Fuzzy Laboratory 2

Components with (FLETPN) Models

1. Laboratory Objectives

* Acquiring the concepts
  + implementing the comparator in Java using the FuzzyP utility,
  + components with FLETPN models,
  + the control of a first-order system with PI and PID implemented with FLETPN.
* Developments and tests

1. The comparator

Considering the mathematical model for the comparator:

x3 = 1 and x4 = -1 ↔ (x1 ± δ) > x2,

The corresponding tokens are: x3 =[0,0,0,0,1] and x4 =[1,0,0,0,0]

x3 = -1 and x4 = 1 ↔ (x1 ± δ) < x2;

x3 = 1 and x4 = 1 ↔ |x1 ± δ| ≤ x2;

with δ ϵ [-0.1, 0.1].

It is required to build a program that accomplishes:

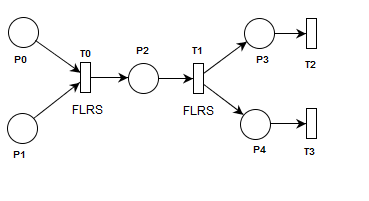
**(x1 – x2) > ε => x5 = 1, x6 = Φ**

**(x2 – x1) > ε => x6 = -1, x5 = Φ**

**| x1 – x2| < ε => x5 ≠ 1 x6 ≠ -1**

Meanning if **| x1 – x2| > ε** only one of the output places will have a token.

Figure 2.1 shows the comparator Petri Net. The comparator can be integrated into a component, where the places P0 and P1 are input ports, and the T2 and T3 transitions are the output ports. The comparator implements the logic: If P0> P1 the output transition T2 is executed with the token <0,0,0,0, 1>, and if P1> P0 executes T3 with the token <1,0,0,0,0>.



**Figure 2.1 The comparator Petri Net**

For implementation, the FLETPN model is used. The model has an executor that is implemented by a thread of execution. The thread is executed cyclically, with a certain period or when an external event is signaled. The thread is awakened when a token appears in the input place or when entering a new cycle.

Next, use the fuzzy set:

FS={NL, NM, ZR, PM, PL}, with extension *EFS = FS ∪ Φ*

Where Φ signifies the non-existence of a token, or equivalent "there is no information about that variable at this time", and in the implementation is coded with FF.

The FLRS tables attached to the transitions, depending on the arcs attached to the transitions, may have:

- 2 inputs and 2 outputs, the representation is being the TwoXTwoTable table;

- 1 input and 1 output, OneXOneTable table

- 2 inputs and 1 output, TwoXOneTable table;

- 1 input and 1 output, OneXOneTable table.

For implementation in Java, a **new package** is created to add a class with the following source code in it:

**public** **class** ComparatorExample {

// FLRS table for T0 that implements P0-P1

String differentiator = "" + //

"{[<ZR><NM><NL><NL><NL>]" + //

" [<PM><ZR><NM><NL><NL>]" + //

" [<PL><PM><ZR><NM><NL>]" + //

" [<PL><PL><PM><ZR><NM>]" + //

" [<PL><PL><PL><PM><ZR>]}";

// FLRS table for T1 that makes the selection according to the result P0-P1 (positive or negative)

String separator="{[<NL,FF><NL,FF><FF,FF><FF,PL><FF,PL>]}";

**public** ComparatorExample() {

// the Petri network is being constructed

TableParser parser = **new** TableParser();

FuzzyPetriNet petriNet = **new** FuzzyPetriNet();

// adding the input places

**int** p0Inp = petriNet.addInputPlace();

**int** p1Inp = petriNet.addInputPlace();

// attaching to the transition t0 the corresponding FLRS table

TwoXOneTable diffTable = parser.parseTwoXOneTable(differentiator);

**int** t0 = petriNet.addTransition(0, diffTable);

// add the arcs and the weights corresponding to the Petri Net

petriNet.addArcFromPlaceToTransition(p0Inp, t0, 1.0);

petriNet.addArcFromPlaceToTransition(p1Inp, t0, 1.0);

// add the places and arc corresponding to the Petri Net

**int** p2 = petriNet.addPlace();

petriNet.addArcFromTransitionToPlace(t0, p2);

**int** t1 = petriNet.addTransition(0,

parser.parseOneXTwoTable(separator));

petriNet.addArcFromPlaceToTransition(p2, t1, 1.0);

**int** p3 = petriNet.addPlace();

petriNet.addArcFromTransitionToPlace(t1, p3);

**int** p4 = petriNet.addPlace();

petriNet.addArcFromTransitionToPlace(t1, p4);

**int** t2Out = petriNet.addOuputTransition(OneXOneTable.*defaultTable*());

petriNet.addArcFromPlaceToTransition(p3, t2Out, 1.0);

// associating an action of the output transition t2

petriNet.addActionForOuputTransition(t2Out, **new** Consumer<FuzzyToken>() {

@Override

**public** **void** accept(FuzzyToken t) {

System.***out***.println( "Output From Transition 2: " + t.shortString());

}

});

**int** t3Out = petriNet.addOuputTransition(OneXOneTable.*defaultTable*());

petriNet.addArcFromPlaceToTransition(p4, t3Out, 1.0);

// associating an action of the output transition t3

petriNet.addActionForOuputTransition(t3Out, **new** Consumer<FuzzyToken>() {

@Override

**public** **void** accept(FuzzyToken t) {

System.***out***.println("Output From Transition 3: " + t.shortString());

}

});

// creating the date Petri Net executor and specifying the period in milliseconds

AsyncronRunnableExecutor executor = **new** AsyncronRunnableExecutor(petriNet, 20);

// creating an object for visualizing the behavior of the Petri net

FullRecorder recorder = **new** FullRecorder();

executor.setRecorder(recorder);

FuzzyDriver driver = FuzzyDriver.*createDriverFromMinMax*(-1.0, 1.0);

// launching the execution of the thread that contains the executor

(**new** Thread(executor)).start();

**for** (**int** i = 0; i < 100; i++) {

// constructing the dictionary collection (map) for inputs

Map<Integer, FuzzyToken> inps = **new** HashMap<>();

**if** (i % 10 < 5) {

// placing the fuzzyficated token

inps.put(p0Inp, driver.fuzzifie(i/100.0));

inps.put(p1Inp, driver.fuzzifie(i/-100.0));

} **else** {

inps.put(p1Inp, driver.fuzzifie(i/100.0));

inps.put(p0Inp, driver.fuzzifie(i/-100.0));

}

// placing the input tokens for the executer

executor.putTokenInInputPlace(inps);

**try** {

Thread.*sleep*(5);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

executor.stop();

// visualizing the Petri Net and its behavoir.

FuzzyPVizualzer.*visualize*(petriNet, recorder);

}

**public** **static** **void** main(String[] main) {

**new** ComparatorExample();

}

}

To run this application we will use the FuzzyP utility that can be downloaded from the site: <https://github.com/AttilaOrs/FuzzP/tree/master/fatJar>.

After the most recent jar is downloaded, it must be added as an external dependence on the eclipse project. Use the "Build Path" menu under "Configure Build Path," and in the "Libraries" table, click the "Add External JARs ..." button and add the downloaded file from the site above.

Classes used in the utility:

* *TableParser* is an assistant class that creates the FLRS tables. The most important methods in this class are: *parseOneXOneTable(), parseOneXTwoTable(), parseTwoXOneTable(), parseTwoXTwoTable().* These methods generate embedded tables in transitions, using strings as a differentiator and separator.
* *FuzzyPetriNet* contains static data about the Petri Nets. It has methods that add places: *addPlace()* and transitions: *addTranzition()*. In the case of transitions, the delay and the fuzzy table must be specified for that transition. The table type must coincide with the number of arcs. For example, if a transition has an input arc and two output arcs, the table must be of the *OneXTwoTable* type.
* *FuzzyPetriNet* has methods that connect places and transitions: *addArcFromTransitionToPlace()* and *addArcFromPlaceToTransition().* Input places: *addInpuPlace()* and output transitions: *addOutputTranzition()* can be added. The difference between the input places and the usual places is that for the input places we can put a token during execution. Output transitions have the ability to execute output actions. These actions can be added by the *addActionForOuputTransition()* method.
* *AsyncronRunnableExecutor* is the class where the *FuzzyPetri* net is executed. The constructor must specify the Petri Net and the period (the duration of a tic). The class implements the Runnable interface, so it's a thread of execution that needs to be run separately. Using the *putTokenInInputPlace()* method, we can place tokens at the input places. Using the *stop()* method, we can stop the execution.
* *FullRecorder* is an assisstant class that stores the status of the net. It is used for viewing.
* *FuzzyPVizualzer* views the behavior of the net. It needs *FullRecorder* and *FuzzyPetriNet* classes.
* *FuzzyDriver* is a fuzzyficator -defuzzyficator.

Exercises:

1. Run the application. Count how many times the FLETPN model runs in one tic. Change the code so that it runs only once.
2. Enter a sine signal at P0 and at P1 a cosine signal.
3. Change the code so that the transition T2 is executed if P0 <P1 and the transition T3 if P1> P0.

3. A controller with a proportional–integral (PI) action

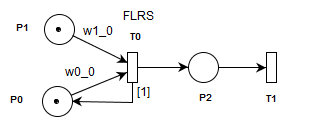
We consider for a first order process, the mathematical model of the form:

y(k+1) =a\*y(k) + b\*u(k), where y is the system status and u is the command of the controller.

In figure 3.1 we have a Petri net that implements the model, the nodes are as follows:

* + p0 stores the current value of the y(k) model;
  + p1 input place (port), modifies the received command u(k);
  + t0 modifies the update of the values of y(k) using FLRS;
  + p2 stores the new y(k + 1) value;
  + t1 output transition (port) that sends the value y(k + 1) to the controller.

The constants are w0\_0 = a and w1\_0 = b.



**Figure 3.1 The Petri Net that implements the process model**

The controller of the type PI has the mathematical model:

u(k) = u(k-1) + Δu(k)

Δu(k) = K∙ e(k) + KI∙e(k-1)

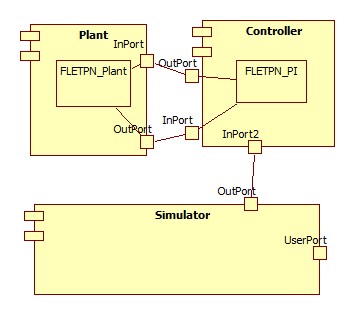
e(k) = r(k) – y(k),

where *r* is a reference and *e* is an error, e(k-1) is equal with e(k) after a time unit.

The diagram of components that simulate the control application in Figure 3.2 is given.

Exercise:

It is required to draw an application class diagram that performs the simulation of the first order process control.



**Figure 3.2 The component diagram**

The PI controller is implemented through a timed Petri net with two inputs and one output according to the model in Figure 3.2. The controller’s outputs (commands) are implemented through transitions, and inputs are implemented by places (reading the references, respectively the state of the system).

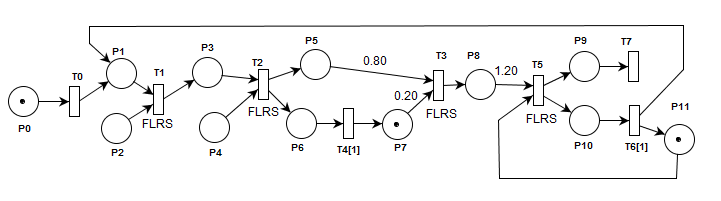


Figure 3.2 Timed Petri Net that models the PI controller

The Interpretation of the Petri Net is as follows:

P2: Takes from the installation the value of output (state), y(k);

T1: Takes the system status from the P2 input port and stores it in the place P3;

P4: is the input port that takes the reference value from the operator, r(k);

T2: Calculates the error e(k) = r(k) - y(k) and stores it in P5 and P6;

T4: Stores the previous error eant(k)= e(k-1) after a delay of 1 t.u. (time unit) at place P7;

T3: calculate the relation w7\_3\*e(k-1) + w5\_3∙e(k) where w7\_3 = 0.20 and w5\_3 = 0.80. Deviation of the command Δu(k) is stored in place P8;

P11: contains the previous value of the command u(k-1), initially set to zero;

T5: Calculates the current value of the command u(k) = u(k-1) + Δu(k) by summing the two values ​​stored in P9 and P10;

T6: Updates with delay of 1 t.u. the command value u(k) = u(k-1) and signals the restart of a new calculation of the order;

T7: Output transition (port) that transmits the command value to the plant.

The coefficients associated with the corresponding arcs from P5 to T3, from P7 to T3 and from P8 to T5, specify the constants of the PI controller. Changing their value may lead to better performance.

The following application implements the controller of a first order system. The system is simulated in the *FirstOrderedSystemThreaded* class. To execute the system, the constants A, B, C, D are specified in the *StateSpace* representation and the period.

**public** **class** FirstOrderPIControl {

/\*

\* This example implements P[k] = a\*e[k] + b\*e[k-1] + P[k-1]

\*/

String reader = "" + //

"{[<NL><NM><ZR><PM><PL>]" + //

" [<NL><NM><ZR><PM><PL>]" + //

" [<NL><NM><ZR><PM><PL>]" + //

" [<NL><NM><ZR><PM><PL>]" + //

" [<NL><NM><ZR><PM><PL>]}"; //

String doubleChannelAdder = ""//

+ "{[<NL,NL><NL,NL><NL,NL><NM,NM><ZR,ZR>]" //

+ " [<NL,NL><NL,NL><NM,NM><ZR,ZR><PM,PM>]" //

+ " [<NL,NL><NM,NM><ZR,ZR><PM,PM><PL,PL>]"//

+ " [<NM,NM><ZR,ZR><PM,PM><PL,PL><PL,PL>]"//

+ " [<ZR,ZR><PM,PM><PL,PL><PL,PL><PL,PL>]}";

String doubleChannelDifferentiator = ""//

+ "{[<ZR,ZR><PM,PM><PL,PL><PL,PL><PL,PL>]" //

+ " [<NM,NM><ZR,ZR><PM,PM><PL,PL><PL,PL>]" //

+ " [<NL,NL><NM,NM><ZR,ZR><PM,PM><PL,PL>]"//

+ " [<NL,NL><NL,NL><NM,NM><ZR,ZR><PM,PM>]"//

+ " [<NL,NL><NL,NL><NL,NL><NM,NM><ZR,ZR>]}";

String doubleChannelDifferentiator2 = ""//

+ "{[<ZR,ZR><nm,nm><nl,nl><nl,nl><nl,nl>]" //

+ " [<pm,pm><ZR,ZR><nm,nm><nl,nl><nl,nl>]" //

+ " [<pl,pl><pm,pm><ZR,ZR><nm,nm><nl,nl>]"//

+ " [<pl,pl><pl,pl><pm,pm><ZR,ZR><nm,nm>]"//

+ " [<pl,pl><pl,pl><pl,pl><pm,pm><ZR,ZR>]}";

String adder = String.*join*("\n", //

"{[<NL><NL><NL><NM><ZR>]", //

" [<NL><NL><NM><ZR><PM>]", //

" [<NL><NM><ZR><PM><PL>]", //

" [<NM><ZR><PM><PL><PL>]", //

" [<ZR><PM><PL><PL><PL>]}");

**public** FirstOrderPIControl() {

// specify constants and period

**long** period = 10;

FirtsOrderSystemThreaded plant = **new**

FirtsOrderSystemThreaded(0.5, 0.7, 0.2, 0.3, period);

// specify the exit interval of the system

FuzzyDriver plantInDriver =

FuzzyDriver.*createDriverFromMinMax*(-0.6, +0.6);

FuzzyDriver userCommandInDriver =

FuzzyDriver.*createDriverFromMinMax*(-0.6, +0.6);

FuzzyDriver controlOutDriver =

FuzzyDriver.*createDriverFromMinMax*(-1.0, 1.0);

// the Petri Net is being built

TableParser parser = **new** TableParser();

FuzzyPetriNet net = **new** FuzzyPetriNet();

**int** p0 = net.addPlace();

net.setInitialMarkingForPlace(p0, FuzzyToken.*zeroToken*());

**int** t0 = net.addTransition(0, OneXOneTable.*defaultTable*());

net.addArcFromPlaceToTransition(p0, t0, 1.0);

**int** p1 = net.addPlace();

net.addArcFromTransitionToPlace(t0, p1);

**int** p2InpSys = net.addInputPlace();

**int** t1 = net.addTransition(0, parser.parseTable(reader));

net.addArcFromPlaceToTransition(p1, t1, 1.0);

net.addArcFromPlaceToTransition(p2InpSys, t1, 1.0);

**int** p3 = net.addPlace();

net.addArcFromTransitionToPlace(t1, p3);

**int** p4InpCmd = net.addInputPlace();

**int** t2 = net.addTransition(0,

parser.parseTable(doubleChannelDifferentiator2));

net.addArcFromPlaceToTransition(p4InpCmd, t2, 1.0);

net.addArcFromPlaceToTransition(p3, t2, 1.0);

**int** p5 = net.addPlace();

net.addArcFromTransitionToPlace(t2, p5);

**int** p6 = net.addPlace();

net.addArcFromTransitionToPlace(t2, p6);

**int** t3 = net.addTransition(0, parser.parseTable(adder));//

**int** t4delay = net.addTransition(1,

OneXOneTable.*defaultTable*());

net.addArcFromPlaceToTransition(p6, t4delay, 1.0);

**int** p7Mem = net.addPlace();

net.setInitialMarkingForPlace(p7Mem, FuzzyToken.*zeroToken*());

net.addArcFromTransitionToPlace(t4delay, p7Mem);

net.addArcFromPlaceToTransition(p7Mem, t3, 0.2);

net.addArcFromPlaceToTransition(p5, t3, 0.8);

**int** p8 = net.addPlace();

net.addArcFromTransitionToPlace(t3, p8);

**int** t5 = net.addTransition(0,

parser.parseTable(doubleChannelAdder));

net.addArcFromPlaceToTransition(p8, t5, 1.2);

**int** p9 = net.addPlace();

net.addArcFromTransitionToPlace(t5, p9);

**int** p10 = net.addPlace();

net.addArcFromTransitionToPlace(t5, p10);

**int** t6delay = net.addTransition(1,

OneXTwoTable.*defaultTable*());

net.addArcFromPlaceToTransition(p10, t6delay, 1.0);

net.addArcFromTransitionToPlace(t6delay, p1);

**int** p11Mem = net.addPlace();

net.setInitialMarkingForPlace(p11Mem, FuzzyToken.*zeroToken*());

net.addArcFromTransitionToPlace(t6delay, p11Mem);

net.addArcFromPlaceToTransition(p11Mem, t5, 1.0);

**int** t7Out =

net.addOuputTransition(OneXOneTable.*defaultTable*());

net.addArcFromPlaceToTransition(p9, t7Out, 1.0);

net.addActionForOuputTransition(t7Out, **new**

Consumer<FuzzyToken>() {

@Override

**public** **void** accept(FuzzyToken t) {

System.***out***.println(

"Output From Transition 2: " + t.shortString());

plant.setCommand(controlOutDriver.defuzzify(t));

}

});

AsyncronRunnableExecutor executor = **new**

AsyncronRunnableExecutor(net, period);

FullRecorder recorder = **new** FullRecorder();

executor.setRecorder(recorder);

// the threads of execution for the controller and system are launched

(**new** Thread(plant)).start();

(**new** Thread(executor)).start();

// a reference is inserted

**double** command = 0.55;

**for** (**int** i = 0; i < 200; i++) {

**if** (i > 100) {

command = 0.35;

}

HashMap<Integer, FuzzyToken> input = **new** HashMap<>();

input.put(p4InpCmd,

userCommandInDriver.fuzzifie(command));

input.put(p2InpSys,

plantInDriver.fuzzifie(plant.curentStatus()));

executor.putTokenInInputPlace(input);

**try** {

Thread.*sleep*(period);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

plant.stop();

executor.stop();

MainView mainView = FuzzyPVizualzer.*visualize*(net, recorder);

}

**public** **static** **void** main(String args[]) {

**new** FirstOrderPIControl();

}

}

**public** **class** FirtsOrderSystemThreaded **implements** Runnable {

**double** a, b, c, d;

**double** x;

**private** **double** currentStatus;

**private** **volatile** **double** command;

**private** **boolean** stop;

**private** **long** period;

**public** FirtsOrderSystemThreaded(**double** a, **double** b, **double** c,

**double** d, **long** period) {

**this**.a = a;

**this**.b = b;

**this**.c = c;

**this**.d = d;

command = 0.0;

x = 0.0;

stop = **false**;

**this**.period = period;

}

// adds the system input

**public** **void** setCommand(**double** cmd) {

command = cmd;

}

**public** **void** stop() {

stop = **true**;

}

// returns the state of the system

**public** **double** curentStatus() {

**return** currentStatus;

}

**private** **void** executeSystem() {

**double** xNew = a \* x + b \* command;

currentStatus = c \* x + d \* command;

x = xNew;}

@Override

**public** **void** run() {

**while** (!stop) {

executeSystem();

**try** {

Thread.*sleep*(period);

} **catch** (InterruptedException e) {

e.printStackTrace();}

}

}

}

4. Exercises:

1. Change the weight of the arcs so that the performance of the system would be as good as possible.

2. Modify the first order system constants. Find the values of the driver and the coefficient values so the system has the best performance behavior.

3. Starting from the previous application, develop an application to implement the proportional integral derivative (PID) controller. The Petri Net will be in Figure 4.1. Where do we need to add a token to make the controller run correctly?

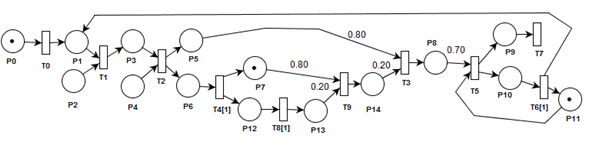


Figure 4.1 FLETPN for PID.

5. Verification of knowledge

1. How to build a Petri Net using the FuzzyP utility?

2. When is the executior launched and what is the execution frequency?

3. How does the performance of the system influence the weight of the arcs?

4. What are the instructions that implement component ports?

5. How to attach FLRS tables to transitions using the FuzzyP utility?