eTrice User Guide

Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, Peter Friese,
Jan Köhnlein, Knut Wannheden, Sebastian Zarnekow and contributors

Chapter 1. Overview

1.1. What is Xtext?

No matter if you want to create a small textual domain-specific language (DSL) or you want to implement a full-blown general purpose programming language. With Xtext you can create your very own languages in a snap. Also if you already have an existing language but it lacks decent tool support, you can use Xtext to create a sophisticated Eclipse-based development environment providing editing experience known from modern Java IDEs in a surprisingly short amount of time. We call Xtext a language development framework.

1.2. How Does It Work?

Xtext provides you with a set of domain-specific languages and modern APIs to describe the different aspects of your programming language. Based on that information it gives you a full implementation of that language running on the JVM. The compiler components of your language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and static analysis aka validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modeling Framework (EMF), which effectively allows you to use Xtext together with other EMF frameworks like for instance the Graphical Modeling Project GMF.

In addition to this nice runtime architecture, you will get a full blown Eclipse-IDE specifically tailored for your language. It already provides great default functionality for all aspects and again comes with DSLs and APIs that allow to configure or change the most common things very easily. And if that's not flexible enough there is Guice to replace the default behaviour with your own implementations.

1.3. Xtext is Highly Configurable

Xtext uses the lightweight dependency injection (DI) framework Google Guice to wire up the whole language as well as the IDE infrastructure. A central, external module is used to configure the DI container. As already mentioned, Xtext comes with decent default implementations and DSLs and APIs for the aspect that are common sweet spots for customization. But if you need something completely different, Google Guice gives you the power to exchange every little class in a non-invasive way.

1.4. Who Uses Xtext?

Xtext is used in many different industries. It is used in the field of mobile devices, automotive development, embedded systems or Java enterprise software projects and game development. People use Xtext-based languages to drive code generators that target Java, C, C++, C#, Objective C, Python, or Ruby code. Although the language infrastructure itself runs on the JVM, you can compile Xtext languages to any existing platform. Xtext-based languages are developed for well known Open-Source projects such as Maven, Eclipse B3, the Eclipse Webtools platform or Google's Protocol Buffers and the framework is also widely used in research projects.

1.5. Who is Behind Xtext?

Xtext is a professional Open-Source project. We, the main developers and the project lead, work for itemis, which is a well known consulting company specialized on model-based development. Therefore we are able to work almost full-time on the project. Xtext is an Eclipse.org project. Besides many other advantages this means that you don't have to fear any IP issues, because the Eclipse Foundation has their own lawyers who take care that no intellectual property is violated.

You may ask: Where does the money for Open-Source development come from? Well, we provide professional services around Xtext. Be it training or on-site consulting, be it development of prototypes or implementation of full-blown IDEs for programming languages. We do not only know the framework very well but we are also experts in programming and domain-specific language design. Don't hesitate to get in contact with us (www.itemis.com).

1.6. What is a Domain-Specific Language

A *Domain-Specific Language (DSL)* is a small programming language, which focuses on a particular domain. Such a domain can be more or less anything. The idea is that its concepts and notation is as close as possible to what you have in mind when you think about a solution in that domain. Of course we are talking about problems which can be solved or processed by computers somehow.

The opposite of a DSL is a so called *GPL*, a *General Purpose Language* such as Java or any other common programming language. With a GPL you can solve every computer problem, but it might not always be the best way to solve it.

Imagine you want to remove the core from an apple. You could of course use a Swiss army knife to cut it out, and this is reasonable if you have to do it just once or twice. But if you need to do that on a regular basis it might be more efficient to use an apple corer.

There are a couple of well-known examples of DSLs. For instance SQL is actually a DSL which focuses on querying relational databases. Other DSLs are regular expressions or even languages provided by tools like MathLab. Also most XML languages are actually domain-specific languages. The whole purpose of XML is to allow for easy creation of new languages. Unfortunately, XML uses a fixed concrete syntax, which is very verbose and yet not adapted to be read by humans. Into the bargain, a generic syntax for everything is a compromise.

Xtext is a sophisticated framework that helps to implement your very own DSL with appropriate IDE support. There is no such limitation as with XML, you are free to define your concrete syntax as you like. It may be as concise and suggestive as possible being a best match for your particular domain. The hard task of reading your model, working with it and writing it back to your syntax is greatly simplified by Xtext.

Chapter 2. Tutorial HelloWorld

2.1. 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

2.2. Create a new model from scratch

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

- the *LogicalSystem*
- at least one *SubSystemClass*
- at least one *ActorClass*

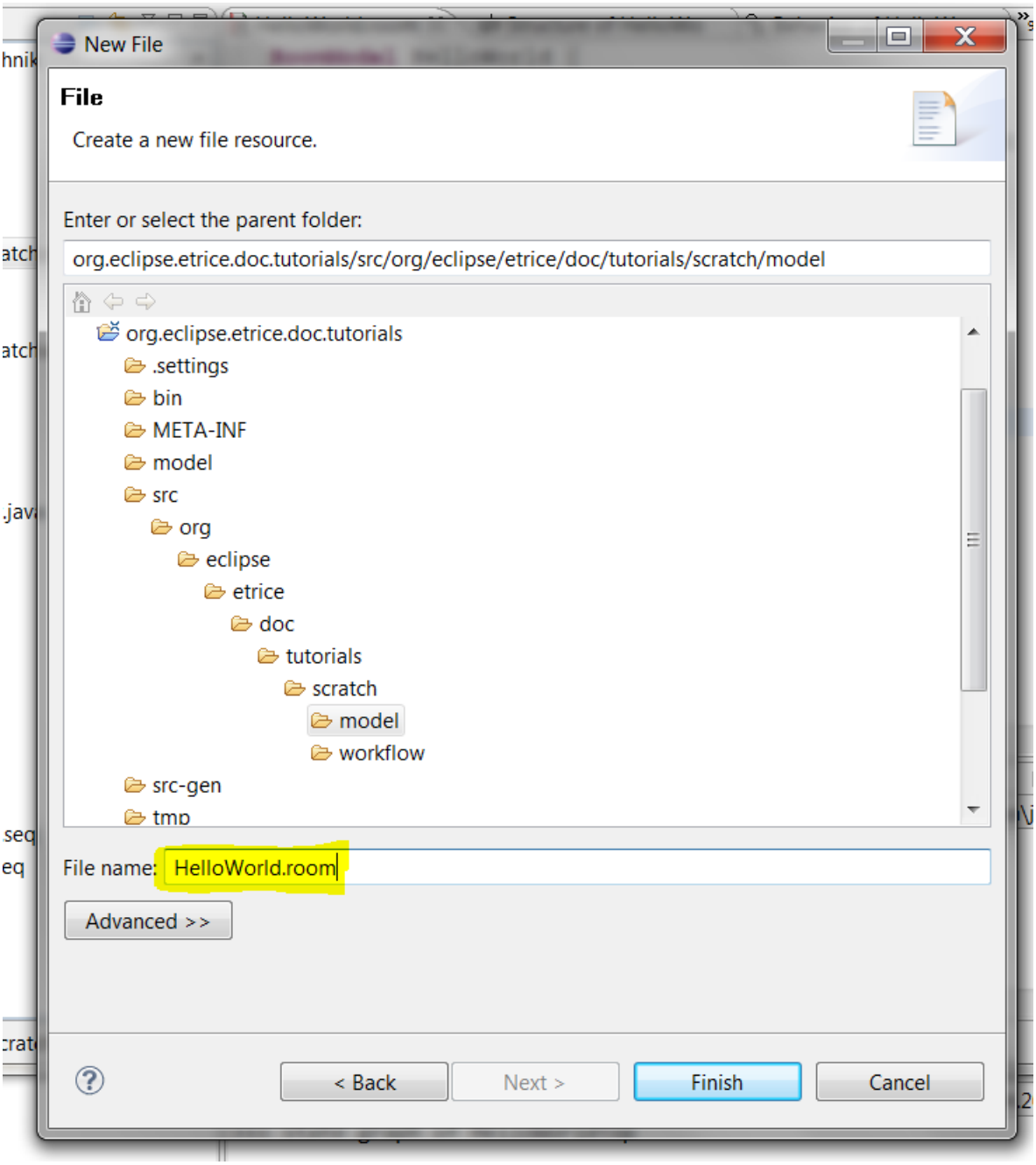
The *LogicalSystem* represents the complete distributed system and contains at least one *SubSystemRef*. The *SubSystemClass* represents an address space and contains at least one *ActorRef*. The *ActorClass* is the building block of which an application will be build of. It is a good idea to define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

The resulting model code looks like this:

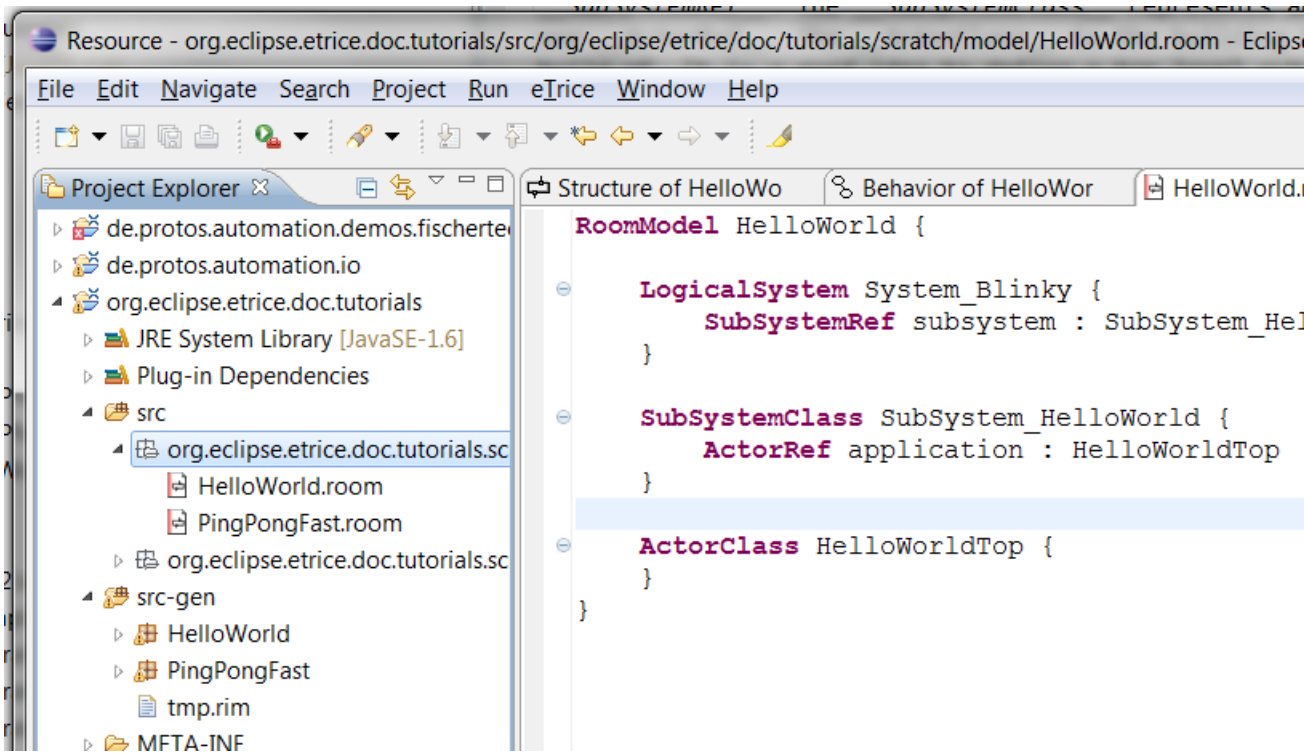
```
RoomModel HelloWorld {

 LogicalSystem System_Blinky {
  SubSystemRef subsystem : SubSystem_HelloWorld
 }

 SubSystemClass SubSystem_HelloWorld {
  ActorRef application : HelloWorldTop
 }

 ActorClass HelloWorldTop {
 }
}
```

2.2.1. Create a new model file

Create a new file in your org.eclipse.etrice.doc.tutorials.scratch.model directory and name it *HelloWorld.room* and select finish.

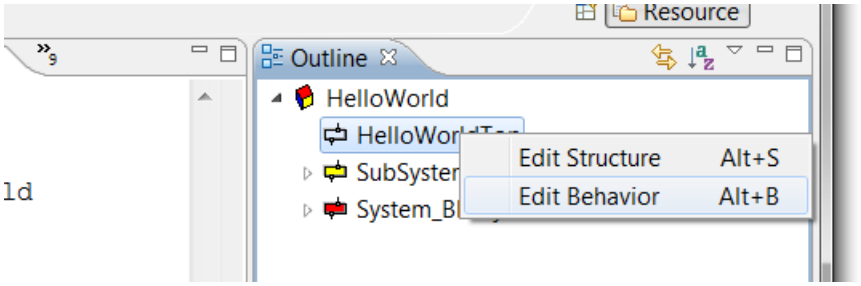


The file ending must be *.room* for selecting the correct editor. Open the file and copy the above code into the editor window. You should see something like this:

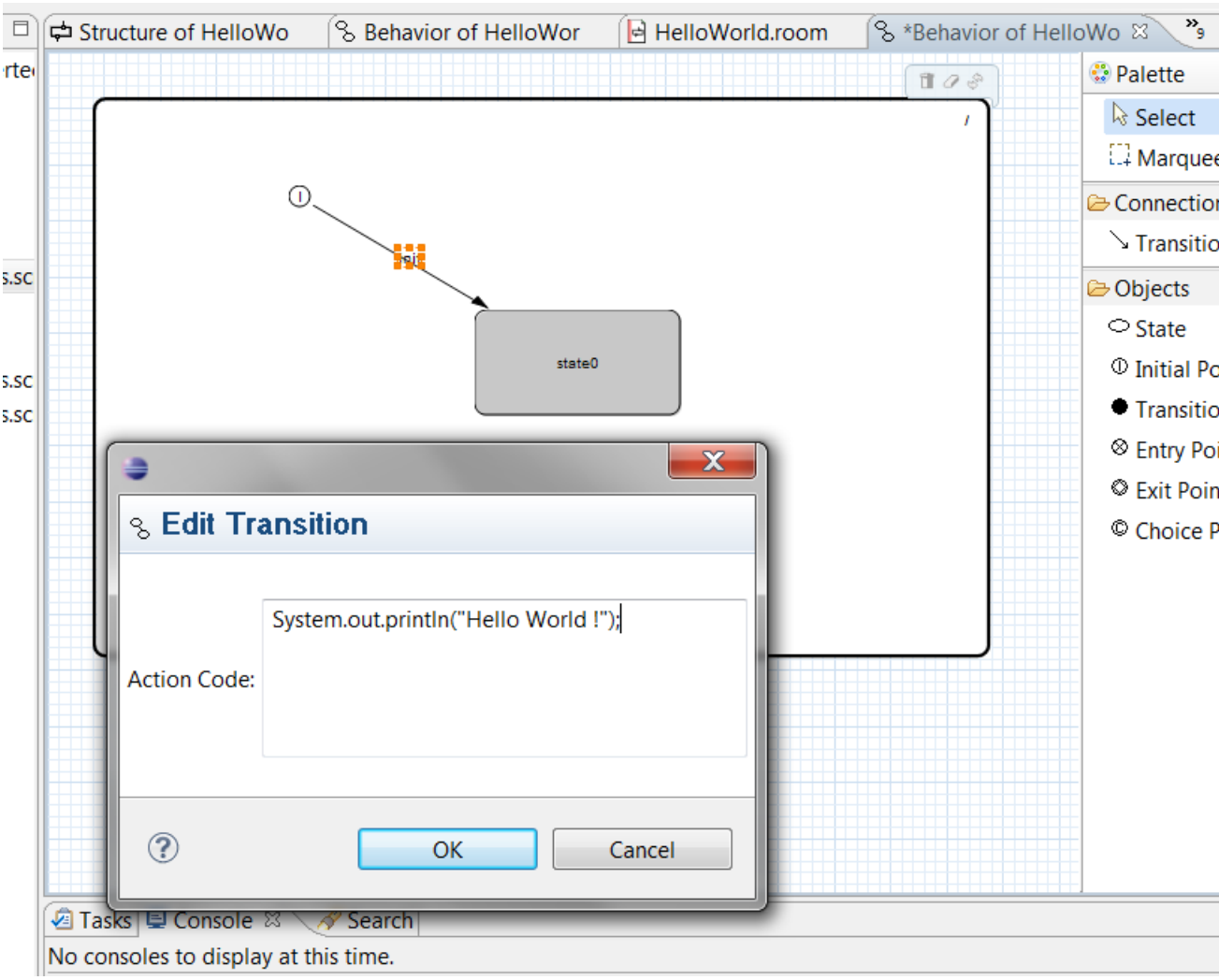


2.2.2. Create a state machine

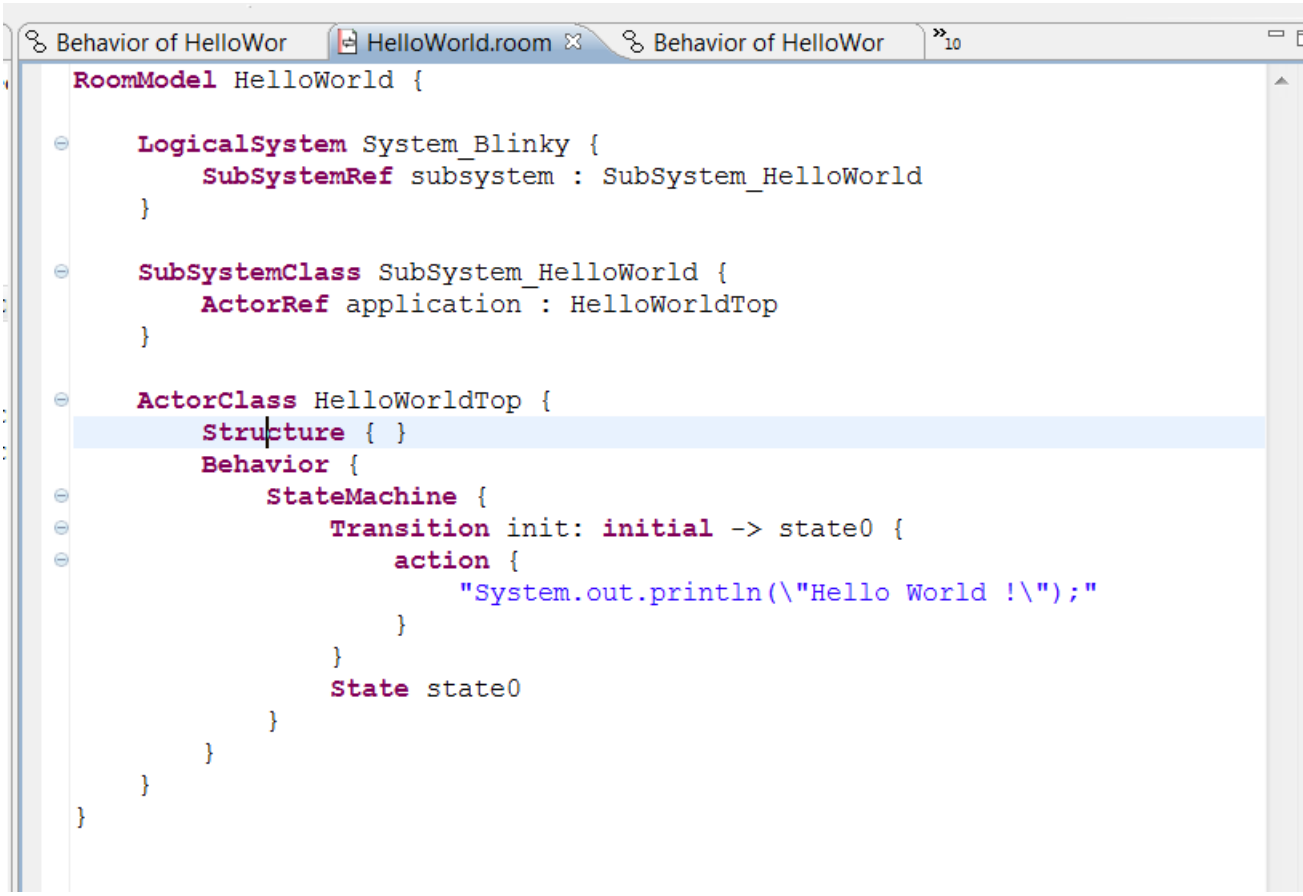
We will implement the Hello World code on the initial transition of the *HelloWorldTop* actor. Therefore open the state machine editor by right clicking the *HelloWorldTop* actor in the outline view and select *Edit Behavior*.



The state machine editor will be opened. Create the state machine as follow. Put the action code to the initial transition.

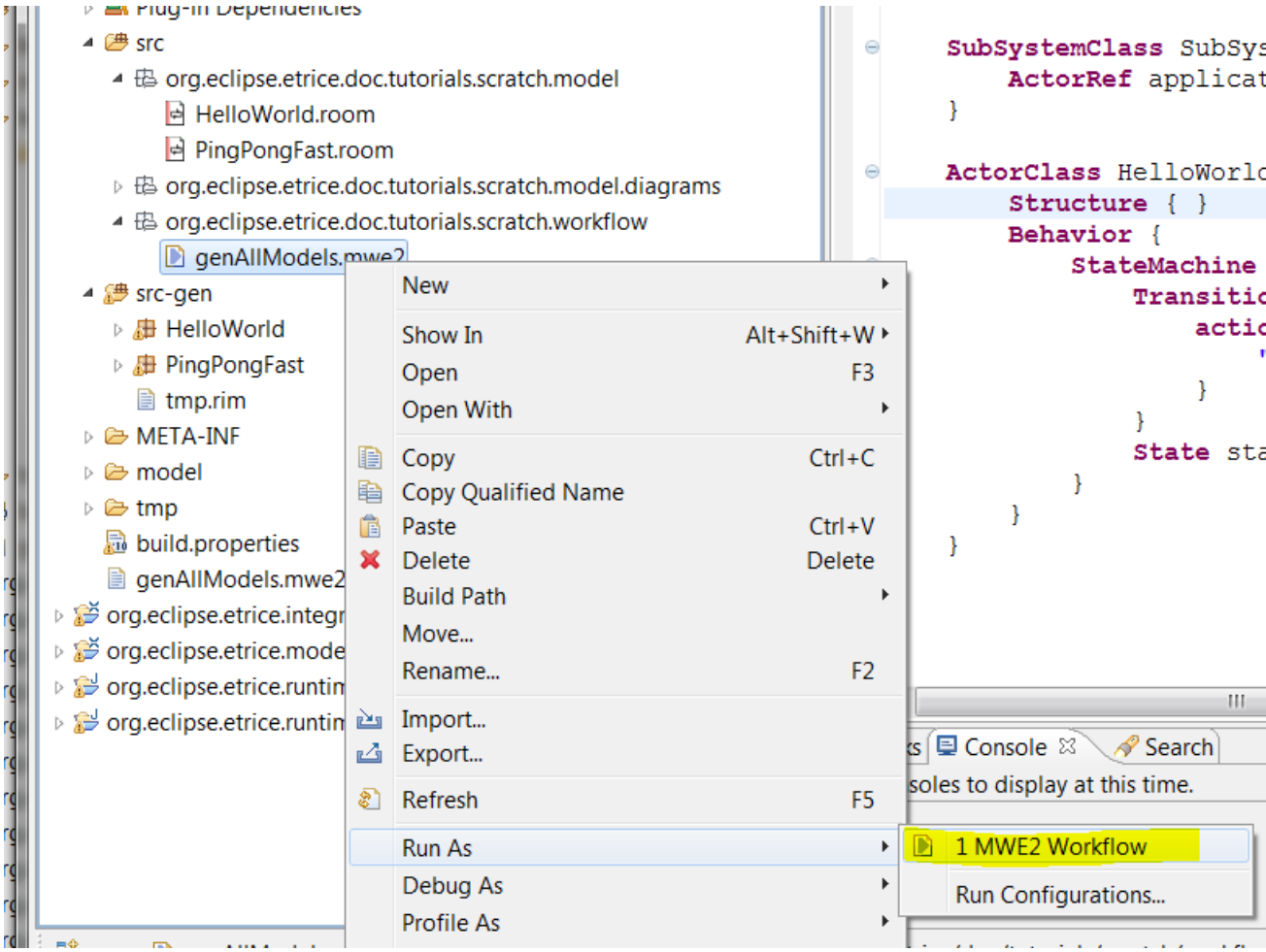


Save the diagram and inspect the model file. Note that the textual representation was created after saving the diagram.

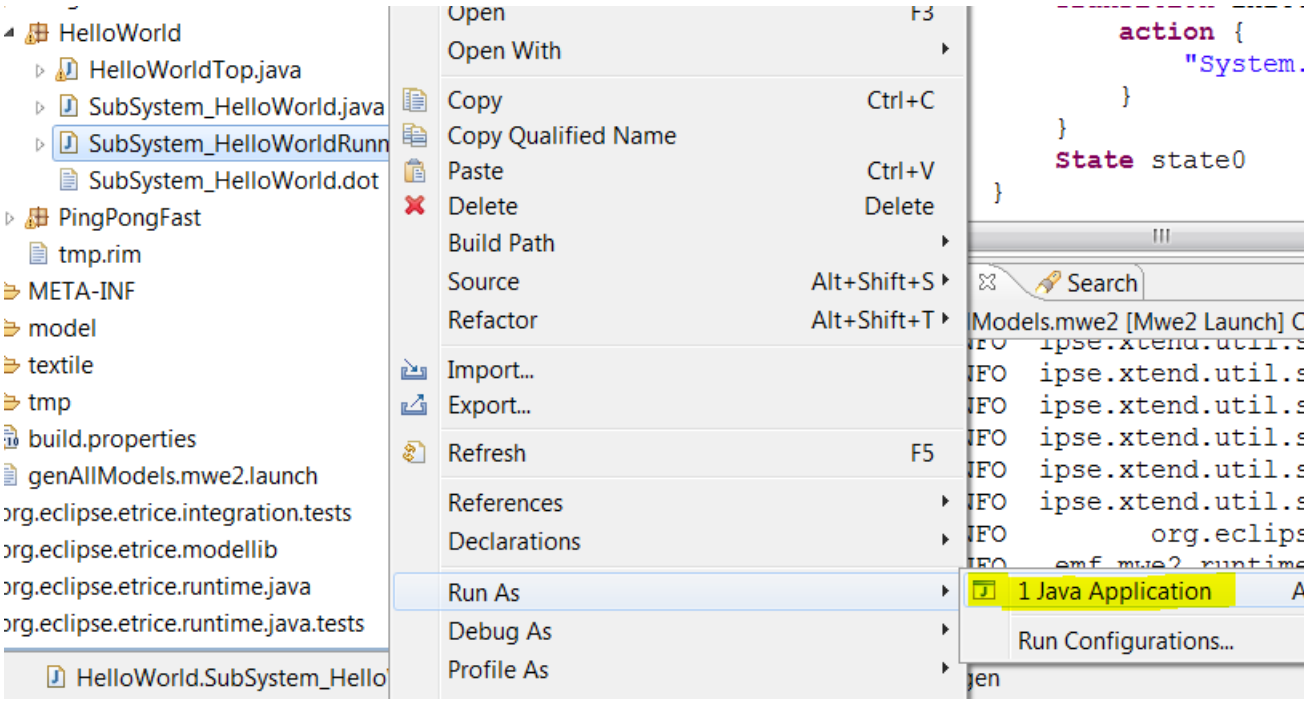


2.2.3. Build and run the model

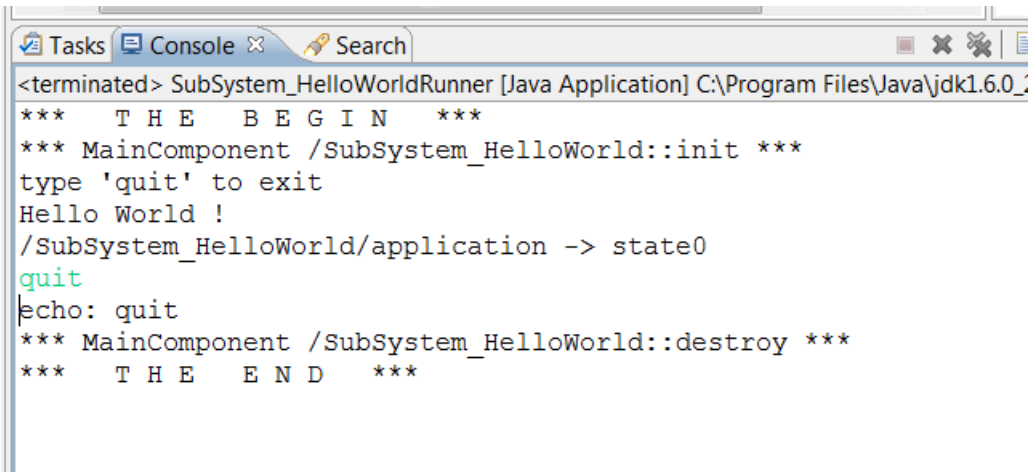
Now the model is finshed and source code can be generated. From org.eclipse.etrice.doc.tutorials.scratch.workflow select genAllModells.mwe2 and run it as MWE2Workflow. Currently all models in the directory will be generated.



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

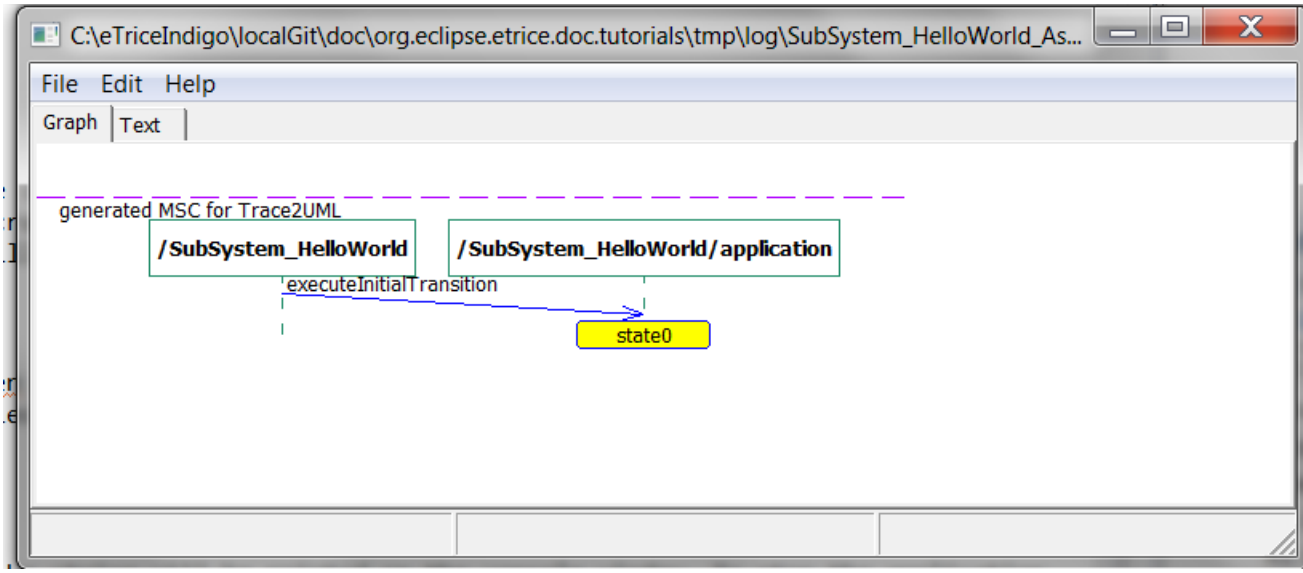


The Hello World application starts and the string will be printed on the console window. To stop the application the user must type *quit* in the console window.



2.2.4. Open the Message Sequence Chart

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



2.3. Summary

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