Multi-agent Environment for Complex SYstems COsimulation (MECSYCO) - User Guide: MECSYCO-visu

Benjamin Camus[1,2], Julien Vaubourg[2], Yannick Presse[2],
Victorien Elvinger[2], Thomas Paris[1,2], Alexandre Tan[2]
Vincent Chevrier[1,2], Laurent Ciarletta[1,2], Christine Bourjot[1,2]
[1]Universite de Lorraine, CNRS, LORIA UMR 7503,
Vandoeuvre-les-Nancy, F-54506, France.
[2]INRIA, Villers-les-Nancy, F-54600, France.
`mecsyco@inria.fr`

June 21, 2016

Contents

Introduction **2**

1 Observing agent **3**

2 Observing artifact **4**
- 2.1 Creation of the dispatcher 4
- 2.2 Connection to the agent 4
- 2.3 Adding an observing artifact 5

3 Requirements **6**
- 3.1 Connection 6
- 3.2 Starting 6

4 Graphics' configuration **7**
- 4.1 Temporal graph 7
- 4.2 XY graphics 8
- 4.3 Bar chart 9
- 4.4 Pie chart 10
- 4.5 Factual representation 11
- 4.6 3D graphic 12

5 Script R **14**
- 5.1 The observing artifact for R 14

6 Data comparator **16**

7 LoggingArtifact **17**
- 7.1 Format 17
- 7.2 Listing of the different constructors 17
  - 7.2.1 One file per output port 17
  - 7.2.2 One file for all output ports 18
    - Line structure 18
    - Column structure 18
- 7.3 Examples 18
  - 7.3.1 One file per output port 18
  - 7.3.2 One file for all output ports, line structure 19
  - 7.3.3 One file for all output ports, column structure 20
    - Checking port mode 20
    - Treatment 21

8 Template **22**
- Java Example Template: run configuration (with observing tools) 22

Introduction

MECSYCO-visu is a set of agents and artifacts dedicated to the display of the results from simulations on different types of graph. It also contains artifacts used for post-treatment. The observer agent has the same behavior than any other MECSYCO's agent, which makes its use and parametrization easier.

MECSYCO-visu is built upon observing agents (agents that manage a visualization tool), an observing dispatcher (artifact that will connect the agent to the observing artifacts) and observing artifacts to connect agents and visualization tools.

All primitives and classes needed for visualization are in *mecsyco-visu-2.0.0.jar*.

All examples (codes and figures) are taken from the case Lorenz created in the *Getting Started* section.

Chapter 1

Observing agent

In order to connect MECSYCO's visualization system, you need to create the observing agent first (*ObservingMAgent*). It takes two arguments to do that:

- **aAgentName:** this defines the agent's name used for information purpose (when displayed in the console, cf Figure 6.1).
- **aMaxSimTime:** the maximum time of the multi-simulation.
- **aInterfaceArtifact:** Defines the*ModelArtifact* to associate with the agent.

ObservingMAgent constructor in Java implementation
public ObservingMAgent (String aAgentName, double aMaxSimTime, ObservingArtifact aInterfaceArtifact) public ObservingMAgent (String aAgentName, double aMaxSimTime)

ObservingMAgent constructor in C++ implementation
Not implemented yet

Example:
ObservingMAgent obsAgent = new ObservingMAgent ("agent_obs", maximulationTime);

Chapter 2

Observing artifact

We explain here the functions used for linking the observing agent to the different observing artifacts.

2.1 Creation of the dispatcher

You have to create an artifact needed to link the observing agent to the different observing artifacts. The dispatcher or *SwingDispatcherArtifact* does not need any argument:

SwingDispatcherArtifact constructor in Java implementation
public SwingDispatcherArtifact ()

SwingDispatcherArtifact constructor in C++ implementation
Not implemented yet

Example:
SwingDispatcherArtifact ObsModelArtifact = new SwingDispatcherArtifact ();

Note:

- The dispatcher is needed because usually we cannot link more than one artifact to the same agent, but with the dispatcher, we will be able to have multiple observing artifacts for the same agent
- As said, we cannot link multiple artifacts to one agent, so we cannot link multiple dispatchers either

2.2 Connection to the agent

If you did not use the first constructor of the *ObservingMAgent*, you need to associate the artifact manually using the following method:

setDispatcherArtifact
Parameters:

- **artifact** - The dispatcher to associate to the agent

setDispatcherArtifact method in Java implementation
public void setDispatcherArtifact(ObservingArtifact artifact)

setDispatcherArtifact method in C++ implementation
Not implemented yet

Example:
obsAgent.setObservingArtifact (ObsModelArtefact);

2.3 Adding an observing artifact

You can now add the observing artifacts. The observing artifacts will link the agent to the visualization tool asked, or the post-treatment to use. To add them, you need to use the following method:

addObservingArtifact
Parameters:

- **aConsumer** - The observing artifact to add to the visualization tools' list

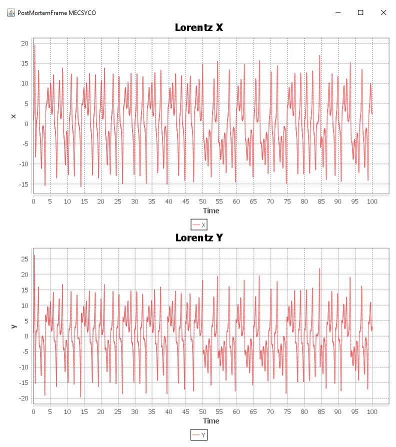
addObservingArtifact method in Java implementation
public void addObservingArtifact (ObservingArtifact aConsumer)

addObservingArtifact method in C++ implementation
Not implemented yet

Example:
ObsModelArtifact.addObservingArtifact (
new ObservingArtifactName (ObservingArtifact's arguments));

Note:

- You can add as many observing artifact as you want. If they have the same type (live or post-mortem), the graphs will even be displayed on the same windows (Figure 2.1a)
- If you want to be sure that all graphs are displayed on different windows, you can create another observing agent and its proper dispatcher (Figure 2.1b)



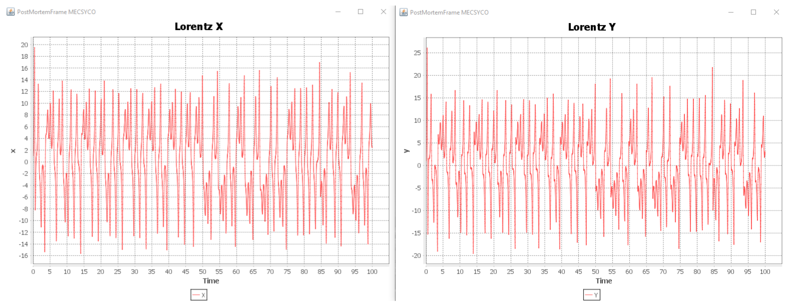


Figure 2.1: (a) Two PostMortem graphs link to the same observing agent. (b) Each PostMortem graph links to its proper observing agent

Chapter 3

Requirements

As explained, the observing agent has the same behavior than any other agent. (cf: *User Guide section "The m-agent"*)

3.1 Connection

The last step of the observer creation is to define what to observe, that is to say which ports. To do so, we have to create the link:

```
CentralizedEventCouplingArtifact Coupl1 = new CentralizedEventCouplingArtifact ();
CentralizedEventCouplingArtifact Coupl2 = new CentralizedEventCouplingArtifact ();
CentralizedEventCouplingArtifact Coupl3 = new CentralizedEventCouplingArtifact ();
...
Agent1.addOutputCouplingArtifact (Coupl1, "name of port1");
Agent2.addOutputCouplingArtifact (Coupl2, "name of port2");
Agent3.addOutputCouplingArtifact (Coupl3, "name of port3");
...
obsAgent.addInputCouplingArtifact (Coupl1, "name of port1");
obsAgent.addInputCouplingArtifact (Coupl2, "name of port2");
obsAgent.addInputCouplingArtifact (Coupl3, "name of port3");
...
```

3.2 Starting

Before launching, as the other agent, the observer need to be properly started:

```
obsAgent.startModelSoftware ();
obsAgent.start ();
```

Chapter 4

Graphics' configuration

The observing artifact proposes 6 types of graphs (all in live or post mortem): classic temporal graph (TX graph, cf subsection 4.1), phase diagram (XY graph, cf section 4.2), pie chart (cf section 4.4), bar chart (cf section 4.3), factual representation (cf section 4.5), and a 3D graph (cf section 4.6).

The observer also proposes a saving results system the *LoggingArtifact* (cf chapter 7) and two post-treatment functions. You can launch an R script (cf chapter 5), or compare the data from the simulation to data from an outside a file (*.csv or *.Json, cf chapter 6).

4.1 Temporal graph

The observing artifact for temporal graphs is *TXGraphic* and its constructor needs 5 arguments:

- **title:** title of the graph.
- **y:** name of the y axis.
- **r:** renderer used (Figure 4.1).
- **aSeries:** list filled with the names of the series.
- **aPorts:** list filled with the names of the ports observed.

TXGraphic constructor in Java implementation
public LiveTXGraphic (String title, String y, Renderer r, String[] aSeries, String[] aPorts) public PostMortemTXGraphic (String title, String y, Renderer r, String[] aSeries, String[] aPorts)

TXGraphic constructor in C++ implementation
Not implemented yet

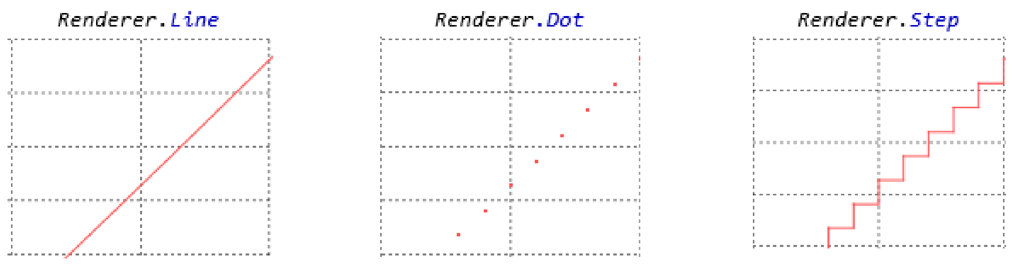


Figure 4.1: List of renderer possible.

Example:
In order to obtain Figure 4.2, we filled the arguments like this:

```
ObsModelArtifact.addObservingArtifact (
newLiveTXGraphic (”Lorenz temporal”, ”Value”, Renderer.Line,
new String [] {”SeriesX”, ”SeriesY”, ”SeriesZ”} ,
new String [] {”X”, ”Y”, ”Z”} ));
or
ObsModelArtifact.addObservingArtifact (
new PostMortemTXGraphic (”Lorenz temporal”, ”Value”, Renderer.Line,
new String [] {”SeriesX”, ”SeriesY”, ”SeriesZ”} ,
new String [] {”X”, ”Y”, ”Z”} ));
```

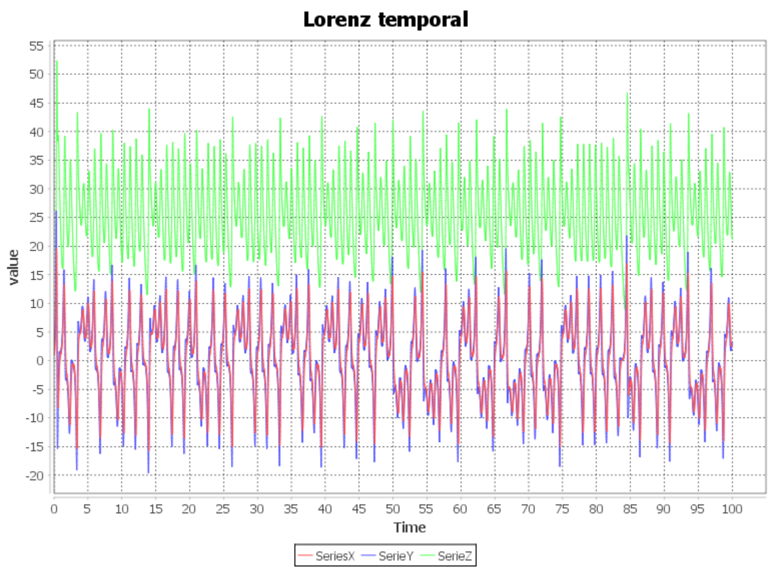


Figure 4.2: Lorenz TXGraphic for ports ”X”, ”Y” and ”Z”.

Note:

- It is possible to add as many series as desired, but too many can cause the simulation to slow down a bit.
- The order of the list of the name of series is IMPORTANT! It has to be in the same order than the list of ports.
- *TXSeries* only manage real values (Double).

4.2 XY graphics

The observing artifact is *XYGraphic* and its constructor needs 6 arguments:

- **aTitle:** title of the graph.
- **aXaxis:** name of the x axis.
- **aYaxis:** name of the y axis.
- **aRenderer:** renderer used (Figure 4.1).
- **aSeries:** name of the series.
- **aPorts:** name of the port observed.

XYGraphic constructor in Java implementation
public LiveXYGraphic (String aTitle, String aXaxis, String aYaxis, Renderer aRenderer, String aSeries, String aPort)
public PostMortemXYGraphic (String aTitle, String aXaxis, String aYaxis, Renderer aRenderer, String aSeries, String aPort)

XYGraphic constructor in C++ implementation
Not implemented yet

Example:
In order to obtain Figure 4.3, we filled the arguments like this:

ObsModelArtifact.addObservingArtifact (
new LiveXYGraphic(”Lorenz phase”, ”X”, ”Y”, Renderer.Line, ”XY”, ”obs”));
or
ObsModelArtifact.addObservingArtifact (
new PostMortemXYGraphic(”Lorenz phase”, ”X”, ” Y ”, Renderer.Line, ”XY”, ”obs”));

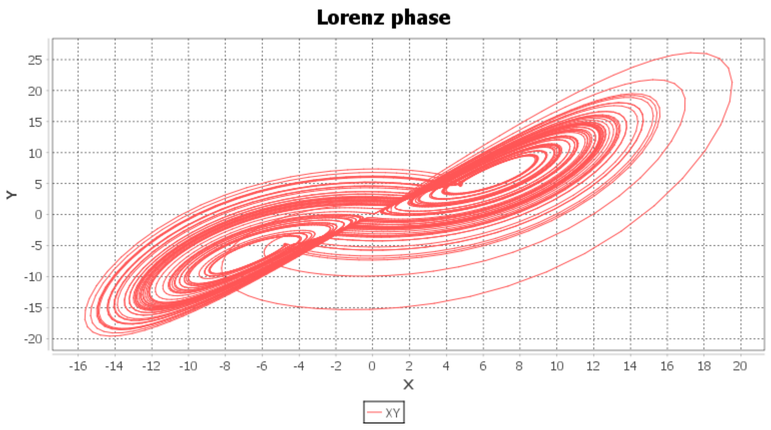


Figure 4.3: Lorenz XY Graphic for port ”obs2D” that combines X and Y

Note:

- *XYGraphic* only manage a *Tuple2* (cf *User Guide* section *Simulation data*) of two real. The first real is for the x axis, and the second for y axis

4.3 Bar chart

The observing artifact is *BarGraphic* and its constructor needs 5 arguments:

- **title:** title of the graph.
- **x:** name of the x axis.
- **y:** name of the y axis.
- **aSeries:** list filled with the names of the bars.
- **aPort:** name of the port observed.

BarGraphic constructor in Java implementation
public LiveBarGraphic (String title, String x, String y, String[] aSeries, String aPort)
public PostMortemBarGraphic (String title, String x, String y, String[] aSeries, String aPort)

BarGraphic constructor in C++ implementation
Not implemented yet

Example:
In order to obtain Figure 4.4, we filled the arguments like this:

ObsModelArtifact.addObservingArtifact (
new LiveBarGraphic(”Lorenz bar”, ”Bars”, ”Value”, new String [] {”X”, ”Y”, ”Z”}, ”XYZ”));
or
ObsModelArtifact.addObservingArtifact (
new PostMortemBarGraphic(”Lorenz bar”, ”Bars”, ”Value”, new String [] {”X”, ”Y”, ”Z”}, ”XYZ”));

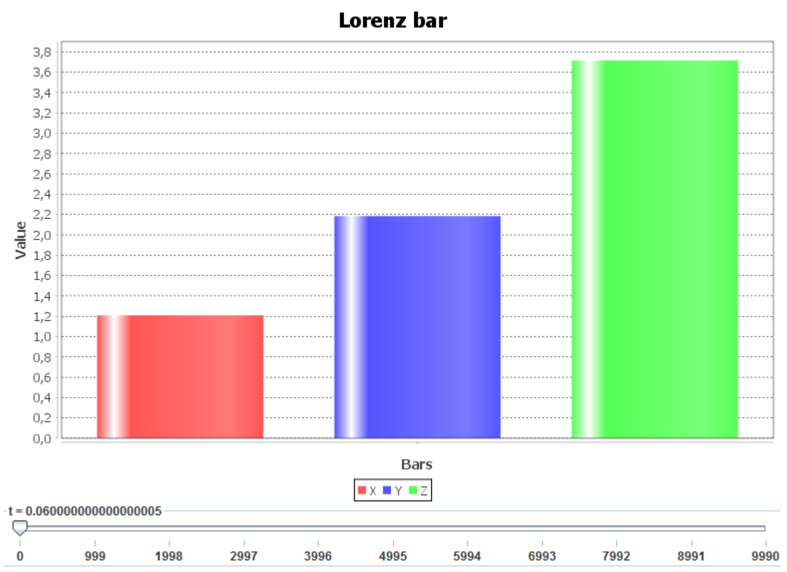


Figure 4.4: Lorenz bar chart for port ”obs3d” that combines X, Y and Z

Note:

- *BarGraphic* only manages a *SimulVector* (cf *User Guide* section *Simulation data*) of real.
- The slide bar is used to indicate which time of the simulation you want visualize.
- Be sure that the names of the bars are in the same order than the vector.

4.4 Pie chart

The observing artifact is *PieGraphic* and its constructor needs 3 arguments:

- **title:** title of the graph.
- **aSeries:** list filled with the names of the sections of the pie chart.
- **aPort:** name of the port observed.

PieGraphic constructor in Java implementation
public LivePieGraphic (String title, String[] aSeries, String aPort)
public PostMortemPieGraphic (String title, String[] aSeries, String aPort)

PieGraphic constructor in C++ implementation
Not implemented yet

Example:
In order to obtain Figure 4.5, we filled the arguments like this:

ObsModelArtifact.addObservingArtifact (
new LivePieGraphic("Lorenz Pie", new String [] {"X", "Y", "Z"}, "XYZ"));
or
ObsModelArtifact.addObservingArtifact (
new PostMortemPieGraphic(""Lorenz Pie", new String [] {"X", "Y", "Z"}, "XYZ"));

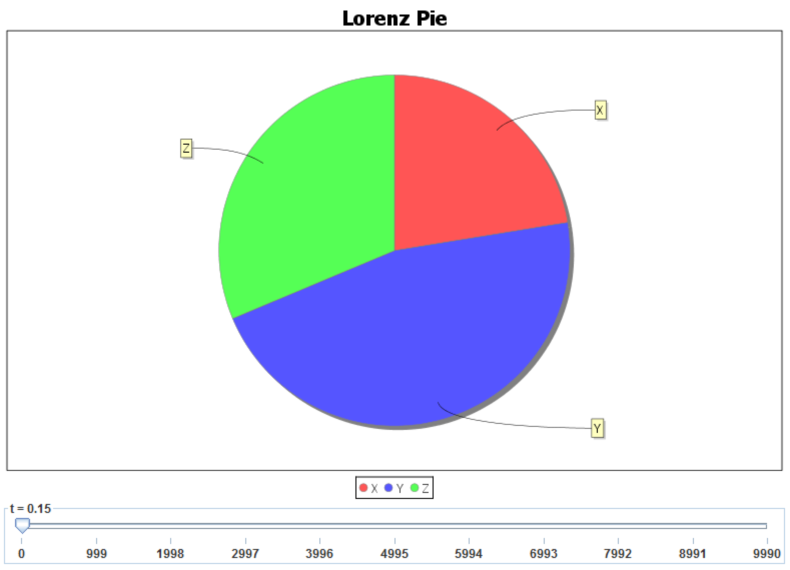


Figure 4.5: Lorenz pie chart for port obs3d that combines X, Y and Z

Note:

- *PieGraphic* only manage a *SimulVector* (cf *User Guide* section *Simulation data*) of real.
- The slide bar is used to indicate which time of the simulation you want visualize.
- Be sure that the names of the pie's section are in the same order than the vector.

4.5 Factual representation

The observing artifact is *EventGraphic* and its constructor needs 5 arguments:

- **aTitle:** title of the graph.
- **aXaxis:** name of the x axis.
- **aYaxis:** name of the y axis.
- **aSeries:** name of the series.
- **aPort:** name of the port observed.

EventGraphic constructor in Java implementation
public LiveEventGraphic (String aTitle, String aXaxis, String aYaxis, String aSeries, String aPort) public PostMortemEventGraphic (String aTitle, String aXaxis, String aYaxis, String aSeries, String aPort)

EventGraphic constructor in C++ implementation
Not implemented yet

Example:
In order to obtain Figure 4.6, we filled the arguments like this:

ObsModelArtifact.addObservingArtifact (
new LiveEventGraphic(”Lorenz factual”, ”time”, ”X”, ”X”, ”X”));
or
ObsModelArtifact.addObservingArtifact (
new PostMortemEventGraphic(”Lorenz factual”, ”time”, ”X”, ”X”, ”X”));

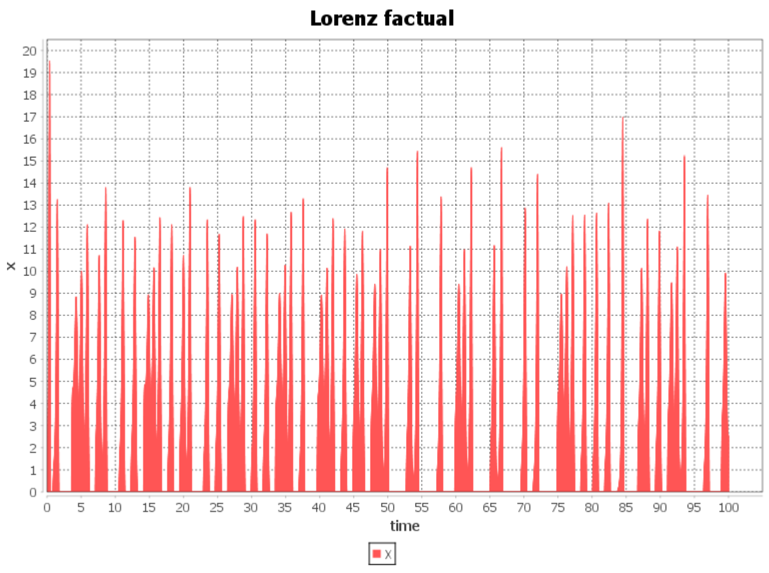


Figure 4.6: Lorenz factual chart for port ”X”

Note:

- *EventGraphic* only manage real.
- This graphic shows a peak for each value change of the port observed. The height of the peak correspond to the new value of the port observed.

4.6 3D graphic

The observing artifact is *3DGraphic* and its constructor needs 5 arguments:

- **aTitle:** title of the graph.
- **xaxis:** name of the x axis.
- **yaxis:** name of the y axis.
- **zaxis:** name of the y axis.
- **aPort:** name of the port observed.

3DGraphic constructor in Java implementation
public Live3DGraphic (String aTitle, String xaxis, String yaxis, String zaxis, String port) public PostMortem3DGraphic (String aTitle, String xaxis, String yaxis, String zaxis, String port)

3DGraphic constructor in C++ implementation
Not implemented yet

Example:
In order to obtain Figure 4.7, we filled the arguments like this:

ObsModelArtifact.addObservingArtifact (
new Live3DGraphic("Lorenz 3D", "X", "Y", "Z", "X,Y,Z"));
or
ObsModelArtifact.addObservingArtifact (
new PostMortem3DGraphic("Lorenz 3D", "X", "Y", "Z", "X,Y,Z"));

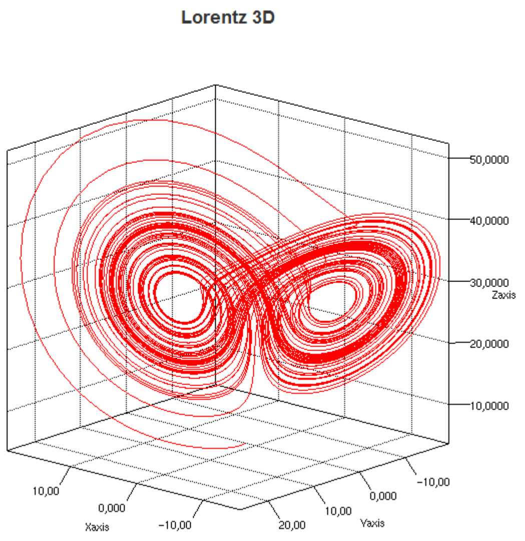


Figure 4.7: Lorenz 3D chart for port "X,Y,Z" that combines X, Y and Z in a different way than" obs3d"

Note:

- *3DGraphic* only manage a *Tuple3* (cf *User Guide* section *Simulation data*) of real.
- First value is for the x axis, second for the y axis and third for z axis.

Chapter 5

Script R

It is also possible to use the observer in order to launch an R script.

5.1 The observing artifact for R

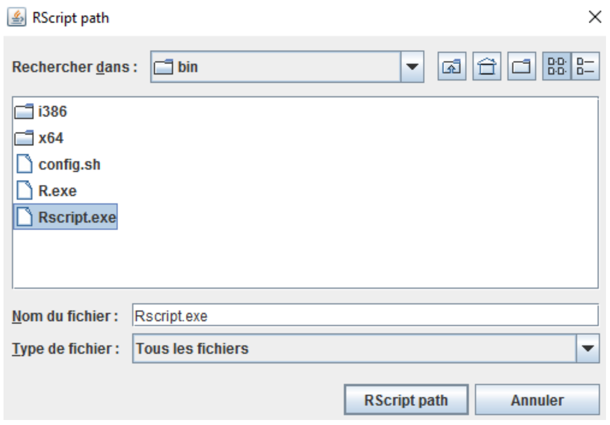
In order to launch the R script after the simulation, you need to add a particular observer to the dispatcher, the *LogToRProject* that needs 2 arguments:

- **aFilePath:** Absolute path to the file where the data will be saved. If the file does not exist, it will be created.
- **aPorts:** list of the names of the ports observed.

LogToRProject constructor in Java implementation
public LogToRProject (String aFilePath, String[] aPorts)

LogToRProject constructor in C++ implementation
Not implemented yet

The file created can be used as input for the Rscript, but you can also ignore it. In the example provided in the *Getting Started* section, the R script does not need input, it will search for a specific file that the user need to point in the R script code directly.
If your R script want to use the file created, you need to know that it was created following the LoggingArtifact creating one file for multiple port in line structure (see section 7.2.2).
At the beginning of the simulation, two windows will then pop-up (Figures 5.1 and 5.2) asking the file path to *Rscript.exe* (user\Program Files\R-3.2.2\bin\i386, for Windows 32 bits, or user\Program Files\R-3.2.2\bin\x64 for Windows 64 bits) then the script to compile.

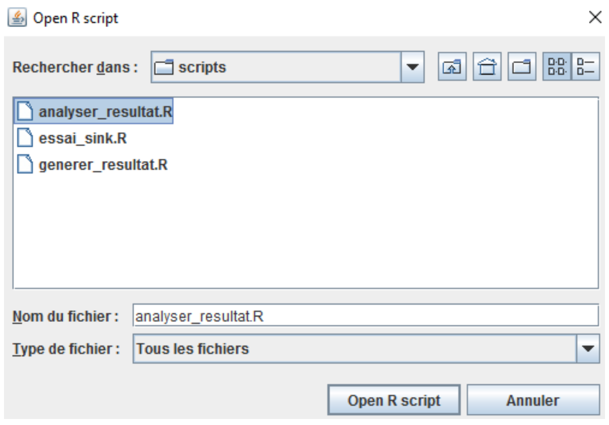

RScript path

Rechercher dans : bin

i386
x64
config.sh
R.exe
Rscript.exe

Nom du fichier : Rscript.exe
Type de fichier : Tous les fichiers

Figure 5.1: Rscript.exe selection


Open R script

Rechercher dans : scripts

analyser_resultat.R
essai_sink.R
generer_resultat.R

Nom du fichier : analyser_resultat.R
Type de fichier : Tous les fichiers

Figure 5.2: R script file to execute

The next version of this artifact will enable to provide the name of the scrit as a parameter instead of prompting it.
Note that the save file asking will have the following format: ”port; time; data”.

Chapter 6

Data comparator

Another function available is the data comparator. This comparator is used to compare the result of the simulation with data taken from a *.csv with the format: "time; value" (without headers). In order to use it, you need to activate the debug system in "info" mode (cf **User Guide: Debug system**).
The observing artifact is the *DataComparator* that needs 2 arguments:

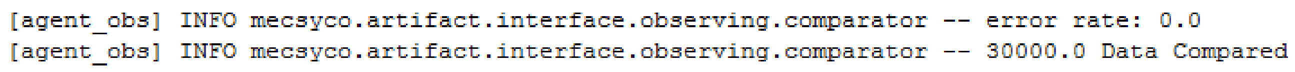
- **fileName:** list filled with the absolute path to the files where the data use as reference are located
- **aPorts:** list filled with the names of the ports observed.

DataComparator constructor in Java implementation
public DataComparator (String[] fileName, String[] aPorts)

DataComparator constructor in C++ implementation
Not implemented yet

Example:
In order to obtain Figure 6.1 in the console, we filled the arguments like this:

ObsModelArtifact.addObservingArtifact (
new DataComparator(
new String[] { "resources/model_checking/lorenzLog_x.csv", "resources/model_checking/lorenzLog_y.csv", "resources/model_checking/lorenzLog_z.csv" },
new String[] { "X", "Y", "Z" }));



```
[agent_obs] INFO mecsyco.artifact.interface.observing.comparator -- error rate: 0.0
[agent_obs] INFO mecsyco.artifact.interface.observing.comparator -- 30000.0 Data Compared
```

Figure 6.1: Lorenz factual chart for port obs3d that combines X, Y and Z

Note:

- *DataComparator* only manage real.
- The order is important! The first file will be used for the first port

Chapter 7

LoggingArtifact

The *LoggingArtifact* saves results of a simulation in one or multiple files. We will first explain the use of this particular artifact, then show examples from Lorenz tutorial.

7.1 Format

The format is a String that defines how to write each line of the results' files. To do that, you need to precise what is the content of each column using *"%port"* for the port's name, *"%time"* for the current simulation time or *"%value"* for the values. You also need to show the separator of the columns by using it.
For example, *"%time;%value"* indicates that the first column will be for the time, the separator will be ";" and the last column the value.
There are two default values for the format:

- **LightDefaultFormat:** "%time;%value".
- **VerboseDefaultFormat:** "%port;%time;%value".

In order to show that it is really a line that you defined, it is really IMPORTANT to add *"\n"* at the end of the format.
It is also possible to choose the decimal separator (by default it is ".") using:
LoggingArtifact.DecimalSeparator=" new_decimal_separator"

7.2 Listing of the different constructors

The *LoggingArtifact* is a particular observing artifact because it has 3 different constructors and each one has its proper impact on the results' file.

7.2.1 One file per output port

This first constructor will create a saving file for each port observed. It needs 3 arguments:

- **aPaths:** list filled with the absolute path to the saving files. If they do not exist, they will be created.
- **aPorts:** list filled with the names of the ports observed.
- **aFormat:** format used. By default, it is *LightDefaultFormat* (cf section 7.1).

LoggingArtifact constructor in Java implementation (one file per output port)

public LoggingArtifact (String [] aPaths, String [] aPorts, String aFormat);
public LoggingArtifact (String [] aPaths, String [] aPorts);

LoggingArtifact constructor in C++ implementation (one file per output port)
Not implemented yet

7.2.2 One file for all output ports

The two following constructors will create a single saving file for all ports observed. The main difference between these two is the structure of the file of results.

Line structure

The line structure means that in the saving file, each line will correspond to an event. The constructor uses 3 arguments:

- **aPath:** the absolute path to the saving file. If it does not exist, it will be created.
- **aPorts:** list filled with the names of the ports observed.
- **aFormat:** format used. By default, it is *VerboseDefaultFormat* (cf section 7.1).

LoggingArtifact constructor in Java implementation (one file for all output ports, line)

public LoggingArtifact (String aPath, String [] aPorts, String aFormat);
public LoggingArtifact (String aPath, String [] aPorts);

LoggingArtifact constructor in C++ implementation (one file for all output ports, line)
Not implemented yet

Column structure

The column structure means that in the saving file, each line will correspond to a simulation time and each column to a port. The constructor uses 5 arguments:

- **aPath:** the absolute path to the saving file. If it does not exist, it will be created.
- **aPorts:** list filled with the names of the ports observed.
- **aFormat:** format used (cf section 7.1).
- **headers_names:** list filled with the headers to use. This list has to be in the same order than the list of ports observed.
- **checkPort: true** for checking port mode, **false** for treatment (cf section 7.3.3).

LoggingArtifact constructor in Java implementation (one file for all output ports, column)
LoggingArtifact (String aPath, String [] aPorts, String aFormat, String [] headers_names, Boolean checkPort);

LoggingArtifact constructor in C++ implementation (one file for all output ports, column)
Not implemented yet

7.3 Examples

In order to make the path argument easier, we defined beforehand the absolute path so we just need to add the name of the files: *String path="data_log/";*

7.3.1 One file per output port

```
ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
new String [] {path+"X.csv",path+"Y.csv",path+"Z.csv", path+"XYZ.csv"},
new String [] {"X", "Y", "Z", "X,Y,Z"},
"%time;%value\n"));
```

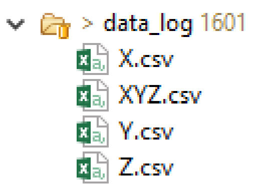
Currently not working well with SimulData different than *Tuple1* (cf *User Guide* section *Simulation data*). The best way to have one file per port in the other case, is to use the one file for all output port multiple times:

ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
path+"X.csv", new String [] {"X"}, "%time;%value\n"));

ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
path+"Y.csv", new String [] {"Y"}, "%time;%value\n"));

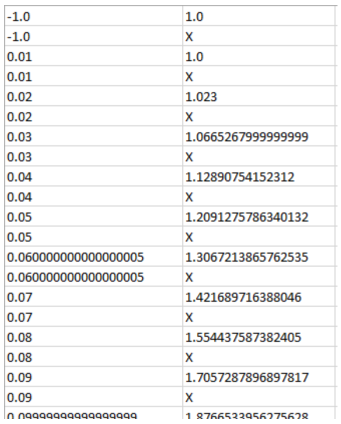
ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
path+"Z.csv", new String [] {"Z"}, "%time;%value\n"));

ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
path+"XYZ.csv", new String [] {"X,Y,Z"}, "%time;%value\n"));



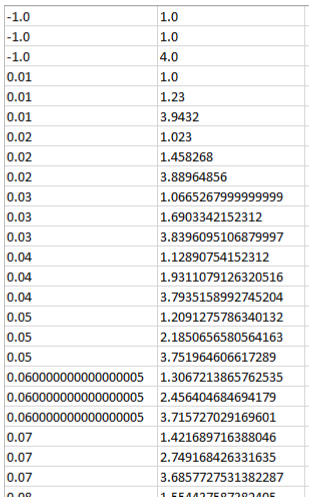
```
data_log 1601
    X.csv
    XYZ.csv
    Y.csv
    Z.csv
```

Figure 7.1: Files created



-1.0	1.0
-1.0	X
0.01	1.0
0.01	X
0.02	1.023
0.02	X
0.03	1.0665267999999999
0.03	X
0.04	1.12890754152312
0.04	X
0.05	1.2091275786340132
0.05	X
0.060000000000000005	1.3067213865762535
0.060000000000000005	X
0.07	1.421689716388046
0.07	X
0.08	1.554437587382405
0.08	X
0.09	1.7057287896897817
0.09	X
0.0999999999999999	1.8766533956275628

Figure 7.2: Result for port X

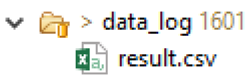


-1.0	1.0
-1.0	1.0
-1.0	4.0
0.01	1.0
0.01	1.23
0.01	3.9432
0.02	1.023
0.02	1.458268
0.02	3.88964856
0.03	1.0665267999999999
0.03	1.6903342152312
0.03	3.8396095106879997
0.04	1.12890754152312
0.04	1.9311079126320516
0.04	3.7935158992745204
0.05	1.2091275786340132
0.05	2.1850656580564163
0.05	3.751964606617289
0.060000000000000005	1.3067213865762535
0.060000000000000005	2.456404684694179
0.060000000000000005	3.715727029169601
0.07	1.421689716388046
0.07	2.749168426331635
0.07	3.6857727531382287
0.08	1.554437587382405

Figure 7.3: Result for port XYZ

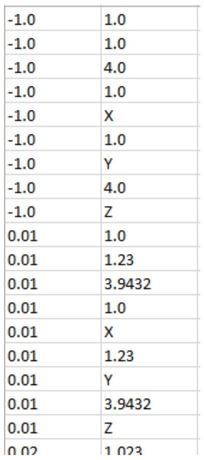
7.3.2 One file for all output ports, line structure

ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
path+"result.csv", new String [] {"X", "Y", "Z", "X,Y,Z"}, "%time;%value\n"));



```
data_log 1601
    result.csv
```

Figure 7.4: File created



-1.0	1.0
-1.0	1.0
-1.0	4.0
-1.0	1.0
-1.0	X
-1.0	1.0
-1.0	Y
-1.0	4.0
-1.0	Z
0.01	1.0
0.01	1.23
0.01	3.9432
0.01	1.0
0.01	X
0.01	1.23
0.01	Y
0.01	3.9432
0.01	Z
0.02	1.023

Figure 7.5: Result for all port (X, Y, Z, XYZ)

The 3 first lines of figure 7.5 correspond to the XYZ port, then we have X then Y then Z port (it follows the order of the simulator).

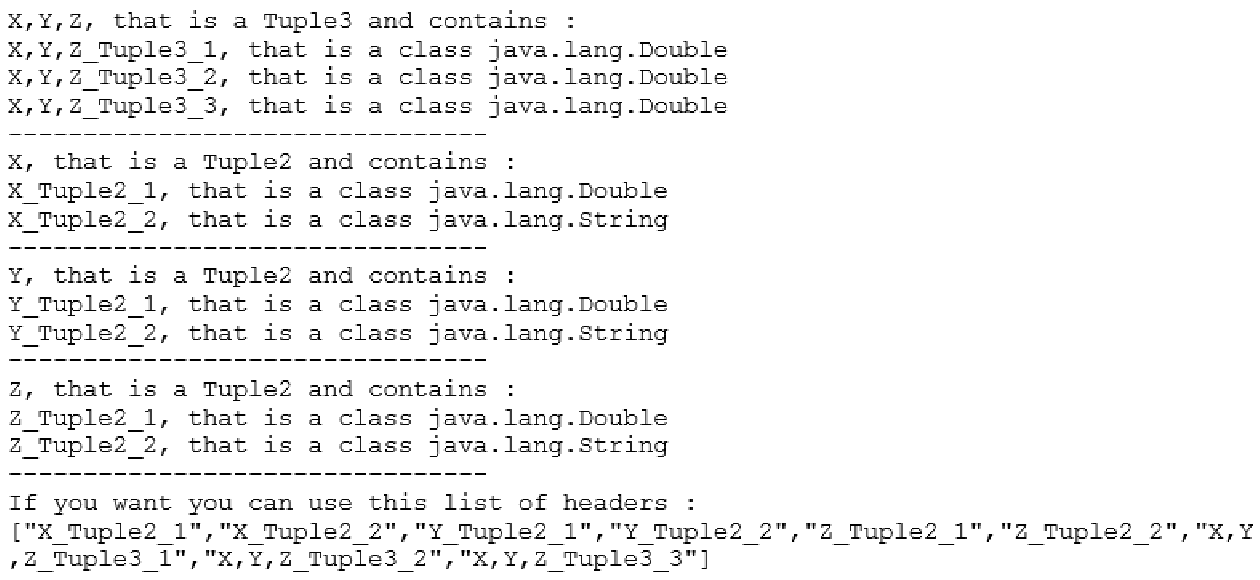
7.3.3 One file for all output ports, column structure

This constructor is particular because it has two different functions depending on the argument *checkPort*.

Checking port mode

It allows you to verify the order of the event, the SimulData's categories of the outputs and it will also give a default header list. As a consequence, the argument *headers_names* does not need to be properly filled.

ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
path+"result.csv", new String [] {"X", "Y", "Z", "X,Y,Z"}, "%time;%value\n",
new String [] {"obsolete argument"}, true));



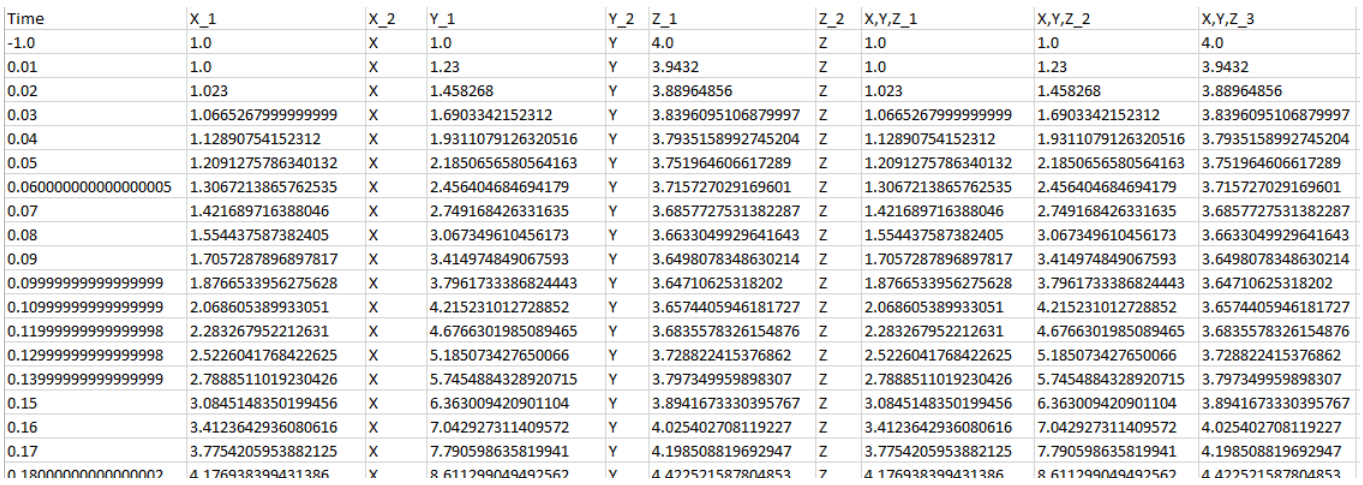
```
X,Y,Z, that is a Tuple3 and contains :
X,Y,Z_Tuple3_1, that is a class java.lang.Double
X,Y,Z_Tuple3_2, that is a class java.lang.Double
X,Y,Z_Tuple3_3, that is a class java.lang.Double
--------------------------------
X, that is a Tuple2 and contains :
X_Tuple2_1, that is a class java.lang.Double
X_Tuple2_2, that is a class java.lang.String
--------------------------------
Y, that is a Tuple2 and contains :
Y_Tuple2_1, that is a class java.lang.Double
Y_Tuple2_2, that is a class java.lang.String
--------------------------------
Z, that is a Tuple2 and contains :
Z_Tuple2_1, that is a class java.lang.Double
Z_Tuple2_2, that is a class java.lang.String
--------------------------------
If you want you can use this list of headers :
["X_Tuple2_1","X_Tuple2_2","Y_Tuple2_1","Y_Tuple2_2","Z_Tuple2_1","Z_Tuple2_2","X,Y
,Z_Tuple3_1","X,Y,Z_Tuple3_2","X,Y,Z_Tuple3_3"]
```

Figure 7.6: Console result for checking port mode

Treatment

This mode is the main function, which means saving result.

ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
path+"result.csv", new String [] {"X", "Y", "Z","X,Y,Z"}, "%time;%value\n",
new String [] {"X_1","X_2","Y_1","Y_2","Z_1","Z_2","X,Y,Z_1","X,Y,Z_2","X,Y,Z_3"},
false));



Time	X_1	X_2	Y_1	Y_2	Z_1	Z_2	X,Y,Z_1	X,Y,Z_2	X,Y,Z_3
-1.0	1.0	X	1.0	Y	4.0	Z	1.0	1.0	4.0
0.01	1.0	X	1.23	Y	3.9432	Z	1.0	1.23	3.9432
0.02	1.023	X	1.458268	Y	3.88964856	Z	1.023	1.458268	3.88964856
0.03	1.0665267999999999	X	1.6903342152312	Y	3.8396095106879997	Z	1.0665267999999999	1.6903342152312	3.8396095106879997
0.04	1.12890754152312	X	1.9311079126320516	Y	3.7935158992745204	Z	1.12890754152312	1.9311079126320516	3.7935158992745204
0.05	1.2091275786340132	X	2.1850656580564163	Y	3.751964606617289	Z	1.2091275786340132	2.1850656580564163	3.751964606617289
0.060000000000000005	1.3067213865762535	X	2.456404684694179	Y	3.715727029169601	Z	1.3067213865762535	2.456404684694179	3.715727029169601
0.07	1.421689716388046	X	2.749168426331635	Y	3.6857727531382287	Z	1.421689716388046	2.749168426331635	3.6857727531382287
0.08	1.554437587382405	X	3.067349610456173	Y	3.6633049929641643	Z	1.554437587382405	3.067349610456173	3.6633049929641643
0.09	1.7057287896897817	X	3.414974849067593	Y	3.6498078348630214	Z	1.7057287896897817	3.414974849067593	3.6498078348630214
0.09999999999999999	1.8766533956275628	X	3.7961733386824443	Y	3.64710625318202	Z	1.8766533956275628	3.7961733386824443	3.64710625318202
0.10999999999999999	2.068605389933051	X	4.215231012728852	Y	3.6574405946181727	Z	2.068605389933051	4.215231012728852	3.6574405946181727
0.11999999999999998	2.283267952212631	X	4.6766301985089465	Y	3.6835578326154876	Z	2.283267952212631	4.6766301985089465	3.6835578326154876
0.12999999999999998	2.5226041768422625	X	5.185073427650066	Y	3.728822415376862	Z	2.5226041768422625	5.185073427650066	3.728822415376862
0.13999999999999999	2.7888511019230426	X	5.7454884328920715	Y	3.797349959898307	Z	2.7888511019230426	5.7454884328920715	3.797349959898307
0.15	3.0845148350199456	X	6.363009420901104	Y	3.8941673330395767	Z	3.0845148350199456	6.363009420901104	3.8941673330395767
0.16	3.4123642936080616	X	7.042927311409572	Y	4.025402708119227	Z	3.4123642936080616	7.042927311409572	4.025402708119227
0.17	3.7754205953882125	X	7.790598635819941	Y	4.198508819692947	Z	3.7754205953882125	7.790598635819941	4.198508819692947
0.18000000000000002	4.176938399431386	X	8.611299049492562	Y	4.422521587804853	Z	4.176938399431386	8.611299049492562	4.422521587804853

Figure 7.7: Console result for one file for all port in column

Chapter 8

Template

We provide here the template of a the original template (*User Guide Appendix A*) with the implementation of all of what MECSYCO-visu offers. Make sure to check the type of the outputs when using an observing artifact.

Java Example Template: run configuration (with observing tools)

```java
import mecsyco.communication.dds.coupling.DDSEventCouplingArtifactReceiver;
import mecsyco.communication.dds.coupling.DDSEventCouplingArtifactSender;
import mecsyco.core.agent.EventMAgent;
import mecsyco.core.agent.ObservingMAgent;
import mecsyco.core.coupling.CentralizedEventCouplingArtifact;
import mecsyco.core.exception.CausalityException;
import mecsyco.core.type.SimulData;
import mecsyco.observing.base.comparator.DataComparator;
import mecsyco.observing.base.logging.LoggingArtifact;
import mecsyco.observing.jfreechart.bar.LiveBarGraphic;
import mecsyco.observing.jfreechart.bar.PostMortemBarGraphic;
import mecsyco.observing.jfreechart.event.LiveEventGraphic;
import mecsyco.observing.jfreechart.event.PostMortemEventGraphic;
import mecsyco.observing.jfreechart.pie.LivePieGraphic;
import mecsyco.observing.jfreechart.pie.PostMortemPieGraphic;
import mecsyco.observing.jfreechart.xy.LiveTXGraphic;
import mecsyco.observing.jfreechart.xy.LiveXYGraphic;
import mecsyco.observing.jfreechart.xy.PostMortemTXGraphic;
import mecsyco.observing.jfreechart.xy.PostMortemXYGraphic;
import mecsyco.observing.jfreechart.xy.Renderer;
import mecsyco.observing.jzy3d.graphic.Live3DGraphic;
import mecsyco.observing.jzy3d.graphic.PostMortem3DGraphic;
import mecsyco.observing.swing.dispatcher.SwingDispatcherArtifact;
import mecsyco.observing.swing.r.LogToRProject;


public class LauncherWithObserving {
    public final static double maxSimulationTime = 10;

    public static void main(String args[]) {

        /**********************************/
        /**** AGENTS & MODEL ARTEFACTS ****/
        /**********************************/

        // First agent with first model (model1)
        EventMAgent agent1 = new EventMAgent("Name1",maxSimulationTime);
        MyModel1Artefact ModelArtefact1 = new MyModel1Artefact();
        agent1.setModelArtefact(ModelArtefact1);

        // Second agent with second model (model2)
        EventMAgent agent2 = new EventMAgent("Name2",maxSimulationTime);
        MyModel2Artefact ModelArtefact2 = new MyModel2Artefact();
        agent2.setModelArtefact(ModelArtefact2);


        /***************************/
        /**** COUPLING ARTEFACTS ****/
        /***************************/

        //        Model1                      Model2
        //   .--------------.            .--------------.
        //   | .---.      .---.        .---.      .---. |
        //   | | y |------| y |<-------| Y |------| Y | |
        //   | '---'      '---'        '---'      '---' |
        //   |              |            |              |
        //   | .---.      .---.        .---.      .---. |
        //   | | X |------| X |------->| x |------| x | |
        //   | '---'      '---'        '---'      '---' |
        //   '--------------'            '--------------'
        //


        CentralizedEventCouplingArtifact couplingFrom1To2 = new CentralizedEventCouplingArtifact();
        CentralizedEventCouplingArtifact couplingFrom2To1 = new CentralizedEventCouplingArtifact();

        // Agent1 will update "y" with the value received from couplingFrom2To1 (input events)
        // Agent2 will update "x" with the value received from couplingFrom1To2 (input events)
        agent1.addInputCouplingArtifact(couplingFrom2To1, "y");
        agent2.addInputCouplingArtifact(couplingFrom1To2, "x");

        // Agent1 will send "X" to couplingFrom1To2 (output events)
        // Agent2 will send "Y" to couplingFrom2To1 (output events)
        agent1.addOutputCouplingArtifact(couplingFrom1To2, "X");
        agent2.addOutputCouplingArtifact(couplingFrom2To1, "Y");

        /************************************************/
        /****LOGGING VISUALIZATION OR POST TREATMENT ****/
        /******** Check User Guide: MECSYCO-visu ********/
        /*** Check User Guide section Simulation data ***/
        /************************************************/

        /* Set the agent name for logging if you didn't named it at the creation
         *(otherwise, an unique default number is attributed)
         */
        agent1.setAgentName("Agentl1");
        agent2.setAgentName("Agentl2");

        /*Create observing Agent and the dispatcher
         * Create both for each different display windows you want
         */
        ObservingMAgent obsAgent = new ObservingMAgent ("ObserverName", maxSimulationTime);
        SwingDispatcherArtifact ObsModelArtifact = new SwingDispatcherArtifact ();
        obsAgent.setDispatcherArtifact(ObsModelArtifact);

        /*
         * Coupling Artifact and connection
         */
        CentralizedEventCouplingArtifact Agent1ToObs = new CentralizedEventCouplingArtifact();
        CentralizedEventCouplingArtifact Agent2ToObs = new CentralizedEventCouplingArtifact();

        agent1.addOutputCouplingArtifact(Agent1ToObs, "X");
        agent2.addOutputCouplingArtifact(Agent2ToObs, "Y");
        //For easy reading, we named the input port as the port we want to observed
        obsAgent.addInputCouplingArtifact(Agent1ToObs, "X");
        obsAgent.addInputCouplingArtifact(Agent2ToObs, "Y");

        /*
         *Visualization in real time  (can slow down the simulation a bit)
         *the name of ports is the one assigned as observer's input port
         *Comment the observing you don't need
         *all real time will be display on the same windows if only one was created
         */

        //Temporal graph (if ports observed are Double)
        ObsModelArtifact.addObservingArtifact (new LiveTXGraphic (
                "Graph name", "Y axis name", Renderer.Line,//or Rendere.Dot or Renderer.Step
                new String [] {"Names for display purpose, one name per port"} ,
                new String [] {"Names of ports you want to display"}));
        //XY graphics (if the port observed is a Tuple2 of Double)
        ObsModelArtifact.addObservingArtifact (new LiveXYGraphic(
                "Graph name", "X axis name", "Y axis name", Renderer.Line, //or Rendere.Dot or Renderer.Step
                "Name for display purpose", "Name of port observed"));
        //Bar chart (if the port observed is a SimulVector of Double)
        ObsModelArtifact.addObservingArtifact (new LiveBarGraphic(
                "Graph name", "X axis name", "Y axis name",
                new String [] {"Names for display purpose, one name vector's component"},
                "Name of port observed"));
        //Pie chart (if the port observed is a SimulVector of Double)
        ObsModelArtifact.addObservingArtifact (new LivePieGraphic(
                "Graph name",
                new String [] {"Names for display purpose, one name vector's component"},
                "Name of port observed"));
        //Factual representation (if the port observed is a Double)
        ObsModelArtifact.addObservingArtifact (new LiveEventGraphic(
                "Graph name", "X axis name", "Y axis name",
                "Name for display purpose", "Name of port observed"));
        //3D graphic (if the port observed is a Tuple3 of Double)
        ObsModelArtifact.addObservingArtifact (new Live3DGraphic(
                "Graph name", "X axis name", "Y axis name", "Z axis name",
                "Name of port observed"));

        /*
         *Visualization in post-mortem
         *same comment as for real time
         */

        //Temporal graph (if ports observed are Double)
        ObsModelArtifact.addObservingArtifact (new PostMortemTXGraphic (
                "Graph name", "Y axis name", Renderer.Line,//or Rendere.Dot or Renderer.Step
                new String [] {"Names for display purpose, one name per port"} ,
                new String [] {"Names of ports you want to display"}));
        //XY graphics (if the port observed is a Tuple2 of Double)
        ObsModelArtifact.addObservingArtifact (new PostMortemXYGraphic(
                "Graph name", "X axis name", "Y axis name", Renderer.Line, //or Rendere.Dot or Renderer.Step
                "Name for display purpose", "Name of port observed"));
        //Bar chart (if the port observed is a SimulVector of Double)
        ObsModelArtifact.addObservingArtifact (new PostMortemBarGraphic(
                "Graph name", "X axis name", "Y axis name",
                new String [] {"Names for display purpose, one name vector's component"},
                "Name of port observed"));
        //Pie chart (if the port observed is a SimulVector of Double)
        ObsModelArtifact.addObservingArtifact (new PostMortemPieGraphic(
                "Graph name",
                new String [] {"Names for display purpose, one name vector's component"},
                "Name of port observed"));
        //Factual representation (if the port observed is a Double)
        ObsModelArtifact.addObservingArtifact (new PostMortemEventGraphic(
                "Graph name", "X axis name", "Y axis name",
                "Name for display purpose", "Name of port observed"));
        //3D graphic (if the port observed is a Tuple3 of Double)
        ObsModelArtifact.addObservingArtifact (new PostMortem3DGraphic(
                "Graph name", "X axis name", "Y axis name", "Z axis name",
                "Name of port observed"));
        //One file for all output ports, column structure
        ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
                path+"name of file",
                new String [] {"Names of ports you want to display"}, "%time;%value \n",
                new String [] {"Headers to use"},
                false));
        //Check the order of the outputs, and give a default header list
        ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
                path+"name of file",
                new String [] {"Names of ports you want to display"}, "%time;%value \n",
                new String [] {"Headers to use"},
                true));

        /*
         *Logging
         *the name of ports is the one assigned as observer's input port
         *the name of files need the extension (.csv or else)
         */

        String path="path to folder/";
        //One file per output port (Work well only with Tuple1)
        ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
                new String [] {path+"name of file1",path+"name of file 2" /**one per port**/},
                new String [] {"Names of ports you want to display"},
                "%time;%value \n"));//column for time, one for value and ";" as column separator
        //One file per output manual fixed for other type(use one method for each file)
        ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
                path+"name of file1", new String [] {"name of port logged in this file"}, "%time;%valuenn"));
        ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
                path+"name of file2", new String [] {"name of port logged in this file"}, "%time;%valuenn"));
        //One file for all output ports, line structure
        ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
                path+"name of file",
                new String [] {"Names of ports you want to display"}, "%time;%value \n"));
        //One file for all output ports, column structure
        ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
                path+"name of file",
                new String [] {"Names of ports you want to display"}, "%time;%value \n",
                new String [] {"Headers to use"},
                false));
        //Check the order of the outputs, and give a default header list
        ObsModelArtifact.addObservingArtifact (new LoggingArtifact (
                path+"name of file",
                new String [] {"Names of ports you want to display"}, "%time;%value \n",
                new String [] {"Headers to use"},
                true));

        /*
         * Post Treatment
         */
        //Invoke Script R
        ObsModelArtifact.addObservingArtifact (new LogToRProject(
                path+"name of file to log",new String [] {"Names of ports you want to study"}));
        //Data comparator
        ObsModelArtifact.addObservingArtifact (new DataComparator(
                new String [] {path+"name of file to use as reference 1" /**one file per port**/},
                new String [] {"Names of ports you want to study"}));


        /*******************************/
        /**** MODELS INITIALIZATION ****/
        /*******************************/

        // Start the simulation softwares associated to model1 and model2
        // This is not systematically necessary, it depends on the simulation software used
        agent1.startModelSoftware();
        agent2.startModelSoftware();

        // Initialize model1 and model2 parameters (in the case values are typed double)
        // e.g. time discretization or constants
        // This is not systematically necessary, it depends on the model
        String [] args_model1 = { "0.001" };
        String [] args_model2 = { "0.01" };
        agent1.setModelParameters(args_model1);
        agent2.setModelParameters(args_model2);

        /***************************************/
        /**** CO-SIMULATION INIT & STARTING ****/
        /***************************************/

        try {
            // Co-initialization with first exchanges
            // This is necessary only when the model initial states are co-dependent
            agent1.coInitialize();
            agent2.coInitialize();

            // Start the co-simulation
            agent1.start();
            agent2.start();

            // This should never happen
        } catch (CausalityException e) {
            e.printStackTrace();
        }
    }
}
```