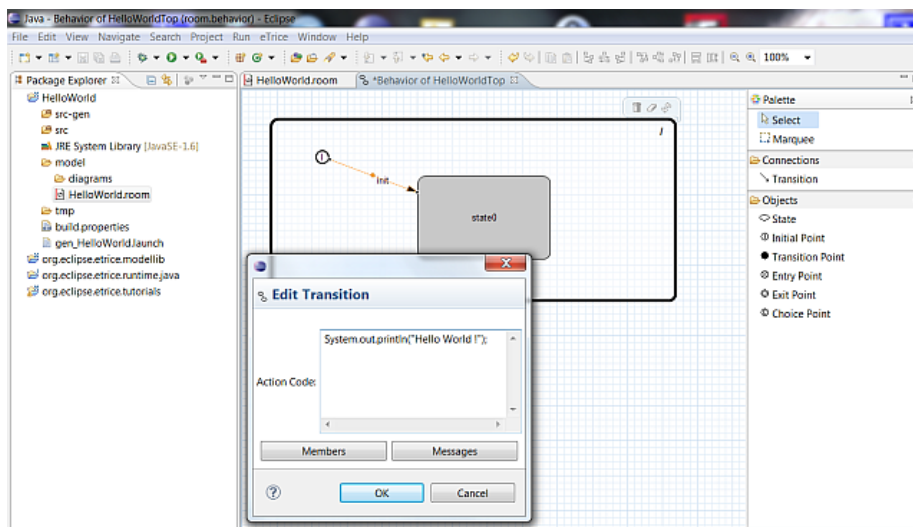
We will implement the Hello World code on the initial transition of the *HelloWorldTop* actor. Therefore open the state machine editor by right clicking the *HelloWorldTop* actor in the outline view and select *Edit Behavior*.



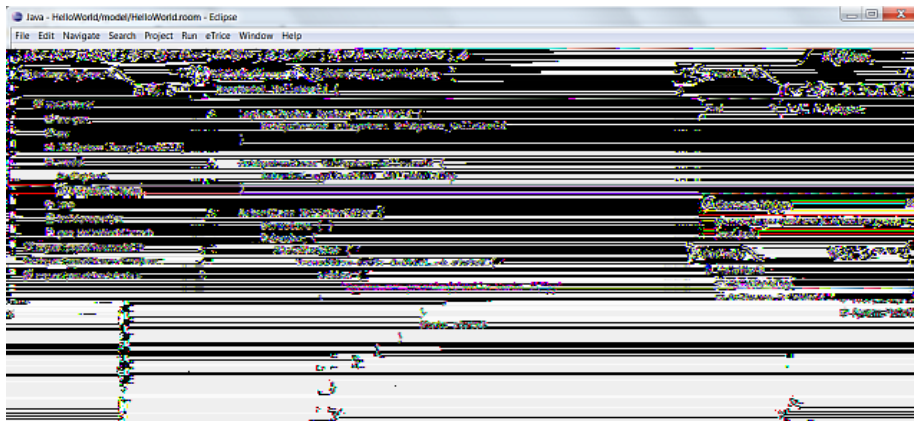
The state machine editor will be opened. Drag and drop an *Initial Point* from the tool box to the diagram into the top level state. Drag and drop a *State* from the tool box to the diagram. Confirm the dialogue with *ok*. Select the *Transition* in the tool box and draw the transition from the *Initial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

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

The result should look like this:

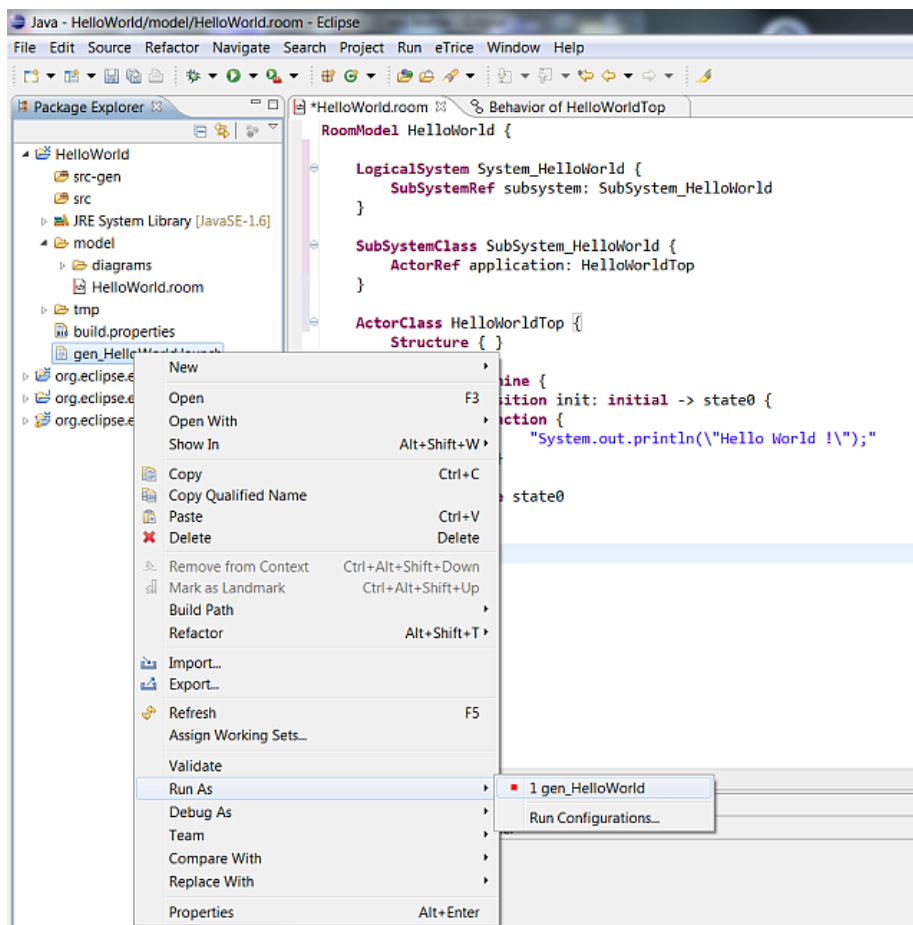


Save the diagram and inspect the model file. Note that the textual representation was created after saving the diagram.

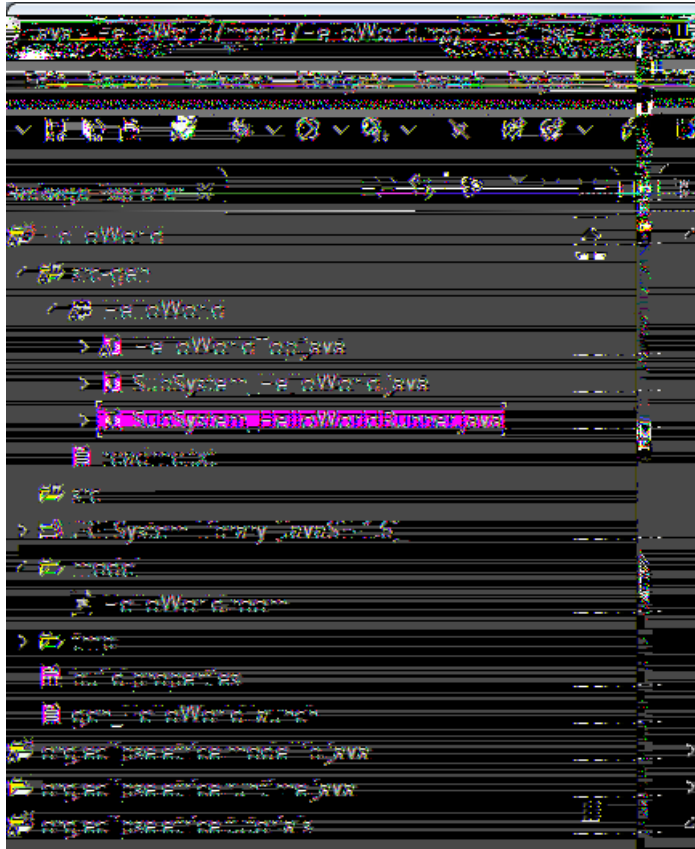


5.4 Build and run the model

Now the model is finished and source code can be generated. The project wizard has created a launch configuration that is responsible for generating the source code. From *HelloWorld/* right click **gen_HelloWorld.launch** and run it as `gen_HelloWorld`. All model files in the model directory will be generated.



The code will be generated to the src-gen directory. The main function 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.

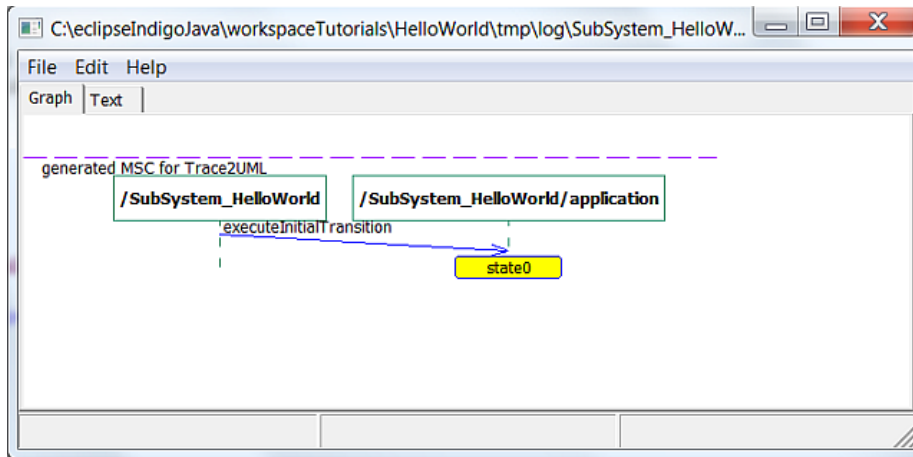
```

Problems @ Javadoc Declaration Console Error Log
<terminated> SubSystem_HelloWorldRunner (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (31.08.2012 19:39:30)
*** THE BEGIN ***
*** MainComponent /SubSystem_HelloWorld::init ***
type 'quit' to exit
HelloWorld !
/SubSystem_HelloWorld/application -> state0
quit
|echo: quit
*** MainComponent /SubSystem_HelloWorld::stop ***
=== done stop MsgSvcCtrl
=== done stop actor instances
*** MainComponent /SubSystem_HelloWorld::destroy ***
=== done destroy actor instances
=== done close loggers
=== done destroy RTServices

*** THE END ***
  
```

5.5 Open the Message Sequence Chart

During runtime the application produced a MSC and wrote it to a file. Open `HelloWorld/tmp/log/SubSystem_HelloWorld_Async.seq` using Trace2UML (it is open source and can be obtained from <http://trace2uml.tigris.org/>). You should see something like this:



5.6 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 will deal with more complex hierarchies in structure and behavior.

Chapter 6

Tutorial Blinky (Java)

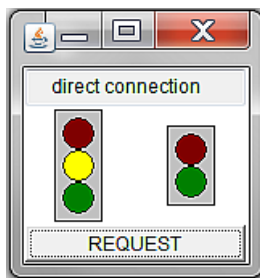
6.1 Scope

This tutorial describes how to use the *TimingService*, how to combine a generated model with manual code and how to model 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.

After the exercise is created you must copy the GUI to your src directory (see below).

The package contains four java classes which implements a small 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 will just toggle 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

6.2 Create a new model from scratch

Remember the exercise *HelloWorld*. Create a new eTrice project and name it *Blinky*.

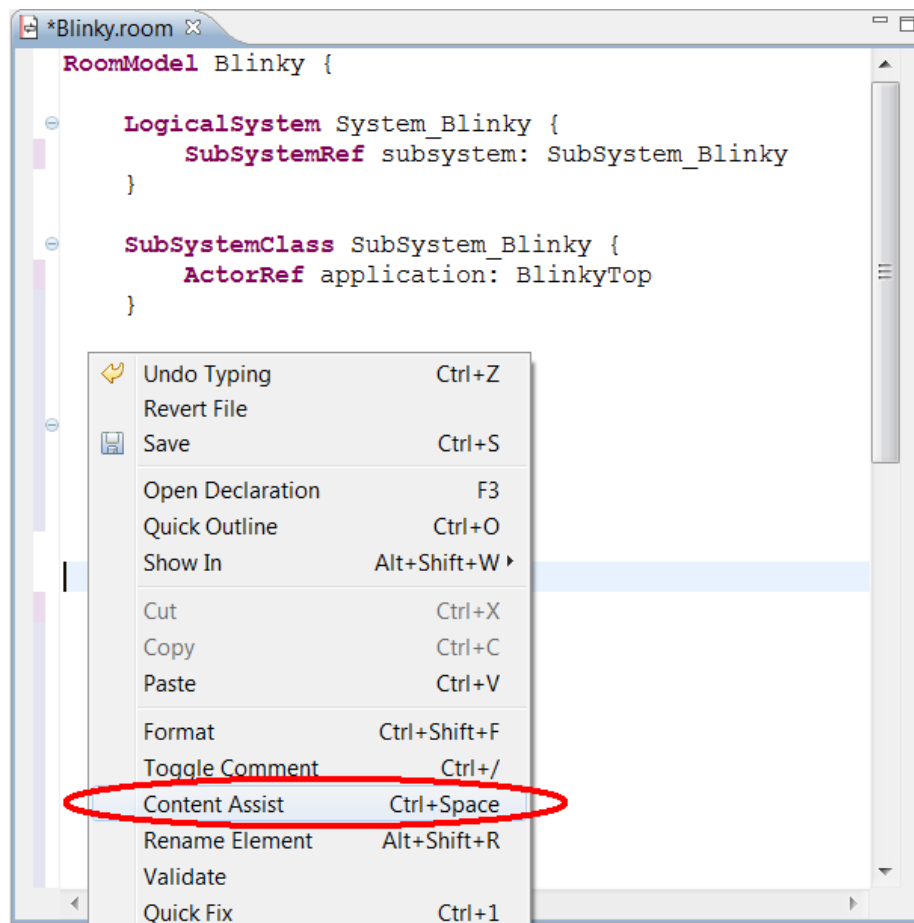
To use the GUI please copy the package *org.eclipse.etrice.tutorials.PedLightGUI* from *org.eclipse.etrice.tutorials/src* to your *src* directory *Blinky/src*. For this tutorial you must remove the error markers by editing the file *Pedestrian-LightWndNoTcp.java*. Appropriate comments are provided to remove the error markers for this tutorial.

Open the *Blinky.room* file and copy the following code into the file or use content assist to create the model.

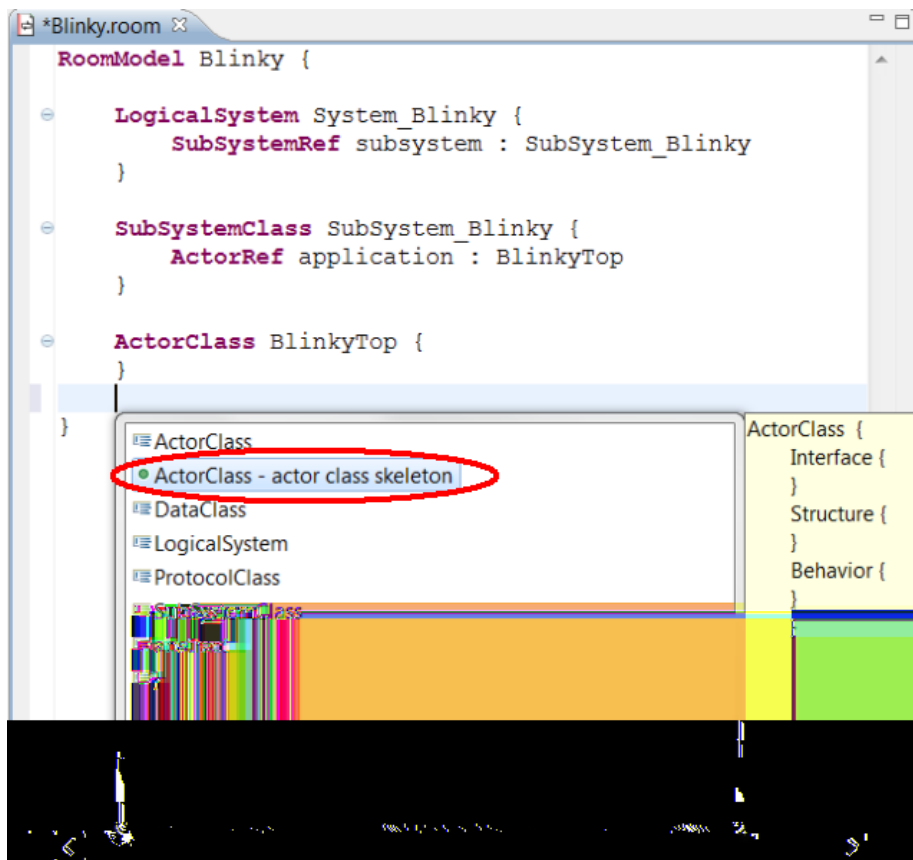
```
RoomModel Blinky {  
  
    LogicalSystem System_Blinky {  
        SubSystemRef subsystem : SubSystem_Blinky  
    }  
  
    SubSystemClass SubSystem_Blinky {  
        ActorRef application : BlinkyTop  
    }  
  
    ActorClass BlinkyTop {  
    }  
}
```

6.3 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*.

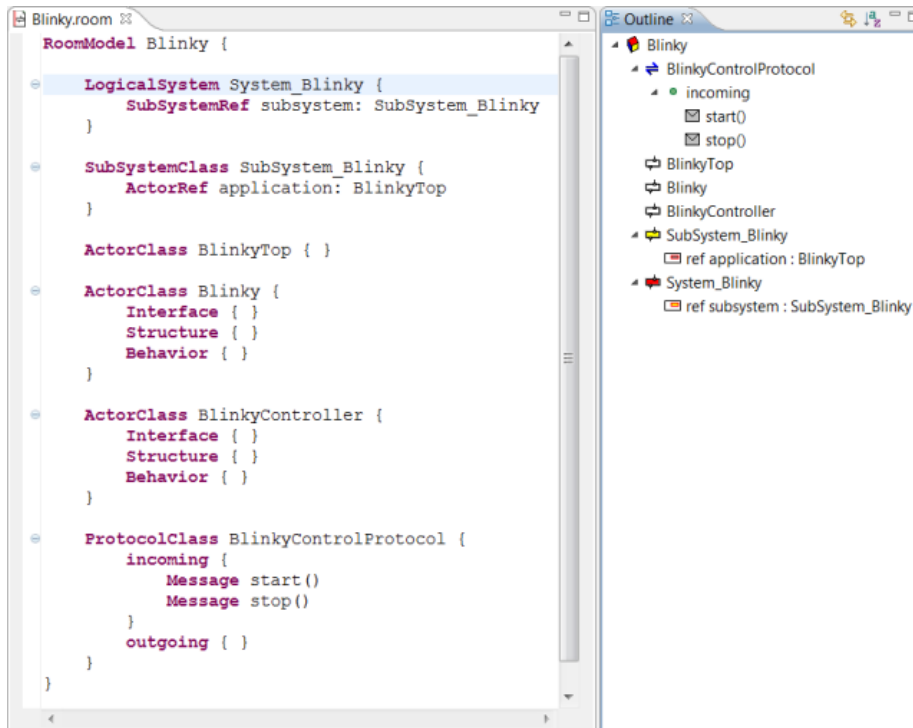
With Ctrl+Shift+F you can beautify the model code.

Save the model and visit the outline view.

6.4 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 incoming messages called *start* and *stop*.

The resulting code should look like this:

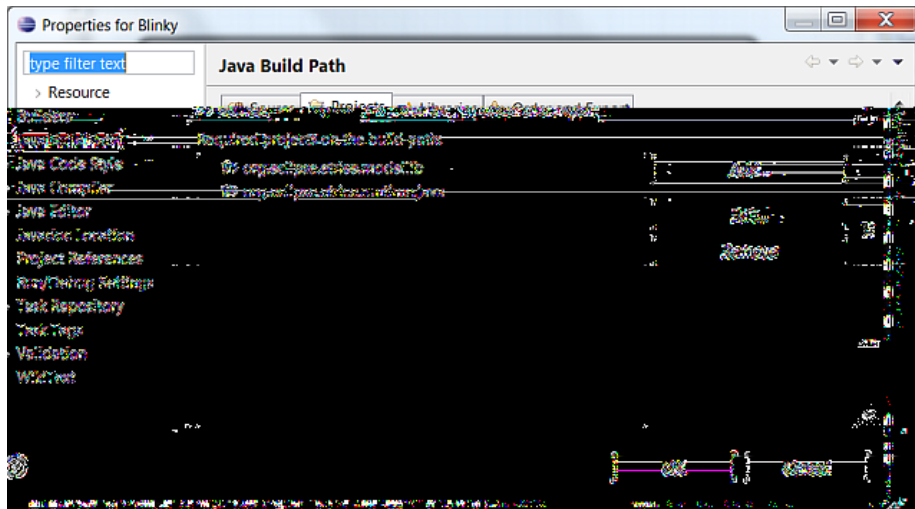


With Ctrl-Shift+F or selecting *Format* from the context menu you can format the text. Note that all elements are displayed in the outline view.

6.5 Import the Timing Service

Switching on and off the LED is timing controlled. The timing service is provided from the model library and must be imported before it can be used from the model.

This is the first time you use an element from the modellib. Make sure that your Java Build Path has the appropriate entry to the modellib. Otherwise the java code, which will be generated from the modellib, can not be referenced. (right click to *Blinky* and select properties. Select the *Java Build Path* tab)



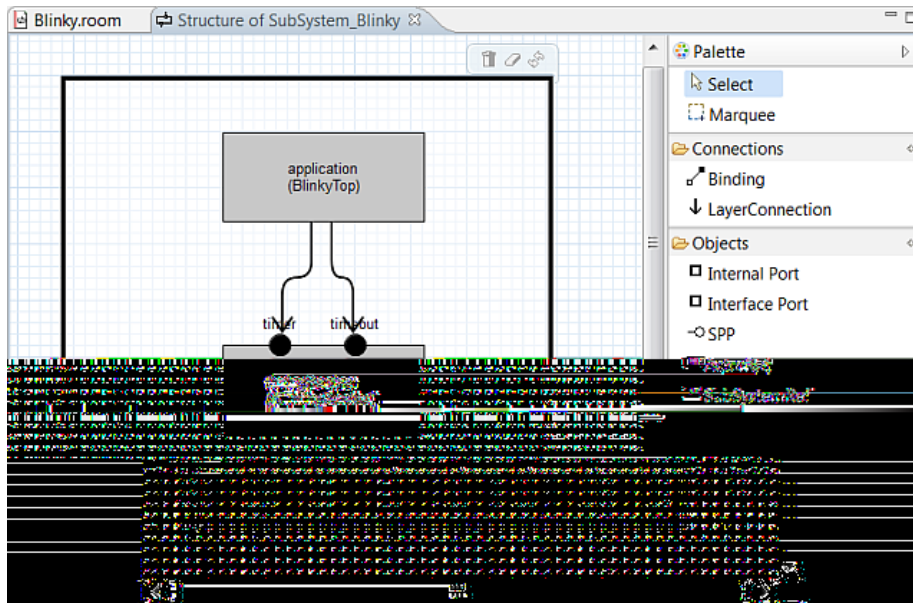
After the build path is set up return to the model and navigate the cursor at the beginning of the model and import the timing service:

```
RoomModel Blinky {
    import room.basic.service.timing.* from
    "../../org.eclipse.etrice.modellib/models/TimingService.room"

    LogicalSystem System_Blinky {
        SubSystemRef subsystem: SubSystem\_Blinky
    }
}
...
```

Make sure that the path fits to your folder structure. The original tutorial code is different due to the folder structure.

Now it can be used within the model. Right click to **SubSystem_Blinky** within the outline view. Select *Edit Structure*. The *application* is already referenced in the subsystem. Drag and Drop an *ActorRef* to the **SubSystem_Blinky** and name it *timingService*. From the actor class 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
```

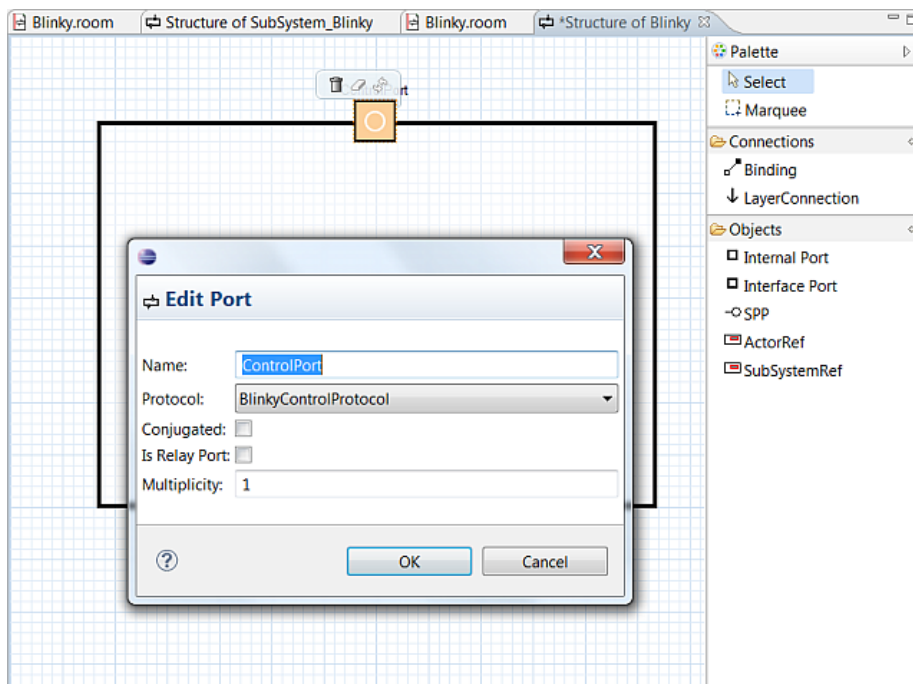
Do the same thing for `BlinkyController`.

The resulting code should look like this:



6.6 Finish the model structure

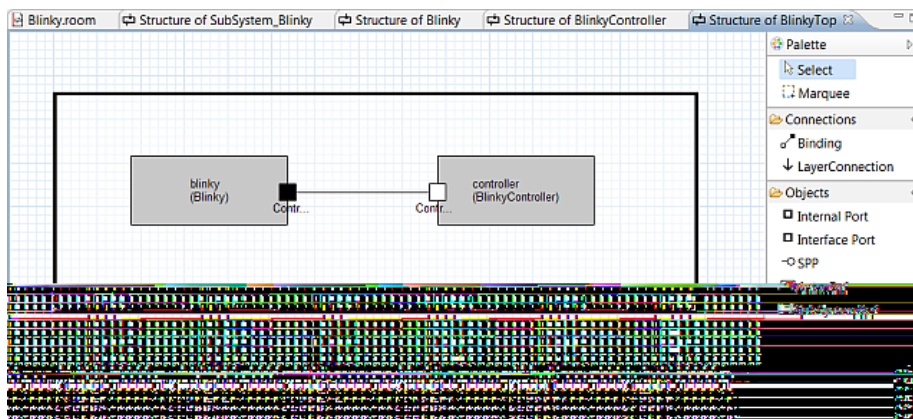
From the outline view right click to *Blinky* and select *Edit Structure*. Drag and Drop an *Interface Port* to the boarder of the *Blinky* actor. Note that an interface port is not possible inside the actor. Name the port *ControlPort* and select *BlinkyControlProtocol* from the drop down list. Uncheck *Conjugated* and *Is Relay Port*. Click *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 regular 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:



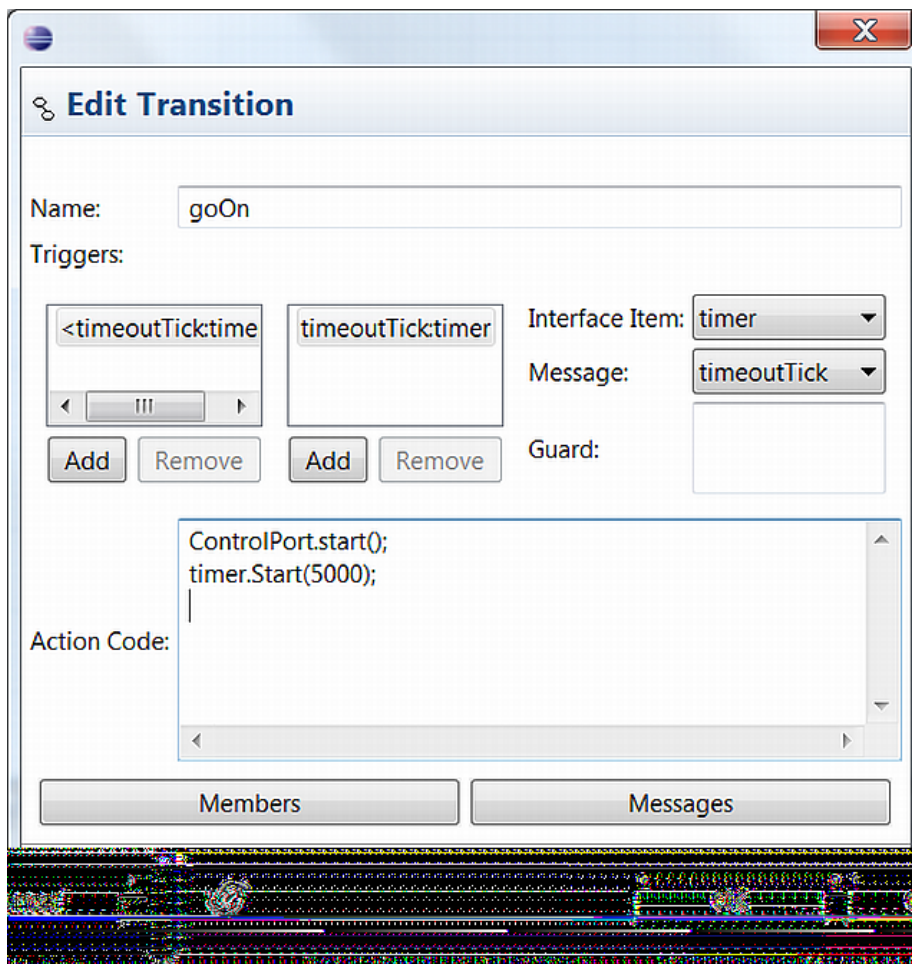
6.7 Implement the Behavior

The application should switch on and off the LED for 5 seconds in a 1 second interval, then 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 *on* 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 transition dialog should look like this:

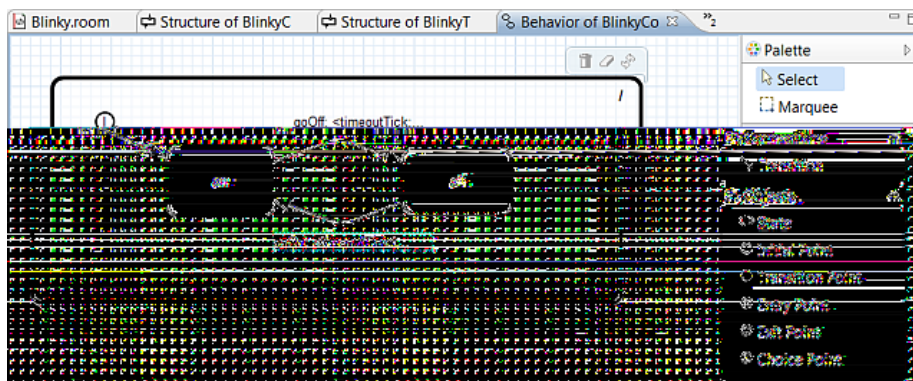


The defined ports will be generated as a member attribute of the actor class

from type of the attached protocol. So, to send a message you must state *port.message(param)*; 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: (Note that the arrows peak changes if the transition contains action code.)



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

```

ActorClass BlinkyController {
  Interface {
    conjugated Port ControlPort: BlinkyControlProtocol
  }
  Structure {
    external Port ControlPort
    SAP timer: room.basic.service.timing.FTimeout
  }
  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: (type it or simply copy it from the tutorial project)

```

ActorClass Blinky {
  Interface {
    conjugated Port ControlPort: BlinkyControlProtocol
  }
  Structure {
    external Port ControlPort
    SAP timer: room.basic.service.timing.FTimeout
  }
  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
  }
}

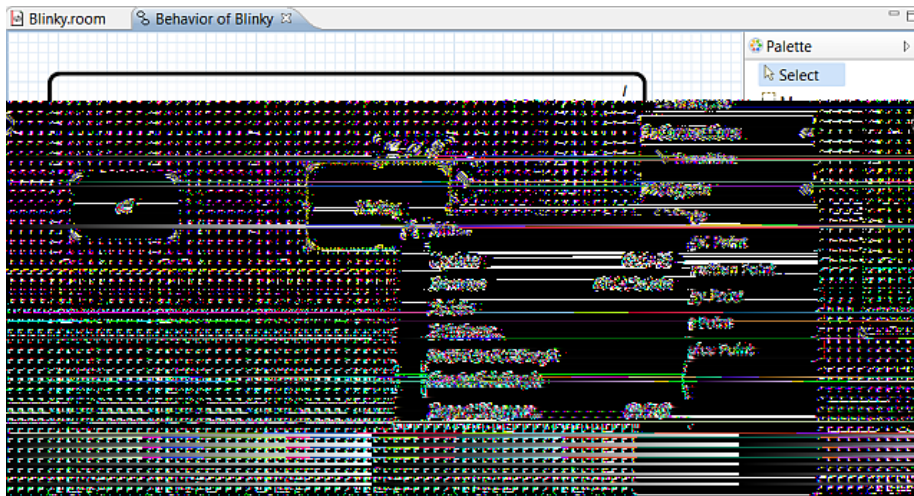
```

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.

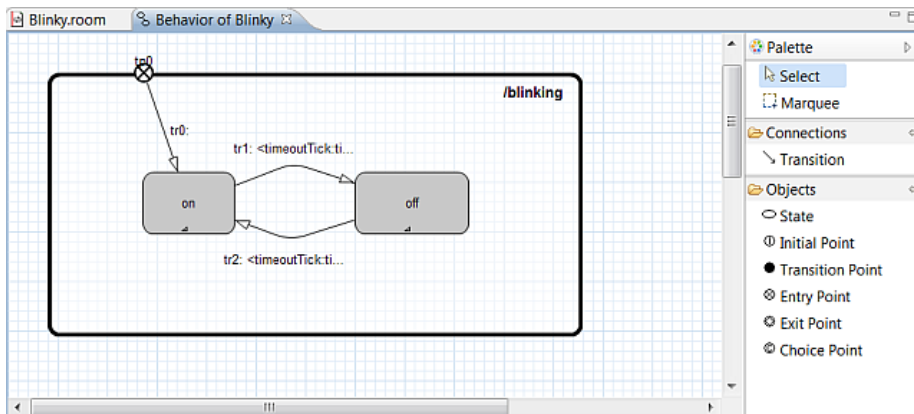
The Operation `destroyUser()` is a predefined operation that will be called during shutdown of the application. Within this operation, cleanup of manual coded classes can be done.

Now design the FSM of *Blinky*. Remember, as the name suggested *blinking* is a state in which the LED must be switched on and off. We will realize that by an hierarchical FSM in which the *blinking* state has two sub states.

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 *Control-Port*. The trigger event from *blinking* to *off* is the *stop* event from the *Control-Port*. Note: The transition from *blinking* to *off* is a so called group transition. This is a outgoing transition from a super state (state with sub states) without specifying the concrete leave state (state without sub states). An incoming transition to a super state is called history transition.

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

The complete model can be found in `/org.eclipse.etrice.tutorials/model/Blinky`.

6.8 Summary

Run the model and take a 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 7

Tutorial Sending Data (Java)

7.1 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 send 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

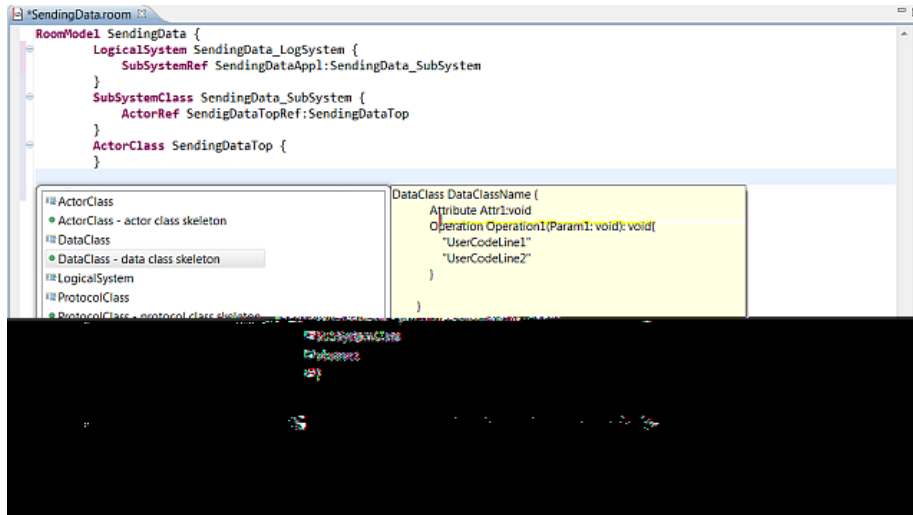
7.2 Create a new model from scratch

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 {
    }
}
```

7.3 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 add 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.

7.4 Create a new protocol

With the help of *Content Assist* create a *ProtocolClass* and name it *PingPong-Protocol*. Create the following messages:

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

```

    }
}

```

7.5 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 { }
    }
}

```

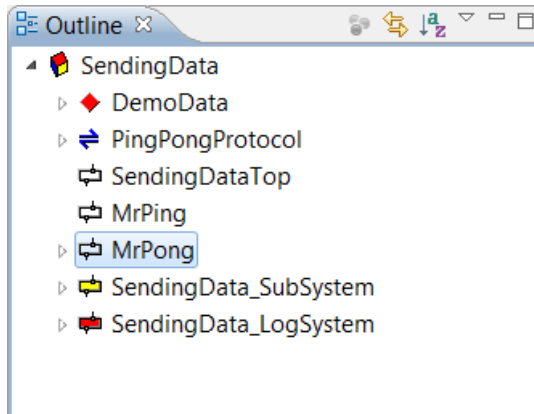


```

    }
}

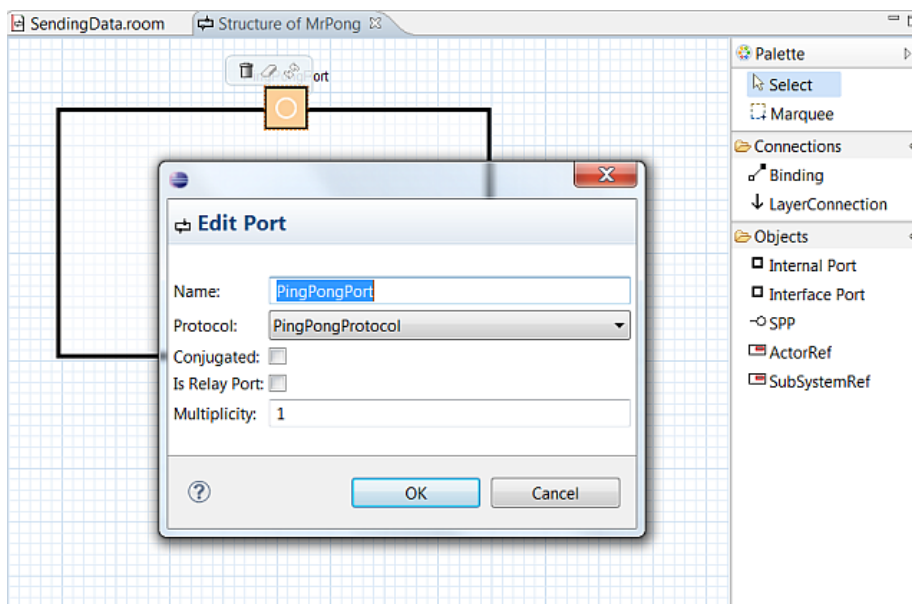
```

The outline view should look like this:



7.6 Define Actor Structure and Behavior

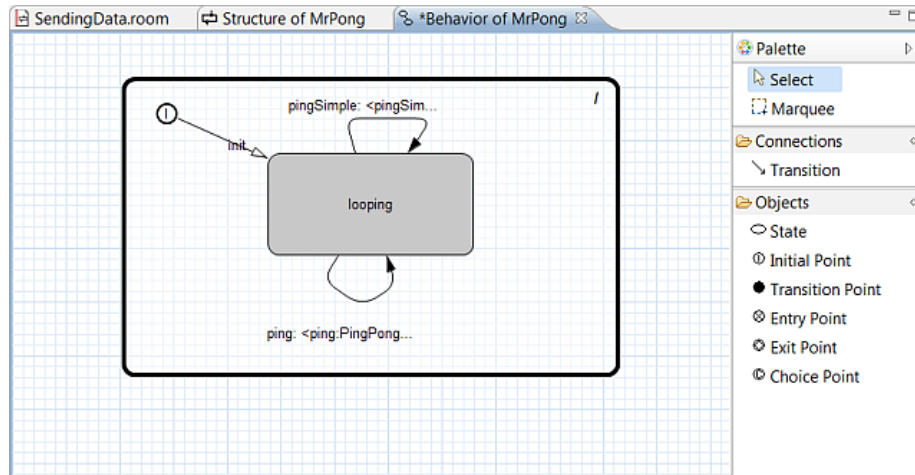
Save the model and visit the outline view. Within the outline view, right click on the *MrPong* actor and select *Edit Structure*. Select an *Interface Port* from the toolbox and add it to *MrPong*. Name the Port *PingPongPort* and select the *PingPongProtocol*.



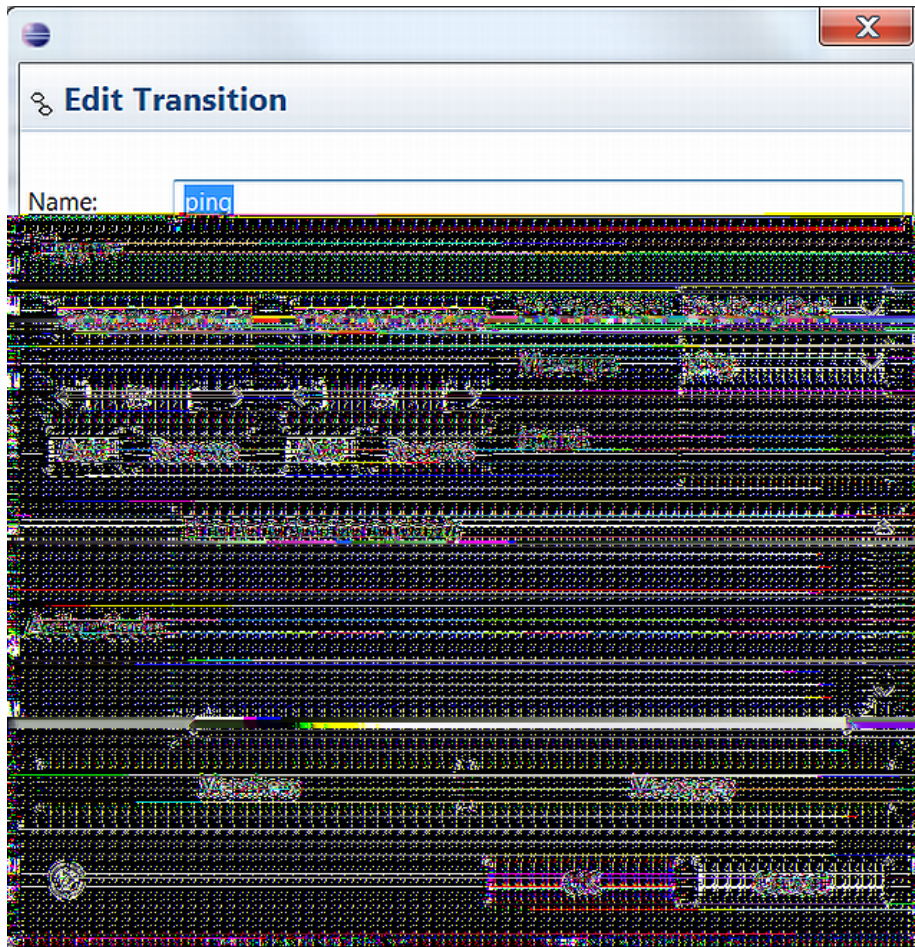
Do the same with *MrPing* but mark the port as *conjugated*

7.6.1 Define MrPongs behavior

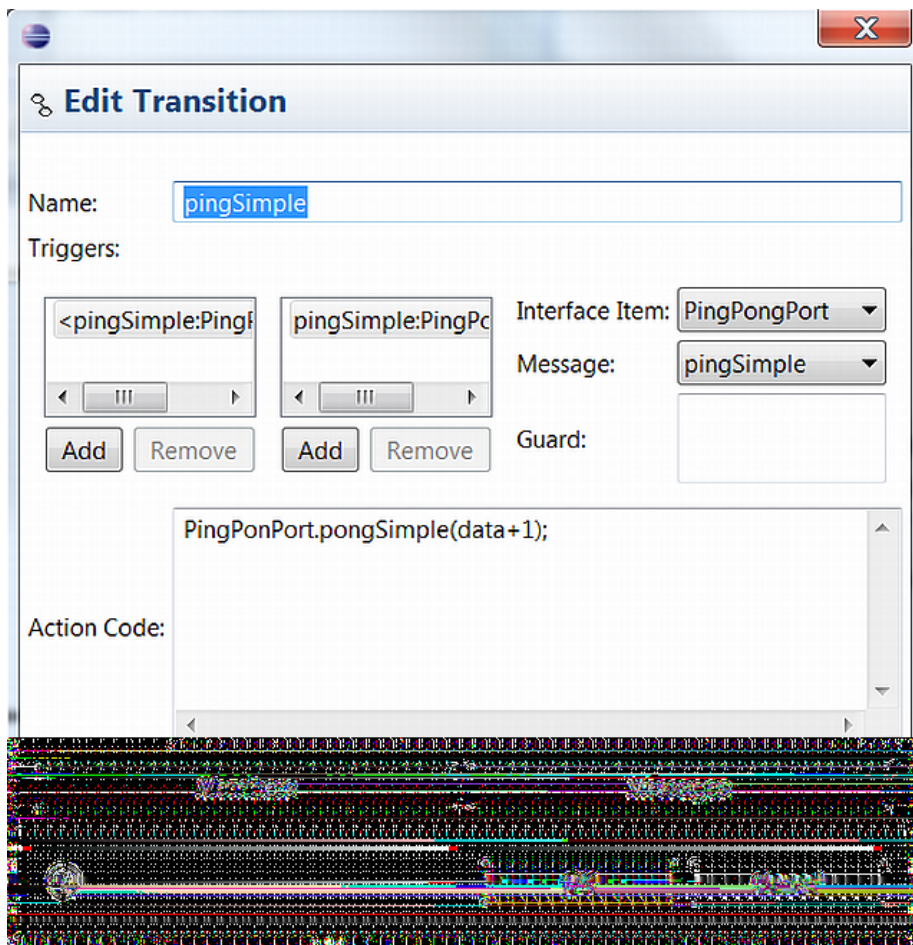
Within the outline view, right click MrPong and select *Edit Behavior*. Create the following state machine:



The transition dialogues should look like this: For *ping*:

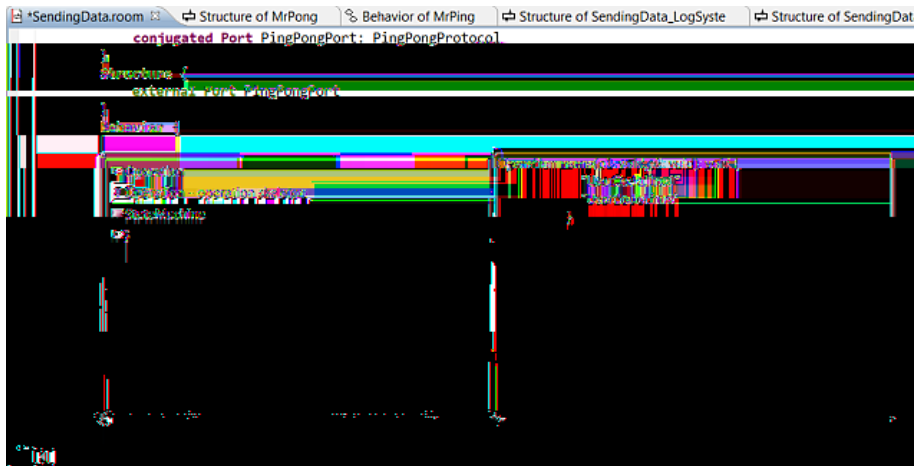


For *pingSimple*:



7.6.2 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.

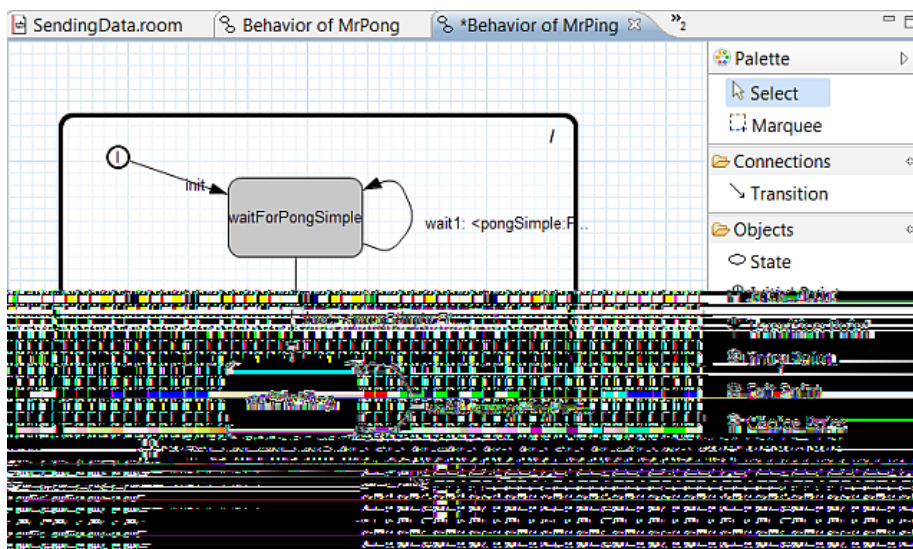


Name the operation *printData* and define the *DemoData* as a parameter.

Fill in the following code:

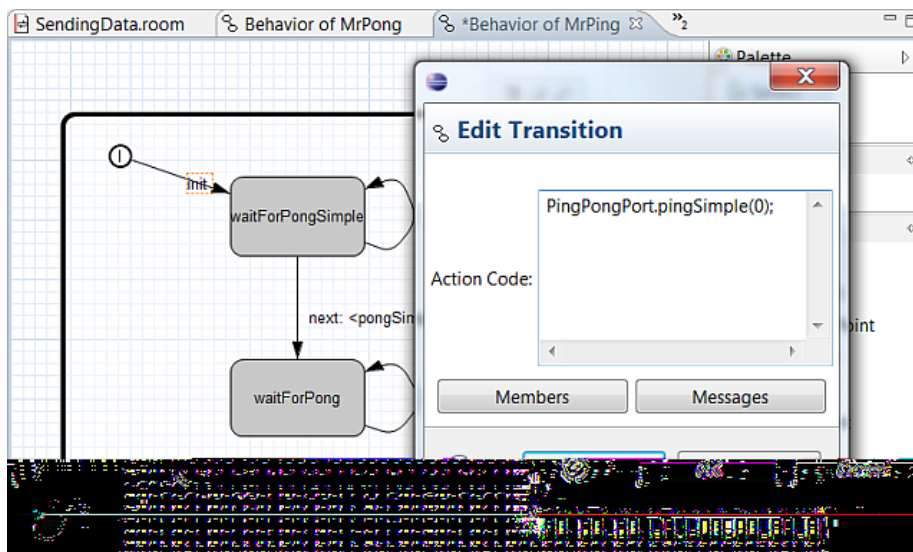
```
Operation printData(d: DemoData) : void {
    "System.out.printf(\"d.int32Val: %d\\n\",d.int32Val);"
    "System.out.printf(\"d.float64Val: %f\\n\",d.float64Val);"
    "System.out.printf(\"d.int8Array: \");"
    "for(int i = 0; i<d.int8Array.length; i++) {"
    "System.out.printf(\"%d \",d.int8Array[i]);}"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

For *MrPing* create the following state machine: (Remember that you can copy and paste the action code from the tutorial directory.)

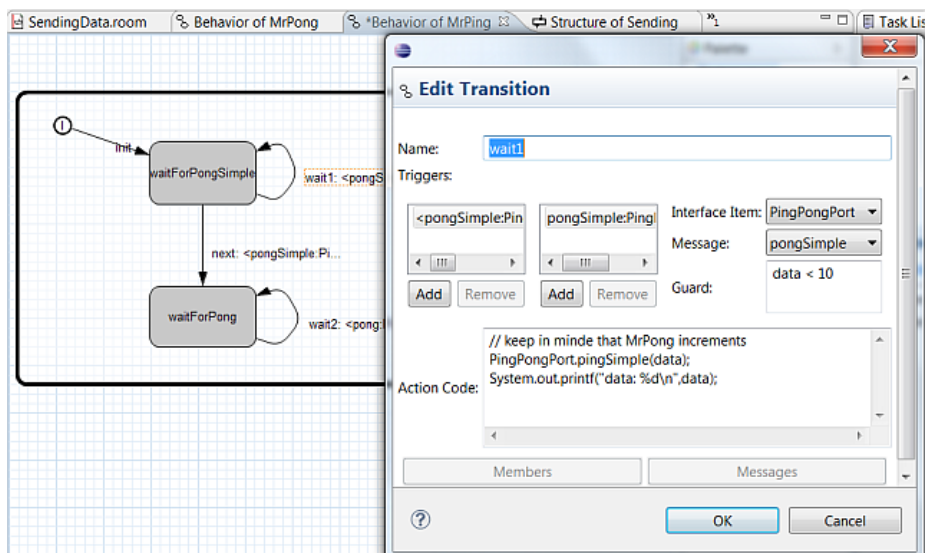


The transition dialogues should look like this:

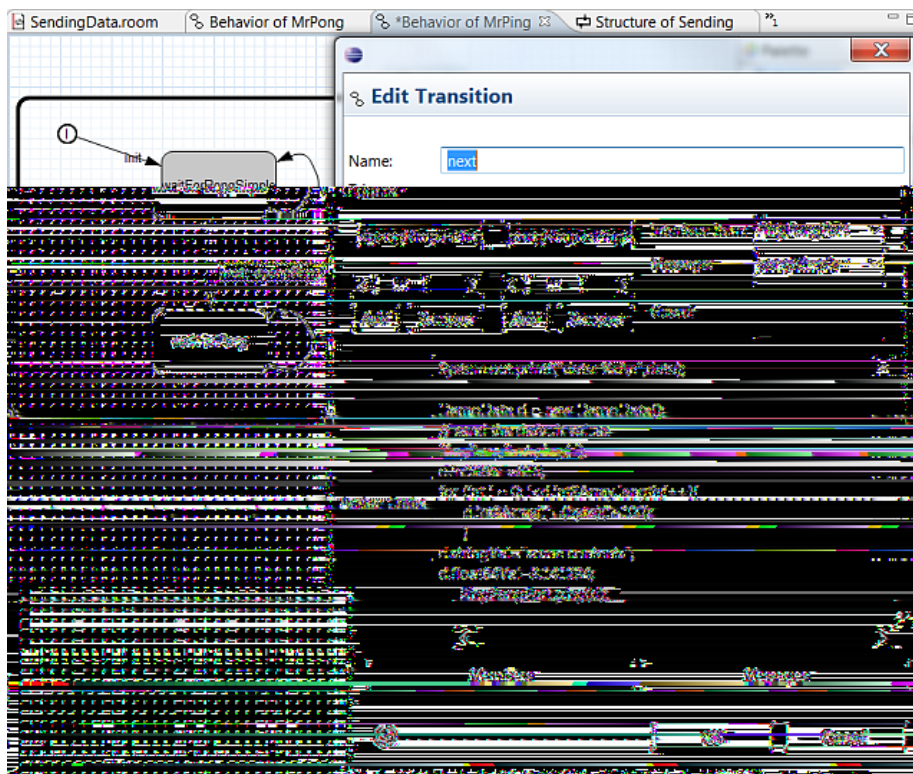
For *init*:



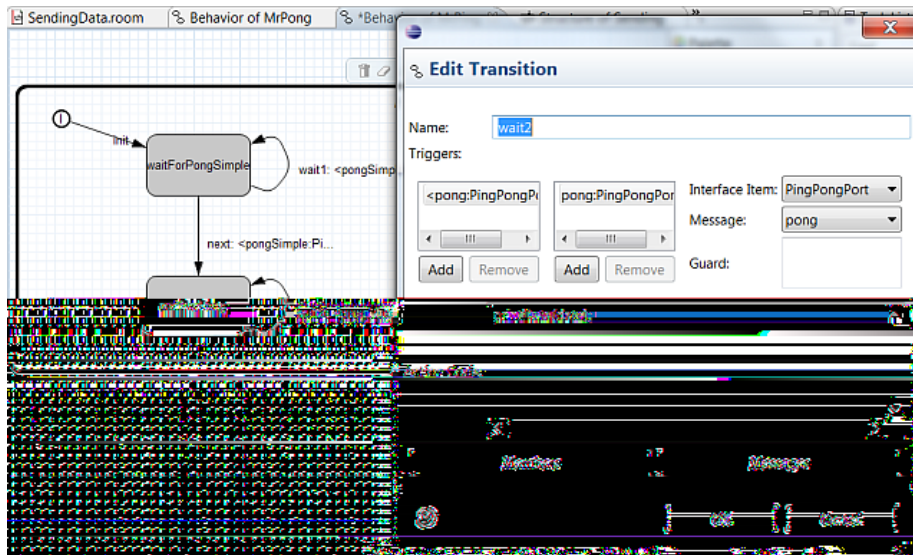
For *wait1*:



For *next*:

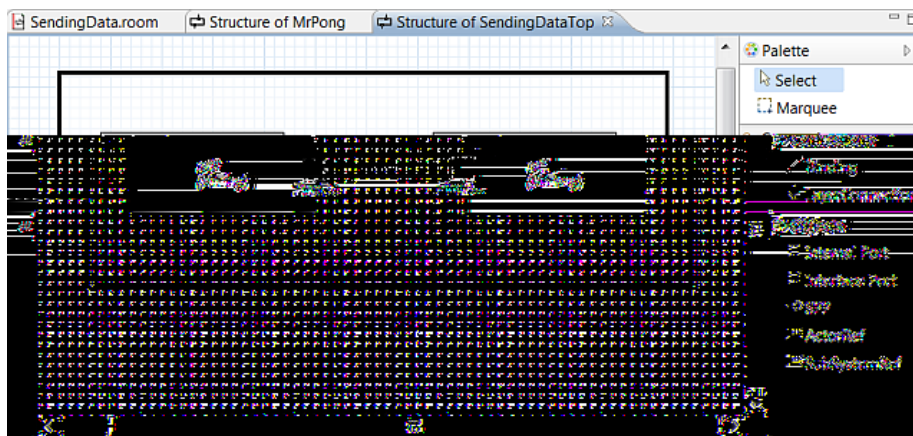


For *wait2*:



7.7 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 can be found in
[/org.eclipse.etrice.tutorials/model/SendingData](http://org.eclipse.etrice.tutorials/model/SendingData).

7.8 Generate and run the model

Generate the code by right click to **gen_SendingData.launch** and run it as **gen_SendingData**. 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

```

7.9 Summary

Within the first loop an integer value will be incremented by *MrPong* and sent back to *MrPing*. As 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. Then *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. This type of data passing is called *sending data by value*. However, for performance reasons some applications requires *sending data by reference*. In this case the user is responsible for the life cycle of the object. In Java the VM takes care of the life cycle of an object. This is not the case for C/C++. Consider that a object which is created within a transition of a state machine will be destroyed when the transition is finished. The receiving FSM would receive an invalid reference. Therefore care must be taken when sending references.

For sending data by reference you simply have to add the keyword *ref* to the protocol definition.

```
Message ping(data: DemoData ref)
```

Make the test and inspect the console output.

Chapter 8

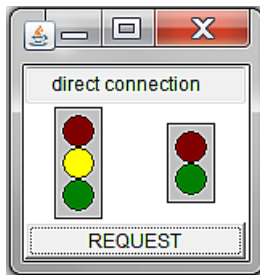
Tutorial Pedestrian Lights (Java)

8.1 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 by the model.

This tutorial is not like hello world or blinky. Being familiar with the basic tool features is mandatory 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 several 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 few steps are necessary:

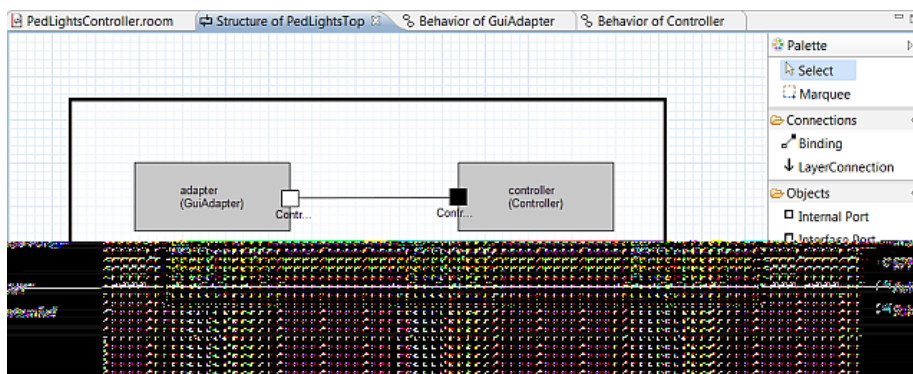
1. specify the messages (within a protocol) which should be sent into the model
2. model an actor with a port (which uses the specified protocol) and connect the port to the receiver
3. the external code should know the port (import of the port class)
4. the external code should provide a registration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

8.2 Setup the model

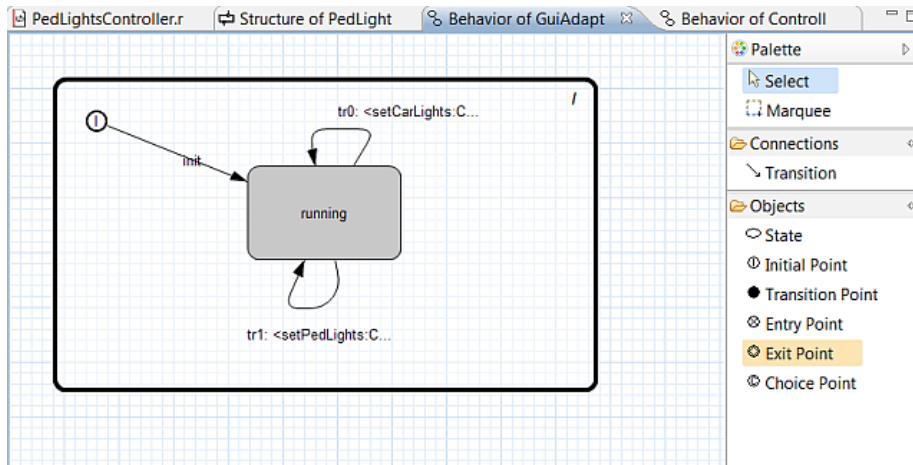
- Use the *New Model Wizard* to create a new eTrice project and name it *PedLightsController*.
- Copy the package *org.eclipse.etrice.tutorials.PedLightGUI* to your *src* directory (see blinky tutorial).
- In *PedestrianLightWndNoTcp.java* uncomment line 15 (import), 36, 122 (usage) and 132-134 (registration). The error markers will disappear after the code is generated from the model.
- Copy the model from
 /org.eclipse.etrice.tutorials/model/PedLightsController to your model file, or run the model directly in the tutorial directory.
- Adapt the import statement to your path.

```
import room.basic.service.timing.* from
"../../../../org.eclipse.etrice.modellib/models/TimingService.room"
```

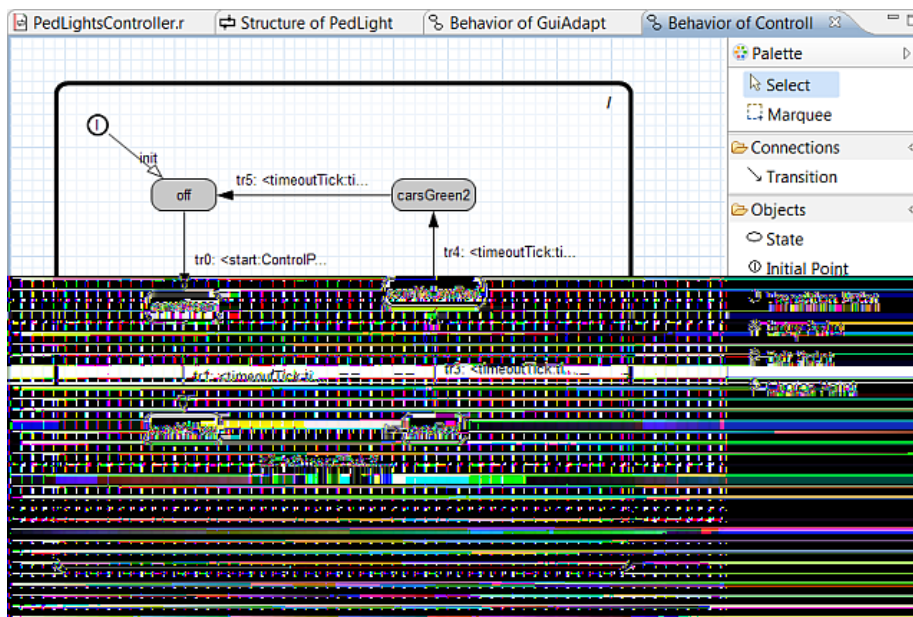
- Generate the code from the model.
- Add the *org.eclipse.etrice.modellib* to the Java Class Path of your project.
- All error markers should be disappeared and the model should be operable.
- Arrange the Structure and the Statemachines to understand the model



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.



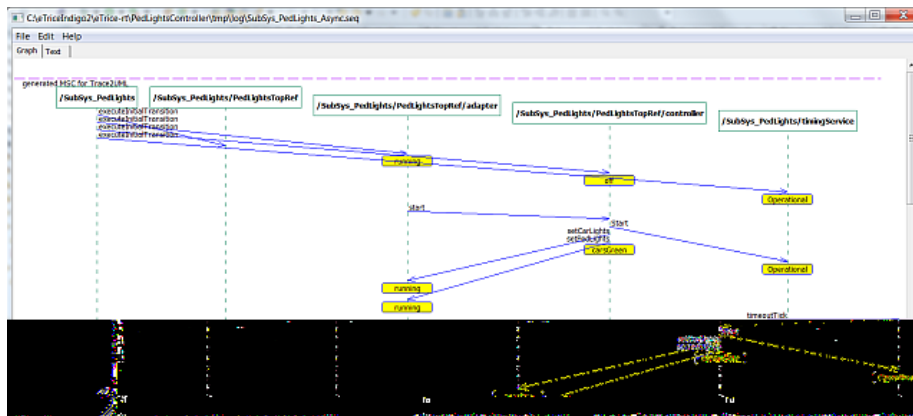
The *Controller* receives the *start* message and controls the timing of the lights. Note that the *start* message will be sent from the external code whenever the *REQUEST* button is pressed.

- 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.



8.3 Why does it work and why is it safe?

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 safe and does not violate 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 9

Tutorial Remove C-Comment (C)

9.1 Scope

In this tutorial you will create a more complex model. The model implements a simple parser that removes comments (block comments and line comments) from a C source file. Therefore we will create two actors. One actor is responsible to perform the file operations, while the second actor implements the parser.

You will perform the following steps:

1. create a new model from scratch for C
2. define a protocol
3. define your own data type
4. create the structure and the behavior by yourself
5. generate, build and run the model

Make sure that you have set up the workspace as described in *Setting up the Workspace for C Projects*.

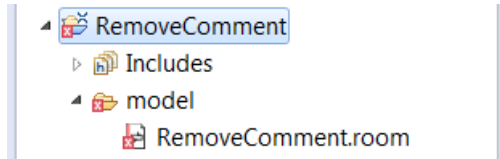
9.2 Create a new model from scratch

Remember the following steps from the previous tutorials:

- select the *C/C++* perspective
- From the main menu select *File->New->C Project*
- Name the project *RemoveComment*
- Project type is *Executable / Empty C Project*

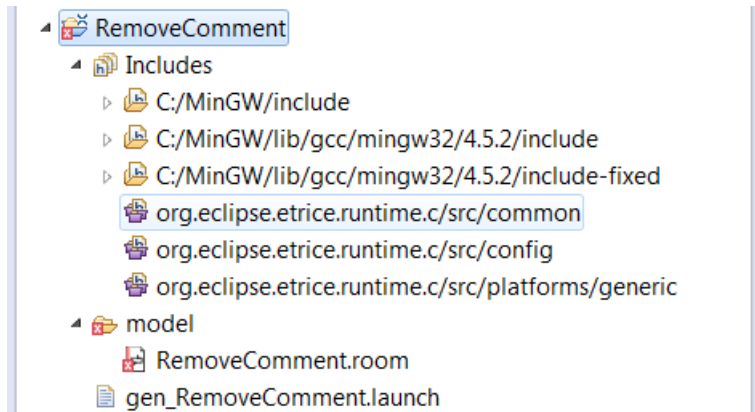
- Toolchain is *MinGW*
- Add the folder *model*
- Add the model file and name it *RemoveComment.room*
- Add the Xtext nature.

The workspace should look like this:



Create a launch configuration for the C generator and add the include path and library as described in *HelloWorldC*.

The workspace should look like this:



Now the model is created and all settings for the code generator, compiler and linker are done.

9.3 Create your own data type

The planned application should read a C source file and remove the comments. Therefore we need a file descriptor which is not part of the basic C types. The type for the file descriptor for MinGW is *FILE*. To make this type available on the model level, you have to declare the type.

Open the file *Types.room* from *org.eclipse.modellib.c* and take a look at the declaration of *string* (last line) which is not a basic C type.

```
PrimitiveType string:ptCharacter -> charPtr default "0"
```

With this declaration, you make the *string* keyword available on model level as a primitive type. This type will be translated to *charPtr* in your C sources. *charPtr* is defined in *etDatatypes.h*. This header file is platform specific (*generic*). With this mechanism you can define your own type system on model level and map the model types to specific target/platform types.

To not interfere with other models, we will declare the type direct in the model. Add the following line to your model:

```
RoomModel RemoveComment {
import room.basic.types.* from
"../../../../org.eclipse.etrice.modellib.c/model/Types.room"

PrimitiveType file:ptInteger -> FILE default "0"
```

FILE is the native type for MinGW. Therefore you don't need a mapping within *etDatatypes.h*. If your model should be portable across different platforms you should not take this shortcut.

9.4 Create the model

Due to the former tutorials you should be familiar with the steps to create the model with protocols, actors and state machines.

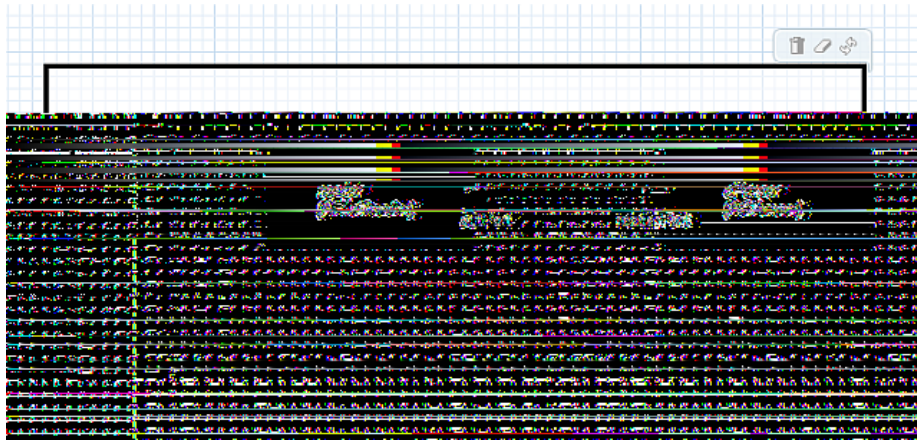
The basic idea of the exercise is to create a file reader actor, which is responsible to open, close and read characters from the source file. Another actor receives the characters and filters the comments (parser). The remaining characters (pure source code) should be print out.

Remember the logical steps:

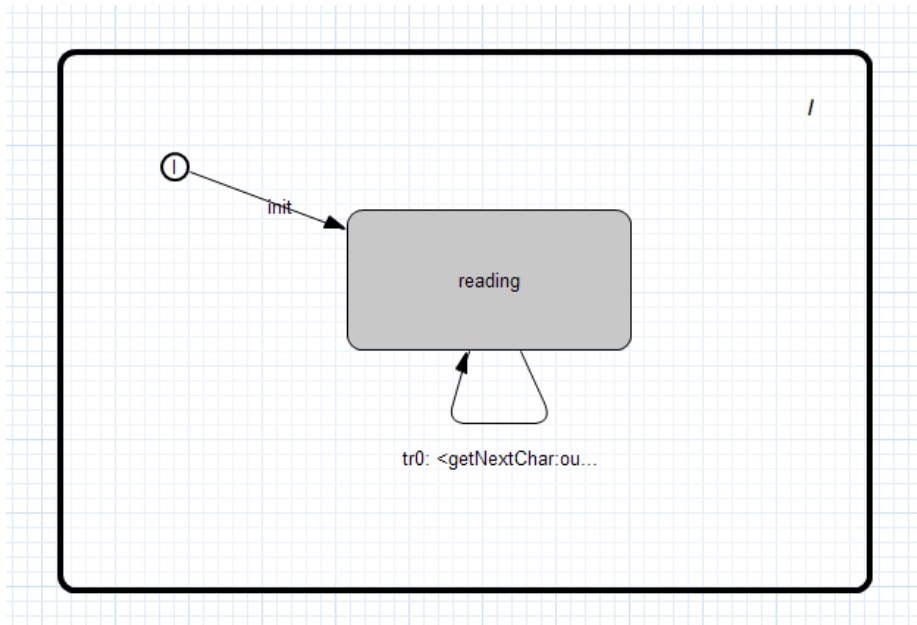
- create the model by the help of content assist (CTRL Space)
- name the model, subsystem and top level actor
- define the protocol (in this case it should be able to send a char, and to request the next char from the file reader)
- create the structure (file reader and parser with an appropriate port, create the references and connect the ports)
- create the state machines

Try to create the model by yourself and take the following solution as an example.

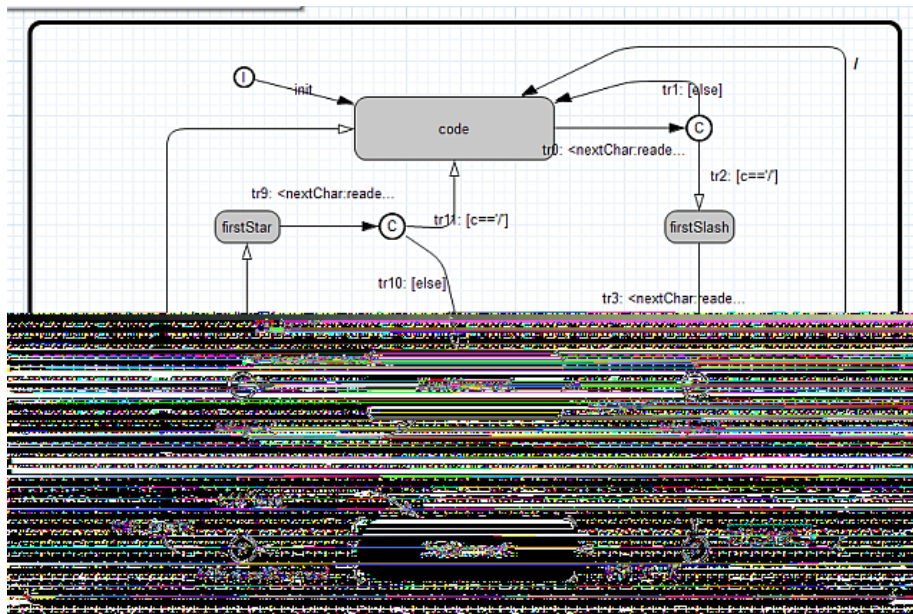
Structure:



File reader FSM:



Parser FSM:



The complete model can be found in `org.eclipse.etrice.tutorials.c`

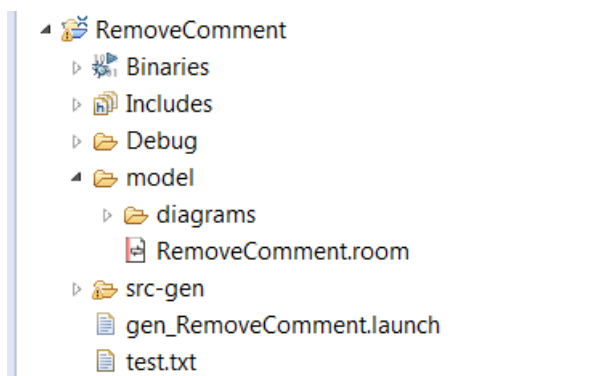
Take a look at the file attribute of the file reader.

Attribute `f:file ref`

`fopen` expects a `FILE *`. `f:file ref` declares a variable `f` from type reference to `file`, which is a pointer to `FILE`.

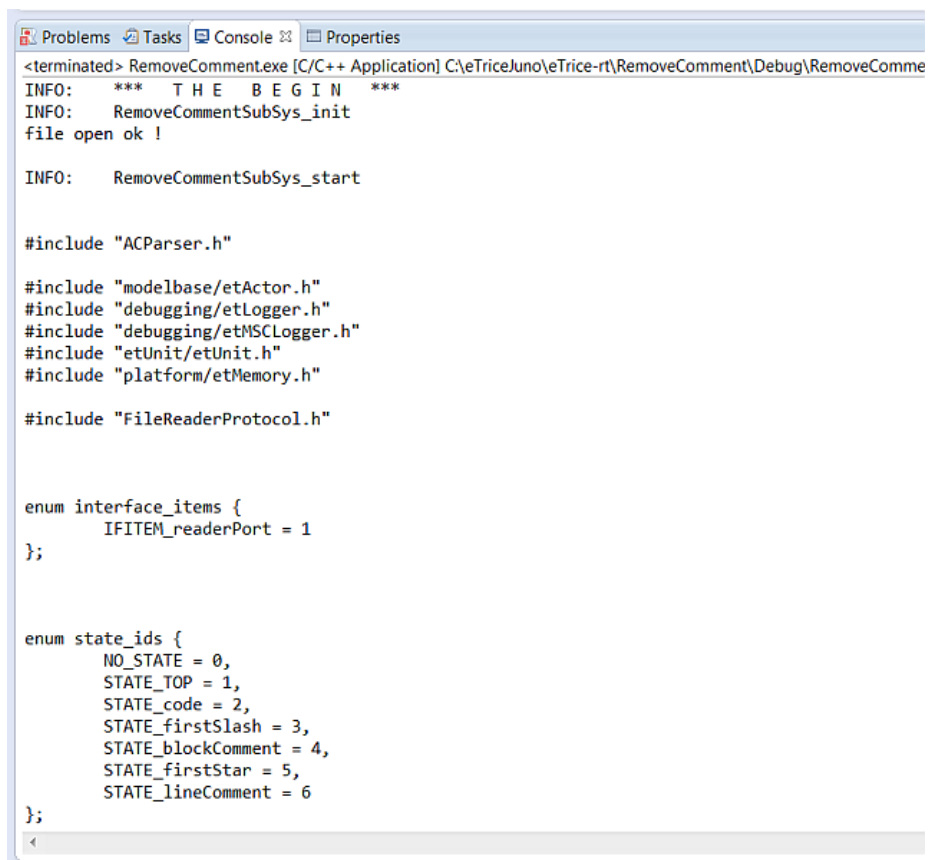
9.5 Generate, build and run the model

Before you can run the model you should copy one of the generated C source files into the project folder and name it `test.txt`.



Generate, build and run the model.

Your output should start like this:



```
<terminated> RemoveComment.exe [C/C++ Application] C:\eTriceJuno\etTrice-rt\RemoveComment\Debug\RemoveComme
INFO:  *** THE BEGIN ***
INFO:  RemoveCommentSubSys_init
file open ok !

INFO:  RemoveCommentSubSys_start

#include "ACParser.h"

#include "modelbase/etActor.h"
#include "debugging/etLogger.h"
#include "debugging/etMSCLogger.h"
#include "etUnit/etUnit.h"
#include "platform/etMemory.h"

#include "FileReaderProtocol.h"

enum interface_items {
    IFITEM_readerPort = 1
};

enum state_ids {
    NO_STATE = 0,
    STATE_TOP = 1,
    STATE_code = 2,
    STATE_firstSlash = 3,
    STATE_blockComment = 4,
    STATE_firstStar = 5,
    STATE_lineComment = 6
};
```

9.6 Summary

This tutorial should help you to train the necessary steps to create a C model. By the way you have seen how to create your own type system for a real embedded project. An additional aspect was to show how simple it is to separate different aspects of the required functionality by the use of actors and protocols and make them reusable.

Chapter 10

ROOM Concepts

This chapter gives an overview over the ROOM language elements and their textual and graphical notation. The formal ROOM grammar based on Xtext (EBNF) you can find here: [ROOM Grammar](#)

10.1 Actors

10.1.1 Description

The actor is the basic structural building block for building systems with ROOM. An actor can be refined hierarchically and thus can be of arbitrarily large scope. Ports define the interface of an actor. An Actor can also have a behavior usually defined by a finite state machine.

10.1.2 Motivation

- Actors enable the construction of hierarchical structures by composition and layering
- Actors have their own logical thread of execution
- Actors can be freely deployed
- Actors define potentially reusable blocks

10.1.3 Notation

10.1.4 Details

Actor Classes, Actor References, Ports and Bindings

An **ActorClass** defines the type (or blueprint) of an actor. Hierarchies are built by ActorClasses that contain **ActorReferences** which have another Ac-

Table 10.1: Actor Class Notation

Element	Graphical Notation	Textual Notation
ActorClass		<pre>ActorClass ActorClass2 { }</pre>
ActorRef		<pre>ActorClass ActorClass1 { Structure { ActorRef ActorReference: ActorClass2 } }</pre>

Table 10.2: Actor Class Example

Graphical Notation	Textual Notation
	<pre>ActorClass ActorClass1 { Interface { Port port1: ProtocolClass1 Port port4: ProtocolClass1 } Structure { external Port port1 conjugated Port port2: ProtocolClass1 conjugated Port port3: ProtocolClass1 ActorRef ActorRef_A: ActorClass2 ActorRef ActorRef_B: ActorClass3 Binding port2 and ActorRef_A.port5 Binding port3 and ActorRef_B.port6 Binding ActorRef_B.port7 and port4 Binding ActorRef_A.port8 and ActorRef_B.port9 } Behavior { } }</pre>

torClass as type. The interface of an ActorClass is always defined by Ports. The ActorClass can also contain Attributes, Operations and a finite state machine.

External Ports define the external interface of an actor and are defined in the **Interface** section of the ActorClass.

Internal Ports define the internal interface of an actor and are defined in the **Structure** section of the ActorClass.

Bindings connect Ports inside an ActorClass.

Example:

- **ActorClass1** contains two ActorReferences (of ActorClass2 and ActorClass3)
- *port1* is a **External End Port**. Since it connects external Actors with the behavior of the ActorClass, it is defined in the **Interface** section and the **Structure** section of the ActorClass.
- *port2* and *port3* are **Internal End Ports** and can only be connected to the ports of contained ActorReferences. Internal End Ports connect the Behavior of an ActorClass with its contained ActorReferences.
- *port4* is a relay port and connects external Actors to contained ActorReferences. This port can not be accessed by the behavior of the ActorClass.
- *port5* through *port9* are Ports of contained ActorReferences. *port8* and *port9* can communicate without interference with the containing ActorClass.

- **Bindings** can connect ports of the ActorClass and its contained ActorReferences.

Attributes

Attributes are part of the Structure of an ActorClass. They can be of a PrimitiveType or a DataClass.

Example:

```
ActorClass ActorClass3 {
  Structure {
    Attribute attribute1: int32 // Attribute of PrimitiveType
    Attribute attribute2: DataClass1 // Attribute of DataClass
  }
}
```

Operations

Operations are part of the Behavior of an ActorClass. Arguments and return values can be of a PrimitiveType or a DataClass. DataClasses can be passed by value (implicit) or by reference (keyword **ref**).

Example:

```
ActorClass ActorClass4 {
  Behavior {
    // no arguments, no return value
    Operation operation1(): void {
      "UserCodeLine1"
    }
    // argument of PrimitiveType, return value of of PrimitiveType
    Operation operation2(Param1: int32, Param2: float64): uint16 {
      "UserCodeLine1"
    }
    // arguments and return value by value
    Operation operation3(Param1: int32, Param2: DataClass1): DataClass1 {
      "UserCodeLine1"
    }
    // arguments and return value by reference, except for PrimitiveTypes
    Operation operation4(Param1: int32, Param2: DataClass1 ref): DataClass1 ref {
      "UserCodeLine1"
    }
  }
}
```

10.2 Protocols

10.2.1 Description

A ProtocolClass defines a set of incoming and outgoing messages that can be exchanged between two ports. The exact semantics of a message is defined by the execution model.

10.2.2 Motivation

- ProtocolClasses provide a reusable interface specification for ports
- ProtocolClasses can optionally specify valid message exchange sequences

10.2.3 Notation

ProtocolClasses have only textual notation. The example defines a ProtocolClass with 2 incoming and two outgoing messages. Messages can have data attached. The data can be of a primitive type (e.g. int32, float64, ...) or a DataClass.

```
ProtocolClass ProtocolClass1 {
  incoming {
    Message m1(data: int32)
    Message m2()
  }
  outgoing {
    Message m3(data: DataClass1)
    Message m4()
  }
}
```

10.3 Ports

10.3.1 Description

Ports are the only interfaces of actors. A port has always a protocol assigned. Service Access Points (SAP) and Service Provision Points (SPP) are specialized ports that are used to define layering.

10.3.2 Motivation

- Ports decouple interface definition (Protocols) from interface usage
- Ports decouple the logical interface from the transport

10.3.3 Notation

Class Ports

These symbols can only appear on the border of an actor class symbol.

Ports that define an external interface of the ActorClass, are defined in the *Interface*. Ports that define an internal interface are defined in the *Structure* (e.g. internal ports).

- **External End Ports** are defined in the Interface and the Structure
- **Internal End Ports** are only defined in the Structure
- **Relay Ports** are only defined in the Interface
- **End Ports** are always connected to the internal behavior of the ActorClass
- **Replicated Ports** can be defined with a fixed replication factor (e.g. *Port port18 [5]: ProtocolClass1*) or a variable replication factor (e.g. *Port port18[*]: ProtocolClass1*)

Table 10.3: Class Port Notation












Element	Graphical Notation	Textual Notation
Class End Port		<p>External Class End Port:</p> <pre> ActorClass ActorClass6 { Interface { Port port12: ProtocolClass1 } Structure { external Port port12 } } </pre> <p>Internal Class End Port:</p> <pre> ActorClass ActorClass6 { Interface { } Structure { Port port20: ProtocolClass1 } } </pre>
Conjugated Class End Port		<p>External Conjugated Class End Port:</p> <pre> ActorClass ActorClass6 { Interface { conjugated Port port13: ProtocolClass1 } Structure { external Port port13 } } </pre> <p>Internal Conjugated Class End Port:</p> <pre> ActorClass ActorClass6 { Interface { } Structure { conjugated Port port21: ProtocolClass1 } } </pre>
Class Relay Port		<pre> ActorClass ActorClass6 { Interface { Port port14: ProtocolClass1 } Structure { Port port14: ProtocolClass1 } } </pre>
Conjugated Class Relay Port		<pre> ActorClass ActorClass6 { Interface { conjugated Port port11: ProtocolClass1 } Structure { Port port11: ProtocolClass1 } } </pre>
Replicated Class End Port		<p>External Replicated Class End Port:</p> <pre> ActorClass ActorClass6 { Interface { Port port16 [2]: ProtocolClass1 } Structure { external Port port16 } } </pre> <p>Internal Replicated Class End Port:</p> <pre> ActorClass ActorClass6 { Interface { } Structure { Port port22 [2]: ProtocolClass1 } } </pre>
Conjugated Replicated Class End Port		<p>External Conjugated Replicated Class End Port:</p> <pre> ActorClass ActorClass6 { Interface { conjugated Port port17 [2]: ProtocolClass1 } Structure { external Port port17 } } </pre> <p>Internal Conjugated Replicated Class End Port:</p> <pre> ActorClass ActorClass6 { Interface { } Structure { conjugated Port port23 [2]: ProtocolClass1 } } </pre>
Replicated Class Relay Port		<pre> ActorClass ActorClass6 { Interface { Port port18 [2]: ProtocolClass1 } Structure { } } </pre>

Table 10.5: Title

Element	Graphical Notation	Textual Notation
Reference Port		<i>implicit</i>
Conjugated Reference Port		<i>implicit</i>
Replicated Reference Port		<i>implicit</i>
Conjugated Replicated Reference Port		<i>implicit</i>

Reference Ports

These symbols can only appear on the border of an ActorReference symbol. Since the type of port is defined in the ActorClass, no textual notation for the Reference Ports exists.

10.4 DataClass

10.4.1 Description

The DataClass enables the modeling of hierarchical complex datatypes and operations on them. The DataClass is the equivalent to a Class in languages like Java or C++, but has less features. The content of a DataClass can always be sent via message between actors (defined as message data in ProtocolClass).

10.4.2 Notation

Example: DataClass using PrimitiveTypes

```

DataClass DataClass1 {
  Attribute attribute1: int32 // attribute of PrimitiveType
  Attribute attribute2: float32 // attribute of PrimitiveType
  // no recursive, no nested value
  Operation operation1() void {
    messageData
  }
}

// summary of PrimitiveType, return value of operation1
Operation operation1() void {
  messageData
}

// summary of PrimitiveType, return value of operation2
Operation operation2() void {
  messageData
}

```

Example: DataClass using other DataClasses:

```

DataClass DataClass2 {
  Attribute attribute1: int32      // attribute of PrimitveType
  Attribute attribute2: DataClass1 // attribute of DataClass

  // arguments and return value by value
  Operation operation1(Param1: int32, Param2: DataClass1, Param3: DataClass1) {
    "end of op1"
  }

  // arguments and return value by reference, except for Primitivetypes
  Operation operation2(Param1: int32, Param2: DataClass1 ref: DataClass1 ref) {
    "end of op2"
  }
}

```

10.5 Layering

10.5.1 Description

In addition to the Actor containment hierarchies, Layering provides another method to hierarchically structure a software system. Layering and actor hierarchies with port to port connections can be mixed on every level of granularity.

1. an ActorClass can define a Service Provision Point (SPP) to publish a specific service, defined by a ProtocolClass
2. an ActorClass can define a Service Access Point (SAP) if it needs a service, defined by a ProtocolClass
3. for a given Actor hierarchy, a LayerConnection defines which SAP will be satisfied by (connected to) which SPP

10.5.2 Notation

10.6 Finite State Machines

10.6.1 Description

Definition from Wikipedia:

A finite-state machine (FSM) or finite-state automaton (plural: automata), or simply a state machine, is a mathematical model used to design computer programs and digital logic circuits. It is conceived as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition, this is called a transition. A particular FSM is defined by a list of the possible states it can transition to from each state, and the triggering condition for each transition.

In ROOM each actor class can implement its behavior using a state machine. Events occurring at the end ports of an actor will be

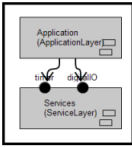
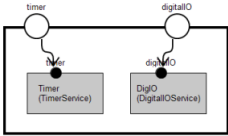
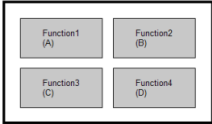




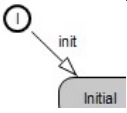
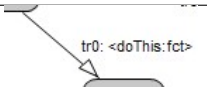
Description	Graphical Notation	Textual Notation
<p>The Layer Connections in this model define which services are provided by the <i>ServiceLayer</i> (<i>digitalIO</i> and <i>timer</i>)</p>		<pre> ActorClass Model { Structure { ActorRef Services: ServiceLayer ActorRef Application: ApplicationLayer LayerConnection ref Application satisfied_by Services.timer LayerConnection ref Application satisfied_by Services.digitalIO } } </pre>
<p>The implementation of the services (SPPs) can be delegated to sub actors. In this case the actor <i>ServiceLayer</i> relays (delegates) the implementation services <i>digitalIO</i> and <i>timer</i> to sub actors</p>		<pre> ActorClass ServiceLayer { Interface { SPP timer: TimerProtocol SPP digitalIO: DigitalIOProtocol } Structure { ActorRef Timer: TimerService ActorRef DigIO: DigitalIOService LayerConnection relay_sap timer satisfied_by Timer.timer LayerConnection relay_sap digitalIO satisfied_by DigIO.digitalIO } } ActorClass TimerService { Interface { SPP timer: TimerProtocol } } ActorClass DigitalIOService { Interface { SPP digitalIO: DigitalIOProtocol } } </pre>
<p>Every Actor inside the <i>ApplicationLayer</i> that contains an SAP with the same Protocol as <i>timer</i> or <i>digitalIO</i> will be connected to the specified SPP</p>		<pre> ActorClass ApplicationLayer { Structure { ActorRef Function1: A ActorRef Function2: B ActorRef Function3: C ActorRef Function4: D } } ActorClass A { Structure { SAP timerSAP: TimerProtocol } } ActorClass B { Structure { SAP timerSAP: TimerProtocol SAP digitalSAP: DigitalIOProtocol } } </pre>

Table 10.6: Title

Description	Graphical Notation	Textual Notation
State		<code>State SomeState</code>
InitialPoint		<i>implicit</i>
TransitionPoint		<code>TransitionPoint tp</code>
ChoicePoint		<code>ChoicePoint chp</code>
Initial Transition		<code>Transition init: initial -> Initial { }</code>
Triggered Transition		<pre>Transition tr0: Initial -> DoingThis { triggers { <doThis: fct> } }</pre>

forwarded to and processed by the state machine. Events possibly trigger state transitions.

10.6.2 Motivation

For event driven systems a finite state machine is ideal for processing the stream of events. Typically during processing new events are produced which are sent to peer actors.

We distinguish flat and hierarchical state machines.

10.6.3 Notation



Flat Finite State Machine

The simpler flat finite state machines are composed of the following elements:

Hierarchical Finite State Machine

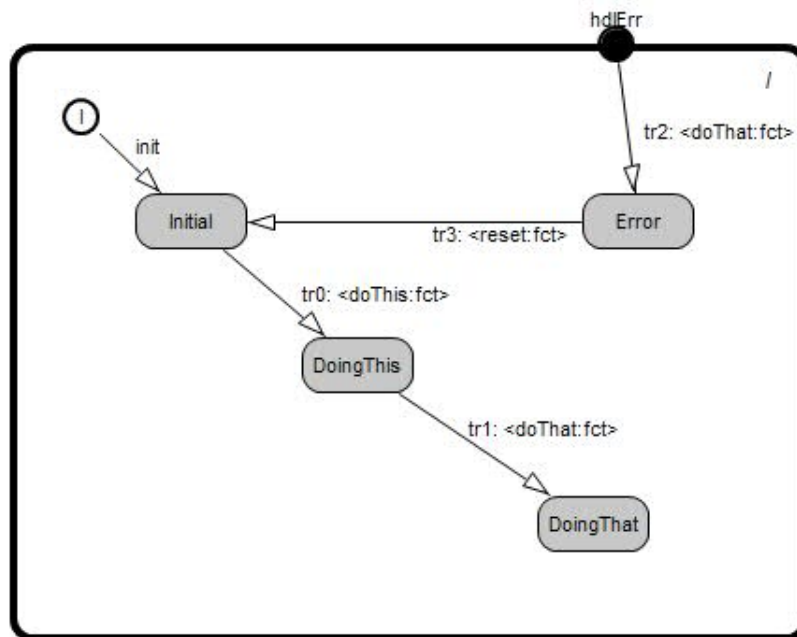
The hierarchical finite state machine adds the notion of a sub state machine nested in a state. A few modeling elements are added to the set listed above:

Table 10.7: Title

Description	Graphical Notation	Textual Notation
State with sub state machine	Parent State 	Sub state machine <pre> State Running { subgraph { Transition init: initial -> Process { } } State Process } </pre>
Entry Point	In sub state machine 	EntryPoint reInit
Exit Point		ExitPoint tp0

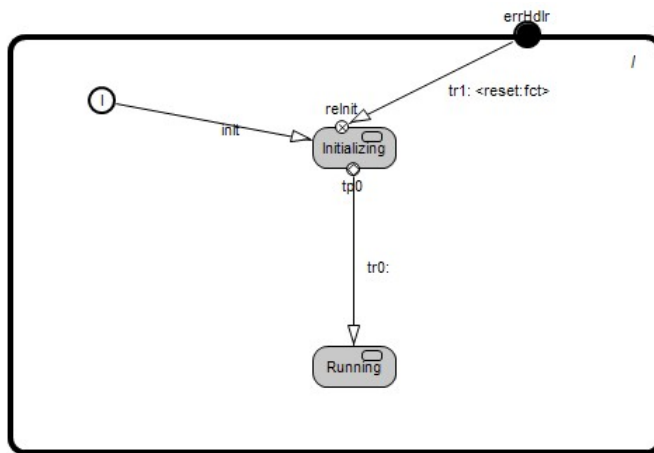
10.6.4 Examples

Example of a flat finite state machine:

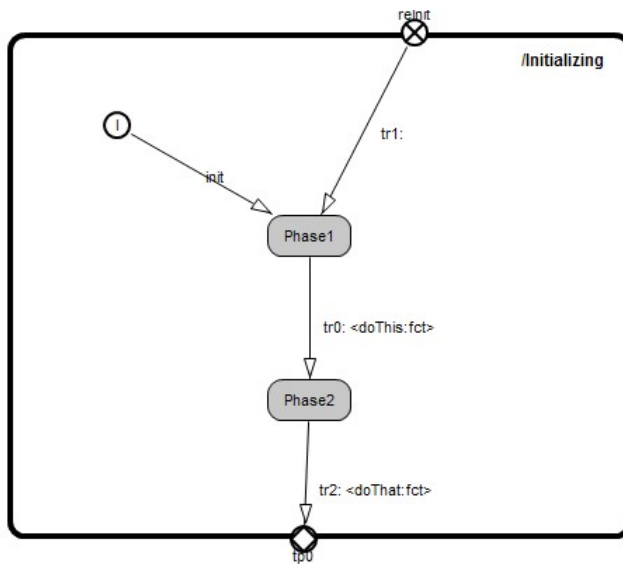


Example of a hierarchical finite state machine:

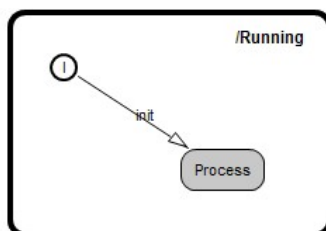
Top level



Sub state machine of Initializing



Sub state machine of Running



Chapter 11

eTrice Features

11.1 Codegenerators

11.1.1 Java Generator

11.1.2 C++ Generator

11.1.3 C Generator

Chapter 12

Codegenerators

Chapter 13

Runtimes

Chapter 14

eTrice Models and Their Relations

eTrice comprises several models:

- the ROOM model (*.room) – defines model classes and the logical structure of the model
- Config model (*.config) – defines configuration values for attributes
- Physical model (*.etphys) – defines the structure and properties of the physical system
- Mapping model (*.etmap) – defines a mapping from logical elements to physical elements

In the following diagram the models and their relations are depicted. The meaning of the arrows is: uses/references.

In the following sections we will describe those models with emphasis of their cross relations.

14.1 The ROOM Model

The ROOM model defines classes for Data, Protocols, Actors, SubSystems and LogicalSystems. Thereby the three latter form a hierarchy. The @LogicalSystem@ is the top level element of the structure. It contains references to SubSystemClass elements. The SubSystemClass in turn contain references to ActorClass elements which again contain (recursively) references to ActorClass elements. The complete structural hierarchy implies a tree which has the LogicalSystem as root and where each reference stands for a new node with possibly further branches.

Let's consider a simple example. It doesn't implement any meaningful and completely omits behavioral and other aspects.

```

RoomModel test {
  LogicalSystem Main {
    SubSystemRef subA: SubA
    SubSystemRef subB: SubB
  }

  SubSystemClass SubA {
    ActorRef actA: ActA
    ActorRef actB: ActB

    LogicalThread dflt
    LogicalThread extra
    ActorInstanceMapping actA/actB1 -> extra {
      ActorInstanceMapping actC1 -> dflt
    }
    ActorInstanceMapping actA/actB2 -> extra
  }

  SubSystemClass SubB {
    ActorRef actA: ActA
    ActorRef actB: ActB

    LogicalThread dflt
    LogicalThread extra
    ActorInstanceMapping actB -> extra
  }

  ActorClass ActA {
    Structure {
      Attribute val: int
      ActorRef actB1: ActB
      ActorRef actB2: ActB
    }
  }

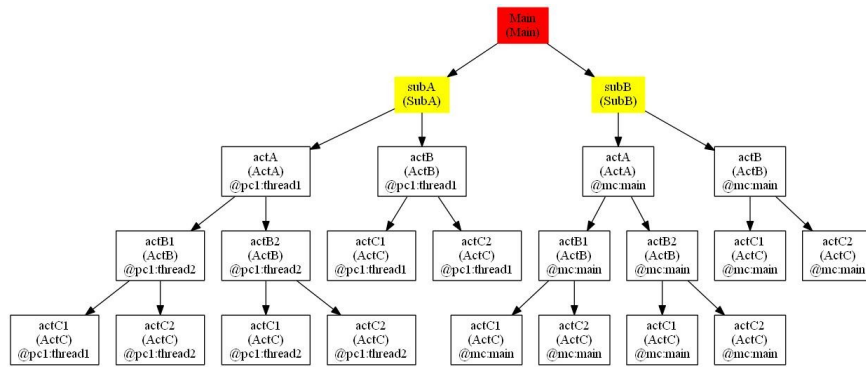
  ActorClass ActB {
    Structure {
      Attribute val: int
      ActorRef actC1: ActC
      ActorRef actC2: ActC
    }
  }

  ActorClass ActC {}

  PrimitiveType int: ptInteger -> int (Integer) default "0"
}

```

When a `LogicalSystem` is instantiated then recursively all of the contained referenced elements are instantiated as instances of the corresponding class. Thus the instance tree of above example looks like this (the third line in the white boxes shows some mapping information, s.b.):



14.2 The Config Model

Once we have the ROOM class model we can configure values using the Config model. This can be done on the class level and/or on the instance level. Values defined for class attributes are used for all instances unless there is an instance value configured for the same attribute.

```
import test.* from "test.room"

ActorClassConfig ActA {
  Attr val = 1
}

ActorClassConfig ActB {
  Attr val = 2
}

ActorInstanceConfig SubA / actA {
  Attr val = 12
}

ActorInstanceConfig SubA / actB {
  Attr val = 13
}

ActorInstanceConfig SubA / actA / actB2 {
  Attr val = 13
}
```

14.3 The Physical Model

The physical model defines the physical resources onto which the logical system will be deployed. It is possible to define runtime classes which (currently) only defines the overall execution model of the platform.

```
PhysicalModel runtimes {  
  RuntimeClass PCRuntime {  
    model = multiThreaded  
  }  
  
  RuntimeClass MSP430Runtime {  
    model = singleThreaded  
  }  
}
```

The physical system is composed of @Node@ references where each @Node@ is defined as a class referencing a @RuntimeClass@ and defining @Threads@.

```

PhysicalModel test {
  import runtimes.* from "runtimes.etphys"
  PhysicalSystem MainPhys {
    NodeRef pc1 : PC_Node
    NodeRef pc2 : PC_Node
    NodeRef mc : MSP430_Node
  }

  NodeClass PC_Node {
    runtime = PCRuntime
    priomin = 1
    priomax = 10
    DefaultThread thread1 {
      execmode = blocked
      prio = 10
      stacksize = 1024
      msgblocksize = 64
      msgpoolsize = 50
    }
    Thread thread2 {
      execmode = polled
      prio = 10
      stacksize = 1024
      msgblocksize = 64
      msgpoolsize = 50
    }
  }

  NodeClass MSP430_Node {
    runtime = MSP430Runtime
    priomin = 1
    priomax = 10
    DefaultThread main {
      execmode = polled
      prio = 10
      stacksize = 256
      msgblocksize = 64
      msgpoolsize = 50
    }
  }
}

```

14.4 The Mapping Model

The last model finally combines all this information by mapping logical to physical entities.

```
MappingModel test {
  import test.* from "test.room"
  import test.* from "test.etphys"

  Mapping Main -> MainPhys {
    SubSystemMapping subA -> pc1 {
      ThreadMapping dflt -> thread1
      ThreadMapping extra -> thread2
    }

    SubSystemMapping subB -> mc {
      ThreadMapping dflt -> main
      ThreadMapping extra -> main
    }
  }
}
```

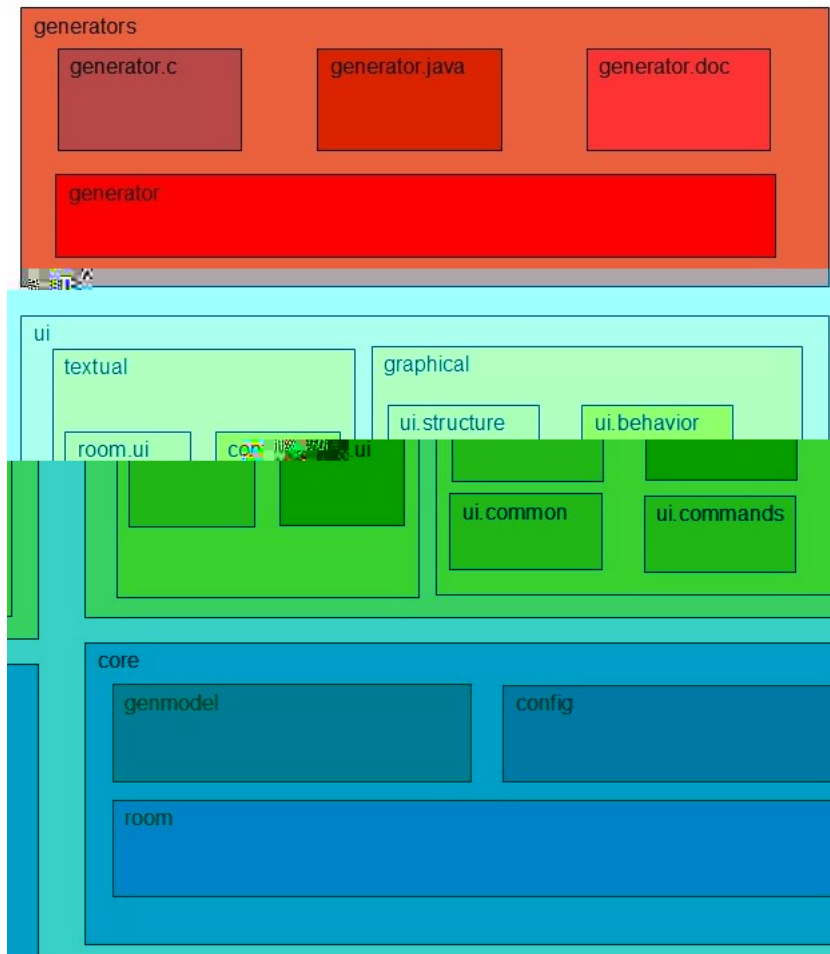
The result of the mapping is also depicted in above tree diagram of the instances. All actor instances (the white boxes) are mapped to a node and a thread running on this node (shown as @ *node* : *thread*).

Chapter 15

eTrice Developer's Reference

15.1 Architecture

The basic components of eTrice are depicted in the following diagram.



Additional to that the eTrice project comprises runtime libraries and unit tests which are treated in subsequent sections.

15.1.1 Editor and Generator Components

- core
 - core.room is an Xtext based language called Room. It consists of the plug-in *org.eclipse.etrice.core.room*. Room is the basic modeling language of eTrice.
 - core.config is an Xtext based language called Config. It consists of the plug-in *org.eclipse.etrice.core.config*. Config is a language designed for the data configuration of model elements. E.g. class and instance attributes can be specified.
 - core.genmodel is an EMF based aggregation layer for Room models. It consists of the plugin *org.eclipse.etrice.core.genmodel*. a Room model can be transformed into a genmodel which allows easy access to implicit relations of the Room model.

- ui
 - textual
 - * room.ui is the ui counterpart of core.room. It consists of the plug-in *org.eclipse.etrice.core.room.ui*. This plug-in realizes IDE concepts like content assist, error markers and navigation by hyper links for the Room language.
 - * config.ui is the ui counterpart of core.config. It consists of the plug-in *org.eclipse.etrice.core.config.ui*. This plug-in realizes IDE concepts like content assist, error markers and navigation by hyper links for the Config language.
 - graphical
 - * ui.common is a set of common code for the two diagram editors. It consists of the plug-in *org.eclipse.etrice.ui.common*.
 - * ui.commands encapsulates some commands related to the navigation between eTrice editors. It consists of the plug-in *org.eclipse.etrice.ui.commands*.
 - * ui.structure is the Graphiti based editor for the Actor structure. It consists of the plug-in *org.eclipse.etrice.ui.structure*.
 - * ui.behavior is the Graphiti based editor for the Actor behavior. It consists of the plug-in *org.eclipse.etrice.ui.behavior*.
- generators
 - generator is a set of general classes and language independent parts of all generators. It consists of the plug-in *org.eclipse.etrice.generator*.
 - generator.c is the generator for the ANSI-C target language. It consists of the plug-in *org.eclipse.etrice.generator.c*.
 - generator.java is the generator for the Java target language. It consists of the plug-in *org.eclipse.etrice.generator.java*.
 - generator.doc is the generator for the model documentation. It consists of the plug-in *org.eclipse.etrice.generator.doc*.

15.1.2 Runtimes

Currently eTrice ships with a C and a Java runtime. The runtimes are libraries

Model Tweaks

The Room EMF model is inferred from the grammar. However, this powerful mechanism has to be tweaked at some places. This is done in the `/org.eclipse.etrice.core.room/src/org/eclipse/etrice/core/RoomPostprocessor.ext` which is written in the legacy Xtend language.

The following parts of the model are changed or added:

- the default
`multiplicity`
of the `Port` is set to 1
- the operation `isReplicated` is added to the `Port`
- the default `size` of the `ActorRef` is set to 1
- an operation `getName` is add to the `State` class
- an operation `getName` is add to the `StateGraphItem` class
- an operation `getGeneralProtocol` is added to the `InterfaceItem`

Imports by URI Using Namespaces

The import mechanism employed is based on URIs. This is configured for one part in the `GenerateRoom.mwe2` model workflow by setting the fragments `ImportURIScopingFragment` and `ImportUriValidator`). For the other part it is configured in the Guice modules by binding

- `PlatformRelativeUriResolver` – this class tries to convert the import URI into a platform relative URI. It also replaces environment variables written in `$` with their respective values.
- `ImportedNamespaceAwareLocalScopeProvider` – this is a standard scope provider which is aware of namespaces
- `GlobalNonPlatformURIEditorOpener` – this editor opener tries to convert general URIs into platform URIs because editors can only open platform URIs
- `ImportAwareHyperlinkHelper` – turns the URI part of an import into a navigatable hyper link

Naming

Helpers

The `RoomHelpers` class provides a great deal of static methods that help retrieve frequently used information from the model. Among many, many others

- `getAllEndPorts(ActorClass)` - returns a list of all end ports of an actor class including inherited ones
- `getInheritedActionCode(Transition, ActorClass)` - get the inherited part of a transition's action code
- `getSignature(Operation)` - returns a string representing the operation signature suited for a label

Validation

Validation is used from various places. Therefore all validation code is accumulated in the `@ValidationUtil@` class. All methods are static and many of them return a `Result` object which contains information about the problem detected as well as object and feature as suited for most validation purposes.

15.2.2 Config Language Overview

Model Tweaks

A couple of operations are added to the `ConfigModel`

- `getActorClassConfigs`
- `getActorInstanceConfigs`
- `getProtocolClassConfigs`
- `getSubSystemConfigs`

Imports by URI Using Namespaces

Imports are treated like in Room language, section *Imports by URI Using Namespaces*.

Util

A set of static utility methods can be found in the `ConfigUtil` class.

15.2.3 Aggregation Layer Overview

The eTrice Generator Model (`genmodel`) serves as an aggregation layer. Its purpose is to allow easy access to information which is implicitly contained in the Room model but not simple to retrieve. Examples of this are the state machine with inherited items or a list of all triggers active at a state in the

order in which they will be evaluated or the actual peer port of an end port (following bindings through relay ports).

The Generator Model is created from a list of Room models by a call of the

```
createGeneratorModel(List<RoomModel>, boolean)
```

method of the `GeneratorModelBuilder` class.

The `Root` object of the resulting Generator Model provides chiefly two things:

- a tree of instances starting at each `SubSystem` with representations of each `ActorInstance` and `PortInstance`
- for each `ActorClass` a corresponding `ExpandedActorClass` with an explicit state machine containing all inherited state graph items

The Instance Model

The instance model allows easy access to instances including their unique paths and object IDs. Also it is possible to get a list of all peer port instances for each port instance without having to bother about port and actor replication.

The Expanded Actor Class

The expanded actor class contains, as already mentioned, the complete state machine of the actor class. This considerably simplifies the task of state machine generation. Note that the generated code always contains the complete state machine of an actor. I.e. no target language inheritance is used to implement the state machine inheritance. Furthermore the `ExpandedActorClass` gives access to

- `getIncomingTransitions(StateGraphNode)` – the set of incoming transition of a `StateGraphNode` (`State`, `ChoicePoint` or `TransitionPoint`)
- `getOutgoingTransitions(StateGraphNode)` – the set of outgoing transition of a `StateGraphNode`
- `getActiveTriggers(State)` – the triggers that are active in this `State` in the order they are evaluated

Transition Chains

By transition chains we denote a connected subset of the (hierarchical) state machine that starts with a transition starting at a state and continues over transitional state graph nodes (choice points and transition points) and continuation transitions until a state is reached. In general a transition chain starts at one state and ends in several states (the chain may branch in choice points). A `TransitionChain` of a transition is retrieved by a call of `getChain(Transition)` of the `ExpandedActorClass`. The `TransitionChain` accepts an `ITransitionChainVisitor` which is called along the chain to generate the action codes of involved transitions and the conditional statements arising from the involved choice points.

15.2.4 Generator Overview

There is one plug-in that consists of base classes and some generic generator parts which are re-used by all language specific generators

Base Classes and Interfaces

We just want to mention the most important classes and interfaces.

- **ITranslationProvider** — this interface is used by the `DetailCodeTranslator` for the language dependent translation of e.g. `port.message()` notation in detail code
- **AbstractGenerator** — concrete language generators should derive from this base class
- **DefaultTranslationProvider** — a stub implementation of **ITranslationProvider** from which clients may derive
- **Indexed** — provides an indexed iterable of a given iterable
- **GeneratorBaseModule** — a Google Guice module that binds a couple of basic services. Concrete language generators should use a module that derives from this

Generic Generator Parts

The generic generator parts provide code generation blocks on a medium granularity. The language dependent top level generators embed those blocks in a larger context (file, class, ...). Language dependent low level constructs are provided by means of an **ILanguageExtension**. This extension and other parts of the generator be configured using Google Guice dependency injection.

GenericActorClassGenerator The `GenericActorClassGenerator` generates constants for the interface items of a actor. Those constants are used by the generated state machine.

GenericProtocolClassGenerator The `GenericProtocolClassGenerator` generates message ID constants for a protocol.

GenericStateMachineGenerator

The `GenericStateMachineGenerator` generates the complete state machine implementation. The skeleton of the generated code is

- definition state ID constants
- definition of transition chain constants
- definition of trigger constants

- entry, exit and action code methods
- the `exitTo` method
- the `executeTransitionChain` method
- the `enterHistory` method
- the `executeInitTransition` method
- the `receiveEvent` method

The state machine works as follows. The main entry method is the `receiveEvent` method. This is the case for both, data driven (polled) and event driven state machines. Then a number of nested switch/case statements evaluates trigger conditions and derives the transition chain that is executed. If a trigger fires then the `exitTo` method is called to execute all exit codes involved. Then the transition chain action codes are executed and the choice point conditions are evaluated in the `executeTransitionChain` method. Finally the history of the state where the chain ends is entered and all entry codes are executed by `enterHistory`.

The Java Generator

The Java generator employs the generic parts of the generator. The `JavaTranslationProvider` is very simple and only handles the case of sending a message from a distinct replicated port: `replPort[2].message()`. Other cases are handled by the base class by returning the original text.

The `DataClassGen` uses Java inheritance for the generated data classes. Otherwise it is pretty much straight forward.

The `ProtocolClassGen` generates a class for the protocol with nested static classes for regular and conjugated ports and similar for replicated ports.

The `ActorClassGen` uses Java inheritance for the generated actor classes. So ports, SAPs and attributes and detail code methods are inherited. Not inherited is the state machine implementation.

The ANSI-C Generator

The C generator translates data, protocol and actor classes into structs together with a set of methods that operate on them and receive a pointer to those data (called `self` in analogy to the implicit C++ `this` pointer). No dynamic memory allocation is employed. All actor instances are statically initialized. One of the design goals for the generated C code was an optimized footprint in terms of memory and performance to be able to utilize modeling with ROOM also for tiny low end micro controllers.

The Documentation Generator

The documentation generator creates documentation in LaTeX format which can be converted into PDF and many other formats.