eTrice

eTrice committers and contributors

July 12, 2013

Contents

Chapter 1

Introduction

1.1 eTrice Overview

1.1.1 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.

1.1.2 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 state machines with inheritance
- team work 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)

1.2   Introduction to the ROOM Language

1.2.1   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 object oriented programming and ROOM?

- What are the benefits of ROOM?

- Which consequences must be taken into account?

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 *ObjecTime*^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.

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 to 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 within 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.

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. The classical 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.

But 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).
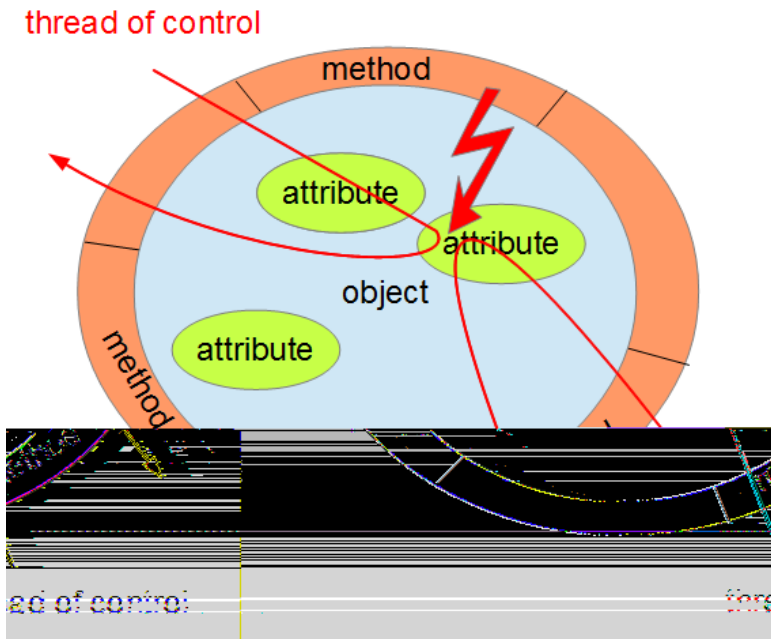
ROOM provides the concept of a logical machine (called actor) with its own thread of control. It provides some kind of cooperative communication infrastructure with *run to completion* semantics. That makes developing of business logic easy and safe (see 1.2.2 Basic Concepts). The logical machine provides an encapsulation shell including concurrency issues (see 1.2.2 Run to Completion).
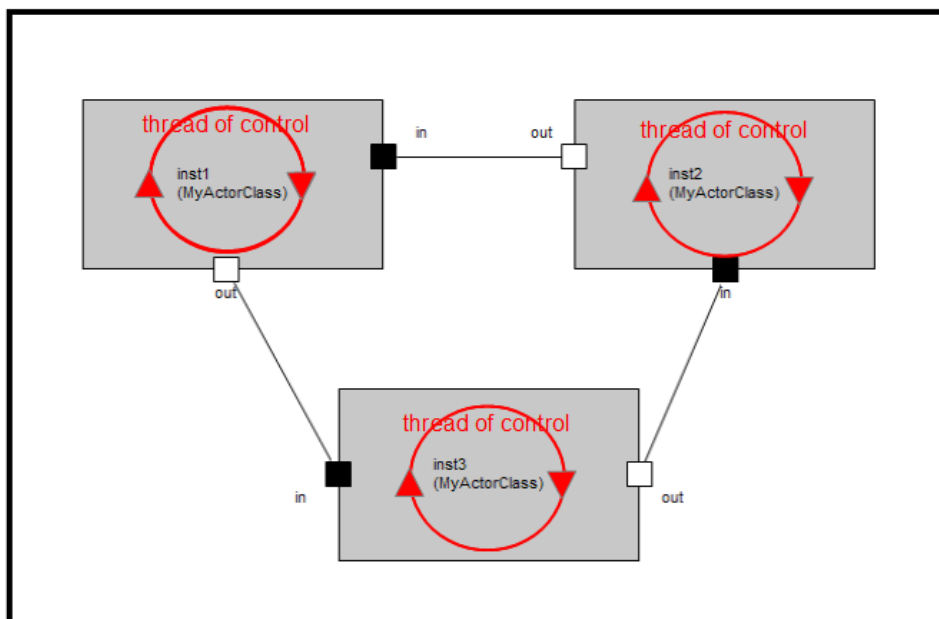
This thinking of an object is much more general than the classic one.

Table 1.1: Actor and Protocol Class Example

Actor with sub actors	Protocol definition
|  | ```
ProtocolClass ProtocolClass1 {
    incoming {
        Message m1(data: int32)
        Message m2()
    }
    outgoing {
        Message m3(data: DataClass1)
        Message m4()
    }
}
``` |

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 e cient, robust and safe due to the fact that it provides some abstract, high level modeling concepts combined with code generation and a small e cient runtime environment.

In detail:

- ROOM models contain well defined interfaces (protocols), which makes it easy to re-use components in di erent 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 e cient 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.

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 o between development costs, time to market and costs in 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.

1.2.2 Basic Concepts

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 actor's behavior (state machine) or will be delegated to the actor's internal structure.

The actor provides access protection for its own attributes (including complex types, i.e. 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 re-usability on the actor level and provides an e ective 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.

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 chapter 3 ROOM Concepts).

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 a 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 actor's point of view, SAPs and SPPs behave almost like 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.

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 by the underlying run time system.

Note: It is very important not to confuse *run to completion* and *cooperative multi threading*. 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 into several actors without the risk to produce access violations or dead locks.

1.2.3   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 supports the **message driven**, the **data driven** and a mixture of both execution models. In future releases maybe also a synchronous execution model will be supported, depending on the requirements of the community.

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.

eTrice currently supports the two former communication methods.

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

eTrice currently supports the two former execution methods.

Execution Models

In present-day's 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 *function calls*. This model in general is not very well suited to support the *run to completion* semantics 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 2

Tutorials

2.1 Working with the tutorials

The eTrice tutorials will help you to learn and understand the eTrice tool and its concepts. eTrice supports several target languages. The first two tutorials are target language specific. The other tutorials work for all target languages. Target language specific aspects are explained for all languages. Currently eTrice supports Java and C. C++ generator and runtime are currently prototypes with no tutorials. You should decide for which target language you want to run the tutorials.

Here an overview over the tutorials:

eTrice generates code out of ROOM models. The generated code relies on the services of a runtime framework (Runtime):

- execution

- communication (e.g. messaging)

- logging

- operating system abstraction (osal)

Additional functionality is provided as model library (Modellib):

- socket server and client

- timing service

- standard types

All tutorial models are provided as examples.

The Runtime, Modellib and Tutorial projects are target language specific and will be set up in the first tutorial "Setting up the workspace for ...".

11

2.2 Setting up the Workspace for Java Projects

Objectives for this tutorial:

- create all needed library projects (runtime.java and modellib.java)

- create the tutorial project with the examples

- create the project with a tra c light simulator

- test the workspace setup by running one of the examples

2.2.1 Create Library, Tutorial and Simulator Projects

After installation of eclipse and the eTrice plug in, your workspace should look like this:

Just the eTrice menu item is visible of the installed eTrice plugins.

Select the menu *File->New->Other*

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

2.2.2 Perform Setup Test

To check the correct setup of your workspace we run a little testproject contained in the tutorial project.

The tutorial models are available in the *org.eclipse.etrice.tutorials.java* project. All tutorials are ready to generate and run without any changes. To test the code generator and the workspace setup simply run *gen_SetupTestJava.launch* as *gen_SetupTestJava*:

The successful generation ends with *Info: − finished code generation* in the Console.

```
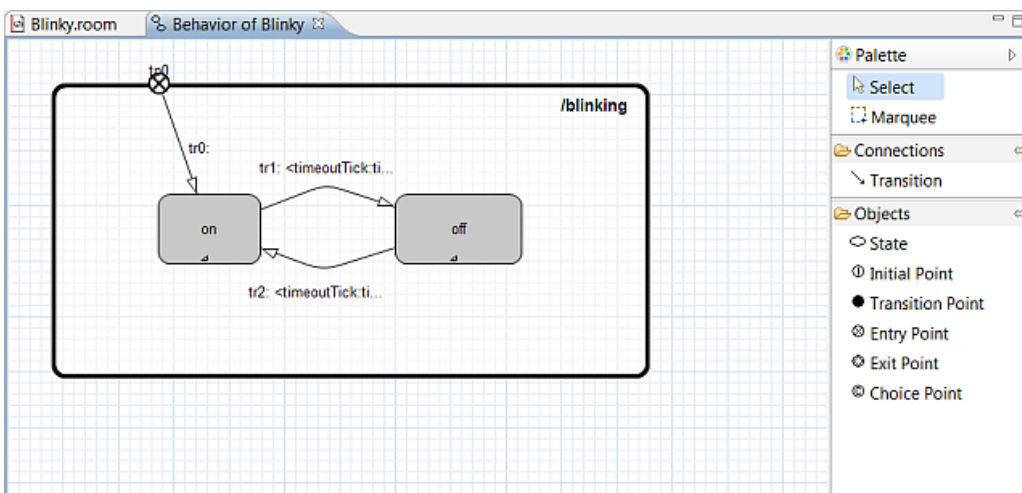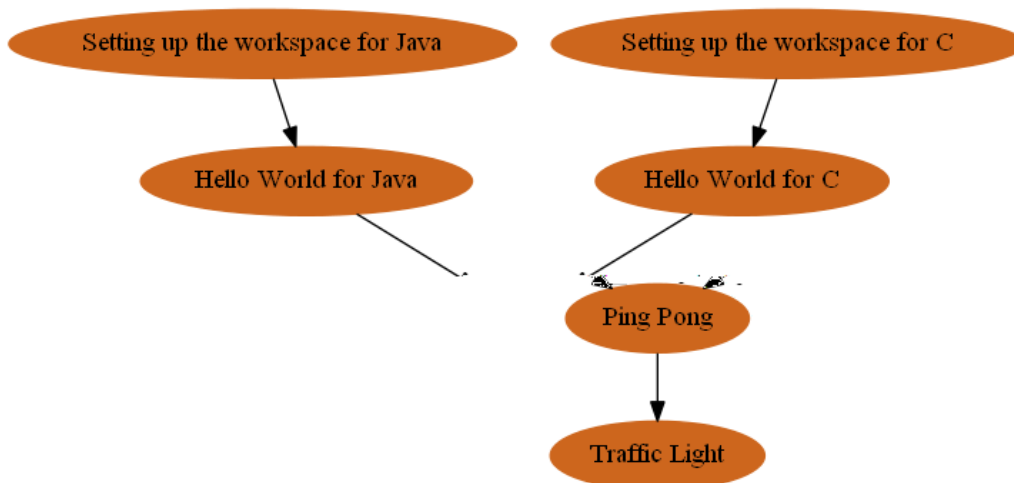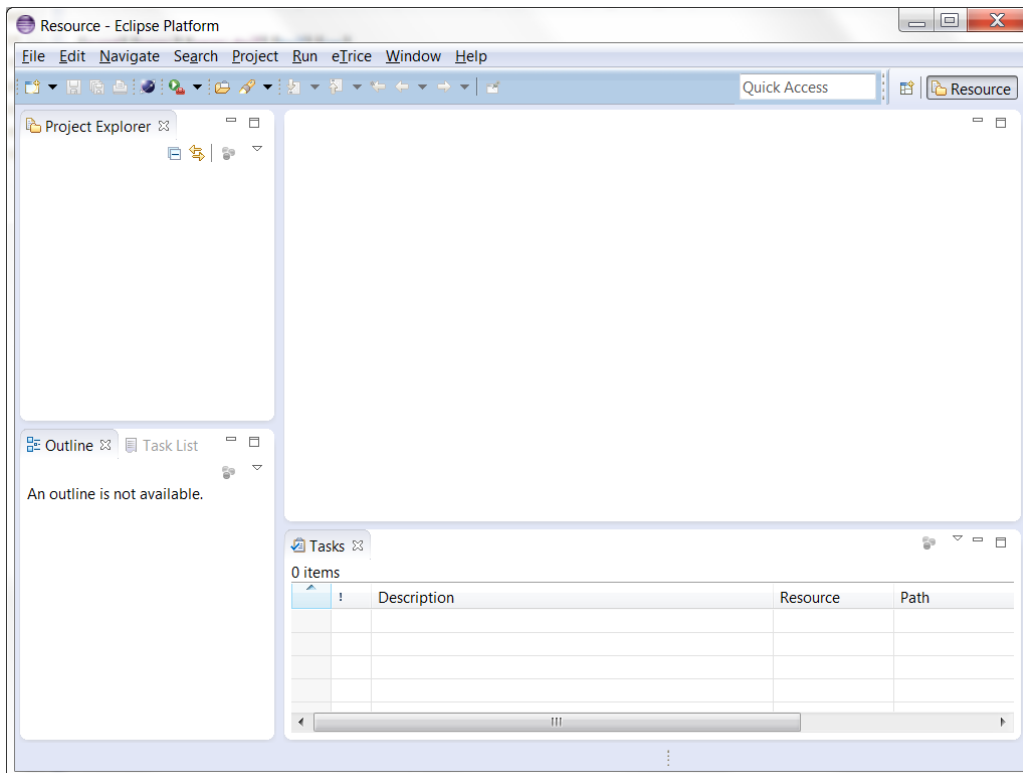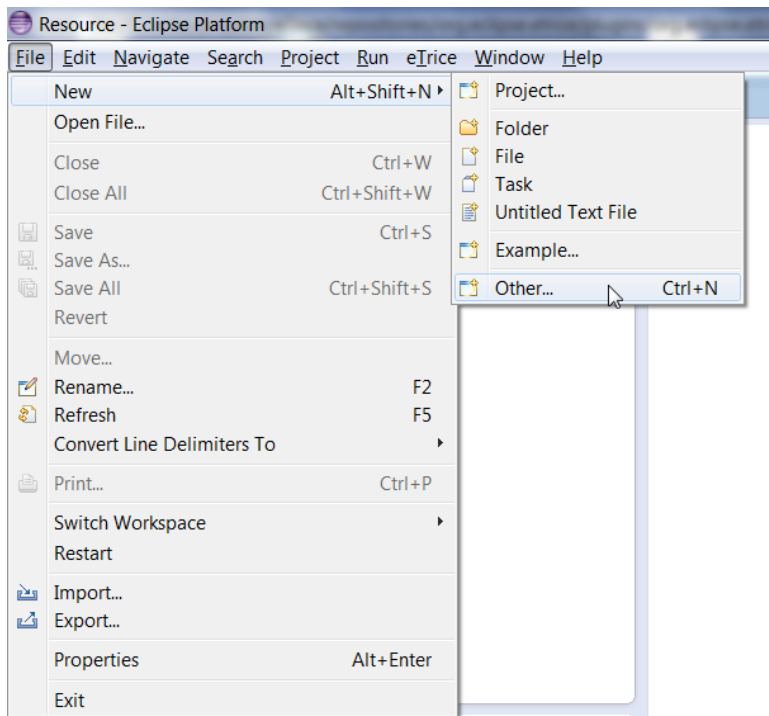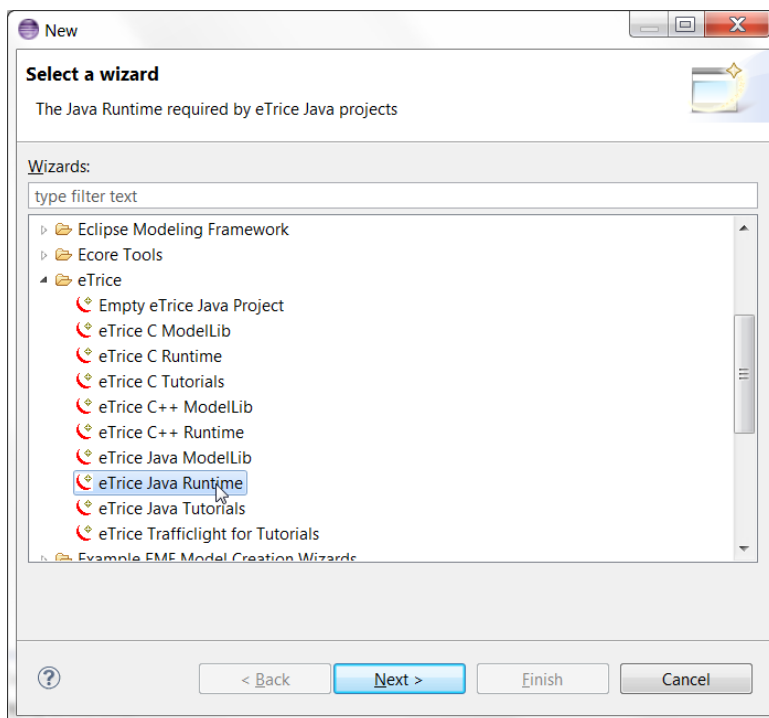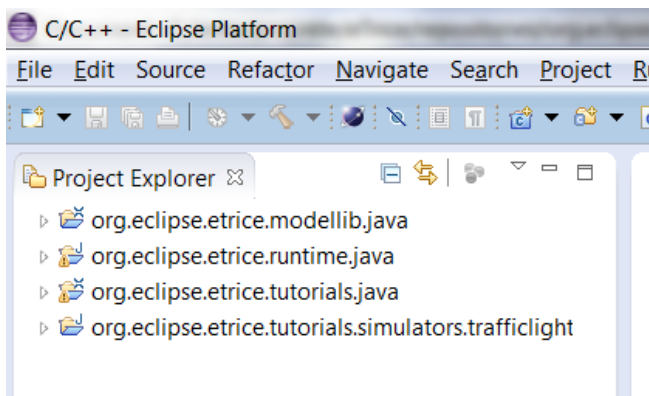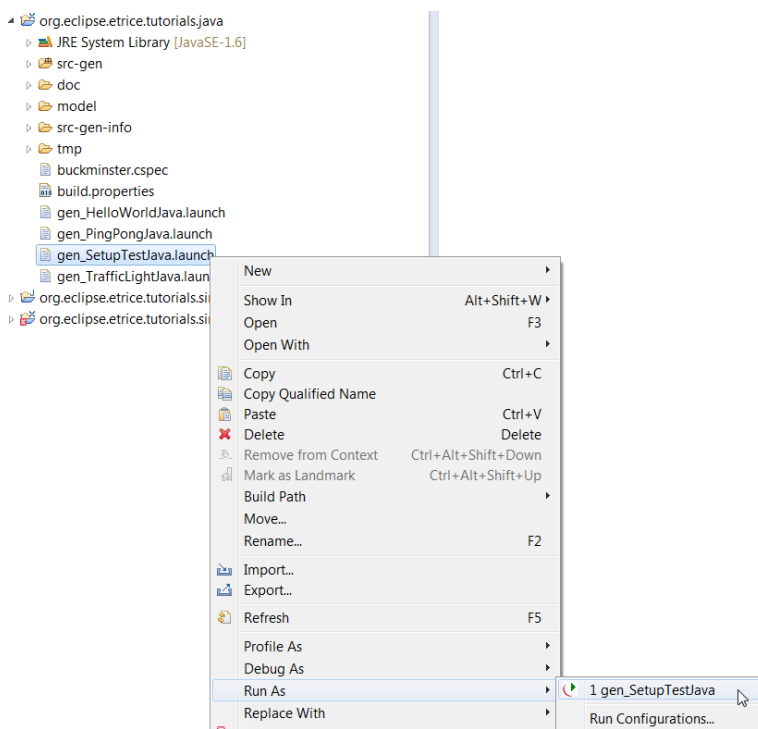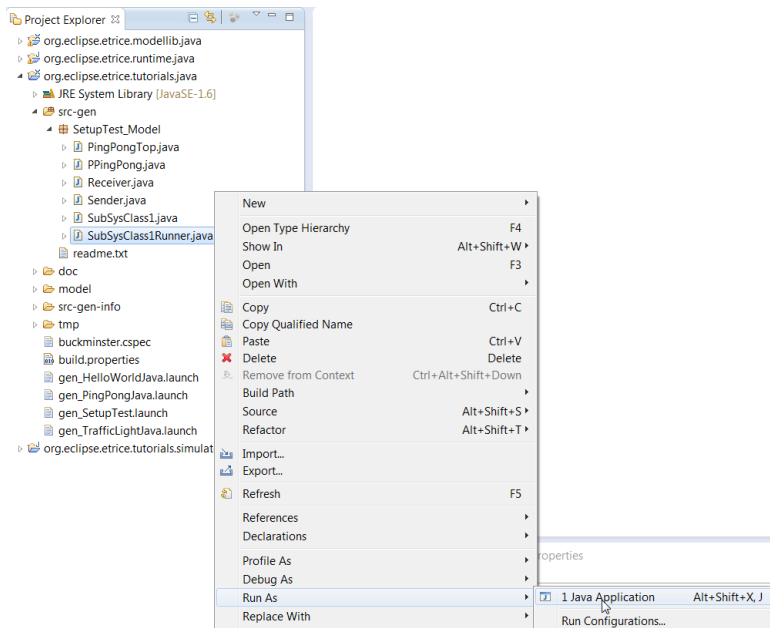 Tasks  Console ✕  Problems
eTrice Java Generator Console
Info: File found ! C:/entw/projekte/eTrice/workspace
Info: generating LaTeX documentation: 'room.basic.ty
Info: generating LaTeX documentation: 'room.basic.se
Info: File found ! C:/entw/projekte/eTrice/workspace
Info: File found ! C:/entw/projekte/eTrice/workspace
Info: File found ! C:/entw/projekte/eTrice/workspace
Info: File found ! C:/entw/projekte/eTrice/workspace
Info: -- finished code generation
```

For each tutorial in the folder src-gen a java package is generated including a java file called *SubSys<...>Runner.java*
. To run the generated application simply run this file as a java application. Try this with the file *src-gen/SetupTest_Model/SubSysClass1Runner.java* :

To stop the application type *quit* in the console window. If your Console contains the lines

```
* * * * * * * * * * * * * * * * * *
*** Setup OK ***
* * * * * * * * * * * * * * * * * *
```

your setup should be ok.

```
 Problems   Tasks  Console ✕  Properties
<terminated> SubSysClass1Runner [Java Application] C:\Program Files (x86)\Java\jdk1.6.0_26\bin\jav
***   T H E   B E G I N   ***
*** MainComponent /LogSys1/subSysRef1::init ***
type 'quit' to exit
******************
*** Setup OK ***
*****************
quit
echo: quit
*** MainComponent /LogSys1/subSysRef1::stop ***
=== done stop MsgSvcCtrl
ActorClass(className=ATimingService, instancePath=/LogSys1/subSysRef1/tim:
=== done stop actor instances
*** MainComponent /LogSys1/subSysRef1::destroy ***
=== done destroy actor instances
=== done close loggers
=== done destroy RTServices
***   T H E   E N D   ***
|
```

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

2.3 Setting up the Workspace for C Projects

Objectives for this tutorial:

- create all needed library projects (runtime.c and modellib.c)

- create the tutorial project with the examples

- create the project with a tra c light simulator

- test the workspace setup by running one of the examples

2.3.1 Create Library, Tutorial and Simulator Projects

Before you can start with C, some preconditions must be fulfilled:

- A C compiler must be installed on your machine. All tutorials are based on MinGW/GCC (Windows) and Posix/GCC (Linux), but currently only tested on Windows with MinGW/GCC

- The CDT-Eclipse plugin must be installed as the C development environment.

After installation of eclipse and the eTrice plug in, your workspace should look like this:

Just the eTrice menu item is visible of the installed eTrice plugins.

Select the menu *File->New->Other*

Open the *eTrice* tab and select *eTrice C Runtime*

Press *Next* and *Finish* to install the Runtime into your workspace.

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

2.3.2   Perform Setup Test

To check the correct setup of your workspace we run a little testproject contained in the tutorial project.

The tutorial models are available in the *org.eclipse.etrice.tutorials.c* project. All tutorials are ready to generate and run without any changes. To test the code generator and the workspace setup simply run *gen_SetupTestC.launch* as *gen_SetupTestC*:

The successful generation ends with *Info: – finished code generation* in the Console.

For each tutorial in the folder src-gen a sub folder is generated which contains the generated code. The file *<...>_Runner.c* contains the main function. To run the generated application you first have to compile the project (with the hammer symbol in the C/C++ Perspective).

If the compillation does not succeed, make sure to clean and compile the projects *org.eclipse.etrice.runtime.c* and *org.eclipse.etrice.modellib.c* with the correct build configuration for your platform. Depending on the setup of your C compiler and CDT you might have to change the pre defined build configurations *MinGWDebug* or *PosixDebug*.

After the successful compilation you can run the application as *Local C/C++ Application*.

To stop the application type *quit* in the console window. If your Console contains the lines

```
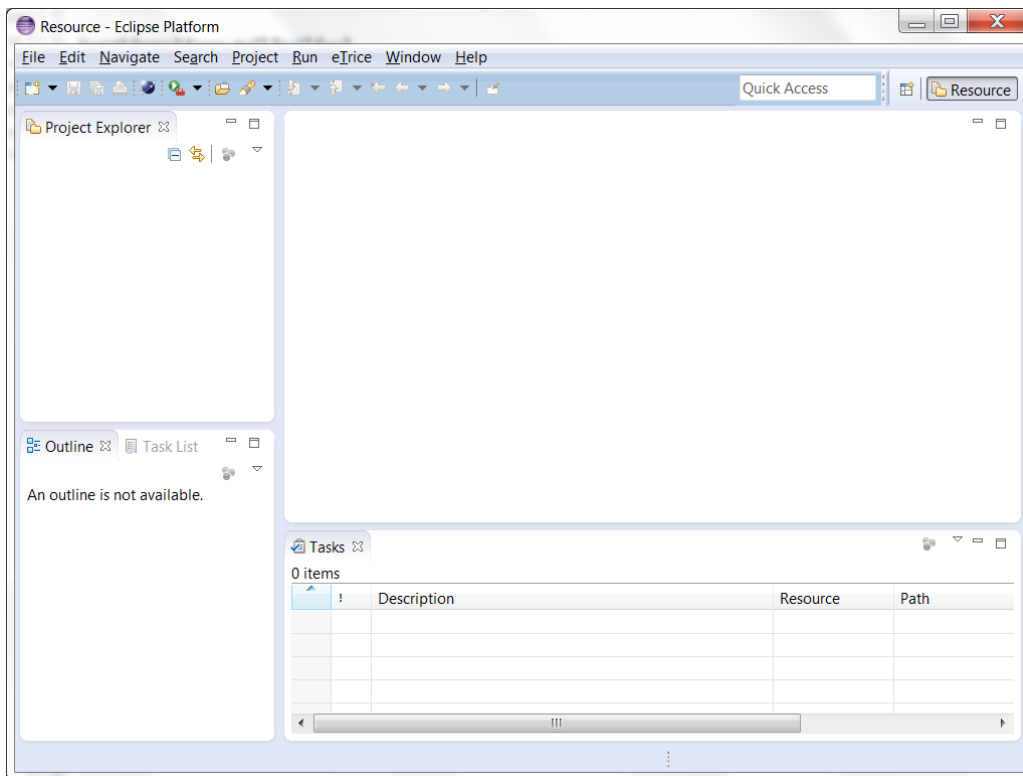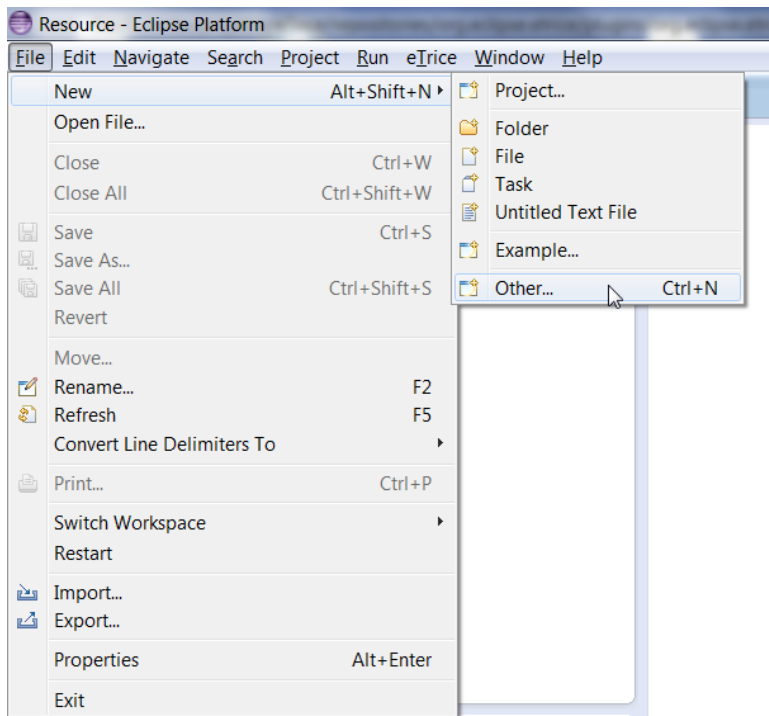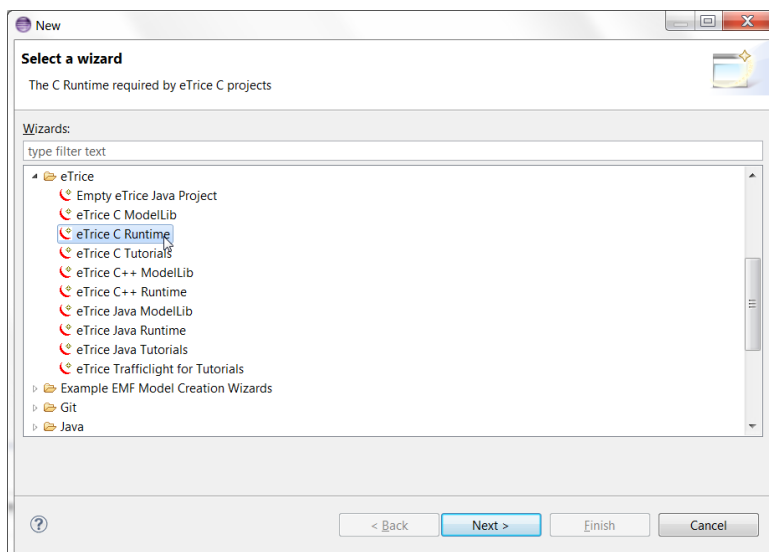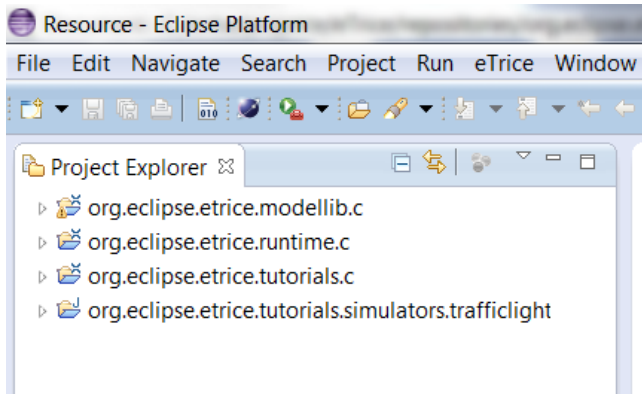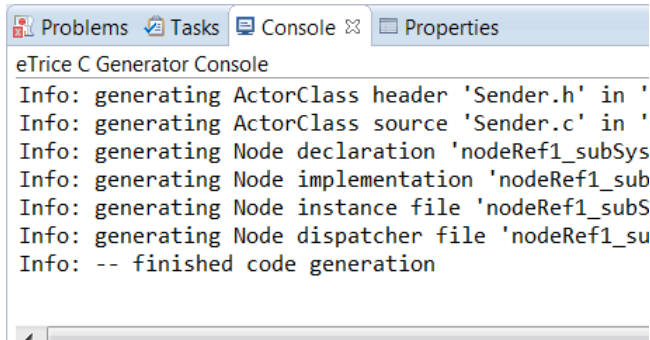*******************
*** Setup OK ***
*******************
```

your setup should be ok.

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

2.4   HelloWorld for Java

2.4.1   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 for Java*.

2.4.2   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 *Empty eTrice Java 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 *RoomModel - 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:

```
1   RoomModel HelloWorld_Model {
2
3       LogicalSystem LogSys1 {
4           SubSystemRef subSysRef1: SubSysClass1
5       }
6
7       SubSystemClass SubSysClass1 {
8           ActorRef actorRef1: HelloWorldTop
9           LogicalThread defaultThread
10      }
11
12      ActorClass HelloWorldTop { }
13
14  }
```

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 (e.g. a linux process or an image for a microcontroller) and contains at least one *ActorRef*. The *ActorClass* is the building block for building the hierachical structure of an application. A good point to start is to define a top level actor that can be used as structural root within the subsystem.

The outline view of the textual ROOM editor shows the main modeling elements in a navigation tree. You can jump to an element in the textual editor by double clicking the element in the outline view.

2.4.3   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. Be aware of the di erent action code in Java and C.

action code for Java	**action code for C**
`System.out.println("Hello World !");`	`printf("Hello World\n");`

The result should look like this:

Save the diagram and inspect the model (HelloWorld.room) file. Note that the textual representation was changed after saving the diagram.

room model for Java

```
1    RoomModel HelloWorld_Model {
2      LogicalSystem LogSys1 {
3        SubSystemRef subSysRef1:SubSysClass1
4      }
5      SubSystemClass SubSysClass1 {
6        ActorRef actorRef1:HelloWorldTop
7        LogicalThread defaultThread
8      }
9      ActorClass HelloWorldTop {
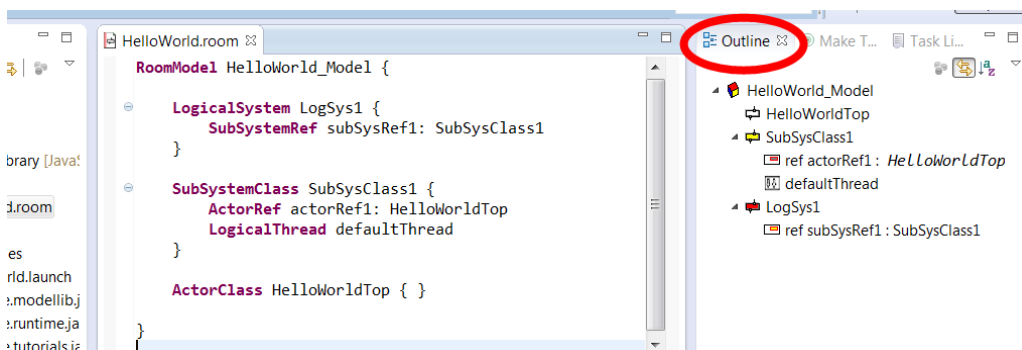10       Structure { }
11       Behavior {
12         StateMachine {
13           Transition init: initial -> state0 {
14             action {
15               "System.out.println(\"Hello World
                   \");"
16             }
17           }
18           State state0
19         }
20       }
21     }
22   }
```

room model for C

```
1    RoomModel HelloWorld_Model {
2      LogicalSystem LogSys1 {
3        SubSystemRef subSysRef1: SubSysClass1
4      }
5      SubSystemClass SubSysClass1 {
6        ActorRef actorRef1: HelloWorldTop
7        LogicalThread defaultThread
8      }
9      ActorClass HelloWorldTop {
10       Structure { }
11       Behavior {
12         StateMachine {
13           Transition init: initial -> state0 {
14             action {
15               "printf(\"Hello World\\n\");"
16             }
17           }
18           State state0
19         }
20       }
21     }
22   }
```

2.4.4 Build and run the model

Now the model is finished and the source code can be generated. The project wizard has created a launch configuration that is responsible for generating the source code. In the project *HelloWorld* right click *_HelloWorld.launch* and run it as *gen_HelloWorld*.

The source code for the model will be generated into the folder *src-gen*. 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 *"Hello World"* will be printed into the console window. To stop the application the user must type *quit* in the console window.

2.4.5   Open the Message Sequence Chart

For debugging and learning purposes, the application produced a Message Sequence Chart and wrote it to a file. Open the file *subSysRef1_Async.seq* or *msc.seq* in the folder *HelloWorld/tmp/log/* using the tool Trace2UML. Create the path if not already there.

Trace2UML is an open source MSC viewer and can be obtained here:

- Trace2UML project home and download of windows version

- download of the linux package of the Astade UML tool which contains Trace2UML

After opening the file, you should see something like this:

The Actor with the instance path */LogSys1/subSysRef1/actorRef1* is in the state *state0*. This is the simplest possible MSC. The MSCs for further tutorials will contain more information.

2.4.6 Summary

Now you have generated your first eTrice model from scratch. You can switch between diagram editor and textual model representation (.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 works. The next tutorials will deal with more complex hierarchies in structure and behavior.

2.5 HelloWorld for C

2.5.1 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. There are some more steps to do in C compared to Java. You will perform the following steps:

You will perform the following steps:

1. create a new model from scratch

2. add a very simple state machine to an actor

3. create a launch configuration for the C code generator

4. setup the C environment

5. generate the source code

6. run the model

7. open the message sequence chart

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

2.5.2 Create a new model from scratch

Before you can create a new C-model, you have to create a new C project as described in *Setting up the Workspace for C Projects*.

1. select the *C/C++* perspective

2. select *File->New->C Project* from the main menue

3. name the project *HelloWorldC*

4. the project type is *Executable / Empty C Project*

5. select the Toolchain for your platform (e.g. *MinGW GCC*)

Your project explorer should look like this:

The next step is to add the model folder: Right click on the new project. Select *New->Folder* and name it *model*.

Add the model file to the folder. Right click on the new folder. Select *New->file* and name it *HelloWorldC.room*.

Since the file extension *.room* is recognised as an Xtext based format, the tool will ask you to add the Xtext nature. Answer with *Yes*.

Open the *HelloWorld.room* file and delete the contents of the file. Open the content assist with <Ctrl>+<Space> and select *RoomModel - 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:

```
1   RoomModel HelloWorld_Model {
2
3     LogicalSystem LogSys1 {
4       SubSystemRef subSysRef1: SubSysClass1
5     }
6
7     SubSystemClass SubSysClass1 {
8       ActorRef actorRef1: HelloWorldTop
9       LogicalThread defaultThread
10    }
11
12    ActorClass HelloWorldTop { }
13
14  }
```

Now create the file model/HelloWorldC.etphys for the physical model and insert (<Ctrl>+<Space>) the code template *PhysicalModel - model skeleton* without changes.

Listing for HelloWorldC.etphys :

```
1   PhysicalModel PhysicalModel1 {
2
3     PhysicalSystem PhysSys1 {
4       NodeRef nodeRef1 : NodeClass1
5     }
6
7     NodeClass NodeClass1 {
8       runtime = RuntimeClass1
9       priomin = -10
10      priomax = 10
11      DefaultThread PhysicalThread1 {
12        execmode = mixed
13        interval = 100 ms
14        prio = 0
15        stacksize = 1024
16        msgblocksize = 32
```

```
17        msgpoolsize = 10
18      }
19    }
20
21    RuntimeClass RuntimeClass1 {
22      model = multiThreaded
23    }
24  }
```

The physical model defines the setup of your nodes with their attributes like threads and mode of execution. In this case we define one node with one thread.

The mapping model we will create now defines the mapping (deployment) of the logical elements in the .room model to the physical elements int the .etphys model. Now create the file model/HelloWorldC.etmap for the mapping model and insert (Ctrl+Space) the code template *MappingModel - model skeleton* with some changes (jump with <Tab> between the template variables):

```
1  MappingModel MappingModel1 {
2    import HelloWorld_Model.* from "HelloWorld.room"
3    import PhysicalModel1.* from "HelloWorld.etphys"
4    Mapping LogSys1 -> PhysSys1 {
5      SubSystemMapping subSysRef1 -> nodeRef1 {
6        ThreadMapping defaultThread -> PhysicalThread1
7      }
8    }
9  }
```

Now the workspace should look like this:

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 *Sub-SystemClass* represents an address space (e.g. a linux process or an image for a microcontroller) and contains at least one *ActorRef*. The *ActorClass* is the building block for building the hierachical structure of an application. A good point to start is to define a top level actor that can be used as structural root within the subsystem.

The outline view of the textual ROOM editor shows the main modeling elements in a navigation tree. You can jump to an element in the textual editor by double clicking the element in the outline view.

2.5.3   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. Be aware of the di erent action code in Java and C.

action code for Java	action code for C
`System.out.println("Hello World !");`	`printf("Hello World\n");`

The result should look like this:

Save the diagram and inspect the model (HelloWorld.room) file. Note that the textual representation was changed after saving the diagram.

```
room model for Java

1   RoomModel HelloWorld_Model {
2     LogicalSystem LogSys1 {
3       SubSystemRef subSysRef1:SubSysClass1
4     }
5     SubSystemClass SubSysClass1 {
6       ActorRef actorRef1:HelloWorldTop
7       LogicalThread defaultThread
8     }
9     ActorClass HelloWorldTop {
10      Structure { }
11      Behavior {
12        StateMachine {
13          Transition init: initial -> state0 {
14            action {
15              "System.out.println(\"Hello World
                   \");"
16            }
17          }
18          State state0
19        }
20      }
21    }
22  }
```

```
room model for C

1   RoomModel HelloWorld_Model {
2     LogicalSystem LogSys1 {
3       SubSystemRef subSysRef1: SubSysClass1
4     }
5     SubSystemClass SubSysClass1 {
6       ActorRef actorRef1: HelloWorldTop
7       LogicalThread defaultThread
8     }
9     ActorClass HelloWorldTop {
10      Structure { }
11      Behavior {
12        StateMachine {
13          Transition init: initial -> state0 {
14            action {
15              "printf(\"Hello World\\n\");"
16            }
17          }
18          State state0
19        }
20      }
21    }
22  }
```

2.5.4   Create a launch configuration to start the C code generator

Unlike in Java a launch configuration for the C code generator must be created.

From the main menu *Run* or the context menu *Run As* in the *Project Explorer* select *Run Configurations* .

Within the dialog select *eTrice C Generator* and click the *New* button to create a new launch configuration.

A new configuration should be created. Name it *gen_HelloWorld* and add the mapping model *HelloWorld.etmap* model via one of the *add* buttons.

The mapping model is the root model for the code generator.

To save your launch configuration, select *Shared file* in the tab *Common* and add the *HelloWorldC* project via the *Browse* button.

Apply your changes. The new configuration should now exist in your workspace.

2.5.5   Generate the code

Now you can generate the code. Right click on the launch configuration and run it as *gen_HelloWorldC*.

The code should be generated and placed in the src-gen folder.

2.5.6 Setup the C build

Before you can build the application you must setup the include and library pathes for the runtime system.

Right click the project and select *Properties -> C/C++ Build -> Settings -> Includes.* Add the include path for the current project *src-gen* and the runtime source folders *common*, *config* and the chosen platform (*MT_WIN_MinGW* or *MT_POSIX_GENERIC_GCC*).

Add the runtime library: *Properties -> C/C++ Build -> Settings -> Libraries*.

The name of the library is *org.eclipse.etrice.runtime.c* but the actual filename for the library is *li-borg.eclipse.etrice.runtime.c.a* .

Exclude the folder *src-gen-info* from the build path. This folder is used to store temporary files for the incremetal code generation and must not be compiled in order to avoid multiple definition of symbols.

2.5.7 Build and run the model

Now you can build the application. Click the build button (hammer symbol) to build the application. Run the application in the folder *Binary* as *Local C/C++ Application*. The output in the Console View should contain the message *Hello World*.

2.5.8 Open the Message Sequence Chart

For debugging and learning purposes, the application produced a Message Sequence Chart and wrote it to a file. Open the file *subSysRef1_Async.seq* or *msc.seq* in the folder *HelloWorld/tmp/log/* using the tool Trace2UML. Create the path if not already there.

Trace2UML is an open source MSC viewer and can be obtained here:

- Trace2UML project home and download of windows version

- download of the linux package of the Astade UML tool which contains Trace2UML

After opening the file, you should see something like this:

The Actor with the instance path */LogSys1/subSysRef1/actorRef1* is in the state *state0*. This is the simplest possible MSC. The MSCs for further tutorials will contain more information.

2.5.9   Summary

You are now familiar with all necessary steps to create, build and run an eTrice C model from scratch. You are able to create a launch configuration to start the code generator and to perform all necessary settings to compile and link the application.

The next tutorial provides an exercise to get more familiar with these working steps.

Chapter 3

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 in the eTrice repository: `http://git.eclipse.org/c/etrice/org.eclipse.etrice.git/plain/plugins/org.eclipse.etrice.core.room/src/org/eclipse/etrice/core/Room.xtext`

3.1 Actors

3.1.1 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.

3.1.2 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 re-usable blocks

Table 3.1: Actor Class Notation

Element	Graphical Notation	Textual Notation
ActorClass		**ActorClass** ActorClass2 {}

ActorRef	ActorReference (ActorClass2)	**ActorClass** ActorClass1 {
		  Structure {
		    ActorRef ActorReference: ActorClass2
		  }
		}

Table 3.2: Actor Class Example

Graphical Notation	Textual Notation
|  | ```
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
  }
}
``` |

3.1.3   Notation

3.1.4   Details

Actor Classes, Actor References, Ports and Bindings

An **ActorClass** defines the type (or blueprint) of an actor. Hierarchies are built by **ActorClass**es that contain **ActorRef**erences which have another **ActorClass** as type. The interface of an **ActorClass** is always defined by **Port**s. The **ActorClass** can also contain **Attribute**s, **Operation**s and a finite **StateMachine**.

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

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

Bindings connect **Port**s inside an **ActorClass**.

Let us have a look at example 3.2:

- *ActorClass1* contains two **ActorRef**erences (of ActorClass2 and ActorClass3)

- *port1* is an *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 **ActorRef**erences. Internal end ports connect the behavior of an **ActorClass** with its contained **ActorRef**erences.

- *port4* is a relay port and connects external Actors to contained **ActorRef**erences. This port can not be accessed by the behavior of the **ActorClass**.

- *port5* through *port9* are ports of contained actor references. *port8* and *port9* can communicate without interference with the containing actor class.

- **Binding**s can connect ports of the actor class and its contained actor references.

Attributes

Attributes are part of the **Structure** of an actor class. They can be of a **PrimitiveType** or a **DataClass**.

Example:

```
ActorClass ActorClass3 {
  Structure {
    Attribute attribute1: int32    // attribute of primitive type
    Attribute attribute2: DataClass1 // attribute of DataClass type
  }
}
```

Operations

Operations are part of the **Behavior** of an actor class. Arguments and return values can be of a **PrimitiveType** or a **DataClass**. Data classes can be passed by value (implicit) or by reference (**ref**).

Example:

```
ActorClass ActorClass4 {
  Behavior {
    // no arguments, no return value
    Operation operation1(): void {
      "UserCodeLine1"
    }
    // argument of primitive type, return value of primitive type
    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 primitive types
    Operation operation4(Param1: int32, Param2: DataClass1 ref): DataClass1 ref {
      "UserCodeLine1"
    }
  }
}
```

3.2 Protocols

3.2.1 Description

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

3.2.2 Motivation

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

3.2.3 Notation

Protocol classes have only textual notation. The example defines a protocol class 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 data class.

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

3.3   Ports

3.3.1   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.

3.3.2   Motivation

- Ports decouple interface definition (protocols) from interface usage

- Ports decouple the logical interface from the transport

3.3.3   Notation

Class Ports

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

Ports that define an external interface of the actor class, 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

The table 3.3 shows all kinds of class ports with textual and graphical notation.

Table 3.3: Class Port Notation

Element	Graphical Notation	Textual Notation

Class End Port		*External Class End Port:*
		
		ActorClass ActorClass6 {
		  Interface {
		    Port port12: ProtocolClass1
		  }
		  Structure {
		    external Port port12
		  }
		}
		Internal Class End Port:
		
		ActorClass ActorClass6 {
		  Interface {
		  }
		  Structure {
		    Port port20
		  }
		}

Table 3.3: Class Port Notation

Element	Graphical Notation	Textual Notation

Conjugated Class End Port	●	*External Conjugated Class End Port:*
		
		ActorClass ActorClass6 {
		  Interface {
		    conjugated Port port13: ProtocolClass1
		  }
		  Structure {
		    external Port port13
		  }
		}
		Internal Conjugated Class End Port:
		
		ActorClass ActorClass6 {
		  Interface {
		  }
		  Structure {
		    conjugated Port port21: ProtocolClass1
		  }
		}

Class Relay Port	□	**ActorClass** ActorClass6 {
		  Interface {
		    Port port10: ProtocolClass1
		  }
		  Structure {
		  }
		}

Conjugated Class Relay Port	■	**ActorClass** ActorClass6 {
		  Interface {
		    conjugated Port port10: ProtocolClass1
		  }
		  Structure {
		  }
		}

Table 3.3: Class Port Notation

Element	Graphical Notation	Textual Notation

Replicated Class End Port		*External Replicated Class End Port:*
		
		ActorClass ActorClass6 {
		  Interface {
		    Port port16[3]:
		       ProtocolClass1
		  }
		  Structure {
		    external Port port16
		  }
		}
		
		Internal Replicated Class End Port:
		
		ActorClass ActorClass6 {
		  Interface {
		  }
		  Structure {
		    Port port16[3]:
		       ProtocolClass1
		  }
		}

Conjugated Replicated Class End Port		*External Conjugated Replicated Class End Port:*
		
		ActorClass ActorClass6 {
		  Interface {
		    conjugated Port port17[3]:
		       ProtocolClass1
		  }
		  Structure {
		    external Port port17
		  }
		}
		
		Internal Conjugated Replicated Class End Port:
		
		ActorClass ActorClass6 {
		  Interface {
		  }
		  Structure {
		    conjugated Port port23[3]:
		       ProtocolClass1
		  }
		}

Table 3.4: Reference Port Notation

Element	Graphical Notation	Textual Notation
Reference Port		*implicit*
Conjugated Reference Port		*implicit*
Replicated Reference Port		*implicit*
Conjugated Replicated Reference Port		*implicit*

Table 3.3: Class Port Notation

Element	Graphical Notation	Textual Notation

Replicated Class Relay Port		**ActorClass** ActorClass6 {
		  Interface {
		    Port port18[3]: ProtocolClass1
		  }
		  Structure {
		  }
		}

Conjugated Replicated Class Relay Port		**ActorClass** ActorClass6 {
		  Interface {
		    conjugated Port port19[3]:
		      ProtocolClass1
		  }
		  Structure {
		  }
		}

Reference Ports

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

The table 3.4 shows all kinds of reference ports with textual and graphical notation.

3.4 DataClass

3.4.1 Description

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

3.4.2 Notation

Example: DataClass using PrimitiveTypes

```
DataClass DataClass1 {
  Attribute attribute1: int32 // attribute of primitive type
  Attribute attribute2: float32 // attribute of another primitive type

  // no arguments, no return value
```

```
Operation operation1(): void {
  "UserCodeLine1"
}
// argument of primitive type, no return value
Operation operation2(Param1: int32): void {
  "UserCodeLine1"
}
// argument of primitive type, return value of primitive type
Operation operation3(Param1: int32): float64 {
  "UserCodeLine1"
}
}
```

Example: DataClass using other DataClasses:

```
DataClass DataClass2 {
  Attribute attribute1: int32    // attribute of primitive type
  Attribute attribute2: DataClass1 // attribute of DataClass

  // arguments and return value by value
  Operation operation1(Param1: int32, Param2: DataClass1): DataClass1 {
    "UserCodeLine1"
  }
  // arguments and return value by reference except for primitive types
  Operation operation2(Param1: int32, Param2: DataClass1 ref): DataClass1 ref {
    "UserCodeLine1"
  }
}
```

3.5   Layering

3.5.1   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 actor class can define a Service Provision Point (**SPPRef**) to publish a specific service, defined by a protocol class

2. an actor class can define a Service Access Point (**SAPRef**) if it needs a service, defined by a protocol class

3. for a given actor hierarchy, a **LayerConnection** defines which SAP will be satisfied by (connected to) which SPP

3.5.2   Notation

For the graphical and textual notation refer to table 3.5

3.6   Finite State Machines

3.6.1   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

Table 3.5: Layering Notation

Description	Graphical Notation	Textual Notation

The layer connections in this model define which services are provided by the *ServiceLayer* (*digitalIO* and *timer*)		**ActorClass** Mode1 {
		  Structure {
		    ActorRef Services: ServiceLayer
		    ActorRef Application: ApplicationLayer
		    LayerConnection ref Application **satisfied_by**
		        Services.timer
		    LayerConnection ref Application **satisfied_by**
		        Services.digitalIO
		  }
		}

The implementation of the services (SPPs) can be delegated to sub actors. In this case the actor *ServiceLayer* relays (delegates) the implementation services *digitalIO* and *timer* to sub actors		**ActorClass** ServiceLayer {
		  Interface {
		    SPP timer: TimerProtocol
		    SPP digitalIO: DigitalIOProtocol
		  }
		  Structure {
		    ActorRef Timer: TimerService
		    ActorRef DigIO: DifitalIOService
		    LayerConnection relay_sap timer **satisfied_by**
		        Timer.timer
		    LayerConnection relay_sap digitalIO
		        satisfied_by DigIO.digitalIO
		  }
		}

Every Actor inside the *ApplicationLayer* that contains an SAP with the same protocol as *timer* or *digitalIO* will be connected to the specified SPP		**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
		  }
		}

Table 3.6: Flat finite state machine notation

Description	Graphical Notation	Textual Notation
State		**State** SomeState
InitialPoint		*implicit*
TransitionPoint		**TransitionPoint** tp
ChoicePoint		**ChoicePoint cp**
Initial Transition		**Transition** init: **initial** -> Initial { }

Triggered Transition		**Transition** tr0: **initial** -> DoingThis {
		  triggers {
		    <doThis: fct>
		  }
		}

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

3.6.2 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.

3.6.3 Notation

We distinguish flat finite state machines (with just one level of hierarchy) and hierarchical ones.

Flat Finite State Machine

The simpler flat finite state machines are composed of the elements shown in table 3.6.

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 listed in table 3.7 are added to the set listed above.

3.6.4 Examples

Table 3.7: Additional notation elements of hierarchical finite state machines

Description	Graphical Notation	Textual Notation

State with sub state machine	Parent State
	 Running
		Sub state machine
		
		 State Running {
		   subgraph {
		      Transition init: **initial** -> Process {}
		      State Process
		   }
		}

Entry Point	In sub state machine reInit ⊗	**EntryPoint** reInit
Exit Point	tp0	**ExitPoint** tp0

Figure 3.1: Example of a flat finite state machine

Figure 3.2: Example of a hierarchical finite state machine – top level

Figure 3.3: Hierarchical finite state machine – sub state machine of *Initializing*

Figure 3.4: Hierarchical finite state machine – sub state machine of *Running*

Chapter 4

eTrice Features

4.1 Automatic Diagram Layout with KIELER

4.1.1 Overview

eTrice now provides a new feature of automatic layout of the ROOM diagrams in its graphical editors. This helps in improving the pragmatics of the diagrams and frees the user from the burden of manually lay-outing the diagrams on the canvas.

The automatic lay-outing has been provided with the help of the well known KIELER framework, which focuses on the pragmatics of model-based system design, which can improve comprehensibility of diagrams, improve development and maintenance time, and improve the analysis of dynamic behavior.

This chapter will answer the following questions

- "How to perform automatic layout in the graphical editors of eTrice?"

- "What are layout options?"

- "How to configure the layout options to alter the diagram layout as desired?"

Moreover, some "special layout options" will also be discussed.

4.1.2 Performing Automatic Layout

Automatic layout could be performed in eTrice graphical editors using the command to layout the current diagram. This command is available in

- The context menu of the diagrams

- Using the *Ctrl+R L* shortcut.

Additionally, an entry in the context menu allows to layout only a selected part of the diagram.

4.1.3 Layout Options

A layout option is a customization point for the layout algorithms, with a specific data type and optionally a default value, used to a ect how the active layout algorithm computes concrete coordinates for the graph elements.

User-configurable layout options for a particular diagram object can be viewed and configured through the Layout View.The Layout View can be opened from the context menu of a selected diagram object by clicking the *Show Layout View* entry.

On opening the layout view, and selecting any layout option, a description of the layout option is available in the footer of eclipse SDK. This is shown in figure 4.1.

Figure 4.1: Layout options

Figure 4.2: Layout view

4.1.4 Configuring Layout Options

The values of the layout options for a particular diagram object (in the visible diagram) can be changed using the Layout View of that diagram object. The initial values are the predefined *default* values. These defaults can be changed using the context menu in Layout View as well as the Layout preference pages provided by eTrice.

The Layout View

The Layout view (figure 4.2) allows flexible customization of layout options for the selected objects in the eTrice diagram. If no object is selected, the view shows the options for the top-level container of the diagram. Options are stored persistently in diagram file (.*structure file* / .behavior file) of the eTrice diagram, so that they are still available after the next Eclipse restart. Of course this requires the diagram to be saved after an option was changed.

The options are grouped according to the function of the selected objects. The group Nodes (respectively Edges, Ports, or Labels) contains options related to the object itself, such as its size or priority, while the group Parents contains options for the elements contained in the selected objects, such as the applied layout algorithm or the spacing between elements. Which layout options are displayed depends on the types of selected objects and the active layout algorithm, since each algorithm supports only a subset of the available options. Furthermore, some options are only visible if the *Show Advanced Properties* button in the view toolbar is activated. The group types can be hidden using the Show Categories button.

An option can be changed by selecting or entering a new value in the corresponding cell of the Value column.

The most important option is Layout Algorithm, which is used to determine the layout algorithm for the contents of the selected element. Here either a specific layout algorithm or a layout type can be chosen; in the latter case, the most suitable layout algorithm of the given type is taken. By changing the active layout algorithm, the content of the layout view is updated to display only those options that are supported by the new layout algorithm.

Selecting *Restore Default Value* in the context menu or the view toolbar (figure 4.3) removes any value for the

Figure 4.3: Layout in context menu

currently selected option that is stored in the current model file, thus resetting the option to its default value. The view menu has an entry Remove all Layout Options which resets all options of the current model by removing persistent data in the model file.

The context menu for a specific layout option has different alternatives to set the currently active value as *default* value:

- *Set as Default for this Diagram*: Changes the open diagram file so that the same value is applied to all similar objects (edit parts) of that diagram.

- *Set as Default for ... in this Context*: Applies the value to all similar objects that are displayed with the any of the eTrice editors (the option is linked to the edit part class of the selected object).

- *Set as Default for all ...*: Links the option value with the domain model element or the diagram type of the selected object (see the context menu depicted above).

These four alternatives have different priorities: if present, the default value for the current diagram is taken first, then the default value for the edit part is checked, then the default value for the domain model element, and then the default value for the diagram type.

Tips:

- The information button of the view toolbar can be used to display some useful details on the current selection, such as the edit part and domain model classes.

- Default values for layout options can most easily be manipulated based on the eTrice domain model elements.

Preference Page

The user-defined *default* values for layout options can also be set using the preference pages provided in eTrice. Three preference pages have been provided for this purpose

- *Layout*: for general preferences regarding layout

- *Behavior*: for setting default values of layout options for eTrice behavior diagrams

- *Structure*: for setting default values of layout options for eTrice structure diagrams

These preference pages can be accessed via *Windows > Preferences > eTrice > Layout*.

Note that the contents of these preference pages are in sync with the *KIELER > Layout* preference page provided by the KIELER. Relevant entries in the *KIELER > Layout* page are shown in the above preference pages.

Layout Preference Page

The *Layout* preference page is meant to configure general options regarding the layout.

If *Set routing style of all edges to oblique* is active, all routing styles and smoothness settings of edges are removed when automatic layout is performed. Since most layouters compute the routing of edges as part of their algorithm, these styles usually do not yield the expected results.

Figure 4.4: Layout preference page

Behavior and *Structure* Preference Page

The *Behavior* and *Structure* sub-preference pages help in setting up the default values of layout options in behavior and structure diagrams respectively.

The *Default Layout Option Values* table is used to manage the default setting for layout options, which can also be modified with the context menu of the layout view (see above). All user-defined settings are displayed here, and the buttons on the right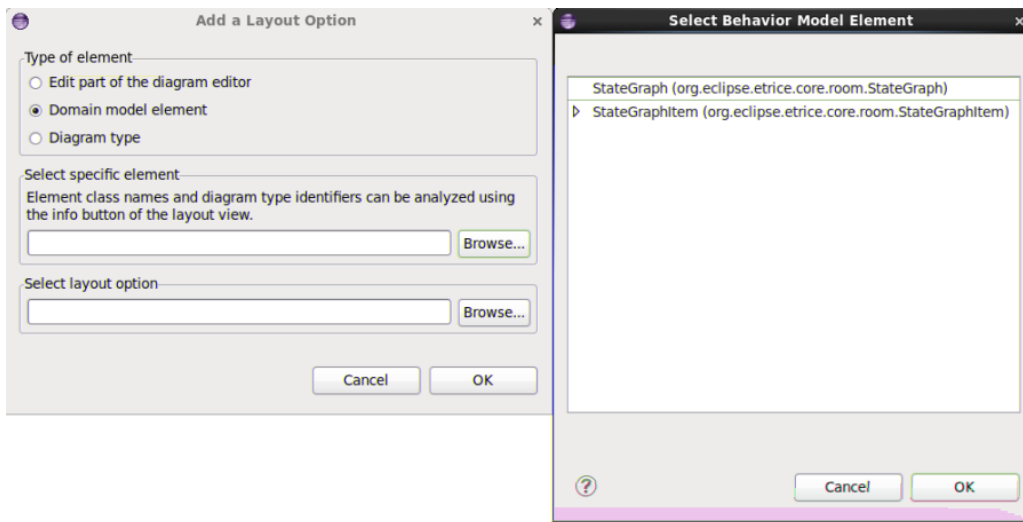 of the table serve to create, edit, and remove entries. The Type column shows the type of element the option is linked with: either edit part, model element, or diagram type. The Element column shows the class name for options that relate to edit parts or domain model elements, and the diagram type name for options that relate to diagram types. Option is the name of the layout option, and Value is the currently set value of the option.

Creating a new entry requires the selection of the type of related element (figure 4.4) and entering its class name or identifier. Class names of edit parts can be explored using the information button of the layout view, while the class names for the domain model elements and the diagram type identifiers for the diagram types can be selected with the Browse button. After that, a layout option has to be selected from the list using the corresponding Browse button. Hitting OK creates an entry, and its value can then be set using the Edit button.

Note that the *Behavior* preference page will show only those entries which hold for the behavior diagrams. Moreover, it will allow setting default values of layout options for only those domain model elements and diagram types which could be present in the behavior editor diagrams. Similar thing holds for the *Structure* preference page.

4.1.5   Special Layout Options

While most layout options are used to a ect how the active layout algorithm computes concrete coordinates for the graph elements, there are some layout options that have a special role.

Layout Algorithm

The option with identifier de.cau.cs.kieler.algorithm specifies which layout algorithm to use for the content of a composite node. The value can be either the identifier of a layout algorithm or the identifier of a layout type. In the latter case the algorithm with highest priority of that type is applied.

For the purpose of automatic diagram layout in eTrice, we use the *Layered* algorithms which are meant for lay-outing hierarchical diagrams and are best suited for behavior and structure diagrams in eTrice. For the behavior diagrams we have used the *Graphviz Dot* algorithm whereas for the structure diagrams we have used the *KLay Layered* algorithm. Though the layout algorithm being used for performing layout can be changed at ones own will, it is recommended to use the defaults.

Diagram Type

Diagram types are used to classify graphical diagrams for setting default layout option values for a set of similar diagrams. The diagram type of an element is specified with the layout option `de.cau.cs.kieler.diagramType`. Thus, these help in

The following diagram types have been defined and used in eTrice:

- *General* - This type is automatically assigned to all diagrams for which no specific type is declared. (Predefined in KIELER)

- *eTrice Behavior Diagrams* - This type has been assigned to the diagram objects in eTrice Behavior Diagrams.

- *eTrice Structure Diagrams* - This type has been assigned to the diagram objects in eTrice Structure Diagrams.

Note that not all diagrams objects in the behavior and structure diagrams are assigned the last two diagram types. Only the top-level container and the visible bounding box has been assigned these diagram types in respective editors.

4.1.6 Further References

Most parts of the above documentation have been taken from the "KIML wiki" (`http://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=328078` and have been modified for automatic layout in eTrice. A more detailed description about the layout algorithms, predefined diagram types and the internal structure of KIELER Infrastructure for Meta-Layout (KIML) can be found there.

4.2 eTrice Models and Their Relations

eTrice comprises several models:

- the ROOM model (*.room) – defines model classes and the logical structure of the model

- the Config model (*.config) – defines configuration values for attributes

- the Physical model (*.etphys) – defines the structure and properties of the physical system

- the 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.

4.2.1 The ROOM Model

The ROOM model defines **DataClass**es, **ProtocolClass**es, **ActorClass**es, **SubSystemClass**es and **LogicalSystem**s. 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 contains 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 anything 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"
}
```

Listing 4.1: ROOM example code

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 the above example looks like in figure 4.5 (the third line in the white boxes shows some mapping information, see section 4.2.4 The Mapping Model):

Figure 4.5: Instances of a ROOM system

4.2.2 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.

```
ConfigModel test {
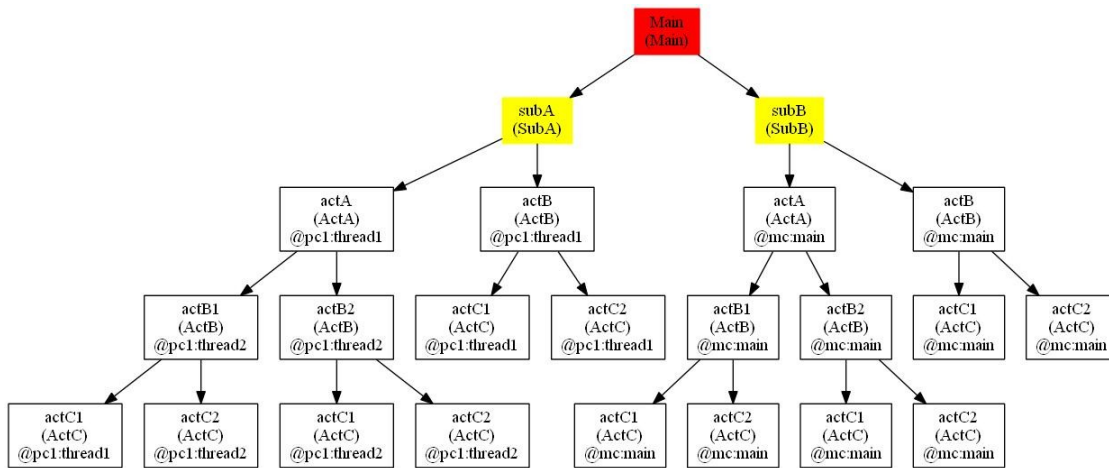
  import test.* from "room-example.room"

  ActorClassConfig ActA {
    Attr val = 1
  }

  ActorClassConfig ActB {
    Attr val = 2
  }

  ActorInstanceConfig Main/subA/actA {
    Attr val = 12
  }

  ActorInstanceConfig Main/subA/actB {
    Attr val = 13
  }

  ActorInstanceConfig Main/subA/actA/actB2 {
    Attr val = 14
  }
}
```

Listing 4.2: Config example code

4.2.3 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 define the overall execution model of the platform.

```
PhysicalModel runtimes {

  RuntimeClass PCRuntime {
    model = multiThreaded
  }

  RuntimeClass MSP430Runtime {
    model = singleThreaded
```

```
  }
}
```
Listing 4.3: etPhys runtime definition

The **PhysicalSystem** is composed of **NodeRef**erences which are instances of **NodeClass**es. Each **NodeClass** is referencing a **RuntimeClass** and is defining **Threads**.

```
PhysicalModel test {

  import test.* from "etphys-runtimes.etphys"

  PhysicalSystem MainPhys {
    NodeRef pc1: PC_Node
    NodeRef pc2: PC_Node
    NodeRef mc: MSP430_Node
  }

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

  NodeClass MSP430_Node {
    runtime = runtimes.MSP430Runtime
    priomin = 1
    priomax = 10
    DefaultThread main {
      execmode = polled
      interval = 10 us
      prio = 10
      stacksize = 256
      msgblocksize = 64
      msgpoolsize = 50
    }
  }
}
```
Listing 4.4: etPhys example code

4.2.4 The Mapping Model

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

```
MappingModel test {

  import test.* from "etphys-example.etphys"
  import test.* from "room-example.room"
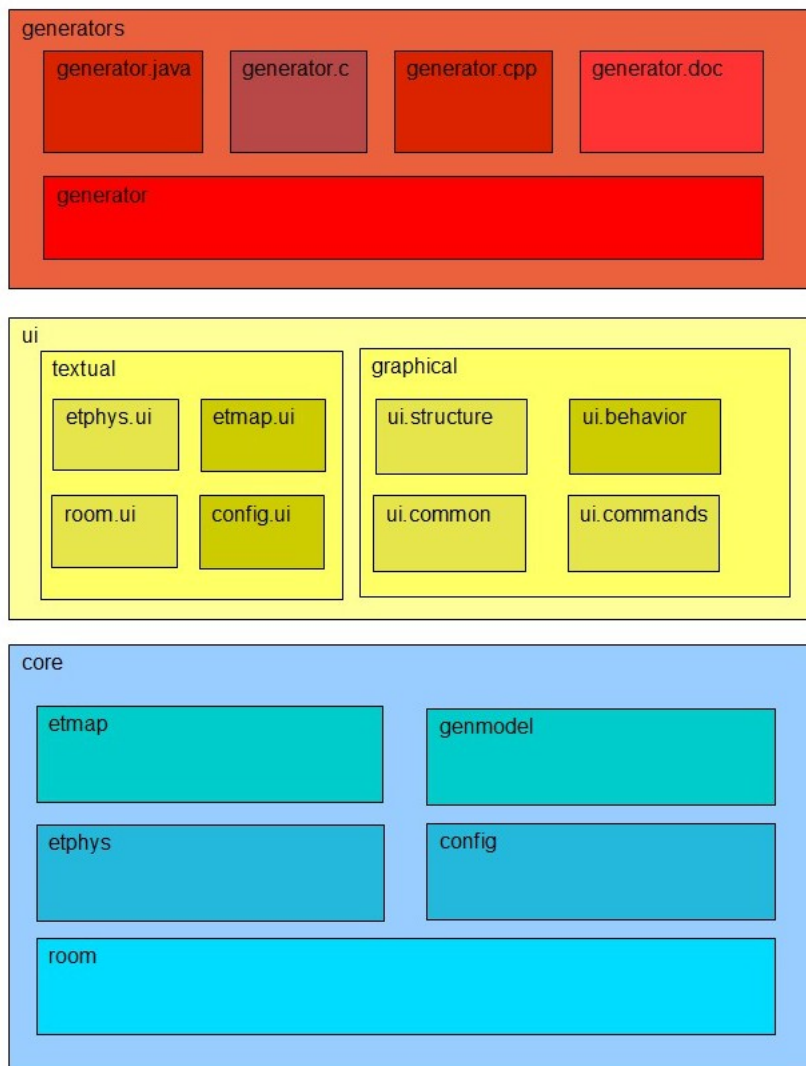
  Mapping Main -> MainPhys {
```

```
    SubSystemMapping subA -> pc1 {
      ThreadMapping dflt -> thread1
      ThreadMapping extra -> thread2
    }
    SubSystemMapping subB -> mc {
      ThreadMapping dflt -> main
      ThreadMapping extra -> main
    }
  }
}
```

Listing 4.5: etMap example code

The result of the mapping is also depicted in above tree diagram (figure 4.5) 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 5

eTrice Tool Developer's Reference

5.1 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.

5.1.1 Editor and Generator Components

- core

  - core.room is an Xtext based language called ROOM. It consists of the plug-ins `org.eclipse.etrice.core.room` and `org.eclipse.etrice.core.room.ui`. ROOM is the basic modeling language of eTrice.

  - core.config is an Xtext based language called Config. It consists of the plug-ins `org.eclipse.etrice.core.config` and `org.eclipse.etrice.core.config.ui`. Config is a language designed for the data configuration of model

  - core.etphys is an Xtext based language called etPhys. It consists of the plug-ins `org.eclipse.etrice.core.etphys` and `org.eclipse.etrice.core.etphys.ui`. etPhys is a language designed for the description of physical systems onto which the logical ROOM systems are deployed.

  - core.etmap is an Xtext based language called etMap. It consists of the plug-ins `org.eclipse.etrice.core.etmap` and `org.eclipse.etrice.core.etmap.ui`. etMap is a language designed for the mapping of logical to physical systems.

  - 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.

    * etphys.ui is the ui counterpart of core.etphys. It consists of the plug-in `org.eclipse.etrice.core.etphys.ui`. This plug-in realizes IDE concepts like content assist, error markers and navigation by hyper links for the etPhys language.

    * etmap.ui is the ui counterpart of core.etmap. It consists of the plug-in `org.eclipse.etrice.core.etmap.ui`. This plug-in realizes IDE concepts like content assist, error markers and navigation by hyper links for the etPhys 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.cpp is the generator for the C++ target language. It consists of the plug-in `org.eclipse.etrice.generator.cpp`.

  - 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`.

5.1.2 Runtimes

Currently eTrice ships with a C and a Java runtime. The C++ runtime is still a prototype. The runtimes are libraries written in the target language against which the generated code is compiled.

5.1.3 Unit Tests

Most plug-ins and other parts of the code have related unit tests.

5.2 Component Overview

5.2.1 Room Language Overview

We assume that the reader is familiar with the Xtext concepts. So we concentrate on the details of our implementation that are worth to be pointed out.

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

Two classes provide object names used for link resolution and for labels. The RoomNameProvider provides frequently used name strings, some of them are hierarchical like State paths. The RoomFragmentProvider serves a more formal purpose since it provides a link between EMF models (as used by the diagram editors) and the textual model representation used by Xtext.

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.

5.2.2 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.

5.2.3 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

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: