Getting Started with Papyrus for RealTime v1.0

1. Introduction

This tutorial will show the creation of a simple model using **Papyrus for Real Time version 1.0** (based on Eclipse Oxygen).

As a precondition to going through this tutorial, you must have Papyrus for Real Time installed.
Please see https://wiki.eclipse.org/Papyrus-RT/User#Installation for information on installing Papyrus for Real Time.

Note: The instructions in this tutorial are illustrated using Linux. Steps and images may differ slightly if the installation is done on a different operating system (both Windows and Mac OS are supported for developing models). Some of these differences have been indicated when known and some may be missing.

This exercise will show the creation of a project and model and how UML-RT concepts can be used to easily create the application's structure and behavior.

At its base, a UML-RT model consists of capsules (UML active classes with composite structure) that communicate through ports defined by protocols (collaboration specifications) These protocols specify the messages (signals) that can be exchanged between capsules, as well as their payloads. Hierarchical state machines are used to represent the behavior of capsules, where transitions between states are triggered by messages received on the capsule's ports.

If you are not familiar with UML-RT and want to know a bit more, you should take a look at the Papyrus-RT Overview page (note that the Papyrus-RT-specific visual elements are not shown on that page, but the concepts stand).

The model that will be created as part of this tutorial is a simple "PingPong" model. In this model, implemented using UML-RT, two players will be playing an eternal game of ping pong.

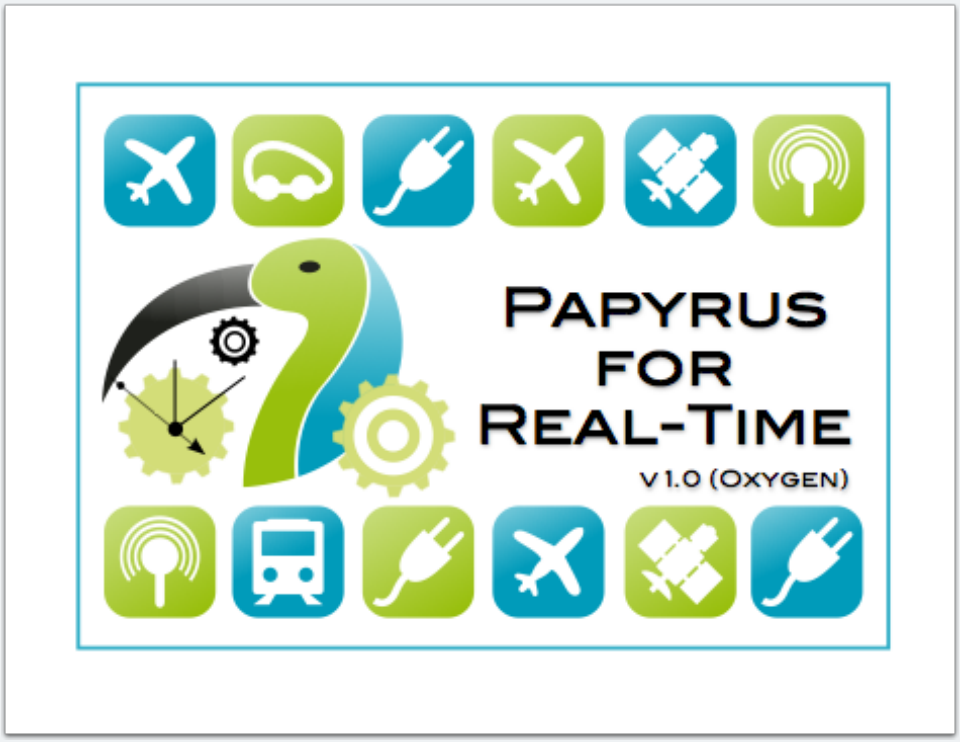
It is a very simple model that will show how a UML-RT model is constructed. Each player will be portrayed using UML-RT capsules and a UML-RT protocol will be used to define how the ball is exchanged between players through ports on each player capsule.

Note: For brevity, "Papyrus-RT," the short form of "Papyrus for Real time," will be used throughout this document.

2. Starting Papyrus for Real Time

Start Papyrus for Real Time. This is typically done by double-clicking on its executable (on some platforms, the executable may be found under the "*papyrusrt*" folder).

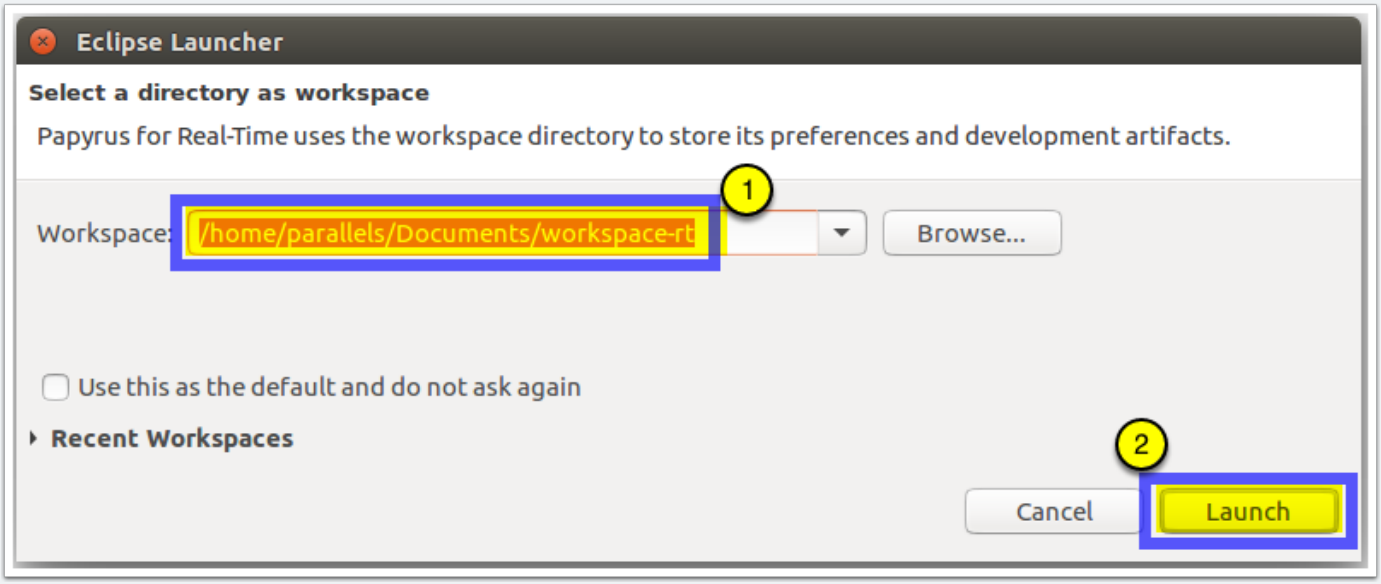
As it starts, the launcher will display the Papyrus for Real Time splash screen.



2.1 Creating a workspace

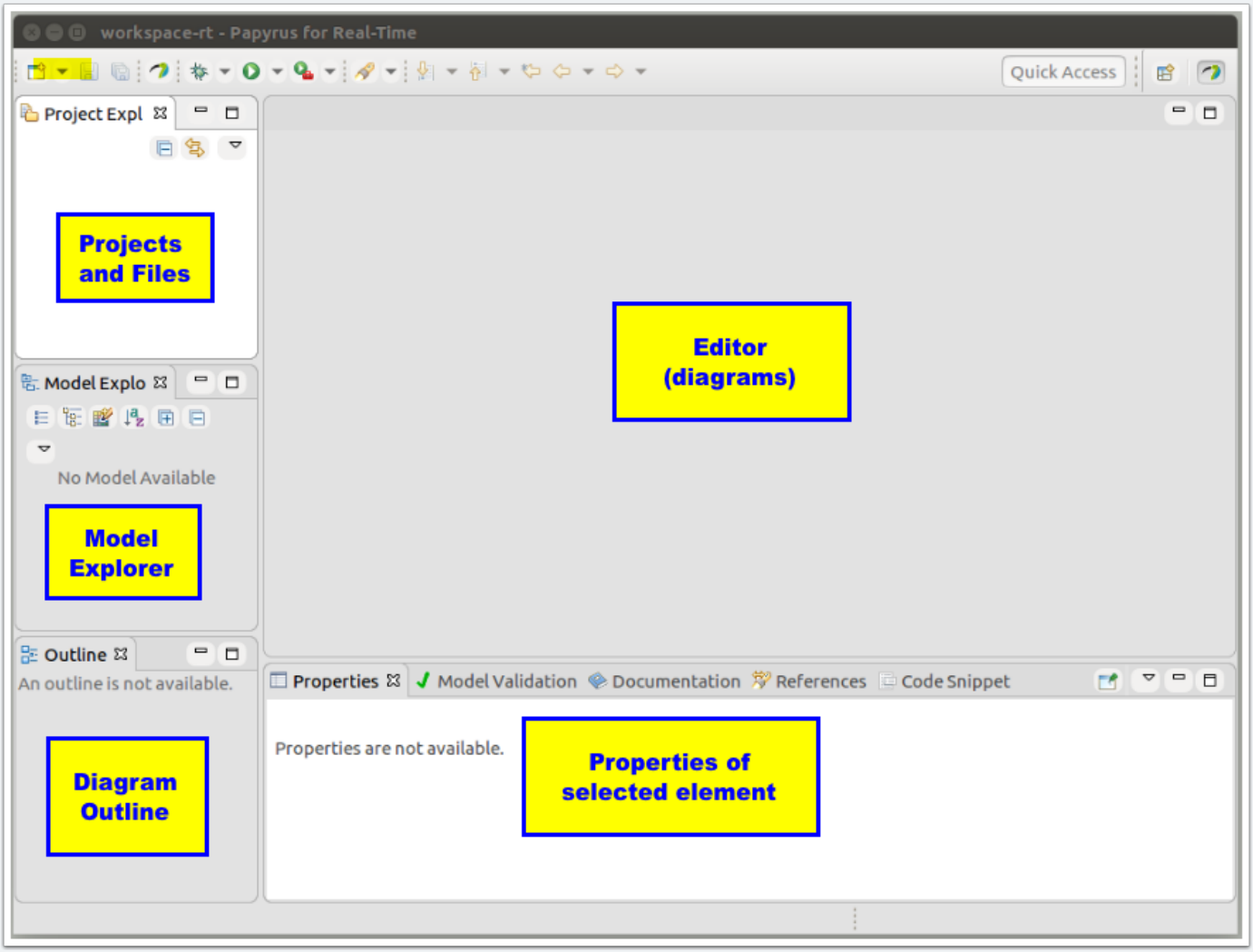
Workspaces are Eclipse's way of creating development environments for various tasks. You can use workspace for different projects, different aspects of a project, or different tasks.

1. After the launch splash screen is dismissed, you will be asked to specify a workspace. You can opt for the provided location or substitute your own.
2. Click on **[Launch]** to continue.



2.2 The workspace

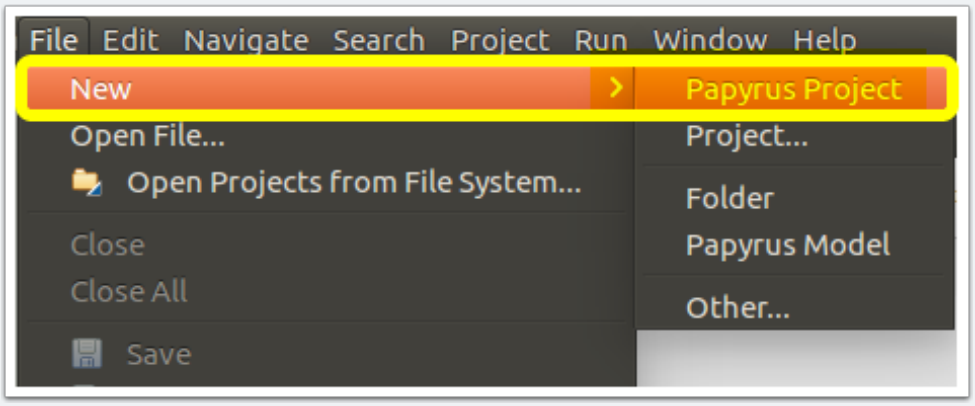
You are now presented with the Eclipse workspace for Papyrus-RT



3. Create a Papyrus for Real Time Project containing a UML-RT model.

Papyrus for Real Time is a Domain-Specific Modeling Language (DSML) tool based on Papyrus. We will use the Papyrus Project creation wizard to create a project configured for Papyrus for Real Time.

3.1 Select File -> New -> Papyrus Project



3.2 Select the architecture context for the model

The architecture context defines the type of model you will be using to build you application. Papyrus can support various architecture model and provides extension points to add more if needed.

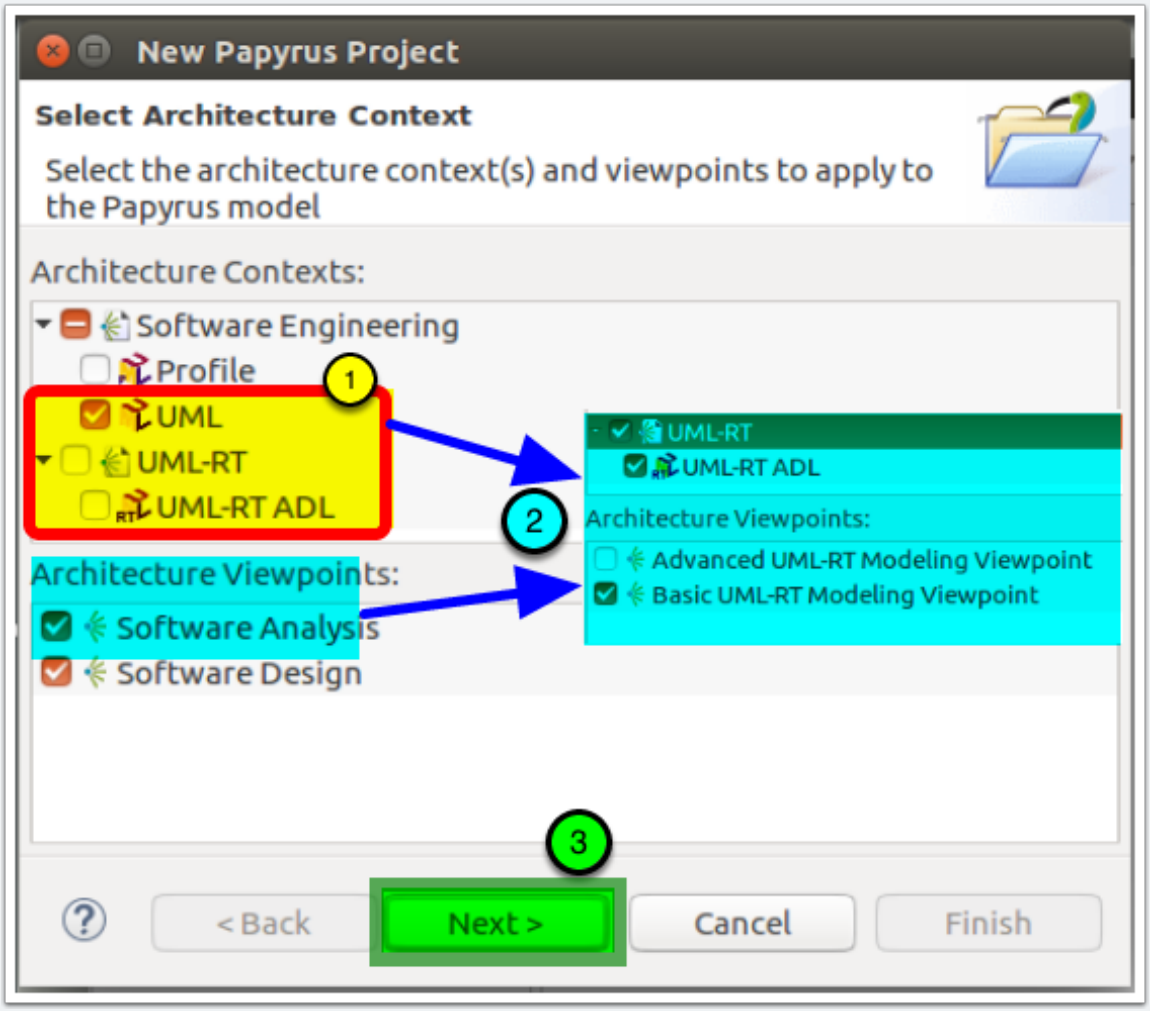
In the displayed dialog:

1. Select the **UML-RT** architecture context

2. Notice that the by clicking on UML-RT, both the architecture context and the viewpoints change to the ones required for UML-RT modeling.
3. Click on **[Next]**.

You will notice that there are two "UML-RT Modeling" viewpoints. The difference between the two is in the amount of information displayed to the user, e.g., in the model explorer. Most users would typically only need to use the "Basic" variant. Some "super-users" and "toolsmiths" could benefit from the added information.

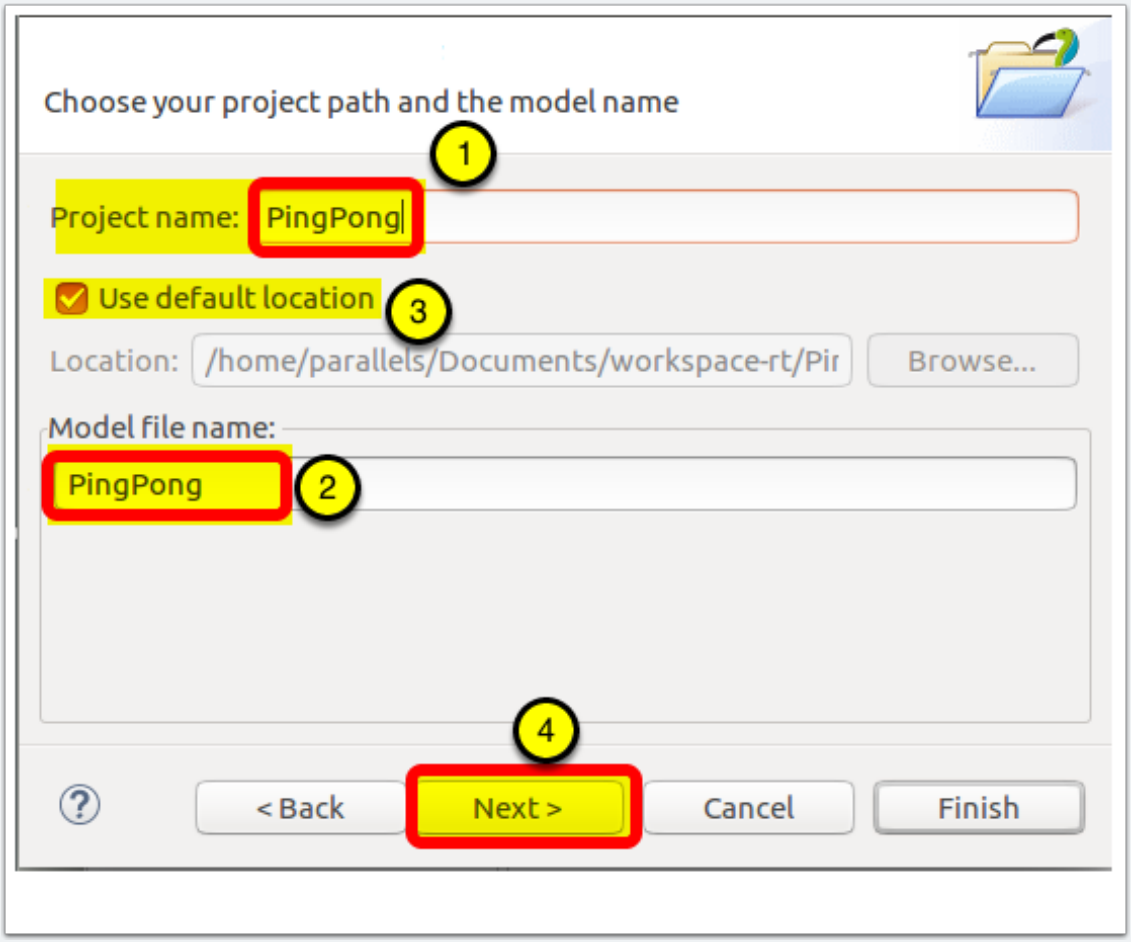
In the context of this tutorial, we will stay with the "Basic UML-RT Modeling Viewpoint."



3.3 Define the project

You can now define the project's name and its location, as well as the name of the model file that will be created.

1. Enter the **name** for the project. This project can hold multiple artefacts and will contain a Papyrus model by default. For this tutorial, we will use the name "**PingPong.**"
2. As you type the name of the project, you will note that the model file name is filled with the same information. It is common to to use the same name for both. If you do need to have a different name for the model file, you can change it from the "Model file name" entry field.
3. By default, the project will be created in the current workspace. You can, however, select an alternative location if, for example, you wish to store your project under source control.
4. Click on **[Next]**



3.4 Provide model initialization information

There is more information that can be provided to create a useful model.

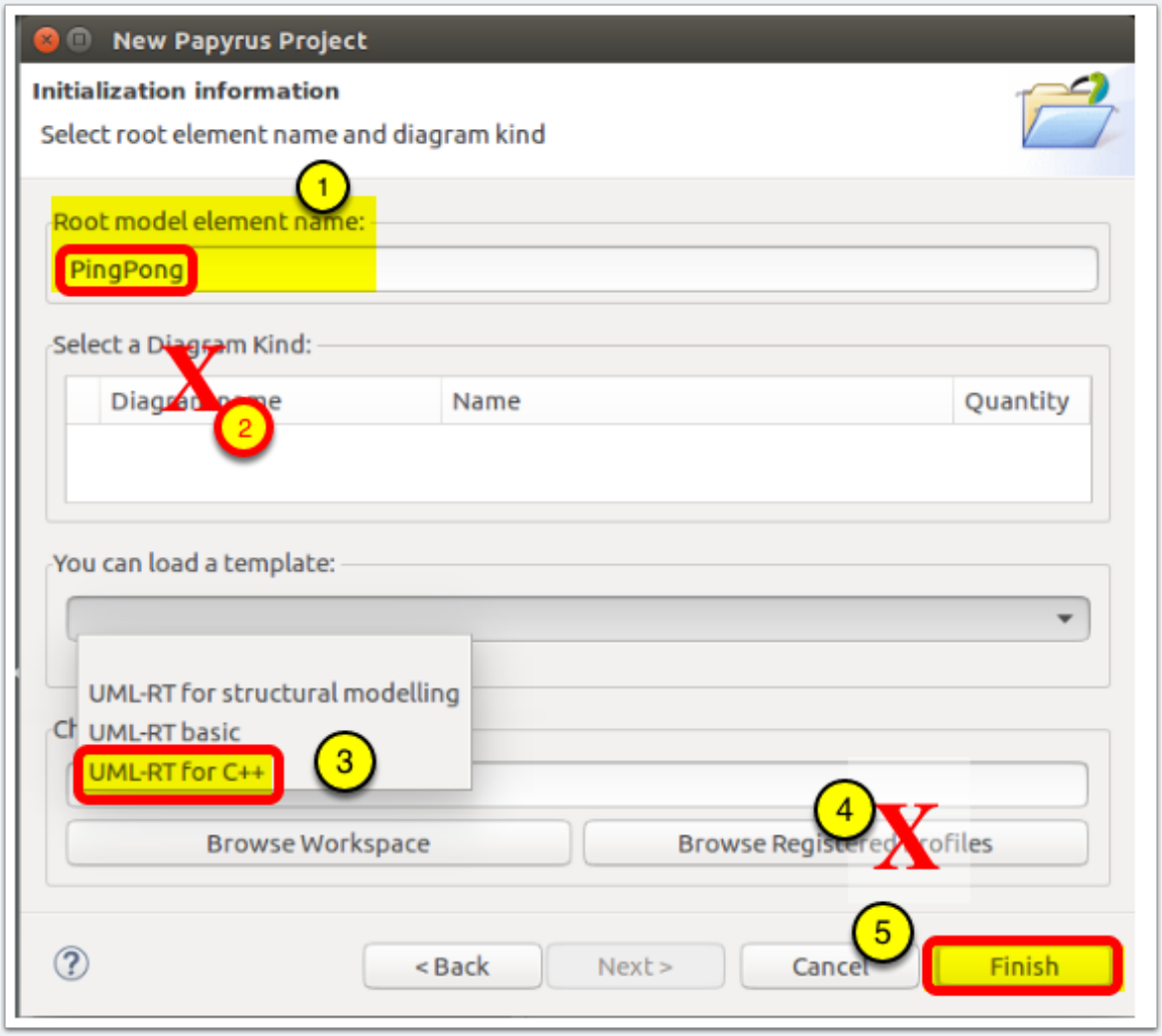
1. The "Root model element" is a representation of the model itself. For this tutorial, and for consistency, we will name it "**PingPong.**"
2. Typically, for UML-RT modeling, we would not start from a diagram. In addition, the two basic diagrams for UML-RT will be created automatically as part of the modeling workflow. As such, we will **not** select any "**Representation Kind**."
3. The wizard let's you select a template of the model you are creating, select "**UML-RT for C++**"

4. We will also **not** select to apply any **profile**. The project/model creation wizard automatically applies those that are required to create a Papyrus-RT model. This part of the dialog could be used to add other profiles that deployed in your environment or that are provided by other add-ons.
5. Click on [**Finish**] to create the model.

There are three template that you can select in the v1.0 version of Papyrus-RT:

1. **UML-RT for Structural Modeling**: only provides only support for capsule, ports, and protocol. Capsule state machines are not created by default.
2. **UML-RT basic**: provides all UML-RT capabilities, including automatic creation of state machines to express the behaviour of capsules. However, no target language is set for code generation
3. **UML-RT for C++**: provides all the capabilities to create UML-RT models that can generate C++ code.

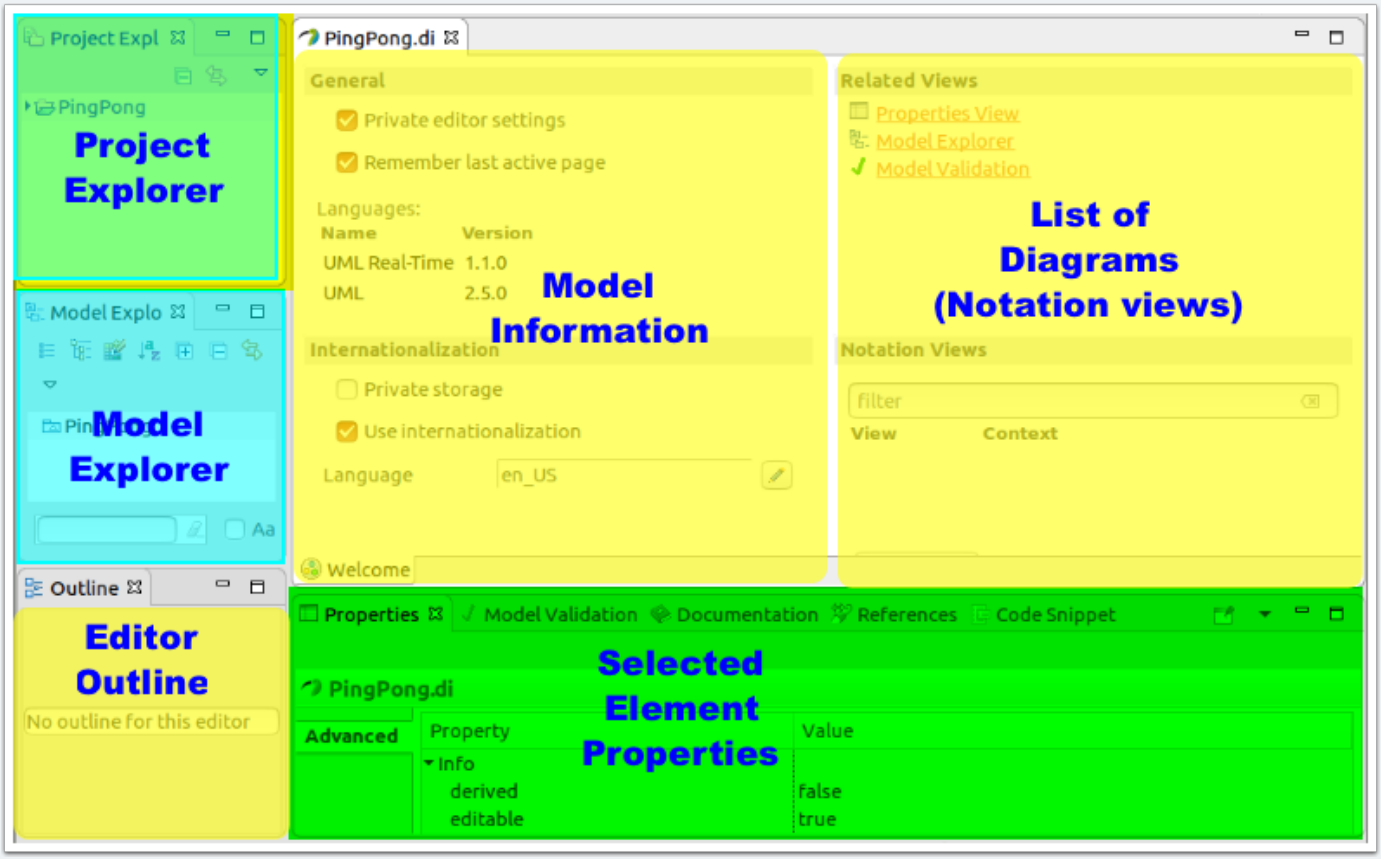
Note that these templates build on top of each other, so even if you select one of the first two template, you can still add the profiles and libraries to get to the "UML-RT for C++" configuration later, once you are in the tool.


New Papyrus Project
Initialization information
Select root element name and diagram kind
1
Root model element name:
PingPong
Select a Diagram Kind:
Name
Quantity
2
You can load a template:
UML-RT for structural modelling
UML-RT basic
UML-RT for C++
3
Browse Workspace
4
5
< Back
Next >
Cancel
Finish

3.5 Project and Model Created

You now have an empty model ready to be populated!

'''Note/Tip''': Many Papyrus-RT users like to have the Code Snippet View displayed along with the Properties View. This can be accomplished by dragging the Code Snippet View to the right of the Properties View.



4. Our project: PingPong

For this tutorial, we will be creating a simple model of a ping pong game, with two players.

The system will consist of two capsules ("active classes) that will communicate to show whose turn it is to hit the ball. Because the system is run by computer, no errors will be made playing so the game could go on forever... To prevent this, we will add a check to ensure that the game only runs for a pre-determined number of exchanges.

So let's get started!

5. Create a protocol

Let's start by determining how the PingPong ball will go from one player to the other. To do this, we can think of the ping pong ball as a message to the other player to which they need to reply (hit the ball back). In UML-RT, the structure that governs the messages that can be exchanged between entities (players in this case) is a protocol.

Protocols contain protocol messages that define how messages can be sent and received between model elements (called "Capsules" in UML-RT - more on that later). These protocol messages can be incoming, outgoing, or symmetrical (i.e., both ingoing and outgoing).

In the case of protocols that are not purely symmetric, i.e., that have either or both incoming and outgoing messages, there is also a need for the concept of conjugation, i.e., of reversing the role of the protocol, a concept that will be addressed further when used later in this tutorial.

In the case of this *Getting Started* tutorial, we could create a symmetric protocol where there would be a single symmetrical protocol message called, for example, "ball". However, to better explore the concept of protocols, we will define our "PingPong" protocol to have one outgoing protocol message called "ping" and one incoming protocol message called "pong".

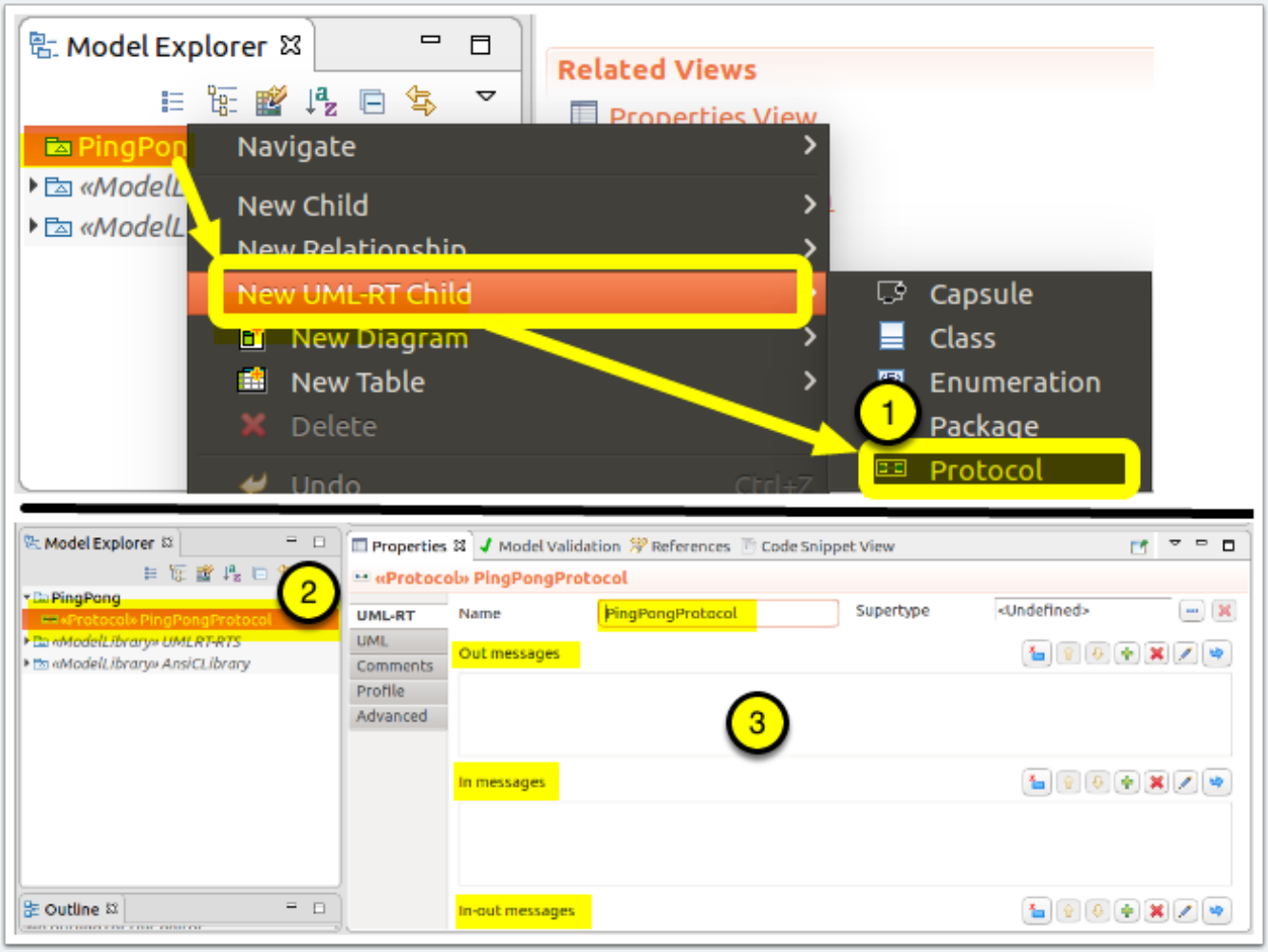
5.1 Create the protocol

The first step is to create the protocol itself.

1. Right-click on the "PingPong" model in the Model Explorer and select "**New UML-RT Child > Protocol**"
2. The name of the protocol is highlighted, name the protocol "**PingPongProtocol**" and hit return.

You can also see the protocol and its messages in the Properties view.

You now have a "protocol" in the Model Explorer.



5.2 Add protocol messages to the protocol

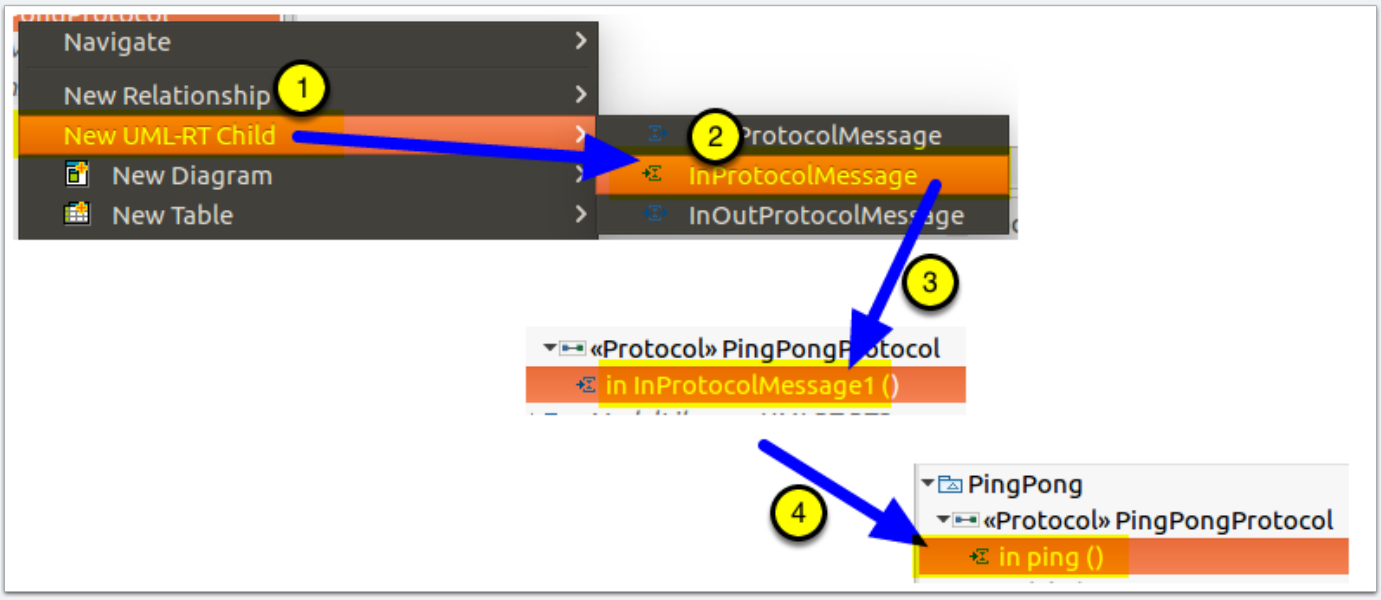
As mentioned previously, a protocol may contain many different "protocol messages" that specify "*operations*" and their associated data.

Our protocol will "*ping*" its opponent and, in return, will respond to a "*pong*". This tells us that there will be a "ping" outgoing protocol message and a "pong" incoming protocol message.

Tip: Protocols should be defined from the client's perspective. In practice, this means that the provider of the service defined by the protocol will have their ports conjugated and the client's ports will by un-conjugated. This makes sense as there will typically be more clients than service providers, so adopting this best practice will reduce the number of ports that need to be conjugated.

In the case of this model, it does not matter as there is no distinct service provider or client. After all, in a game of ping pong, both sides "serve"! For this tutorial, we will consider that the player that starts the game will be sending the "ping" and will therefore be the "server" / "Service Provider."

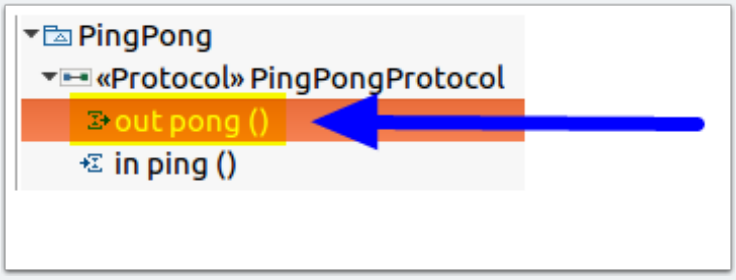
1. Right-click on the protocol
2. Select to create a "**New UML-RT Child > InProtocolMessage**".
3. The protocol message is created and its name is ready to be edited in the model explorer in the model explorer .
4. Rename the protocol message to "ping" (remember, as the server, the port will be conjugated, so it will have to be the opposite of the action we expect, so incoming).



5.3 Add the "pong" Protocol Message

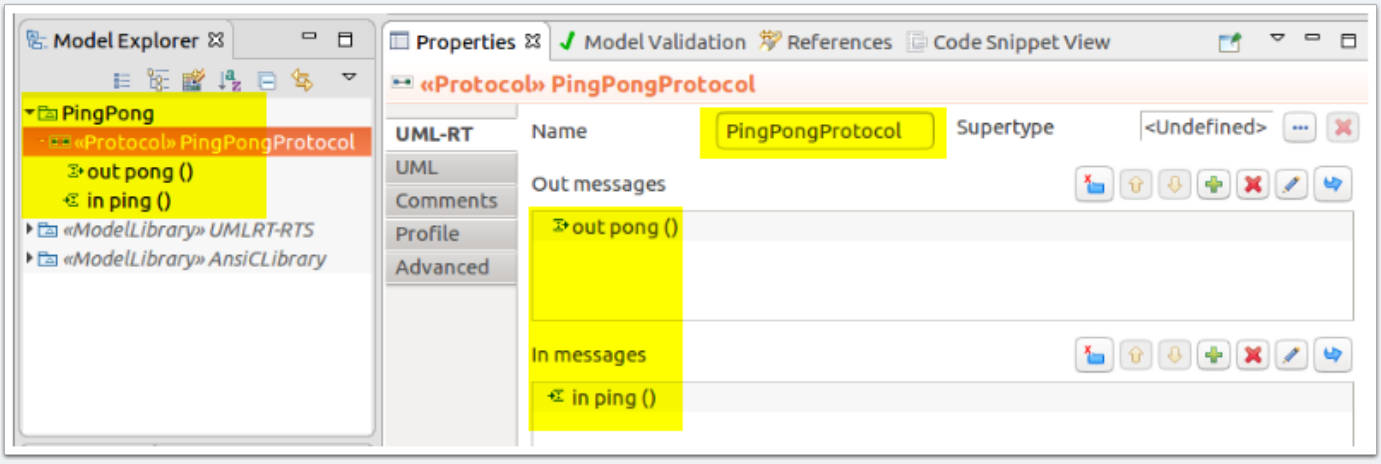
To create the "pong" message, use the same process as for the creation of the "ping" protocol message, above, except that you will instead create a "ProtocolMessageOut" and name the protocol message "pong"

1. Right-click on the PingPong protocol and select "**New UML-RT Child > OutProtocolMessage**"
2. Rename the resulting protocol message to "**pong**"
3. You now have a "**pong**" protocol message in addition to "ping".



5.4 The "PingPong" protocol is now complete

The protocol and its protocol messages can now be seen in both the Model Explorer and the Properties view.



6. Defining the Tutorial's "PingPong" System's Structure

Now that we have a protocol, we can move on to creating the structure of the "PingPong System."

We will need to create three capsules for this tutorial model:

- **Two capsules that will be representing the two players: "Pinger"** ("player 1") **and "Ponger"** ("player 2").
  We will be creating two capsules to highlight some aspects of the communication mechanisms, especially related to how ports are used with asymmetric protocols. If we had a symmetric protocol, we could simply use two instances of the same capsule, but that would be a less interesting model for a tutorial. Also note that, in this example, the two capsules will be slightly different as one has the added responsibility to start the game

- **A top capsule representing the complete system ("Top")**
  In UML-RT, the building blocks are capsules and there is always a top capsule that represents the system to be built. The complete set of capsules that are required to implement the system's functionality are then included by the containment hierarchy from this top capsule.

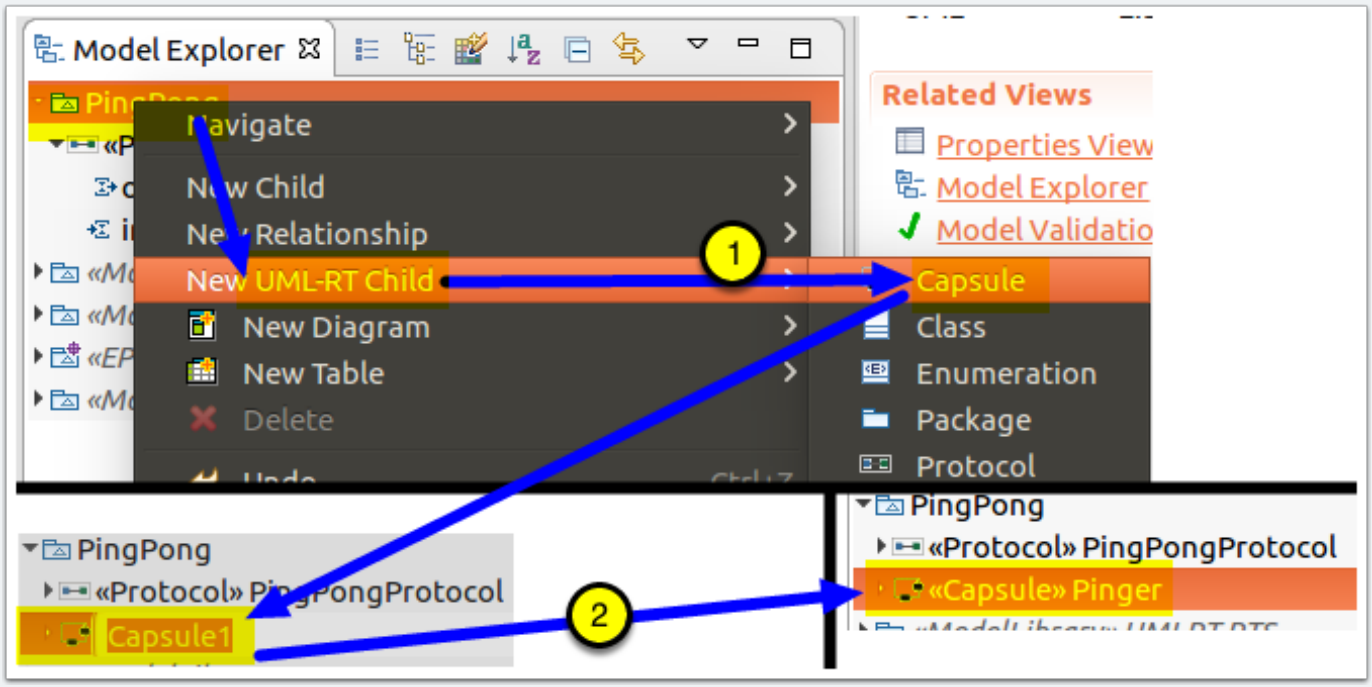
7. Create the Pinger capsule

This capsule will represent the starting player in the game. As such, it will be responsible for sending the first "ball" (message). It will also have to react when receiving the ball from the other player.

7.1 Create the Pinger ("player 1") capsule

Let's start by creating the Pinger capsule.

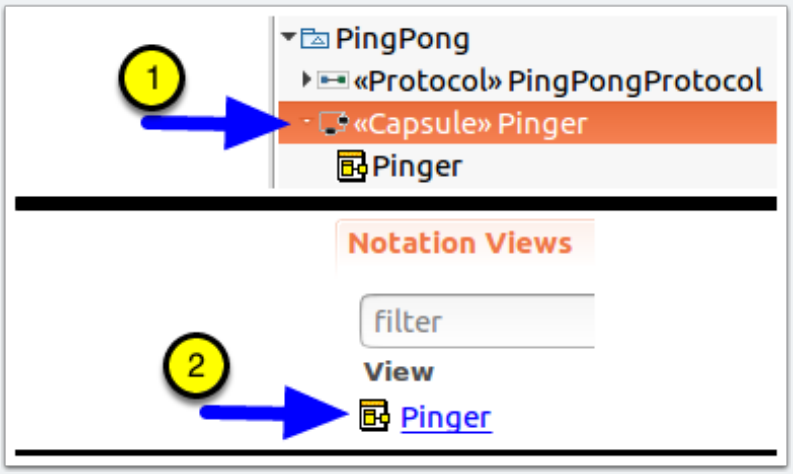
1. Right-click on the PingPong model in the Model Explorer and select "**New UML-RT Child > Capsule**"
2. After the capsule is created, it's name is selected for edition, name it "**Pinger**"



7.2 Look at the Pinger capsule and open it's diagram

Now that we have a capsule, let's have a look at what it contains and at its diagram.

1. Expand the **Pinger** capsule in the model explorer by clicking on the triangle at the beginning of its Model Explorer entry. You will notice that there is already a contained element: **Pinger**'s diagram link. In Papyrus-RT, every capsule has a capsule diagram, this link is a quick way to open it
2. That diagram link is also accessible from the "**Editor**" under the **View** heading.

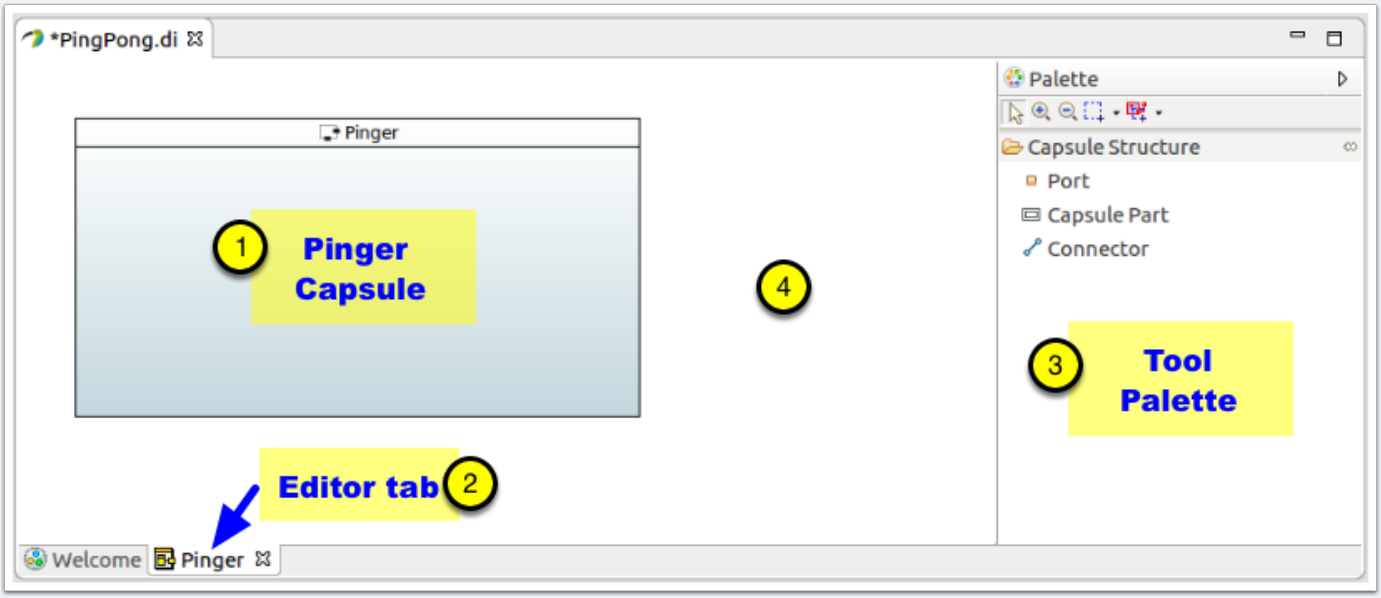


7.3 Open Pinger's capsule diagram

To do this, simply double-click on either of the diagram described in the previous step (or single-click on the blue textual link in the editor view list)

One open, you will see:

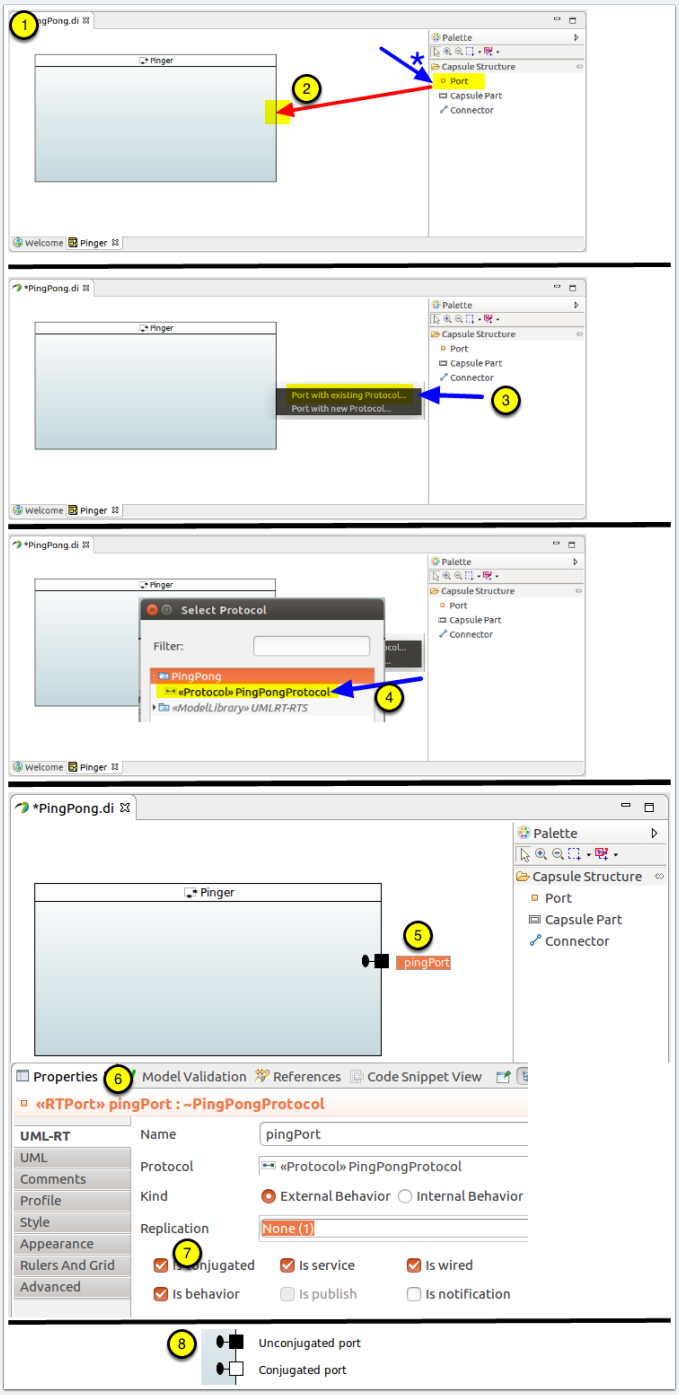
1. The representation of the **Pinger** capsule. It does not contain much right now, but we will work on that in this tutorial.
2. A editor tab is added at the bottom of the editor. This is useful when you have multiple diagram open so you can easily navigate between them
3. A tool palette providing you with the various tools that are relevant when working on a diagram in the context of the capsule's structure (in this case).



7.4 Add an external port to Pinger

In order to be able to communicate, Pinger will need an external port through which it can send messages to other capsules. Let's add this port now.

1. Click on "**Port**" in the tool palette.
2. Click on the right border of the **Pinger** capsule.
3. In the resulting dialog, select "**Port with Existing Protocol**."
4. In the resulting dialog, expand the "**PingPong**" model entry and select the **PingPongProtocol** that was created earlier and click [OK] to close the dialog.
5. The port name will be set for editing on the diagram, name the port "**pingPort**" and hit Enter.
6. Since we have decided that **Ping** would be the "server**"** capsule, we also have to change the conjugation of its port. To do so, look in the **Properties** view just below the **Editor**, and make sure that the "**UML-RT**" tab is selected
7. Click on the box to the left of "**Is conjugated**."
8. When the port is conjugated, you will notice that its graphical element will change from being all black to getting a white fill, graphically showing the conjugation state.

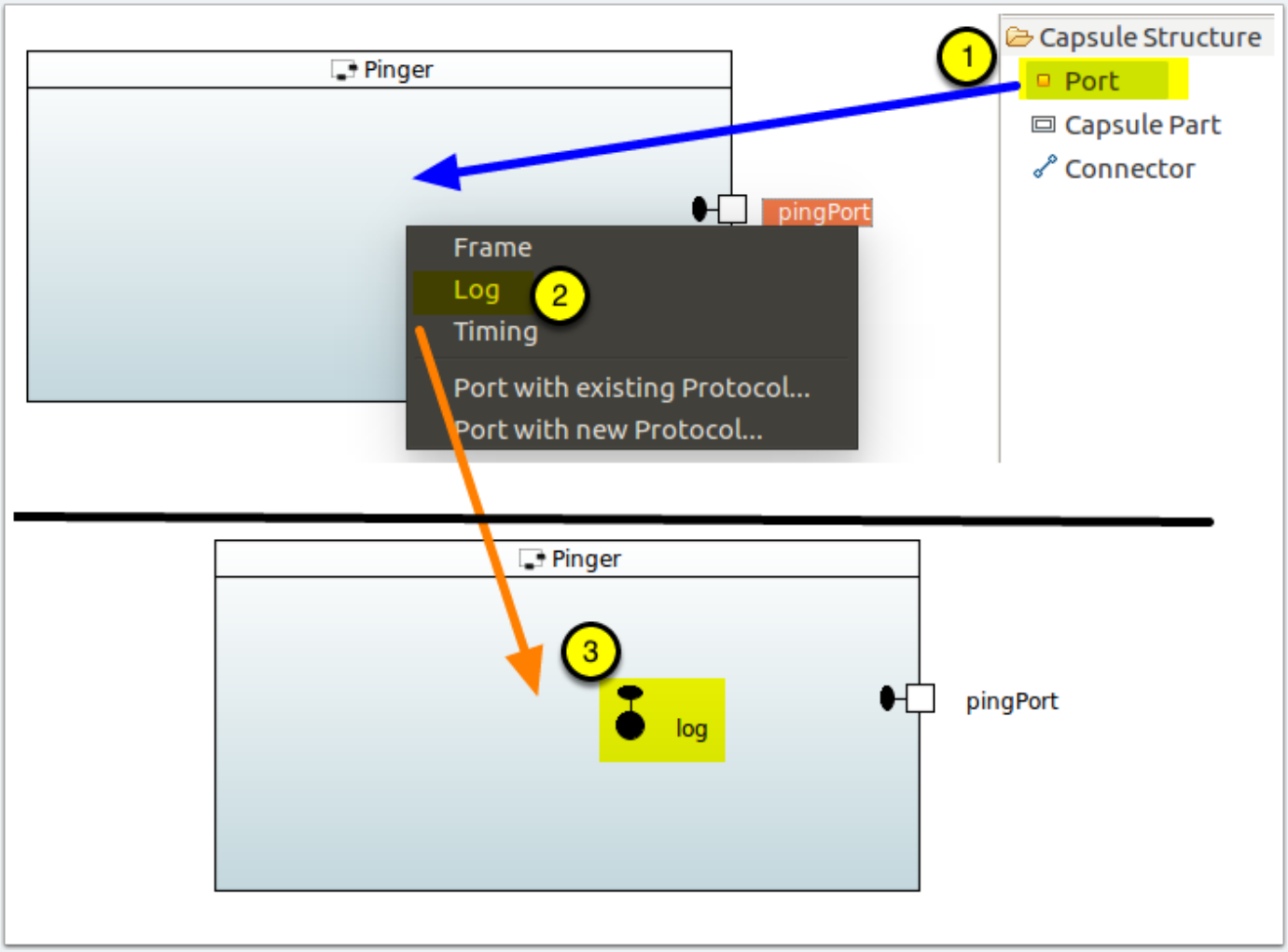

1
2
3
4
5
6
7
8
Pinger
Palette
Capsule Structure
Port
Capsule Part
Connector
Port with existing Protocol...
Port with new Protocol...
Select Protocol
Filter:
PingPong
«Protocol» PingPongProtocol
«ModelLibrary» UMLRT-RTS
pingPort
Properties
Model Validation
References
Code Snippet View
«RTPort» pingPort : ~PingPongProtocol
UML-RT
UML
Comments
Profile
Style
Appearance
Rulers And Grid
Advanced
Name
pingPort
Protocol
«Protocol» PingPongProtocol
Kind
External Behavior
Internal Behavior
Replication
None (1)
Is service
Is wired
Is behavior
Is publish
Is notification
Unconjugated port
Conjugated port

7.5 Add a log port

In order to make sure that the model actually runs correctly, we will need to display some information to the user. The Papyrus-RT runtime provides a logging service that allows models to print out information to the standard output (e.g., the screen). In the current version, it only does this to stdout, but this capability is extensible to use other output targets.

So all that is needed to be able to log event to the screen is to add a log port to the capsule.

1. Select the Port tool and create a port in the middle of the capsule
2. In the resulting dialog, select the "Log" entry
3. Leave the name as is. You have now create a log port through which you can print messages. We will see how to use it when we add the behaviour to the capsule.

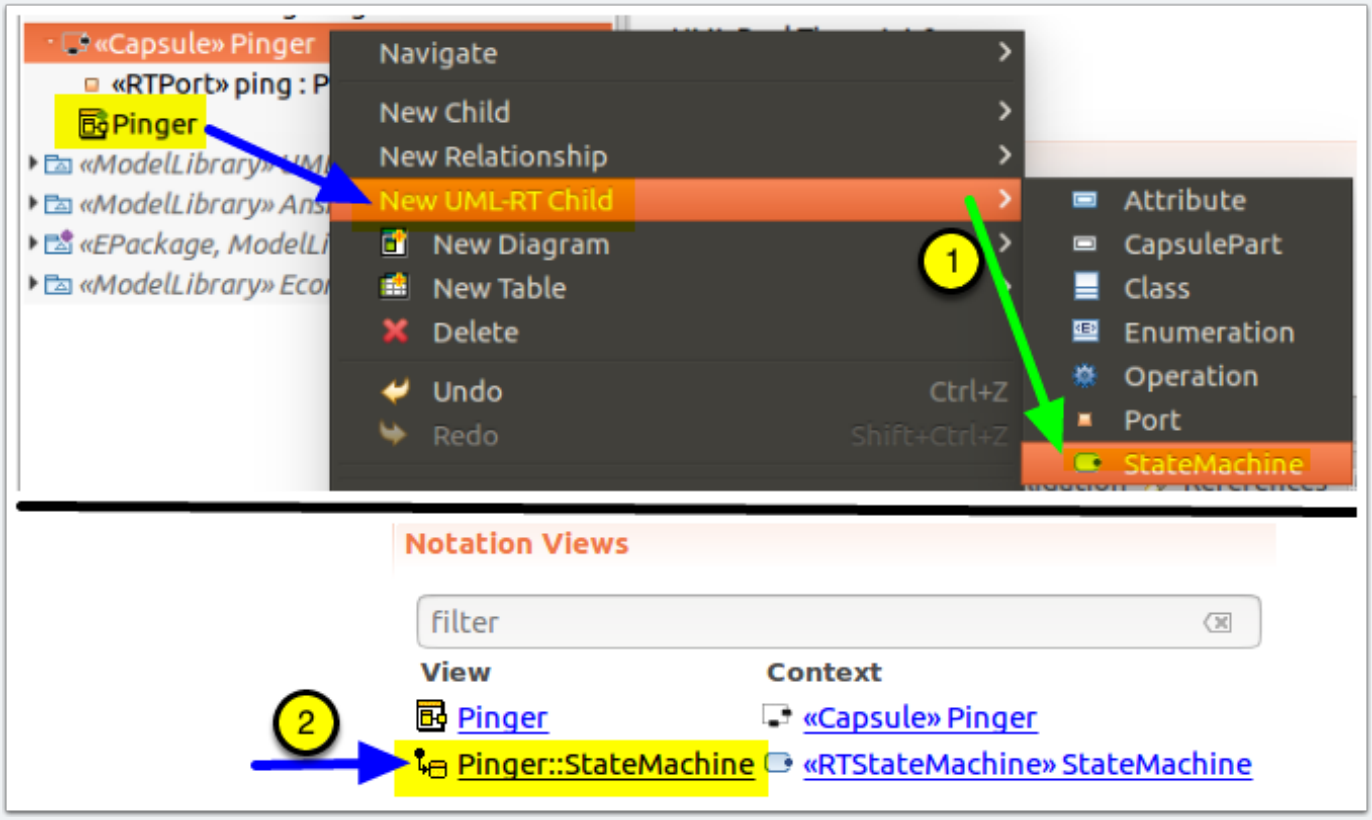


7.6 Create Pinger's state machine

As mentioned, the behaviour of a capsule is represented by a state machine, but one is not provided by default as some capsules may not need to have behaviour. Examples of capsules not needing behaviour are for container capsules, such as "Top", and some patterns, such as dynamic forwarding.

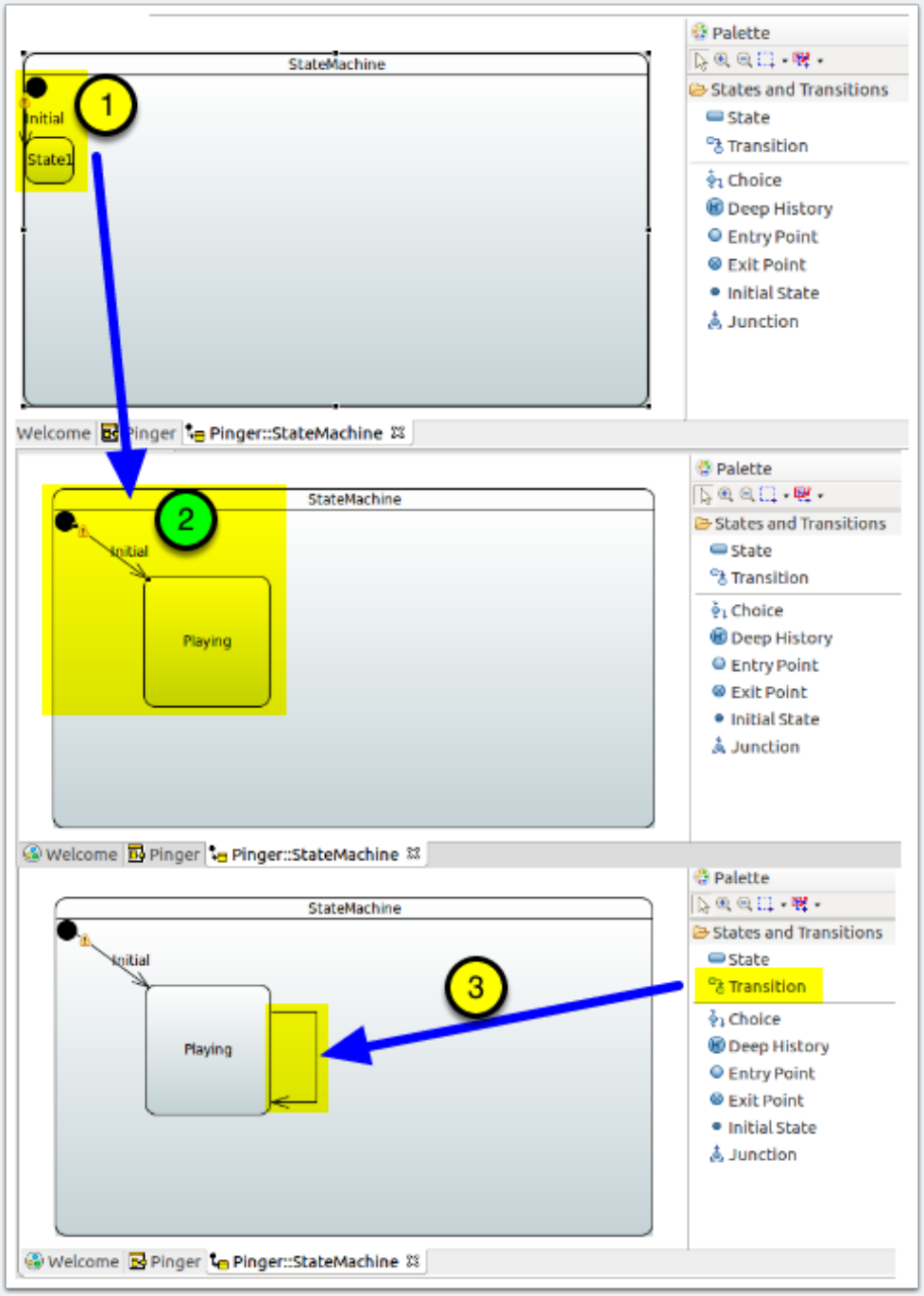
1. Right-click on the **Pinger** capsule in the Model Explorer and select "**New UML-RT Child > StateMachine**" to create the state machine.

2. The state machine is created and its diagram is available from the **Editor**'s **View** list (click on the "**Welcome**" tab at the bottom of the editor area to display the information view).



7.7 Add the Pinger StateMachine behaviour

1. Now that we have a state machine, let's add the behaviour to it: open the **Pinger::StateMachine.** You will notice that we already provide you with the initial pseudostate and a state ("State1") to get you started
2. Move these two initial elements to better positions, as shown and rename "**State1**" to "**Playing**" (hint: use the Properties tab or slowly click twice on the state's border)
3. We will also add a transition that will be taken when the other player returns the ball. Using the transition tool draw a transition from the "**Playing**" state to itself.


StateMachine
Initial
State1
1
Palette
States and Transitions
State
Transition
Choice
Deep History
Entry Point
Exit Point
Initial State
Junction
Welcome
Pinger
Pinger::StateMachine
StateMachine
Initial
Playing
2
Palette
States and Transitions
State
Transition
Choice
Deep History
Entry Point
Exit Point
Initial State
Junction
Welcome
Pinger
Pinger::StateMachine
StateMachine
Initial
Playing
3
Palette
States and Transitions
State
Transition
Choice
Deep History
Entry Point
Exit Point
Initial State
Junction
Welcome
Pinger
Pinger::StateMachine

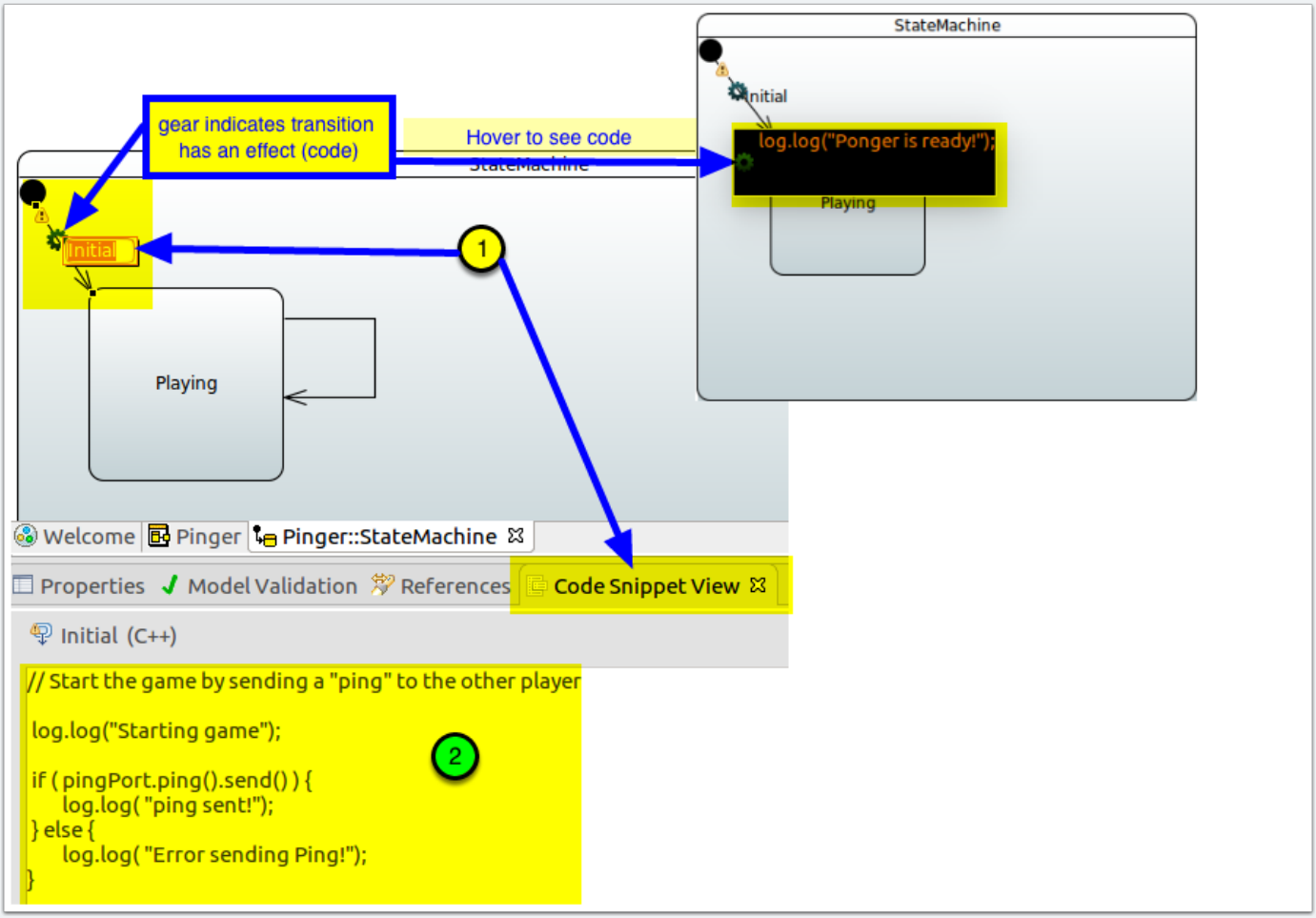
7.8 Add initial transition action

All that is now left is to add some triggers (which allow for transitions to be taken when a message is received) and code blocks (remember, we are using C++ as the "Action Language".

Let's start with the Initial Transition. Note that the initial transition is always taken once, when the capsule is instantiated, so it does not need an trigger.

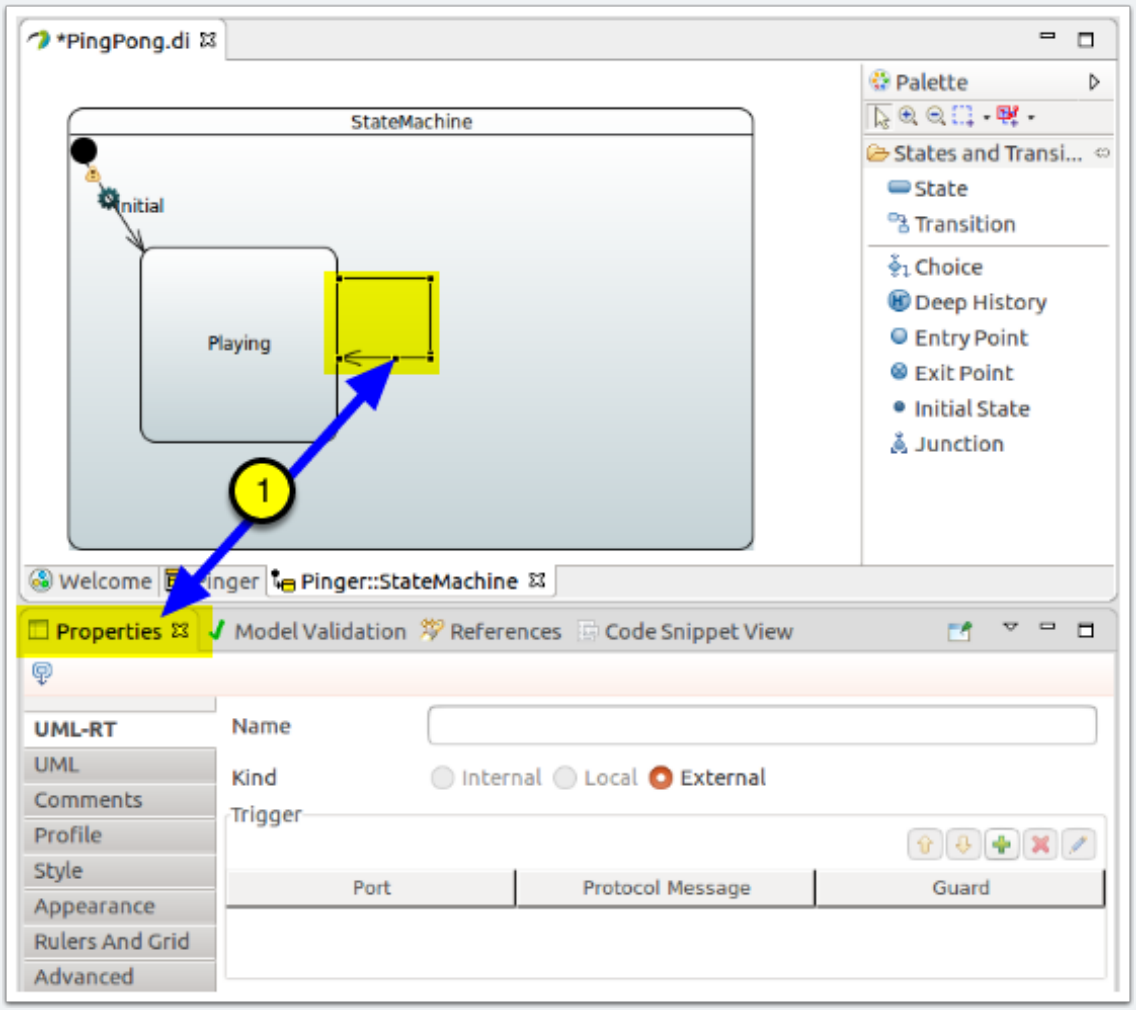
1. Click ont the **Initial** transition and then open the **Code Snippet View**" in the Properties View area. Make sure that the "**Effect**" tab is selected at the bottom of the view. (hint: you may want to move the code snippet view to the right of the properties view, especially if you have a large monitor)
2. In the code snippet view, add the following code: (*note the log messages so we can see the execution and the check on message send success*)

```
// Start the game by sending a "ping" to the other player
log.log("Starting game");
if ( pingPort.ping().send() ) {
        log.log( "ping sent!");
} else {
        log.log( "Error sending Ping!");
}
```


StateMachine
Initial
log.log("Ponger is ready!");
Playing
gear indicates transition has an effect (code)
Hover to see code
Initial
Playing
1
Welcome
Pinger
Pinger::StateMachine
Properties
Model Validation
References
Code Snippet View
Initial (C++)
// Start the game by sending a "ping" to the other player
log.log("Starting game");
if (pingPort.ping().send()) {
log.log("ping sent!");
} else {
log.log("Error sending Ping!");
}
2

7.9 Edit the trigger and code for self transition

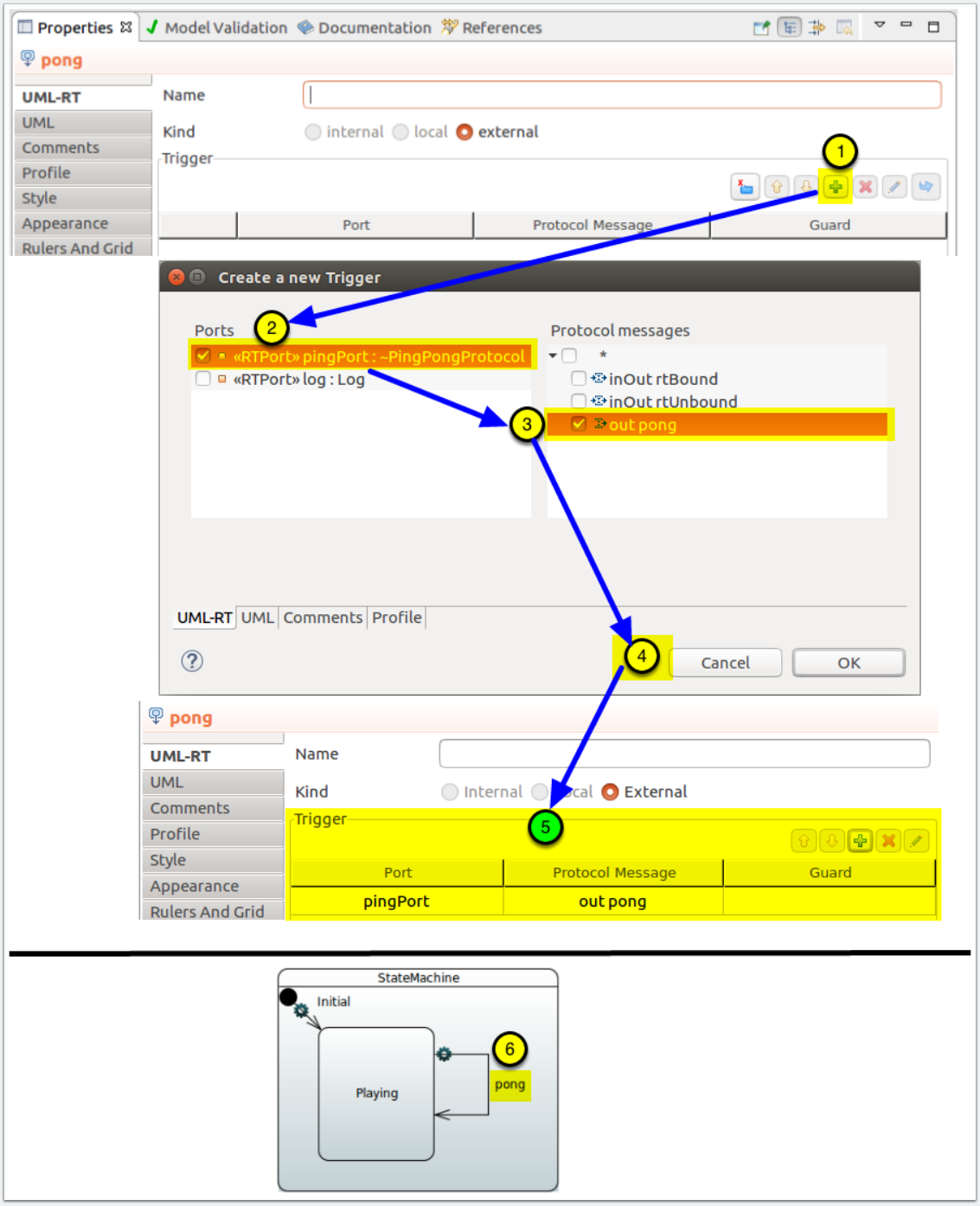
1. Select the transition in the diagram and switch to the **Properties** view.



7.10 Add the transition trigger

1. In the Properties view, click on the [+] button in the **Trigger** section to create a new trigger.
2. From the resulting dialog, select the **pingPort**.
3. From the list of protocol messages, select the **pong** protocol message.

4. Click OK to set the trigger.
5. You now have a trigger defined for this transition. When the model runs, any message received while in the **Playing** state will result in this transition being taken.
6. Notice that the transition name has be set to the name of the protocol message selected, providing a clue on the diagram as to when the transition is triggered.

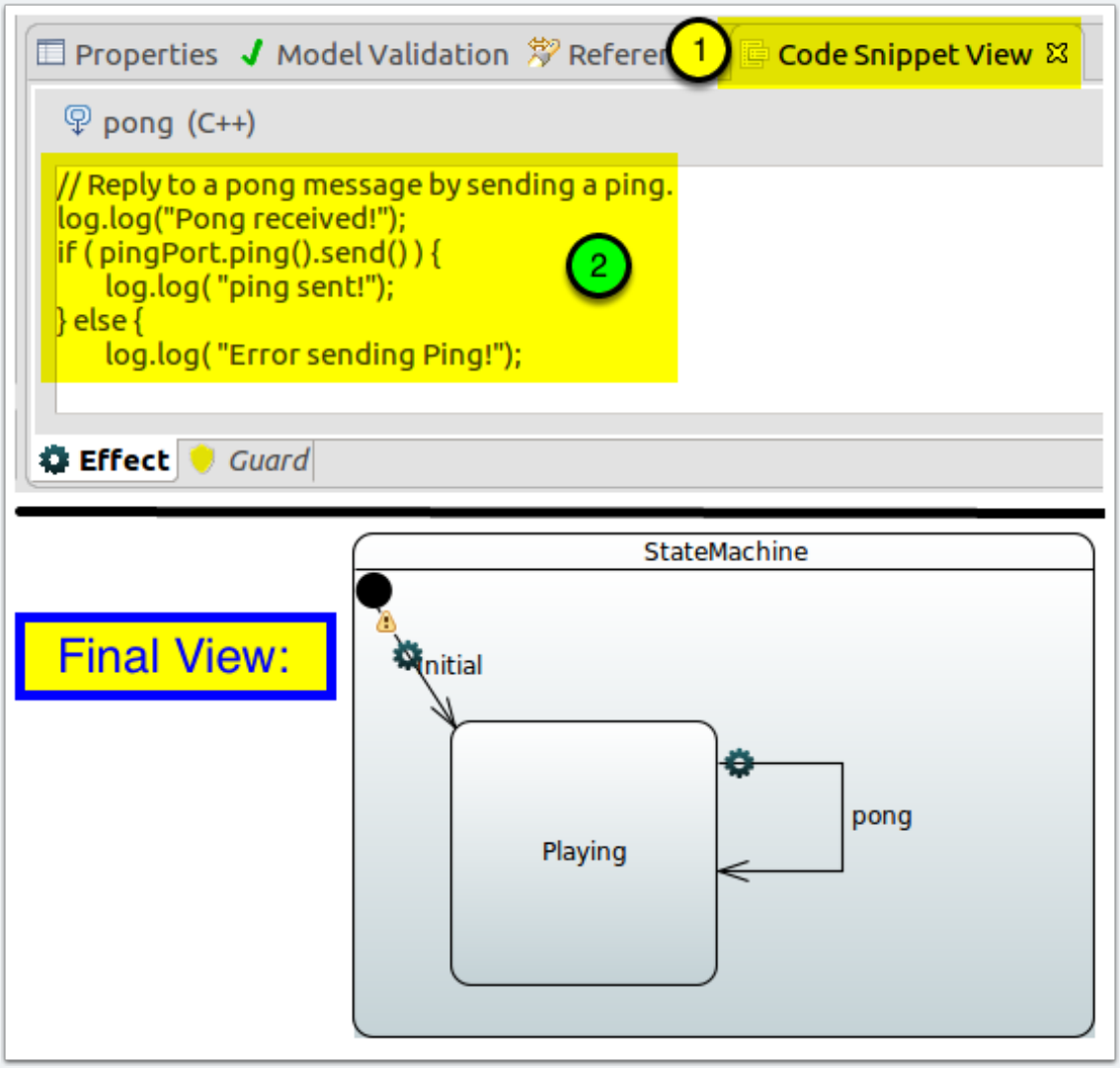

Properties
Model Validation
Documentation
References
pong
UML-RT
UML
Comments
Profile
Style
Appearance
Rulers And Grid
Name
Kind
internal
local
external
Trigger
Port
Protocol Message
Guard
1
Create a new Trigger
Ports
2
«RTPort» pingPort : ~PingPongProtocol
«RTPort» log : Log
Protocol messages
*
inOut rtBound
inOut rtUnbound
3
out pong
UML-RT
UML
Comments
Profile
4
Cancel
OK
pong
UML-RT
UML
Comments
Profile
Style
Appearance
Rulers And Grid
Name
Kind
Internal
External
Trigger
5
Port
Protocol Message
Guard
pingPort
out pong
StateMachine
Initial
Playing
6
pong

7.11 Add the code for the transition

The only thing left to do for this transition is to add its code. The basic steps are the same as those taken previously to add the code for the initial transition

1. Click on the transition and the **Code Snippet View** to bring up the C++ editor for the transition. Make sure that the "**Effect**" tab is selected at the bottom of the view.
2. Type in the following code (you will notice that it is very similar to that of the initial transition):

```
// Reply to a pong message by sending a ping.
log.log("Pong received!");
if ( pingPort.ping().send() ) {
        log.log( "ping sent!");
} else {
        log.log( "Error sending Ping!");
}
```



7.12 You are now done with the creation of the Pinger capsule!

8. Create the "Ponger" capsule

To create the "**Ponger**" capsule, simply follow the same steps as when **Pinger** was created, with the following differnces:

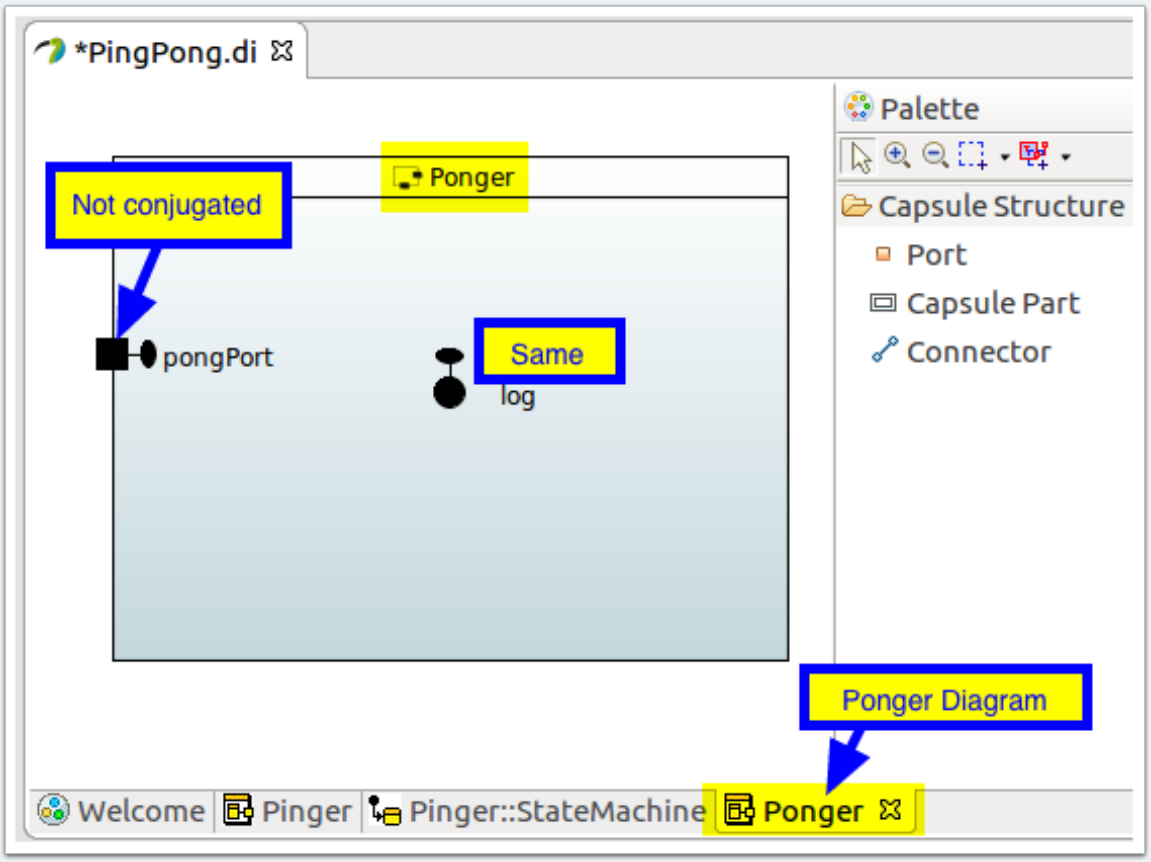
- The capsule will be named "**Ponger**" instead of "**Pinger**"
- The port is named "pong" and is **not** conjugated.
- You should place the port on the left edge of the capsule, instead of the right. This will allow us to put both capsules side by side so that their ports will be easy to connect.
- The capsule's state machine will be different as it does not need to start the game and it will not return the fifth ball in order to stop the game.

8.1 Create the "Ponger" capsule's structure

Create the **Ponger** capsule's structure using the same instructions as for the **Pinger** capsule structure, but with the following changes:

- The capsule is named "**Ponger**"
- The external port is named "pongPort"
- The external port is **not** conjugated.
- The external port is placed on the left edge of the capsule, instead of the right. This will make it easier for us to connect the capsules when we put both side by side.
- The log port is created in the same way

The result should be as shown below.

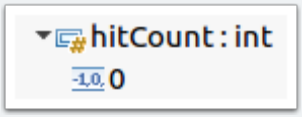


8.2 Add an attribute to Ponger

In order to limit the game (and not get a screenful of fast-streaming logs), we will add an attribute that will be used to limit the number of "pongs" that can be sent as a response to "pings", thereby allowing the game to end after a predetermined number of returns ("pongs").

1. Select the Ponger capsule in the Model Explorer.
2. Click on the UML tab in the Properties view
3. Click on the [+] to the right of "Owned attribute" and select "Property."
4. In the resulting dialog, name the property "hitCount" and make its visibility "protected;"
5. set its type by clicking on the [...] next to "Type", expanding the "AnsiClibrary" entry and select "int;"
6. and set its default value to a Literal Integer "0" (zero) by clicking on the [+] next to "Default Value," selecting "Literal Integer," and accepting the default value of "0" (zero).

You can now see this attribute in the model explorer:



8.3 Create the Ponger Capsule's statemachine

Create the **Ponger** capsule's statemachine using the same instructions as for the **Pinger** capsule structure, but with the following changes:

- The intial transition code will simply log that **Ponger** is ready to play:

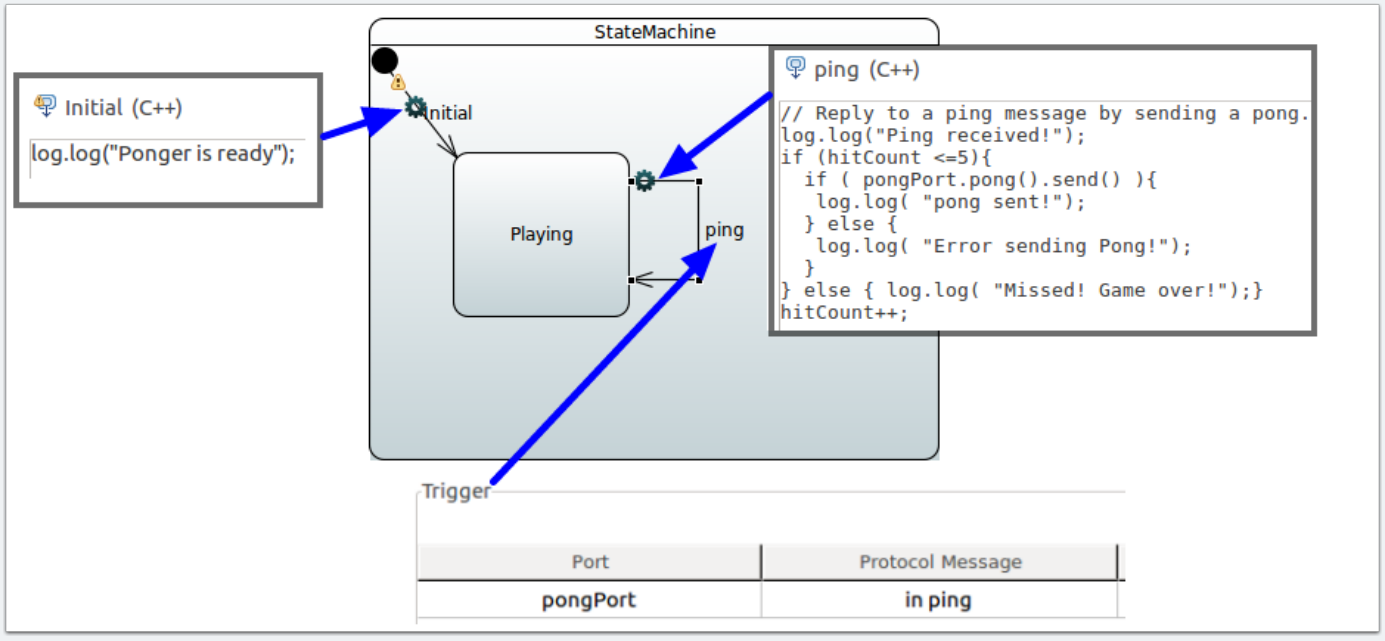
```
log.log("Ponger is ready");
```

- The self-transition's trigger will be on **PongPort**'s **ping** protocol message.
- The self-transition's code will simply send a **pong** for every **Ping** received, until it "misses."

Code:

```
// Reply to a ping message by sending a pong.
log.log("Ping received!");
if (hitCount <=5){
  if ( pongPort.pong().send() ){
        log.log( "pong sent!");
  } else {
        log.log( "Error sending Pong!");
  }
} else { log.log( "Missed! Game over!");}
hitCount++;
```

The result should be as shown below.



9. The "Top" system capsule

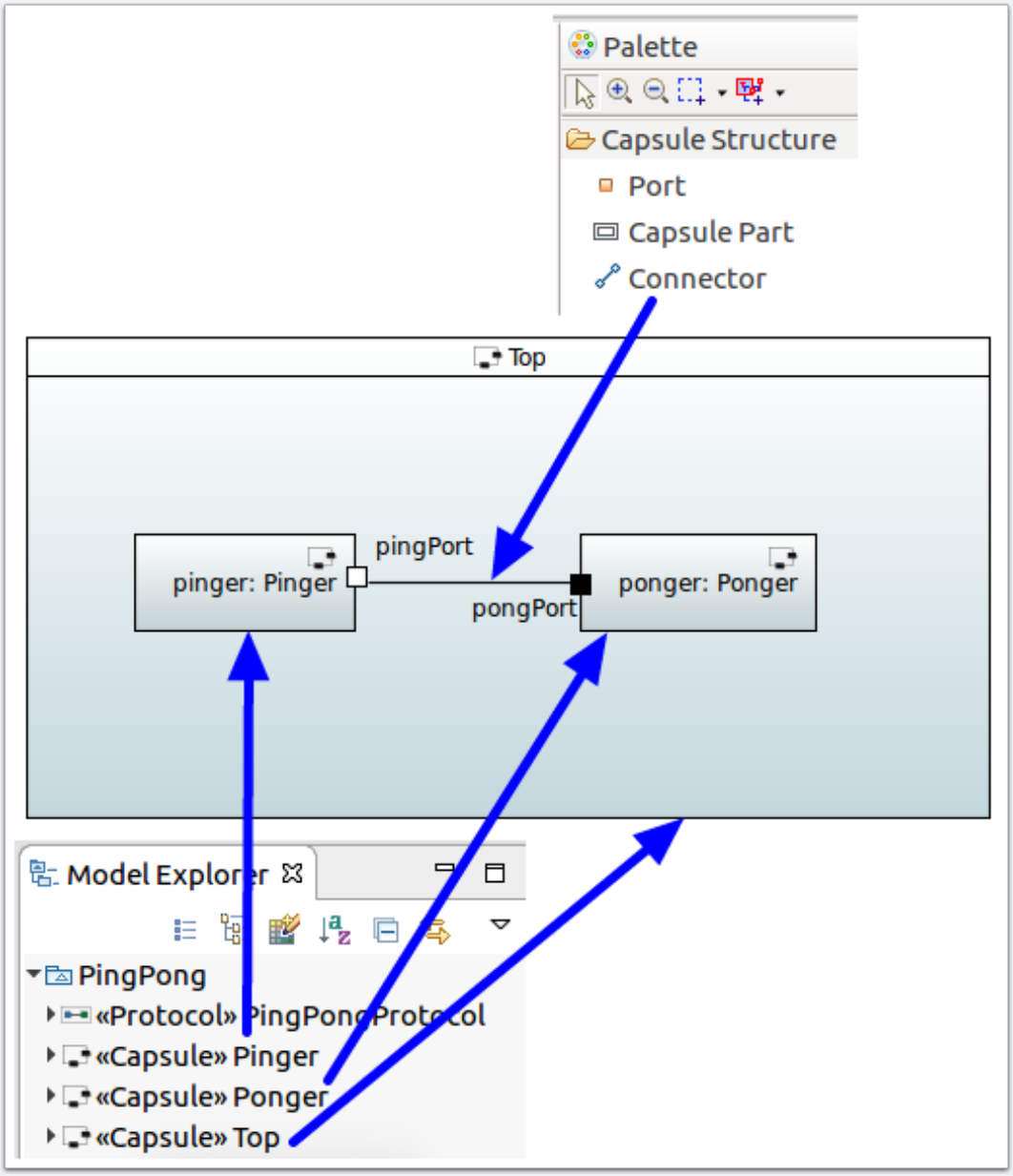
Although it is possible to generate the various capsules on their own, the interactions between them would not happen until their ports are connected.

To do this, we create a "Top" capsule that will contain instances of both the Pinger and Ponger capsules so that we can connect their ports. Once this is done, we can generate the code for that "Top" capsule and execute it. Generating the code for "Top" will automatically bring in all the other related model elements.

Note that, although it is not a requirement, the "**Top**" capsule we will create in here will be only structural, it will not, by itself, implement behaviour, other than that of its contained capsule parts.

9.1 Create the "Top" capsule

1. Create a new capsule in the model and name it "**Top**".
2. Open Top's capsule diagram.
3. From the model explorer, drag and drop a Pinger capsule into Top's compartment, on the left side.
4. From the model explorer, drag and drop a Ponger capsule into Top's compartment, on the right side. Aim to have their ports vertically aligned.
5. Use the Connector tool from the palette to draw a connector between each capsule part's ports.
6. That's it! You have created the Top capsule with two capsule parts that can nos communicate with each other!



10. Execute the model

Now that the model is complete, we can execute it.

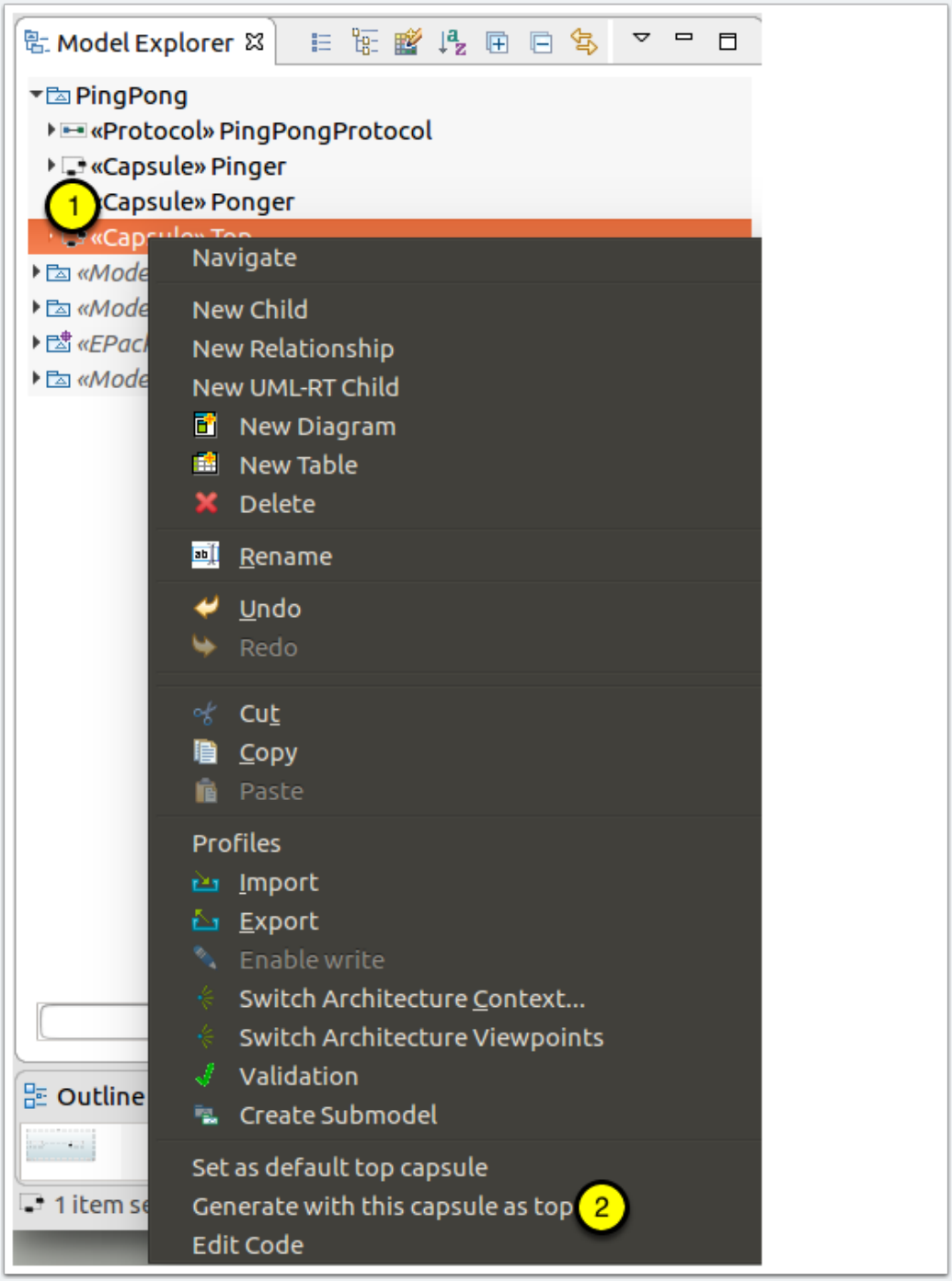
10.1 Top Capsule

In order to generate the code, we need to determine which capsule will be the"top" capsule, that is the capsule that will represent the system for the generated code. The code generator will recursively look at all the capsules that are used as part of this top capsule to generate the complete application. This is useful since each capsule can be executed on its own (e.g., for test purposes. This also allows for easy managements of "test harness" capsules for individual parts of the system. You can also set a default "Top" capsule that is reused by code generation commands.

In this tutorial, we will simply generate the code for the selected capsule.

10.2 Generate the model

1. Right-click on the "**Top**" capsule in the Model Explorer to bring up the context menu
2. Select "**Generate with this capsule as top.**"
3. A CDT project is created in the Project Explorer and the C++ code is generated within it.

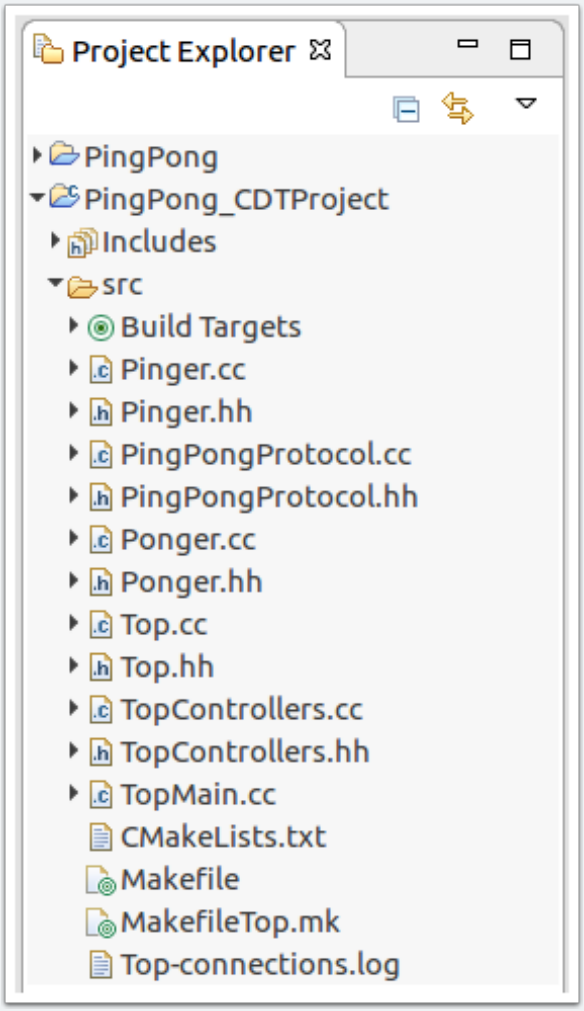

Model Explorer
PingPong
«Protocol» PingPongProtocol
«Capsule» Pinger
«Capsule» Ponger
1
Navigate
New Child
New Relationship
New UML-RT Child
New Diagram
New Table
Delete
Rename
Undo
Redo
Cut
Copy
Paste
Profiles
Import
Export
Enable write
Switch Architecture Context...
Switch Architecture Viewpoints
Validation
Create Submodel
Set as default top capsule
Generate with this capsule as top
2
Edit Code
Outline

10.3 Alternative

If you will be generating the code often for a particular capsule, you can also designate a capsule as being the top capsule. You would then be able to just re-generate more easily.

10.4 Generated model

When generating the model, a CDT project is created to hold the generated code.



10.5 Compile the model

To compile and run the model, you will need a compatible build environment. At present, we support Linux as the primary target platform with more limited support for Windows and MacOS.

Note | CDT Integration | The integration with CDT is not yet complete. To build the system, you will have to go to the command line or, if you are familiar with setting project within the CDT, you can try to configure the project yourself (*hint: the Papyrus-RT runtime library imports may be missing*).

Note | OS other than Linux | If you are using an operating system other than Linux, you can still compile and run your model. Go to Compiling_and_running_Papyrus_for_Real_Time_applications for alternatives

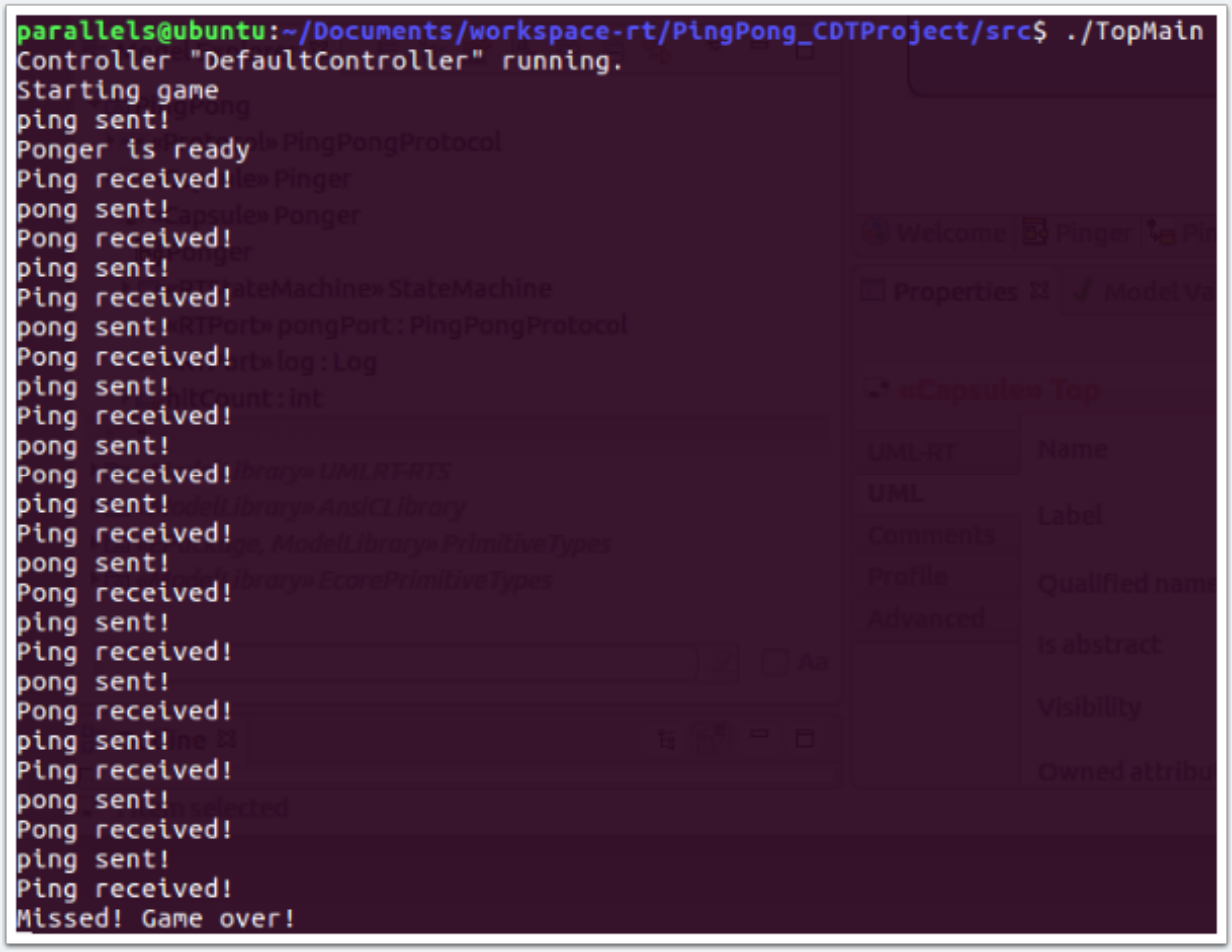
1. Open a terminal and go to the folder where the code was generated, in this case, the folder name would be ***<workspace>*/PingPong_CDTProject/src**, replacing "<*workspace*>" with the path to your workspace location, e.g., "~/workspaces/GettingStarted".
2. Type "make" at the command prompt to compile and link the model's generated code.

```
make[1]: Entering directory '/home/parallels/Documents/workspace-rt/PingPong_CDTProject/src'
MakefileTop.mk:3: warning: TARGETOS not defined. Choosing linux
MakefileTop.mk:9: warning: BUILDTOOLS not defined. Choosing x86-gcc-4.6.3
g++ TopMain.cc -c -Wall   -I/home/parallels/Apps/Papyrus-RT/plugins/org.eclipse.papyrusrt.rts_1.0.0.20
1707181457/umlrts/include -oTopMain.o
g++ PingPongProtocol.cc -c -Wall   -I/home/parallels/Apps/Papyrus-RT/plugins/org.eclipse.papyrusrt.rts
_1.0.0.201707181457/umlrts/include -oPingPongProtocol.o
g++ Pinger.cc -c -Wall   -I/home/parallels/Apps/Papyrus-RT/plugins/org.eclipse.papyrusrt.rts_1.0.0.201
707181457/umlrts/include -oPinger.o
g++ Ponger.cc -c -Wall   -I/home/parallels/Apps/Papyrus-RT/plugins/org.eclipse.papyrusrt.rts_1.0.0.201
707181457/umlrts/include -oPonger.o
g++ Top.cc -c -Wall   -I/home/parallels/Apps/Papyrus-RT/plugins/org.eclipse.papyrusrt.rts_1.0.0.201707
181457/umlrts/include -oTop.o
g++ TopControllers.cc -c -Wall   -I/home/parallels/Apps/Papyrus-RT/plugins/org.eclipse.papyrusrt.rts_1
.0.0.201707181457/umlrts/include -oTopControllers.o
g++  TopMain.o PingPongProtocol.o Pinger.o Ponger.o Top.o TopControllers.o   -L/home/parallels/Apps/Pa
pyrus-RT/plugins/org.eclipse.papyrusrt.rts_1.0.0.201707181457/umlrts/lib/linux.x86-gcc-4.6.3  -lrts  -
lpthread  -lrt -oTopMain
make[1]: Leaving directory '/home/parallels/Documents/workspace-rt/PingPong_CDTProject/src'
parallels@ubuntu:~/Documents/workspace-rt/PingPong_CDTProject/src$
```

10.6 Run the model's executable

You can then run the executable to see the log showing the (short) game!

1. At the command prompt, type "**./TopMain**"
2. Observe the results:



11. Congratulations!