Real-Time Software Development

Exercise 2

University of Twente

June 2019

Argyrios Kokkinis
a.kokkinis@student.utwente.nl
s2252406

Jiahao Cui
j.cui-2@student.utwente.nl
s2094509

Table of Contents

PART 1

➢ 2.1.1

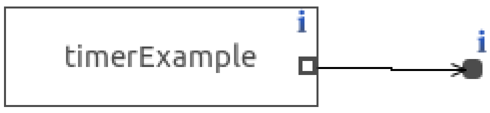


Figure 1. Architecture model with the timer

The timerExample model contains the producer-consumer model and a port used as a timer , creating a delay between two consecutive executions.

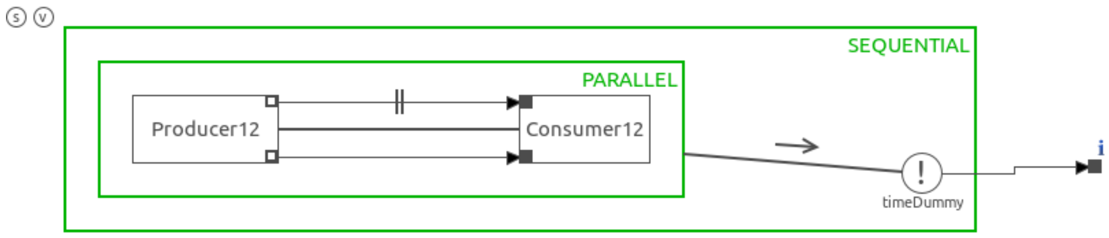
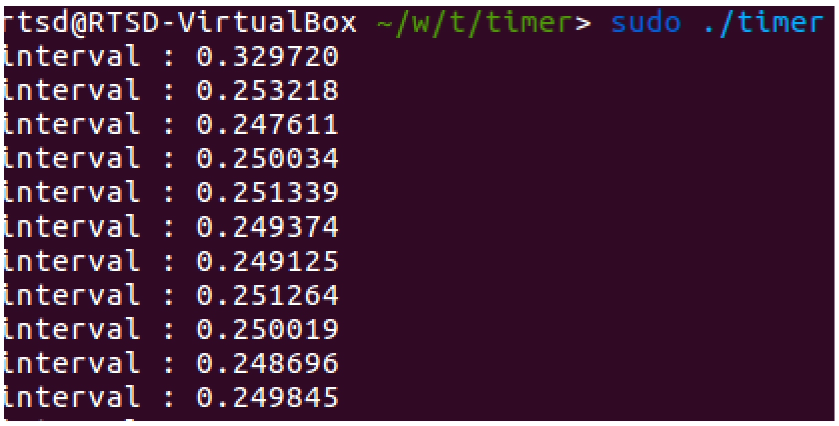


Figure 2. Producer-Consumer sub model

The sub model above is the deadlock free model of the Producer-Consumer example provided in exercise 1.

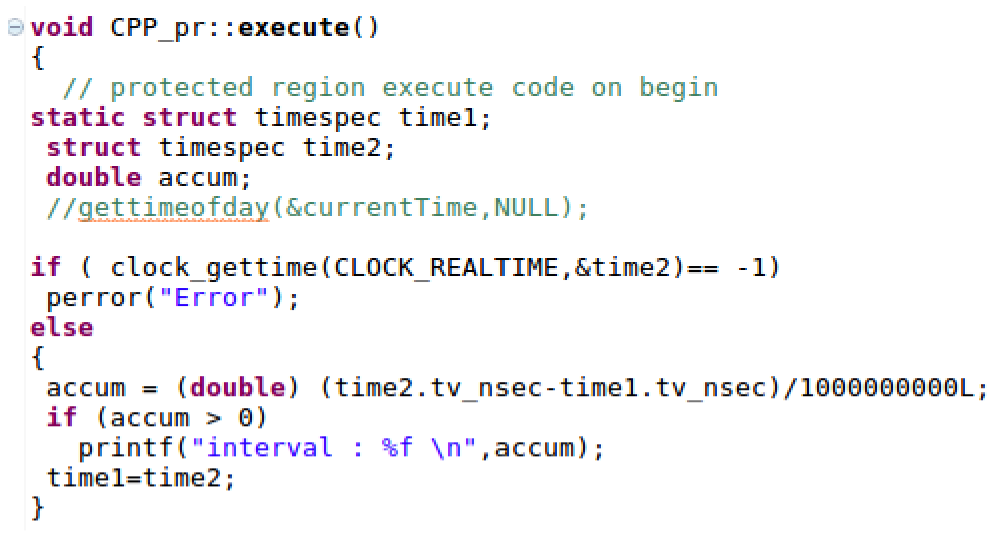
➢ 2.1.2

The delay was set to 0.25 sec. We see that the Producer-Consumer model is executed about every 0.25 sec. To make that effect visible we measure the timing difference between two consecutive executions of the producer’s code block and we print a relevant message on the terminal.



```
rtsd@RTSD-VirtualBox ~/w/t/timer> sudo ./timer
interval : 0.329720
interval : 0.253218
interval : 0.247611
interval : 0.250034
interval : 0.251339
interval : 0.249374
interval : 0.249125
interval : 0.251264
interval : 0.250019
interval : 0.248696
interval : 0.249845
```

Figure 3. Printing the timing intervals



```
void CPP_pr::execute()
{
  // protected region execute code on begin
static struct timespec time1;
 struct timespec time2;
 double accum;
 //gettimeofday(&currentTime,NULL);

if ( clock_gettime(CLOCK_REALTIME,&time2)== -1)
 perror("Error");
else
{
 accum = (double) (time2.tv_nsec-time1.tv_nsec)/1000000000L;
 if (accum > 0)
   printf("interval : %f \n",accum);
 time1=time2;
}
```

Figure 4.producer's code

➢ 2.1.3

The delay was set to 0.01 sec and we extracted a graph showing the interval over 570 executions . In this case we expect the timing interval to be at 0.01 sec.

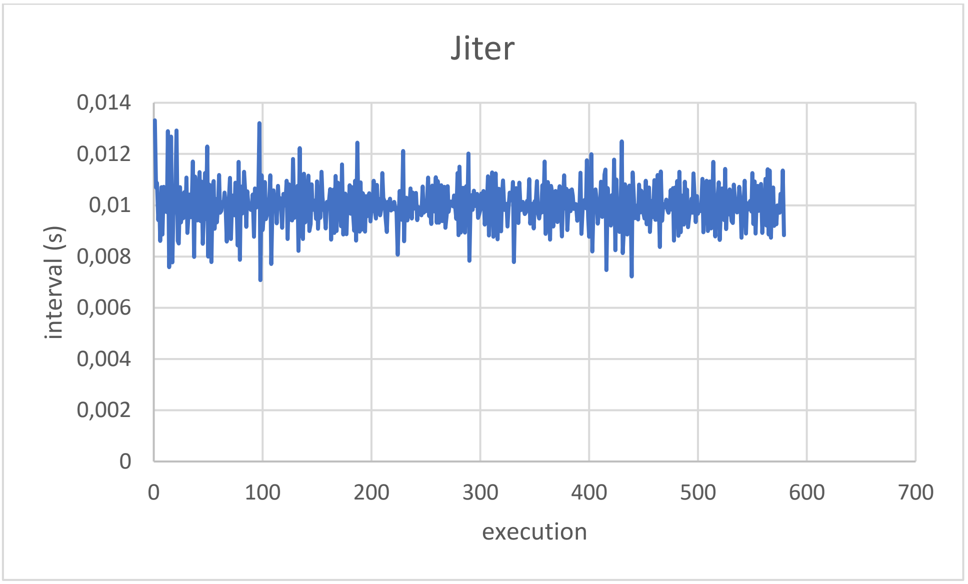


Figure 5.timing intervals over 570 executions

➢ 2.1.4

From the graph in question 2.1.3 we see that there is quite a large variation and the time difference between two consecutive executions cannot be considered constant. This is not what we expected to happen(as we set it precisely to 0.01). The standard deviation is 0.000869 sec with the maximum interval being at 0.013314 sec and the average being at 0.009998 sec (close to what we expected). Although timing variations are not something we expected to happen at first, we know that performing timing simulations using a virtual machine is not a valid way of understanding thoroughly the timing behavior of our system. Virtual machine is executed by the host OS and this is the one that schedules its execution. As long as our host OS is not real-time we shouldn't expect to have precise timing simulations from our model.

➢ 2.1.5

From the above graph we may assume that the maximum (not expected) delay is at 3.314 ms. Whether this delay would be an issue in a real time system or not depends on the system itself. For instance , in a soft real time system this delay shouldn't be considered a problem as those systems aren't supposed the be extremely punctual. However, hard real time systems aren't supposed to have delays greater than 1 ms . Systems such as artificial cardiac pacemakers or brake systems should be able to take decisions within a very limited time gap and be extremely punctual, otherwise they are considered dangerous. Also, those timing variations aren't bounded and thus we can't know whether the delay of 3.314 ms is really the maximum one or not.

PART 2

➢ 2.2.1

Closed feedback loop control system with a P-controller $u = k(SP - MV) = 1(1 - y) = 1 - x$

With state equations $\dot{x} = Ax + Bu = -0.3x + u = -1.3x + 1\,,\; x(0) = 0$ and $y = Cx = x$

Based on that we expect output to be $y(t) = 0.769231 - 0.769231e^{-1.3t}$.

Modeling the above system on 20-sim:

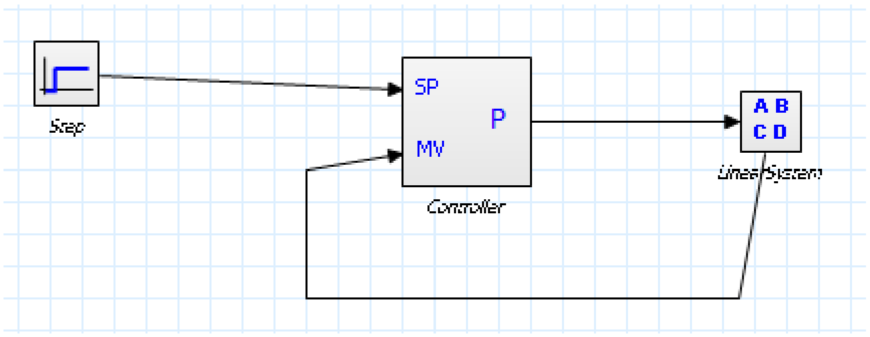


Figure 6.Control system in 20-sim

Having an euler integration method with stepsize 0.01 and running it four times we have an output which is what we expected based on the system's equations.

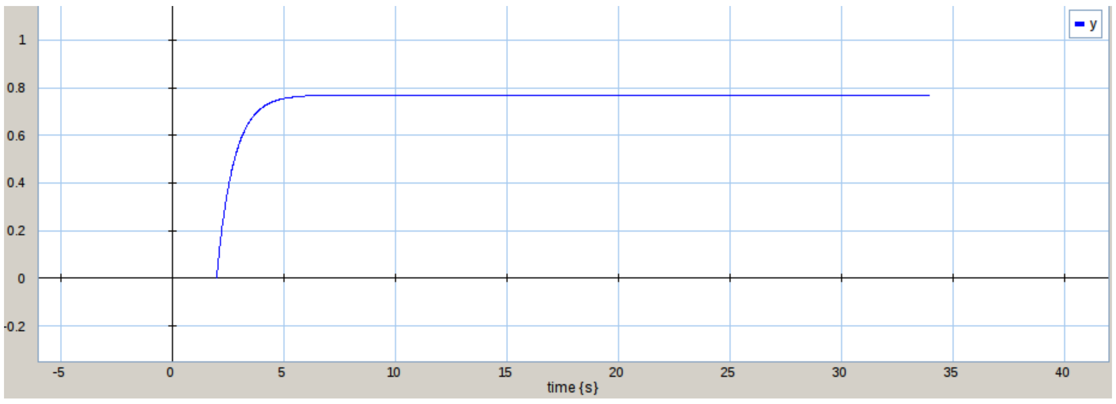


Figure 7.20-sim simulation

➢ 2.2.2

The above system in TERRA using normal processes:

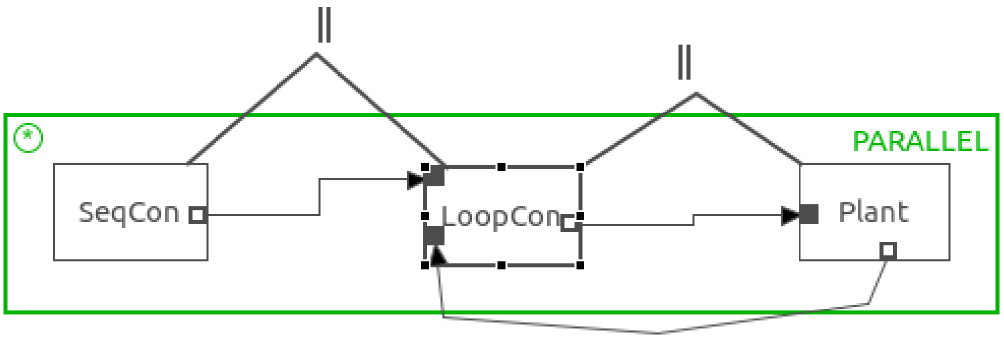


Figure 8.TERRA system

Because of the closed loop a normal implementation(read-compute-write) of the sub-processes would result to a deadlock, because the process Plant would wait for an input from the LoopController and the LoopController would wait for a feedback from the Plant.

For that reason we reversed the sequence.

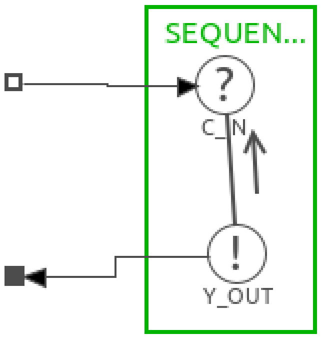


Figure 9.Plant

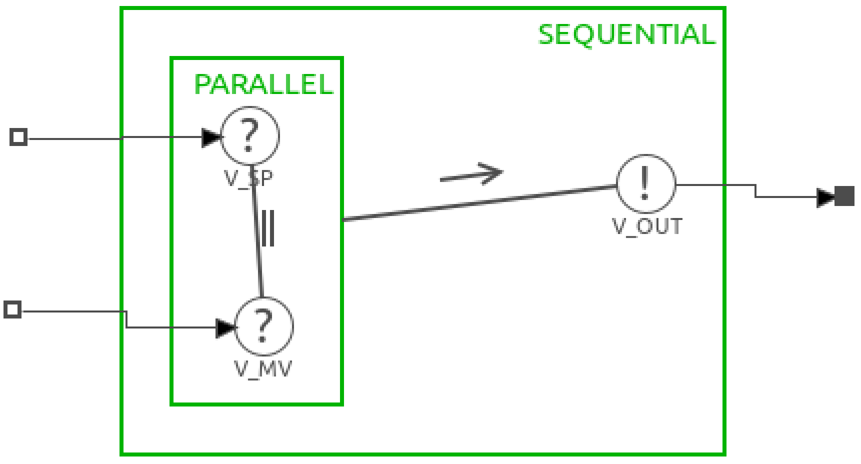
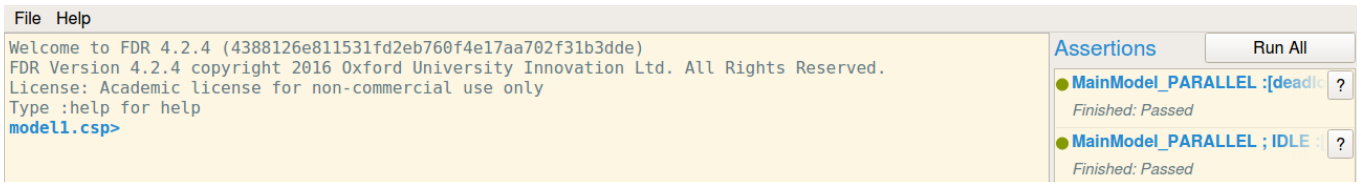


Figure 10.Loop Controller

Running FDR we confirm that a system that has the above structure is deadlock free



```
Welcome to FDR 4.2.4 (4388126e811531fd2eb760f4e17aa702f31b3dde)
FDR Version 4.2.4 copyright 2016 Oxford University Innovation Ltd. All Rights Reserved.
License: Academic license for non-commercial use only
Type :help for help
model1.csp>
```

```
● MainModel_PARALLEL :[deadl
Finished: Passed
● MainModel_PARALLEL ; IDLE
Finished: Passed
```

Figure 11.testing the control system for deadlocks

- ## 2.2.3

Using the code blocks extracted from 20-sim and keeping the above deadlock free structure:

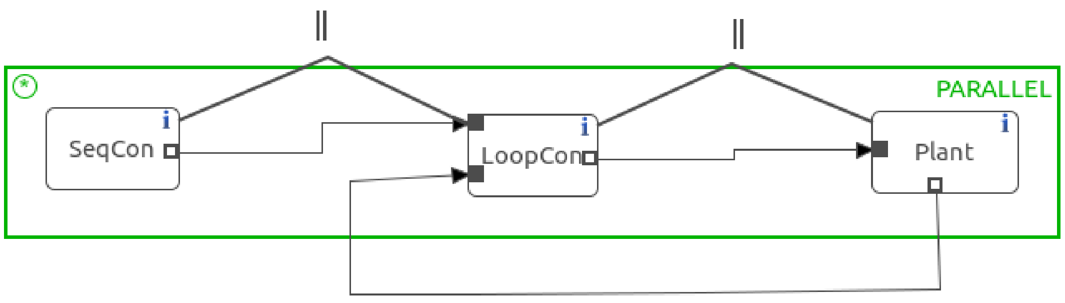


Figure 12.Top-level architecture

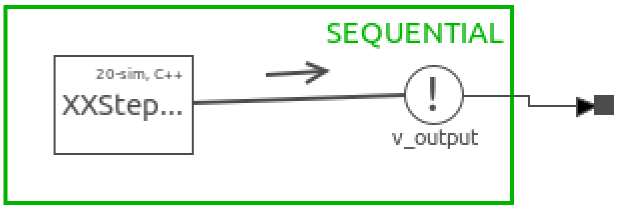


Figure 14.Sequence Controller

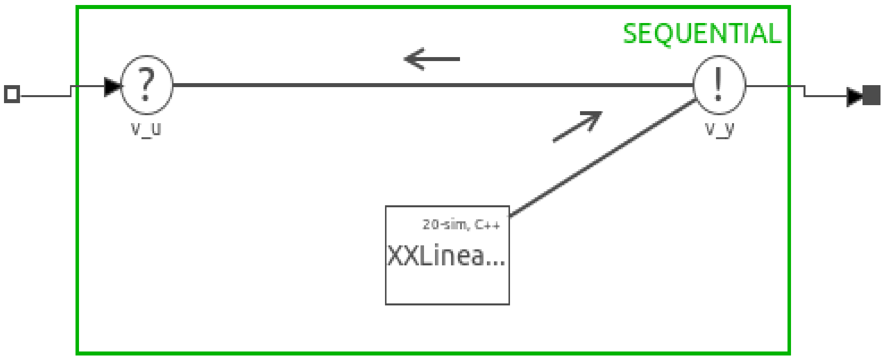


Figure 13.Plant

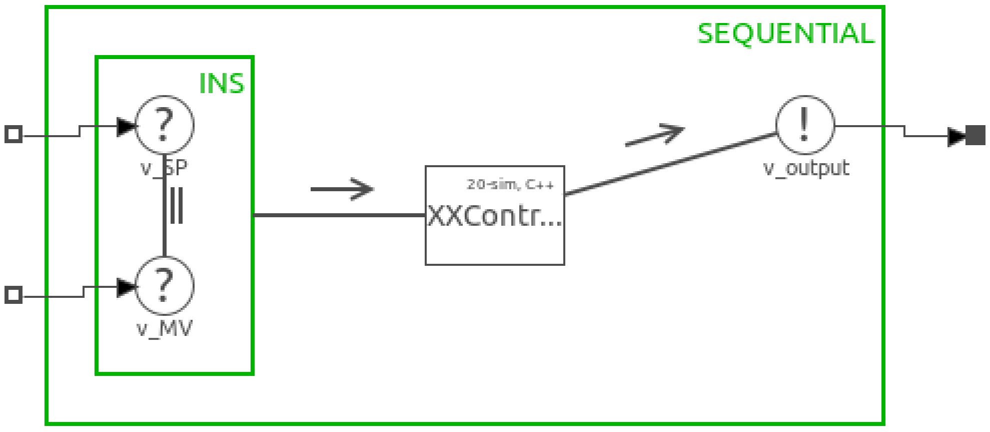


Figure 15.Loop Controller

Running a log event we see that the output is similar to the one that was produced from 20-sim

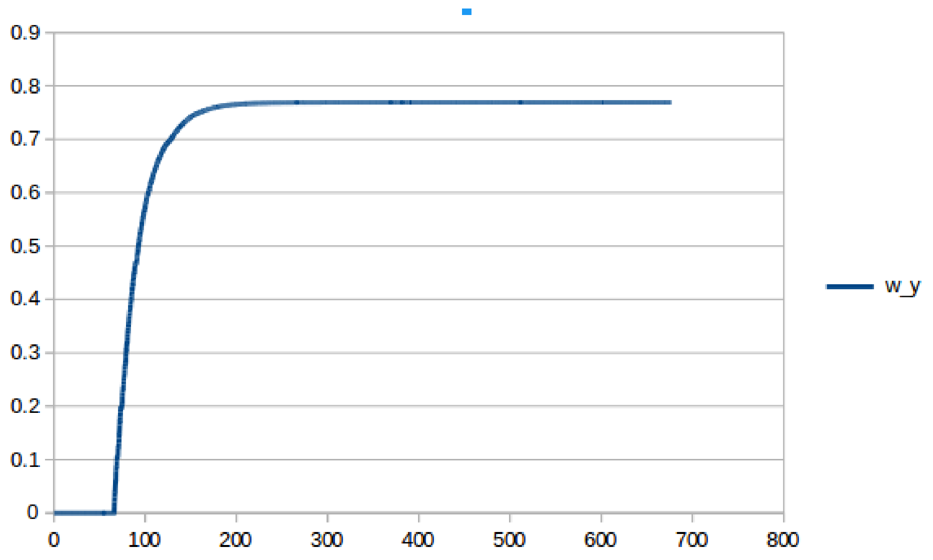


Figure 16.Output from TERRA

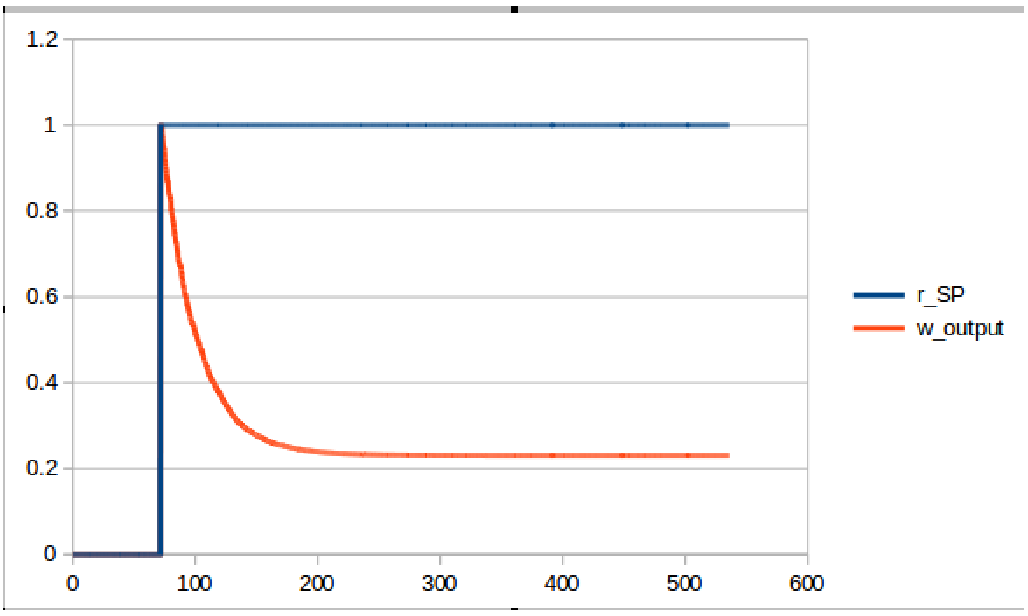
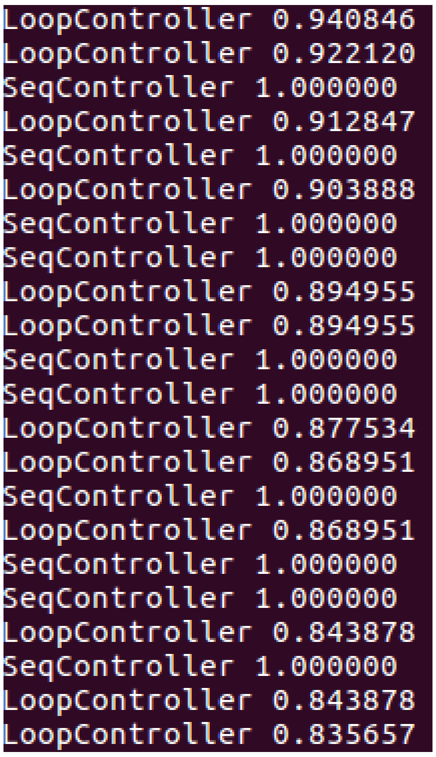


Figure 17.Control signal and pulse

On the above graph we see the output of the sequence controller (blue) and the loop controller (orange). The output of the loop controller is actually the value 1- x . Starting from x being zero and reaching the final value of 0.769231 we validate that the controller's output is correct.

Also letting the processes printing on the terminal we can also check their outputs over time.



```
LoopController 0.940846
LoopController 0.922120
SeqController 1.000000
LoopController 0.912847
SeqController 1.000000
LoopController 0.903888
SeqController 1.000000
SeqController 1.000000
LoopController 0.894955
LoopController 0.894955
SeqController 1.000000
SeqController 1.000000
LoopController 0.877534
LoopController 0.868951
SeqController 1.000000
LoopController 0.868951
SeqController 1.000000
SeqController 1.000000
LoopController 0.843878
SeqController 1.000000
LoopController 0.843878
LoopController 0.835657
```

Figure 18.printing the values on terminal

➢ 2.2.4

For splitting the system into one “hard” real time control system and one “soft” real time we used buffered channels and the priority parallel configuration between the two systems.

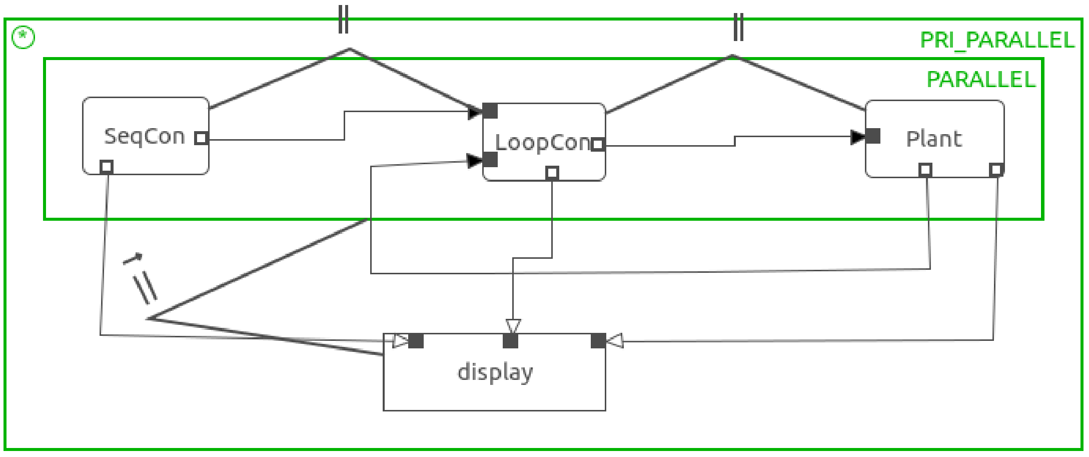


Figure 19.Decoupled system

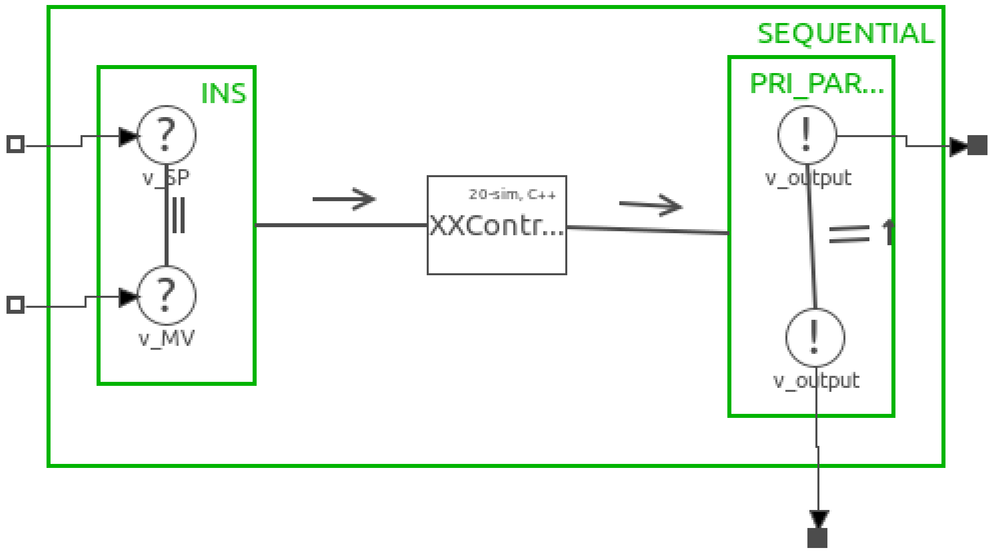


Figure 20.Loop Controller in the new system

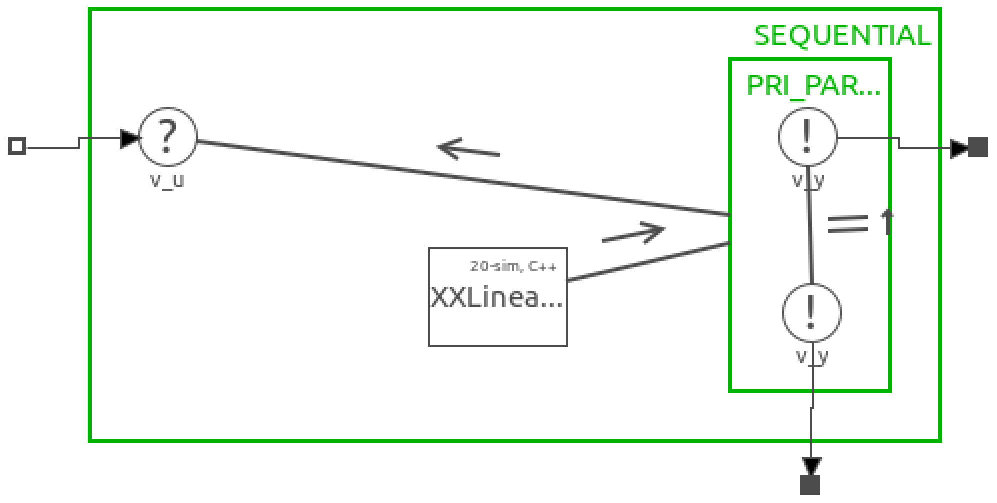


Figure 21.Plant in the new system

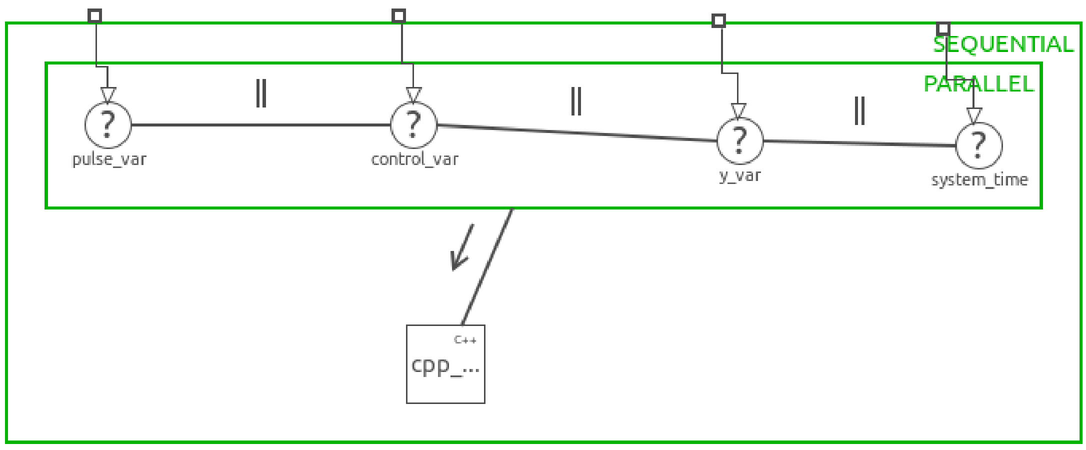
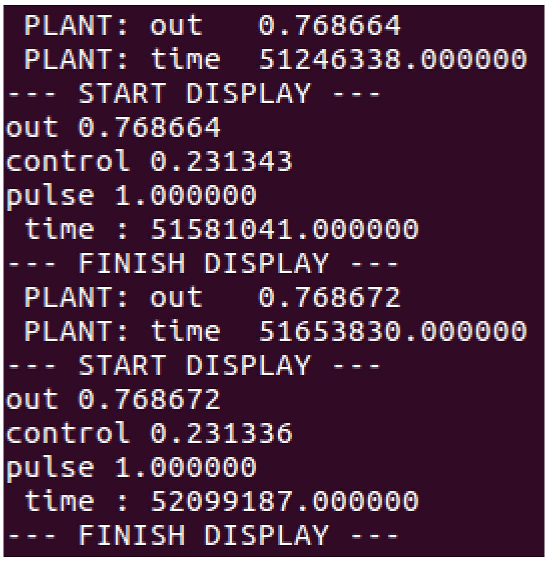


Figure 22.Display process with an extra reader for 2.2.5

- ## 2.2.5

For understanding the timing difference in the execution of the two systems (control system and display) we let both systems printing a message showing the time and the output when the calculate it. What we see is the plant being executed first and after some μsec the display process following.



```
 PLANT: out    0.768664
 PLANT: time   51246338.000000
--- START DISPLAY ---
out 0.768664
control 0.231343
pulse 1.000000
 time : 51581041.000000
--- FINISH DISPLAY ---
 PLANT: out    0.768672
 PLANT: time   51653830.000000
--- START DISPLAY ---
out 0.768672
control 0.231336
pulse 1.000000
 time : 52099187.000000
--- FINISH DISPLAY ---
```

Figure 23.Showing the values on terminal

Also we added a second writer to the plant, sending its execution time to the display process which apart from printing messages calculates and the timing difference between the two executions.

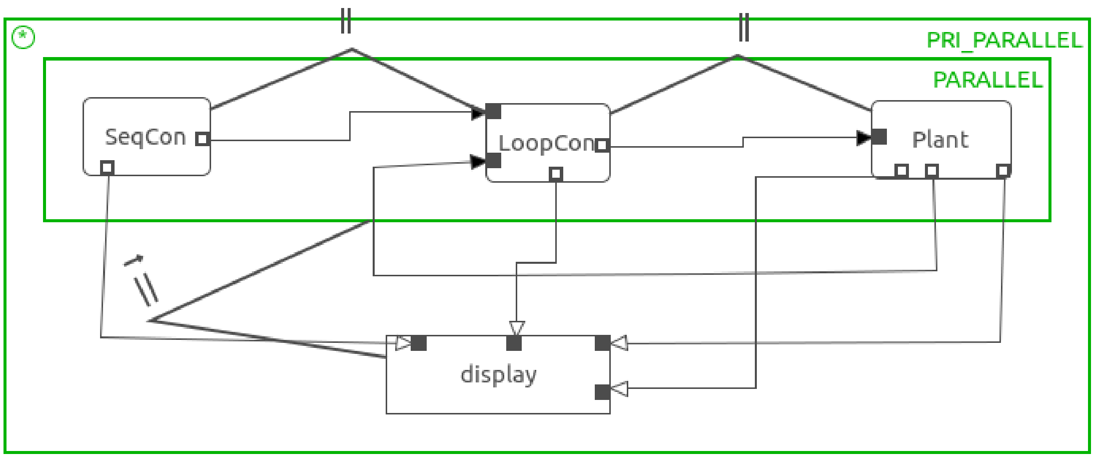


Figure 24.Extra channel for timing purposes

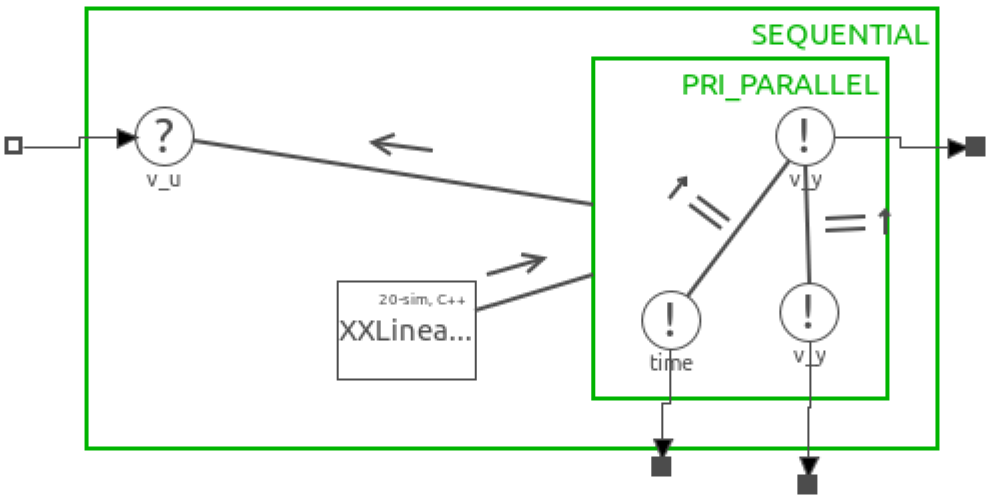
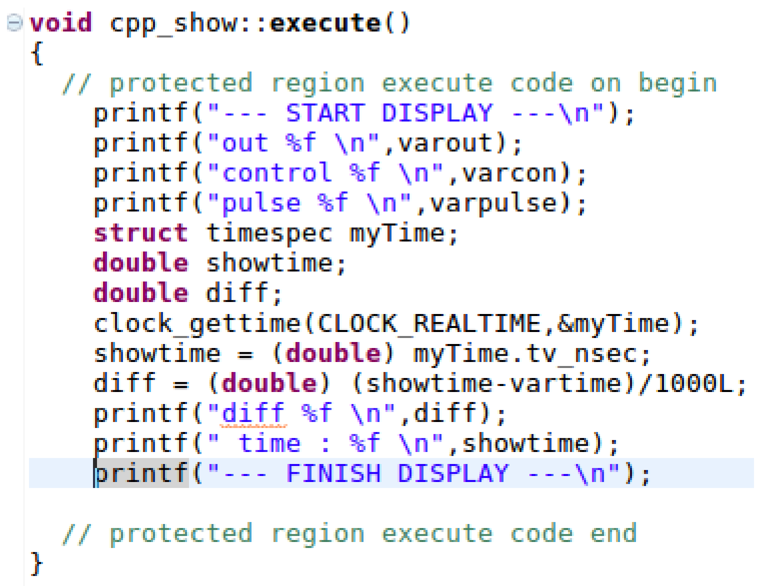


Figure 25.Plant with an extra writer



```
void cpp_show::execute()
{
  // protected region execute code on begin
    printf("--- START DISPLAY ---\n");
    printf("out %f \n",varout);
    printf("control %f \n",varcon);
    printf("pulse %f \n",varpulse);
    struct timespec myTime;
    double showtime;
    double diff;
    clock_gettime(CLOCK_REALTIME,&myTime);
    showtime = (double) myTime.tv_nsec;
    diff = (double) (showtime-vartime)/1000L;
    printf("diff %f \n",diff);
    printf(" time : %f \n",showtime);
    printf("--- FINISH DISPLAY ---\n");

  // protected region execute code end
}
```

Figure 26.Code in the display process

From that we created a graph that shows the timing differences.

Because of the buffered channels and the priority parallel relation the control part is being executed without having to wait for the display process to print the results. So , always the display part will be executed in a later stage.

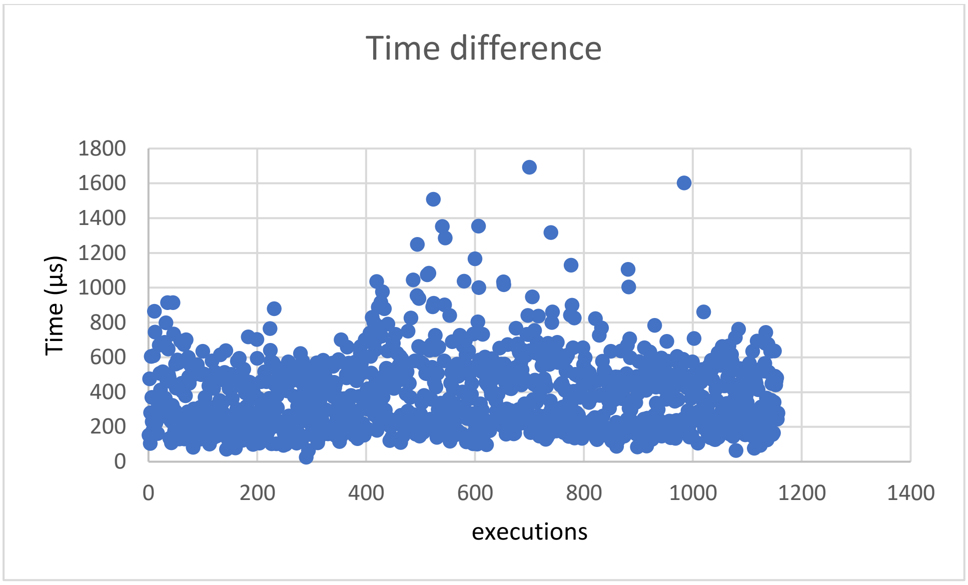


Figure 27.Graph showing the difference in execution

We see that the delay can be up to about 1.8 msec in this simulation.

➢ 2.2.6

FDR is used mainly for checking whether the system is deadlock free or not. What FDR does is checking the structure of the model. It can't recognize the fact that the channels are buffered but it understands the connections and the configuration between the processes. FDR works on lower more abstract level.

➢ 2.2.7

In case that we had a more complicated plant with different sub systems then each sub system would be executed at a different time and at each time we would have a different output from each sub system. A way that we could make it work is to extend the "soft" real time system in a way that for each plant there will be one display process and all display systems will be executed in parallel.

PART 3

- 2.3.1

Running the JIWY setup on 20-sim using the original implementation.

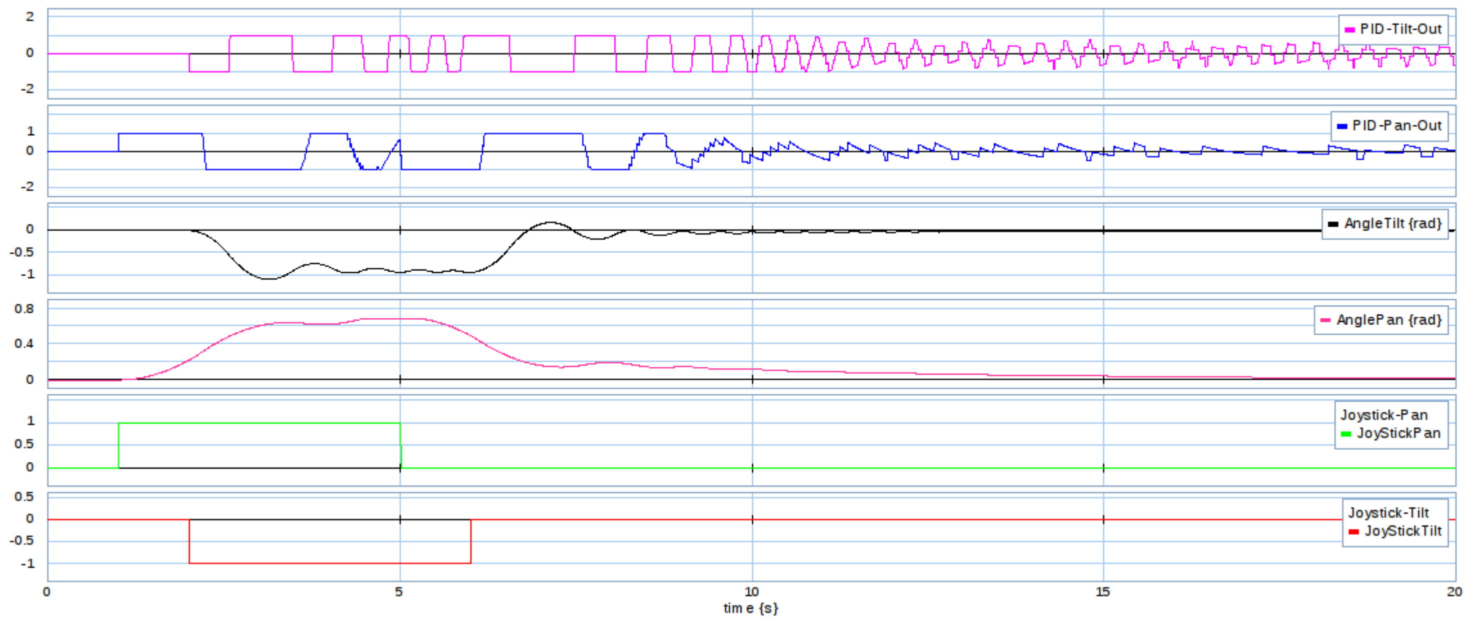


Figure 28.20-sim Controller simulation

For creating an implementation in TERRA we extracted the FMUs for the Joystick , Pan Controller and Tilt Controller and we used those FMUs in a TERRA model.

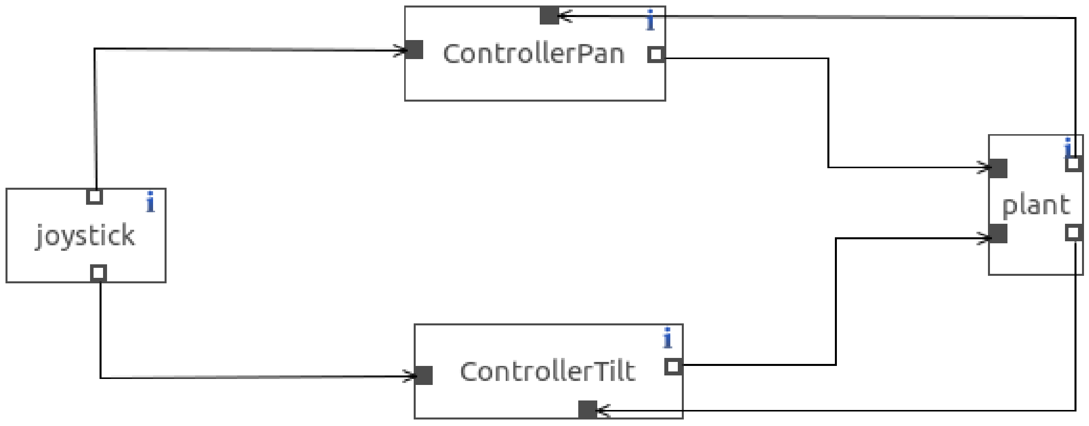


Figure 29.Architecture model of JIWY

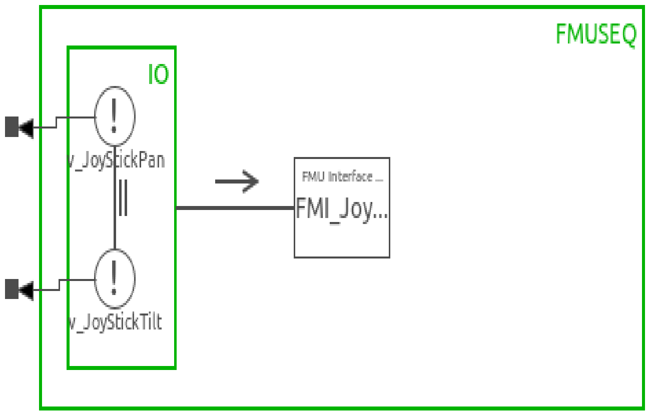


Figure 31.Joystick model

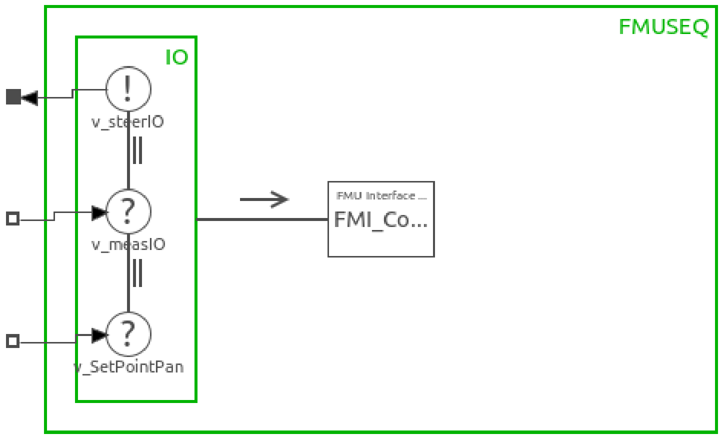


Figure 30.Pan Controller

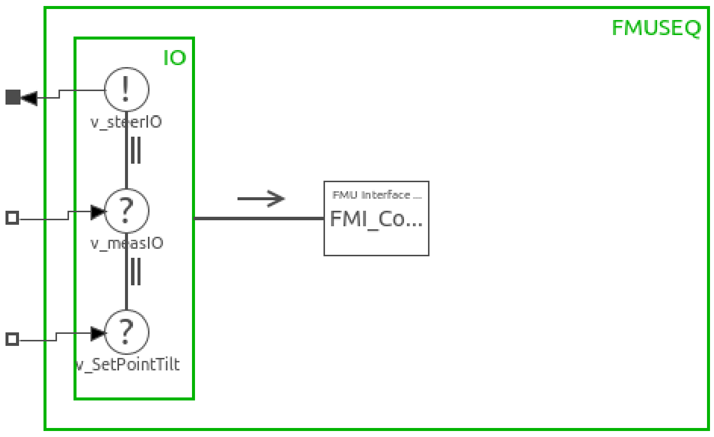


Figure 32.Tilt Controller

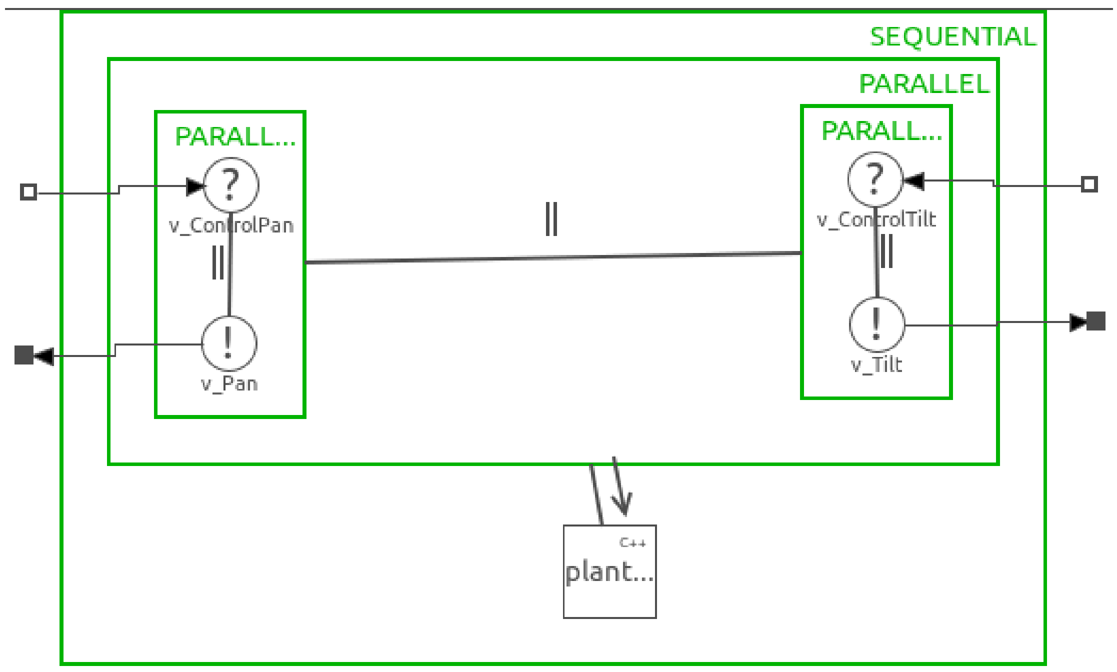
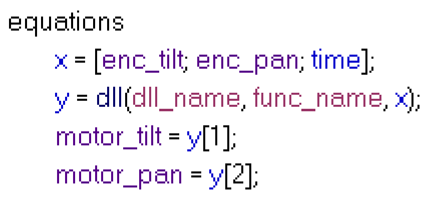


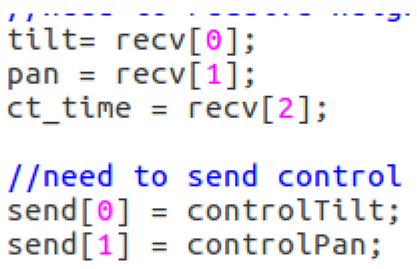
Figure 33.Plant

For performing simulation using the zmq we added the necessary libraries and we changed the inputs and outputs in the plant code block so that they will correspond to the inputs and outputs of our system. The inputs are the control signals for tilt and pan , and the outputs are the tilt and pan that are produced from our device.



```
equations
    x = [enc_tilt; enc_pan; time];
    y = dll(dll_name, func_name, x);
    motor_tilt = y[1];
    motor_pan = y[2];
```

Figure 34. I/O as they are expected to be



```
tilt= recv[0];
pan = recv[1];
ct_time = recv[2];

//need to send control
send[0] = controlTilt;
send[1] = controlPan;
```

Figure 35.Confuguring the I/O in the c++ block

Having the above TERRA implementation of our controller we run a 20-sim simulation in which we expected to have the same results with the simulation that we performed earlier using the original implementation.

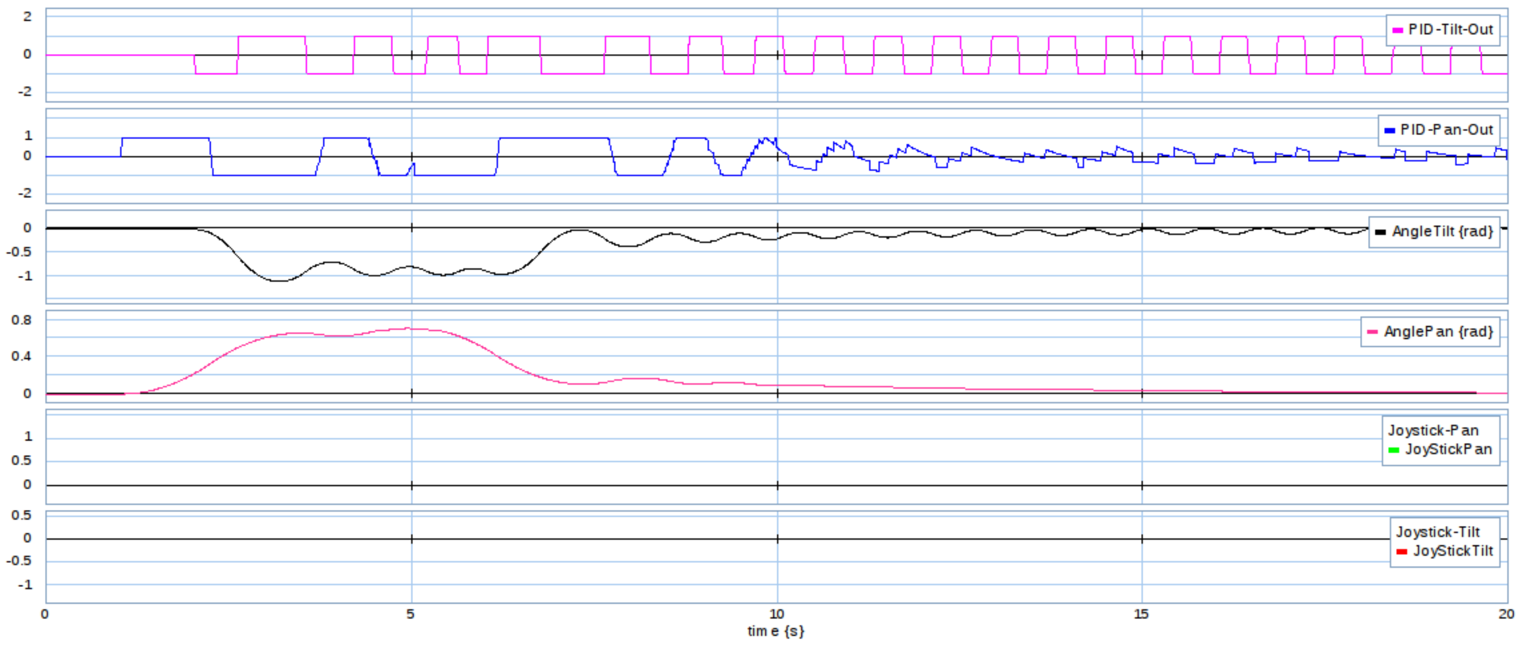


Figure 36. Simulation using the TERRA implementation

In the simulation using the TERRA model of our controller we don't see the graphs showing the Joystick Tilt and Pan values over time . This is because those signals aren't set in the TERRA implementation as we can see from *figure 34.*

The results are quite close to the ones from the first simulation however we see that the outputs appear to be smoother and both control signals and outputs having a small offset from the original simulation.

Also there is difference in the tilt control signal which has become extremely smooth in the TERRA simulation.

In general we see that the second simulation can be considered accurate enough for having a rough estimation of our system's behavior however it doesn't show correctly abrupt changes that may happen in the control and output signals.

- 2.3.2

Because the simulation using TERRA implementation doesn’t show correct abrupt changes in the signals it cannot be used for simulating safety critical systems. What can be done is to change the models in the ControllerPan and the ControllerTilt by altering the code that was produced from 20-sim in a way that the control signals will become more edgy. Also, the controlTilt and controlPan signals can by multiplied with a noise function before entering the plant. In this way we can have a simulation showing the output when there is noise in the system, having a better insight of how the system responds in extreme conditions.

- 2.3.3

FMUs is a standard interface for simulations which is relatively simple. We can easily extract the xml files that contain information about our system , having the equations that describe the system’s behavior over time (and models of the different controllers). Those xml files can be imported in different environments (TERRA in our case) and use them in co-simulation. It is a simple and effective method of translating a model of a physical system into a form that can be used for simulations. ZeroMQ is a library used for asynchronous messaging. In our case we can easily use it alongside with the FMUs by adding the appropriate libraries and code , establishing a communication protocol between the plant and the controllers.

PART 4

- 2.4.1

Creating a JIWY controller that is a composed of a model responsible for the horizontal motion , one for the vertical and a Check process that is in between the controller and the plant.
For testing the system below we had to add dummy processes(only readers and writers) for not having the ports open ended.

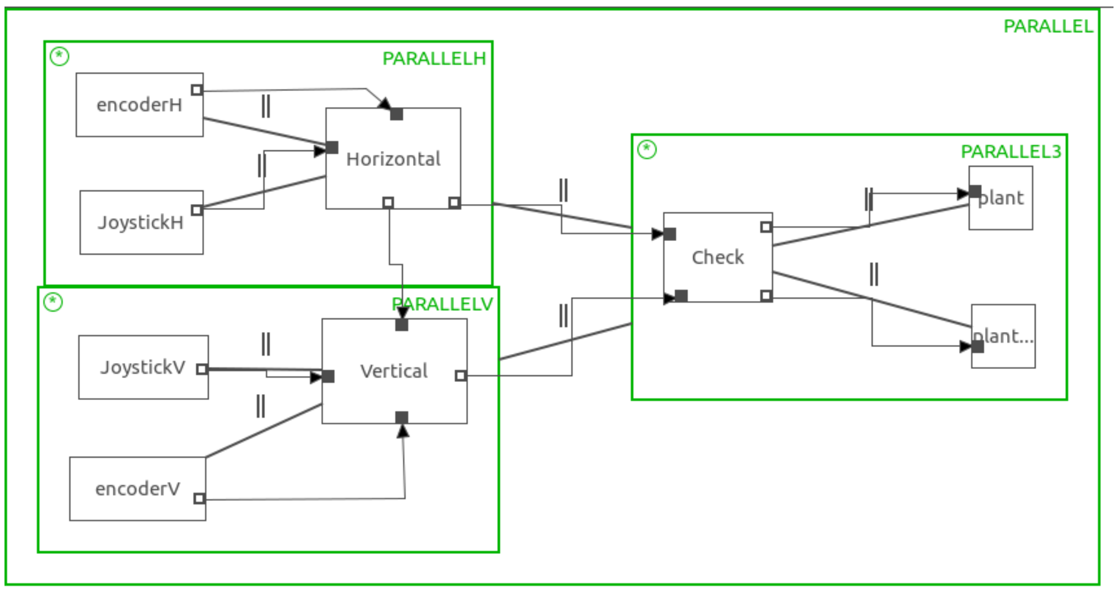


Figure 37.Top level JIWY with dummies

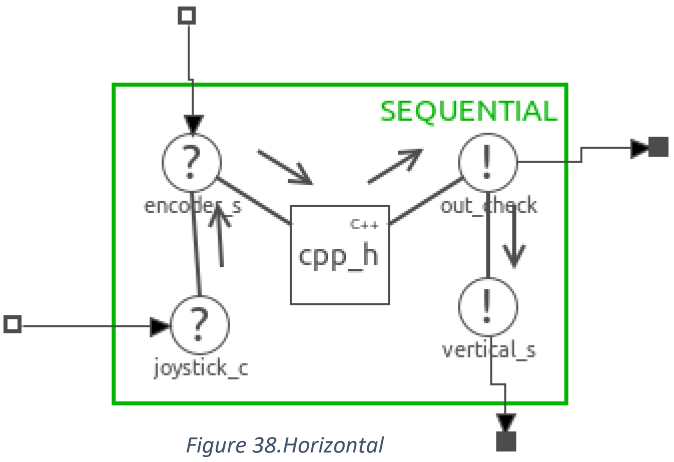


Figure 38.Horizontal

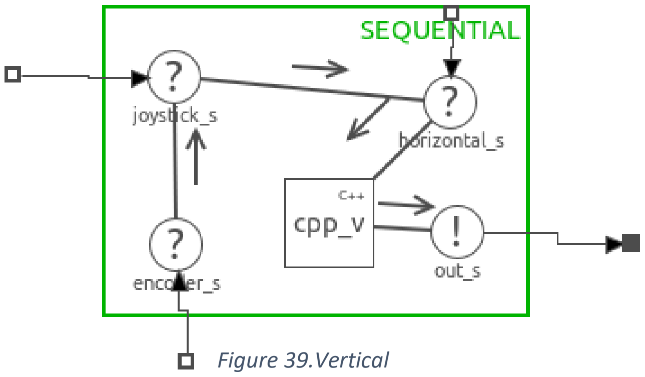


Figure 39.Vertical

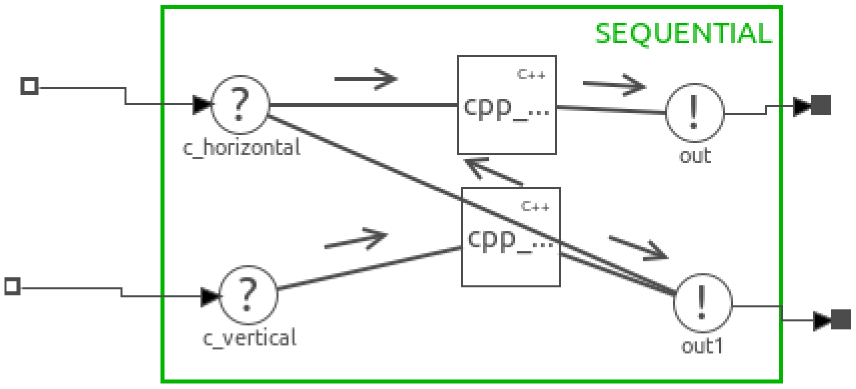
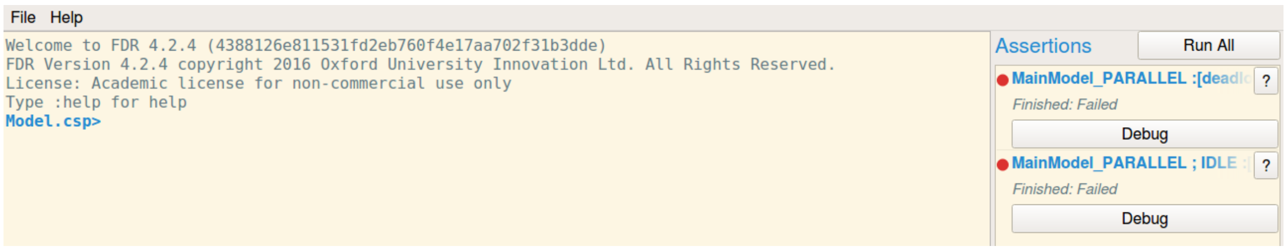


Figure 40.Check DC

Performing FDR test to the above system we see that it fails.



```
Welcome to FDR 4.2.4 (4388126e811531fd2eb760f4e17aa702f31b3dde)
FDR Version 4.2.4 copyright 2016 Oxford University Innovation Ltd. All Rights Reserved.
License: Academic license for non-commercial use only
Type :help for help
Model.csp>
```

```
● MainModel_PARALLEL :[deadl
Finished: Failed
● MainModel_PARALLEL ; IDLE
Finished: Failed
```

Figure 41. The system deadlocks

➢ 2.4.2

By altering the reading sequence in the check process we can make the above system deadlock free. That happens because of the communication dependency between the horizontal and vertical processes. In order for the vertical process to produce an output the horizontal process should have finished first. By having the check process starting from the vertical port we keep all processes in a constant waiting. Horizontal process waits for check process to take its output , check process waits for vertical process to give its output and the vertical process waits for horizontal's communication. This results to a deadlock. By altering the reading sequence the above problem is solved as horizontal is the first process that is finished then vertical continues and finally check makes its computations first with the horizontal control input and then with the vertical.

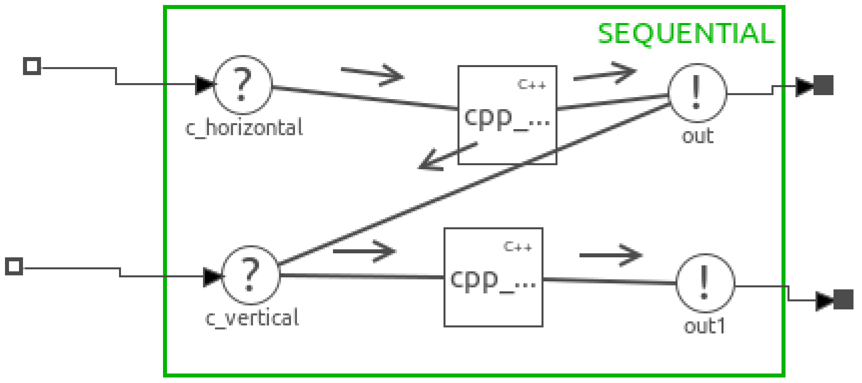
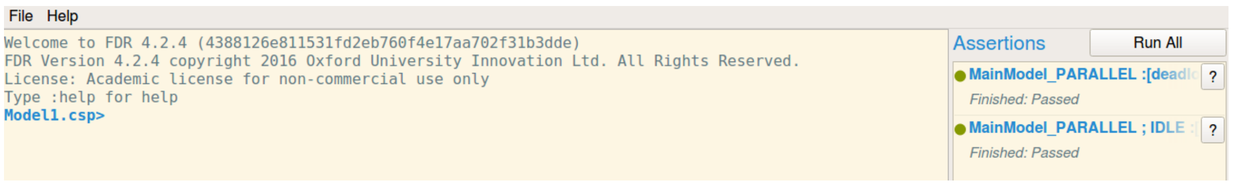


Figure 42.Check DF

Using FDR test we validate that this change produces a correct system.



```
Welcome to FDR 4.2.4 (4388126e811531fd2eb760f4e17aa702f31b3dde)
FDR Version 4.2.4 copyright 2016 Oxford University Innovation Ltd. All Rights Reserved.
License: Academic license for non-commercial use only
Type :help for help
Model1.csp>
```

● MainModel_PARALLEL :[deadl
Finished: Passed
● MainModel_PARALLEL ; IDLE
Finished: Passed

Figure 43.FDR success

➢ 2.4.3

Adding I/O in the models the top level remains the same.(Figure 37)

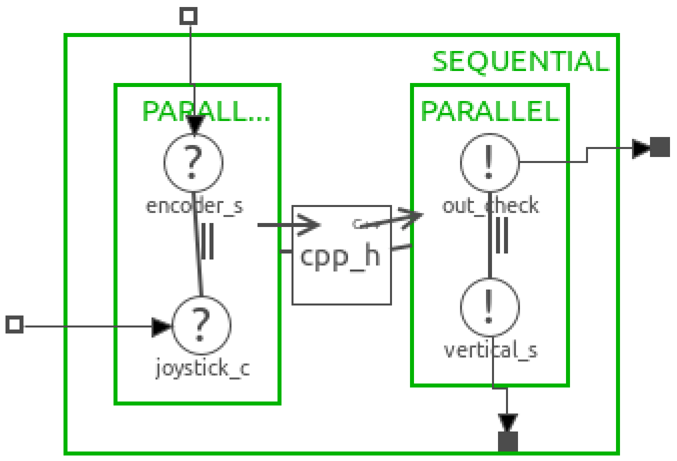


Figure 45.I/O Horizontal

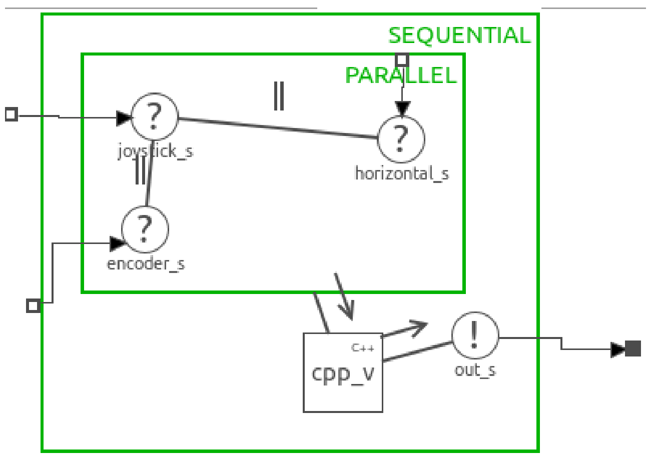


Figure 44.I/O Vertical

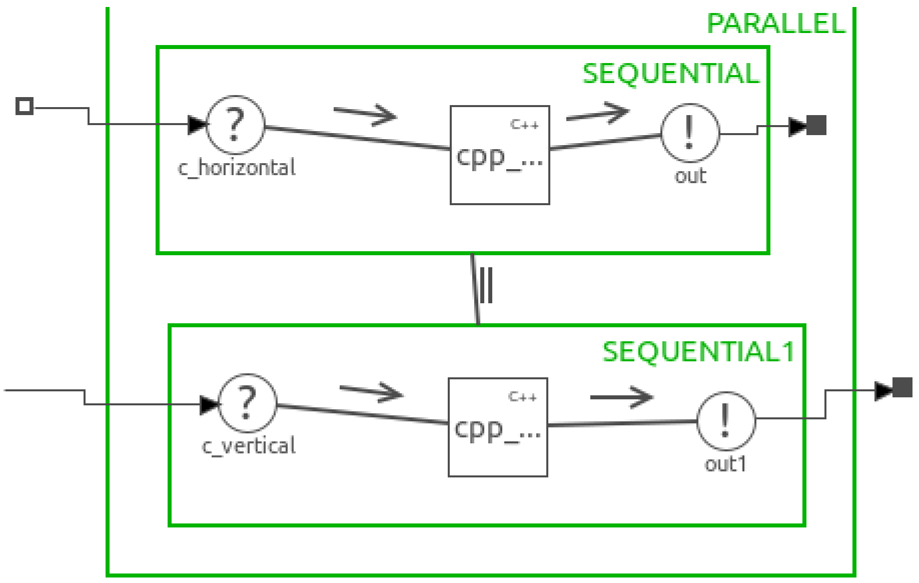
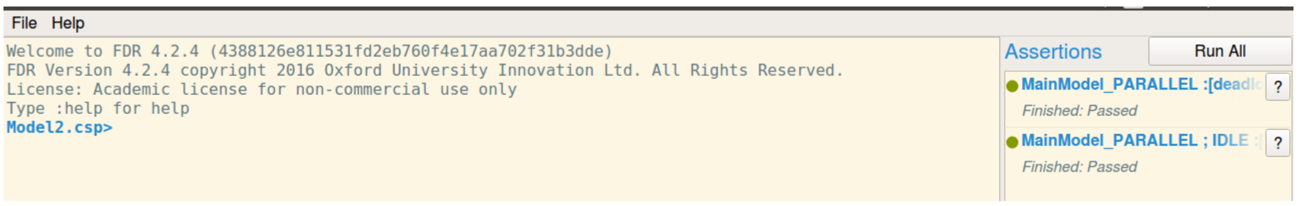


Figure 46. I/O check

Testing the I/O system using FDR we validate that it is deadlock free.



```
Welcome to FDR 4.2.4 (4388126e811531fd2eb760f4e17aa702f31b3dde)
FDR Version 4.2.4 copyright 2016 Oxford University Innovation Ltd. All Rights Reserved.
License: Academic license for non-commercial use only
Type :help for help
Model2.csp>
```

```
● MainModel_PARALLEL :[deadl
  Finished: Passed
● MainModel_PARALLEL ; IDLE
  Finished: Passed
```

Figure 47. I/O FDR success

➢ 2.4.4

The I/O model is a little bit more complex than the first implementation. However, due to the parallelism between the readers and the writers in the I/O design we don't care which input will be taken first ,joystick may send its output first or the encoder. That makes things more structured as we have a clear general view of the fact that input must be taken first , then we make the computation and finally we write the output. On the first implementation we have sequential relation between the processes which can be considered as a more "hard-coded" approach which can easily result to deadlock if we aren't careful.

➢ 2.4.5

In this question we made JIWY a separate process model with the rest processes being dummy models having a correct structure that can be extended with code and create a fully functional model.

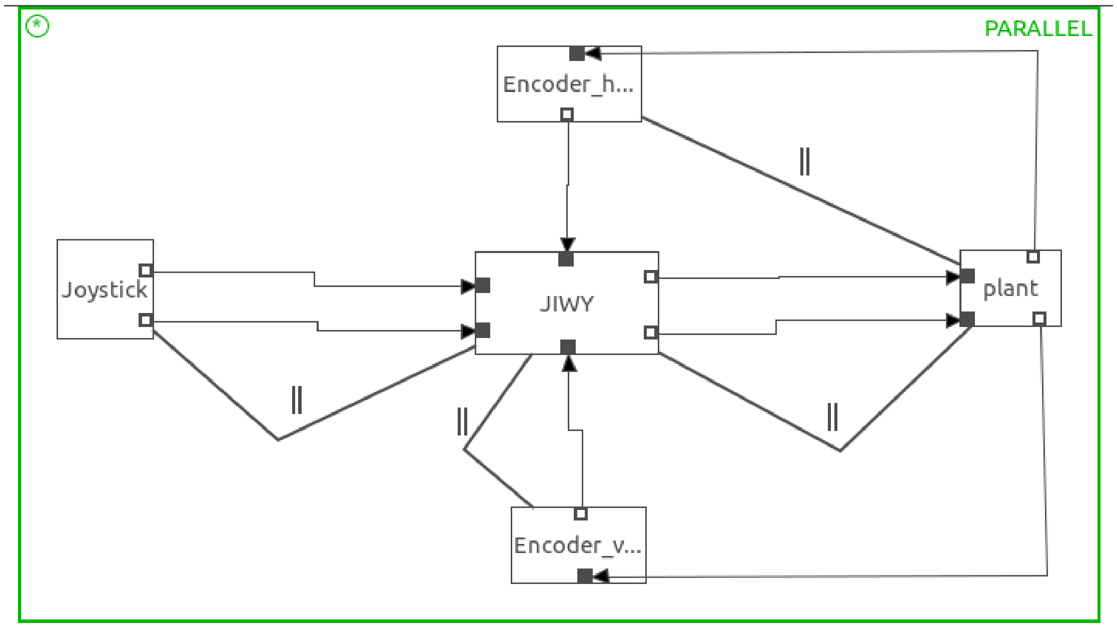


Figure 48.Top level system

At the top level we see that we have a closed loop system with the plant sending its output back to the encoders. This implementation can result to a deadlock if the plant's structure is *read-compute-write.*

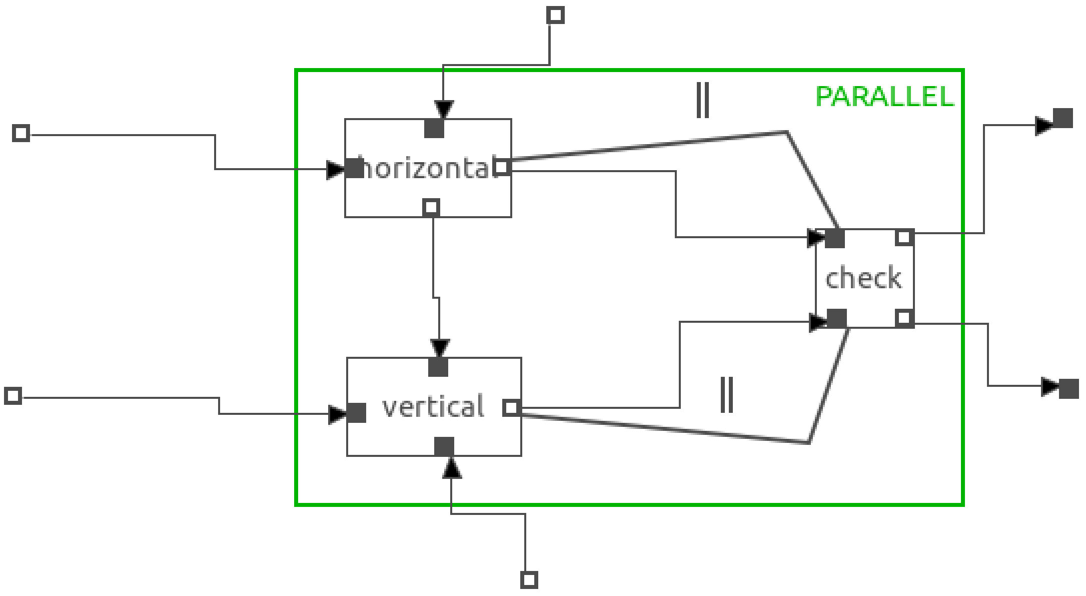


Figure 49.JIWY Controller

The structure of the JIWY is the same with question's 2.4.2 but here we can test the whole system as one.

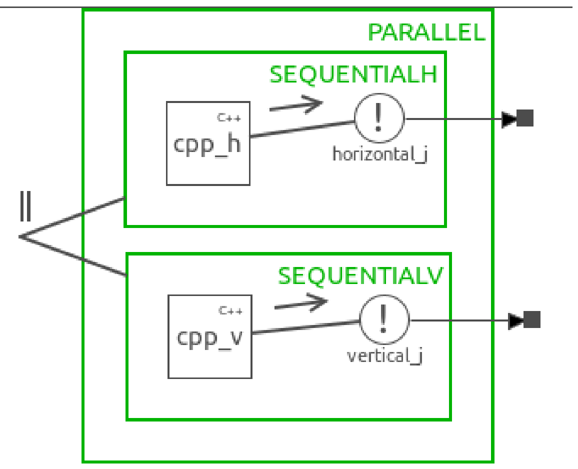


Figure 51.Joystick

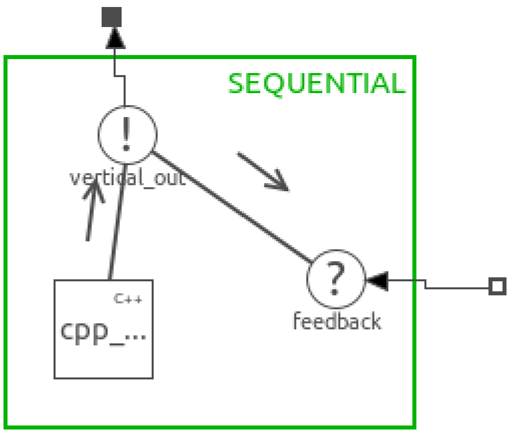


Figure 50.Encoder

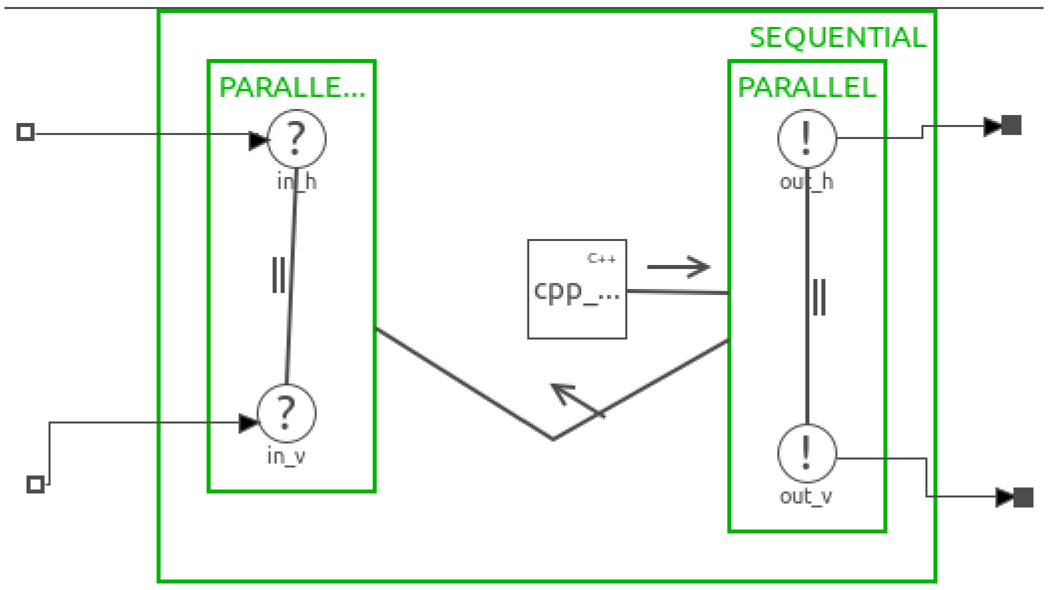
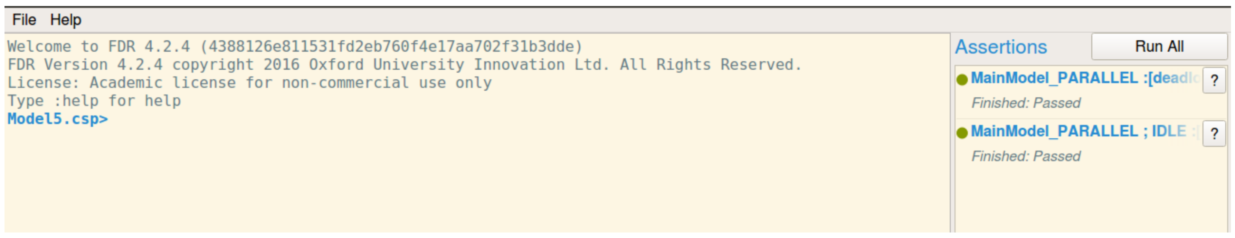


Figure 52. Plant

The composition that was used is the *write-read-compute* in order to avoid deadlock.

Using FDR we validate that the whole system is deadlock free.



```
Welcome to FDR 4.2.4 (4388126e811531fd2eb760f4e17aa702f31b3dde)
FDR Version 4.2.4 copyright 2016 Oxford University Innovation Ltd. All Rights Reserved.
License: Academic license for non-commercial use only
Type :help for help
Model5.csp>
```

● MainModel_PARALLEL :[deadl
Finished: Passed
● MainModel_PARALLEL ; IDLE
Finished: Passed

Figure 53.FDR success

- ## 2.4.6

Using producers and consumers for closing the JIWY ports would work. But if we had a more complex system then we would miss the dependencies that may exist between our under test model and the different sub-models (for instance having a closed loop system). That would create problems in a latter design stage.

- ## 2.4.7

Having only readers and writers for keeping the ports closed and testing the controller would also be a valid method for testing the controller sub-model. However, we see that in our case we have a closed loop system which means that there is a direct relation between controller's and plant's behavior. If we neglect plant's structure and replace it with just two readers and two writers that don't interact with each other(and do the same to the encoders) then most probably we'll miss the fact that this system can potentially deadlock and the time that we'll first try to implement we'll probably have to redesign.