Exercise 1

Rick Veens Studentno: 0912292 Huib Donkers Studentno: 0769015
r.veens@student.tue.nl h.t.donkers@student.tue.nl

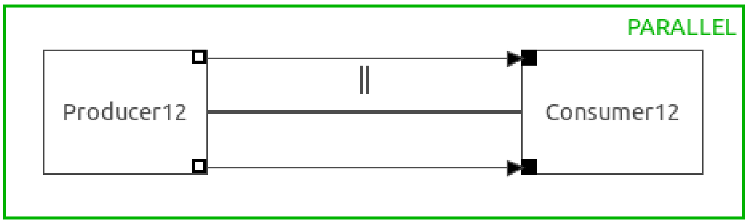
May 30, 2015

1

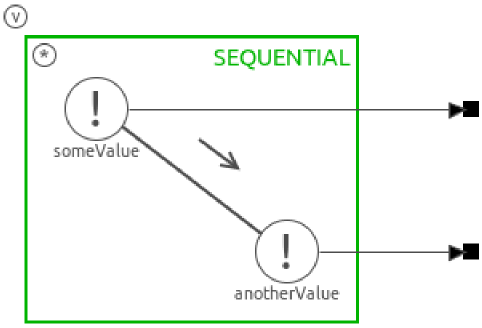
1.1

1.1.1

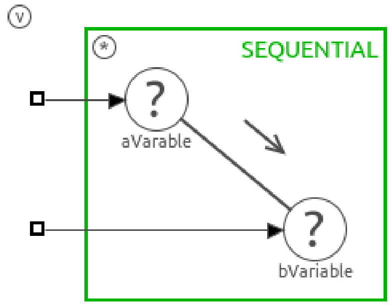
We modelled the deadlock free system as show in figure 1, and the deadlocked system as shown in figure 3. As expected, FDR accepts the deadlock free system (firgure 2), but not the deadlocked system (figure 4).



(a) Composition of the Producer and Consumer processes.



(b) The producer process.



(c) The consumer process.

Figure 1: The producer-consumer (DF) model.

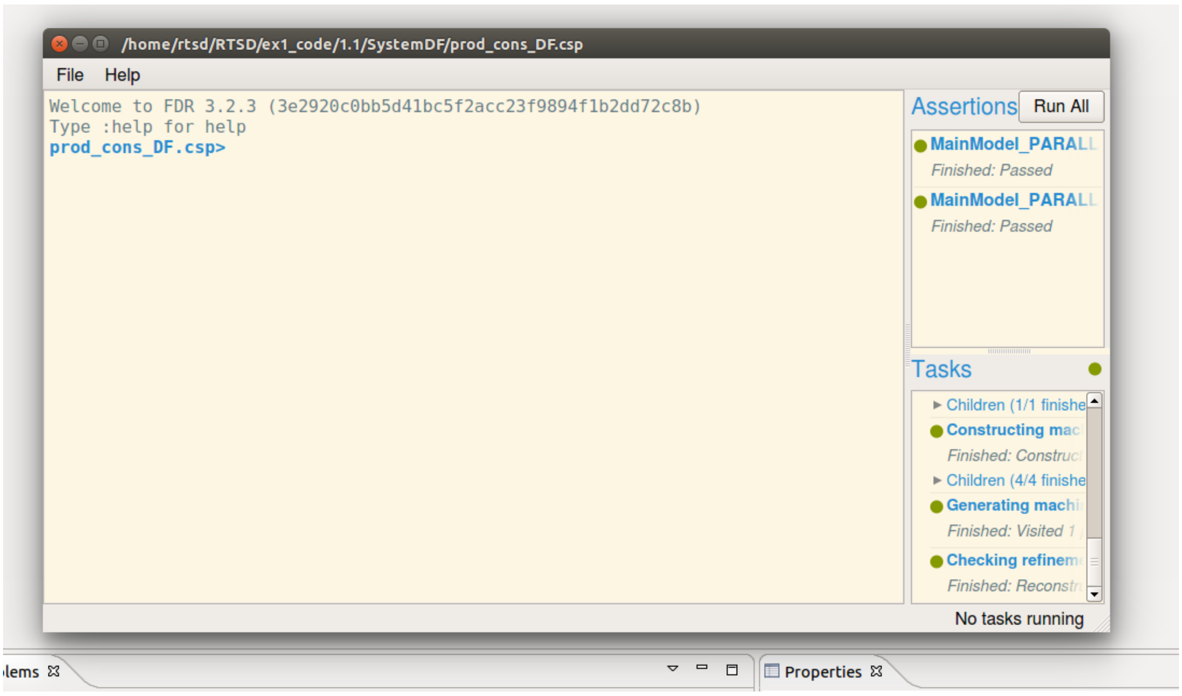
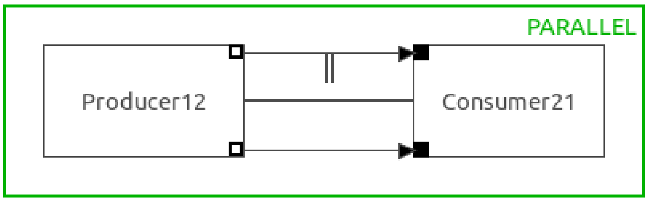
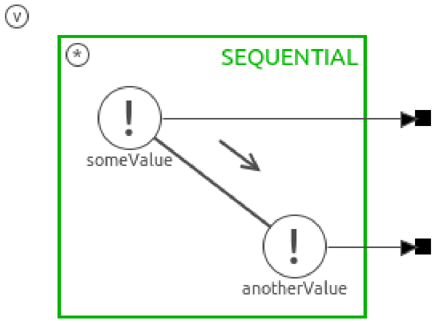


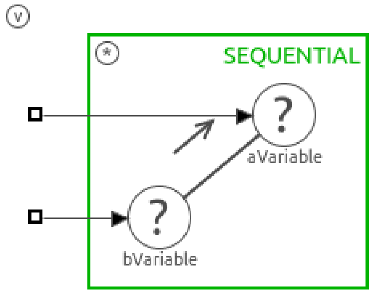
Figure 2: Successfully passes tests in FDR.



(a) Composition of Producer and Consumer processes.



(b) The producer process.



(c) The consumer process.

Figure 3: The producer-consumer (DC) model.

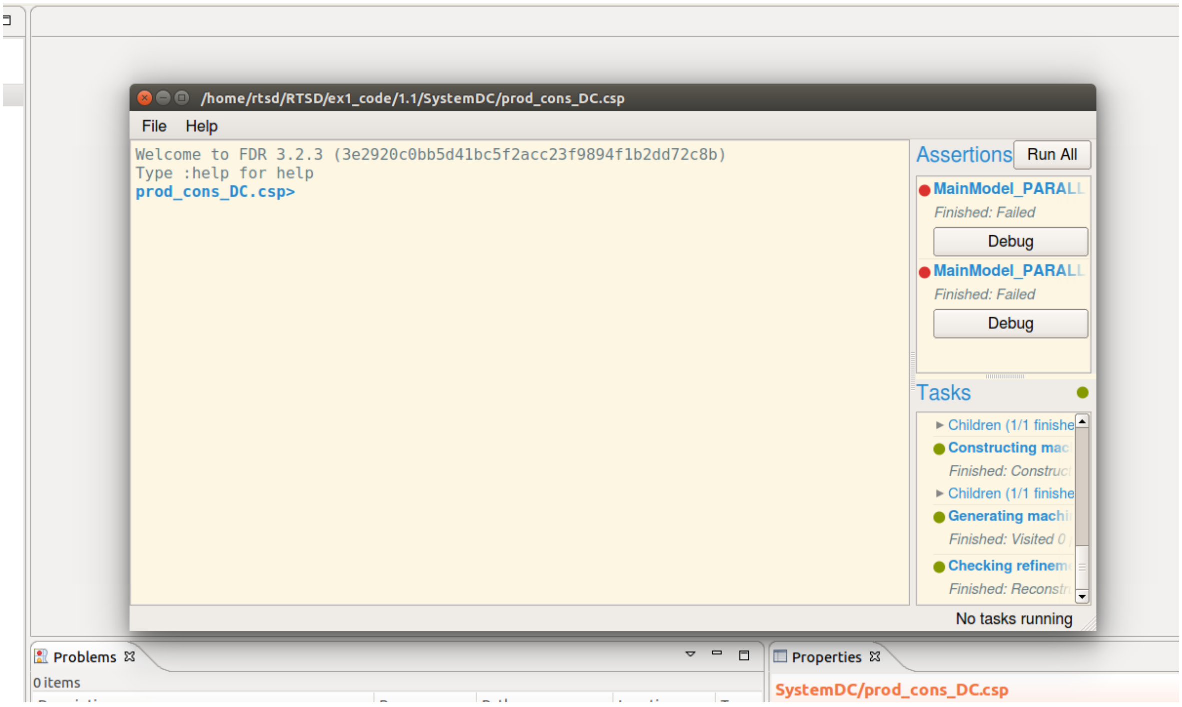


Figure 4: Fails the tests in FDR.

1.1.2

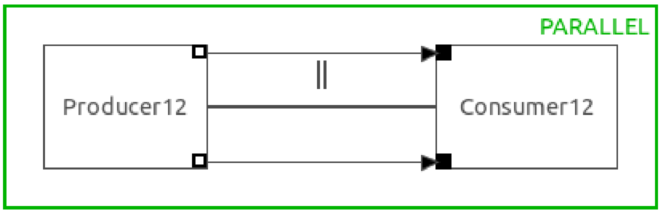
We extended the producer consumer (DF) model from figure 1 with code blocks, and added `C++` implementation. The extended model is shown in figure 5 and the code in included in blocks 1 and 2. The initial few lines of the output is shown in figure 6.

Codeblock 1: Consumer12/CCode.cpp

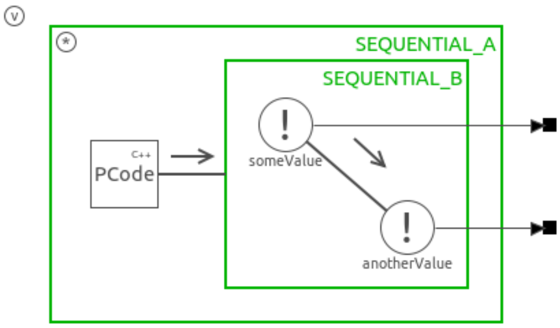
```cpp
/**
 * Source file for the CCode model
 * Generated by the TERRA CSPm2LUNA generator version 1.1.1
 *
 * protected region document description on begin
 *
 * protected region document description end
 */

#include "Consumer12/CCode.h"
// protected region additional headers on begin
// Each additional header should get a corresponding dependency in the Makefile
// protected region additional headers end

namespace MainModel { namespace Consumer12 { namespace CCode {

CCode::CCode(int &CCode_aVariable, int &CCode_bVariable) :
    CodeBlock(), CCode_aVariable(CCode_aVariable), CCode_bVariable(CCode_bVariable){
  SETNAME(this, "CCode");

  // protected region constructor on begin
  // protected region constructor end
}

CCode::~CCode()
{
  // protected region destructor on begin
  // protected region destructor end
}

void CCode::execute()
{
  // protected region execute code on begin
    if (this->CCode_aVariable == -1 || this->CCode_bVariable == -1)
        exit();
    else {
        printf("Receiving: CCode_aVariable: \t'%c'\n", this->CCode_aVariable);
        printf("Receiving: CCode_bVariable: \t'%c'\n", this->CCode_bVariable);

        printf("\n");
    }
  // protected region execute code end
}

// protected region additional functions on begin
// protected region additional functions end

// Close namespace(s)
} } }
```

Codeblock 2: Producer12/PCode.cpp

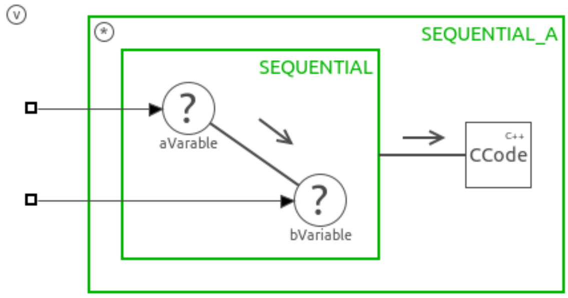
```cpp
/**
 * Source file for the PCode model
```



(a) Overview diagram of the producer consumer system.



(b) The producer process.



(c) The consumer process.

Figure 5: The producer-consumer (DF) model extended with code.

```
 * Generated by the TERRA CSPm2LUNA generator version 1.1.1
 *
 * protected region document description on begin
 *
 * protected region document description end
 */

#include "Producer12/PCode.h"
#include "string.h"
// protected region additional headers on begin
// Each additional header should get a corresponding dependency in the Makefile
// protected region additional headers end

namespace MainModel { namespace Producer12 { namespace PCode {

PCode::PCode(int &PCode_anotherValue, int &PCode_someValue) :
    CodeBlock(), PCode_anotherValue(PCode_anotherValue), PCode_someValue(↩
        PCode_someValue){
  SETNAME(this, "PCode");

  // protected region constructor on begin
  // protected region constructor end
}

PCode::~PCode()
{
  // protected region destructor on begin
  // protected region destructor end
}

void PCode::execute()
{
  // protected region execute code on begin
```

```
    static int index = 0;

    char *stuff = "Appelflap";

    if (index == strlen(stuff)) {
        this->PCode_someValue = -1;
        this->PCode_anotherValue = -1;
    } else {
        this->PCode_someValue = stuff[index];
        this->PCode_anotherValue = stuff[index++];

        printf("Sending: PCode_someValue: \t'%c'\n", this->PCode_someValue);
        printf("Sending: PCode_anotherValue: \t'%c'\n", this->PCode_anotherValue);
    }


  // protected region execute code end
}

// protected region additional functions on begin
// protected region additional functions end

// Close namespace(s)
} } }
```

```
rtsd@rtsd-ubuntu-64bit: ~/RTSD/ex1_code/1.2/prod_cons_DF
g++ -L/opt/LUNA/target-x86_64-linux_x86_64_CSP/lib64 obj/main.o obj/MainModel.o
obj/Producer12.o obj/Producer12/PCode.o obj/Consumer12.o obj/Consumer12/CCode.o
-lLUNA -lpthread -o prod_cons_DF
rtsd@rtsd-ubuntu-64bit:~/RTSD/ex1_code/1.2/prod_cons_DF$ ./prod_cons_DF
Sending: PCode_someValue:       'A'
Sending: PCode_anotherValue:    'A'
Sending: PCode_someValue:       'p'
Sending: PCode_anotherValue:    'p'
Receiving: CCode_aVariable:     'A'
Receiving: CCode_bVariable:     'A'

Sending: PCode_someValue:       'p'
Sending: PCode_anotherValue:    'p'
Receiving: CCode_aVariable:     'p'
Receiving: CCode_bVariable:     'p'

Sending: PCode_someValue:       'e'
Sending: PCode_anotherValue:    'e'
Receiving: CCode_aVariable:     'p'
Receiving: CCode_bVariable:     'p'

Sending: PCode_someValue:       'l'
Sending: PCode_anotherValue:    'l'
Receiving: CCode_aVariable:     'e'
Receiving: CCode_bVariable:     'e'

Sending: PCode_someValue:       'f'
Sending: PCode_anotherValue:    'f'
Receiving: CCode_aVariable:     'l'
Receiving: CCode_bVariable:     'l'

Sending: PCode_someValue:       'l'
Sending: PCode_anotherValue:    'l'
Receiving: CCode_aVariable:     'f'
Receiving: CCode_bVariable:     'f'

Sending: PCode_someValue:       'a'
Sending: PCode_anotherValue:    'a'
Receiving: CCode_aVariable:     'l'
Receiving: CCode_bVariable:     'l'

Sending: PCode_someValue:       'p'
Sending: PCode_anotherValue:    'p'
Receiving: CCode_aVariable:     'a'
Receiving: CCode_bVariable:     'a'

Receiving: CCode_aVariable:     'p'
Receiving: CCode_bVariable:     'p'
```

Figure 6: The output that our system produces.

1.1.3

The consumer process cannot start before something is written on the channels. The producer process can start with the execution of its `C++` code. The parallel composition of the two processes can therefor only start with the execution of the `C++` code of the producer, hence the first two lines of the output originate from the producer.
Now, for the first channel, both the reader and the writer are available, and no other actions can be done by either process. The data is sent over the channel. After that, the same goes for the second channel.
Next, the producer can start over, starting with the execution of the `C++` code, but now the consumer can also execute its `C++` code. We expect either two lines of output produced by the producer's `C++` code, or two lines produced by the consumers `C++` code. And indeed we see the output of the producer's `C++` code.
Next, the writers in the producer cannot write to the channels, because the readers of the consumer are not ready. First the consumer's `C++` code needs to be executed before the process can recur and execute its readers. Hence the next two lines are produced by the consumer.
Now, the readers and writers do their thing and, again, either the producer's `C++` code can be executed, or the consumer's. We see for the complete initial part of the run that the producer's `C++` code is executed before the consumer's `C++` code.

1.1.4

ProBE is an interpreter for CSPm scripts, the user can manually inspect the possible execution paths. When a process is deadlock free, its execution paths are infinite. ProBE will never see the end of these executionpaths, nor will it know whether it has an end. FDR recognises this infinite behaviour, or rather, it recognises finite behaviour. When it finds the process is able to terminate, it shows that the process fails the deadlock free-test. Similarly it can check for other properties without needing to simulate the complete execution paths likes ProBE does. ProBE can be used to inspect how a process deadlocks.

1.1.5

If we adapt the `C++` code of the producer to start at the beginning of the word again when all characters have been sent, we can observe the behaviour of the process when it runs for a while. We saw previously that the consumer only continued when the producer had to wait. This does not seem fair parallel. But when we watch what happens when the code runs for a while, we occasionally see the consumer receiving four values in a row, meaning that its `C++` code executed before the producer's `C++` code could prepare the next two values for the writers. So when both processes can execute, the parallel composition does not consistently execute one of them before the other. This seems fair.

1.1.6

The composition $A|||B$ does not deadlock. Whichever process gets to execute its `C++` code first, it will need to wait for the other process directly after. When both have executed their `C++` code, execution can continue by writing and reading on the channels. At no point do the processes wait for the other.

1.2

1.2.1

See Figure 7 for some screenshots of the model. We chose to implement three channels.

1.2.2

See Codeblock 3 for the code of `C++` code block PCode. The other code blocks in the program are of no interest, because they merely print their input variable.

Figure 9 shows some of the output of the program after extending it with `C++` code.
The decision was made to have the program send each of the three values once. After sending the prompt appears again.
The guards on the producer side (see Figure 7b) are configured in the model in such a way that a value '1' gets send to channel 1, value '2' to channel 2 and value '3' to channel 3. Any other value that is entered will result in the "Prullenbak" `C++` code being executed.
Note that some of the messages are out of order, the same stuff is happening as discussed in exercise 1.1.3.
In short, order we see occurs because the top producer and consumer process are linked in parallel. The communication of the consumer/producer processes are synchronized, but not the code being executed in their internal `C++` code blocks.

1.2.3

Note that in Figure 7b a `C++` codeblock is added called "Prullenbak". This "Prullenbak" `C++` code block is executed only when none of the other guard expressions in the ALTERNATIVE block apply. The function of "Prullenbak" is to catch and print input data that is not to be send to the consumer, it is a garbage can.

1.2.4

It is wise to start with the process-structure before implementing `C++` code.
Generally, if you have to choose, you want most of the functionality in the model instead of in `C++` code blocks. The model also serves as documentation. Making sure the model is of decent quality first helps in containing most of the functionality in the model instead of the `C++` code.

1.2.5

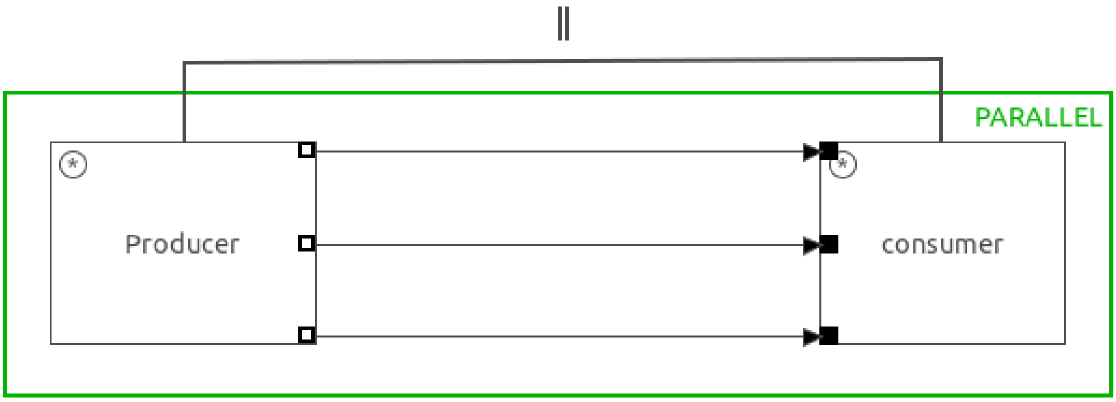
The method we chose to implement is to add another guard that is triggered when none of the other guards are triggered. It is a garbage bin exit.

Another possible method is to copy the guard logic of the alternative blocks in the producer to the "PCode" `C++` code block, and do a check on the input data to make sure it it checks out with your guard logic. This idea is less desired, because now you have the guard logic in *two* places: in the model and in the `C++` code.
You could also remove the alternative structure, have only one instead of three channels, and put all your guard logic in the first `C++` code block (PCode). This idea makes the program harder to understand, so not recommended.

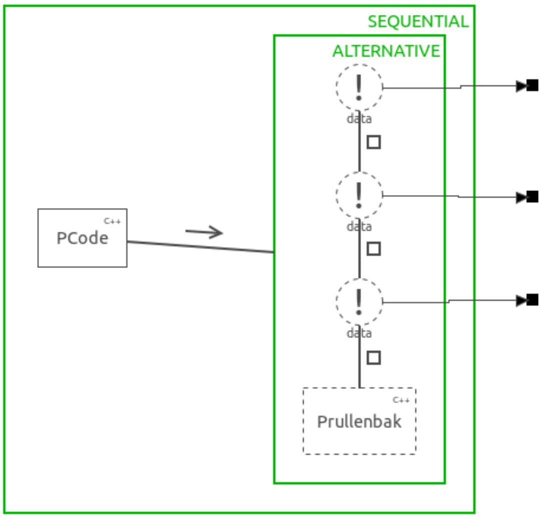
Codeblock 3: Producer/PCode.cpp

```
/**
 * Source file for the PCode model
 * Generated by the TERRA CSPm2LUNA generator version 1.1.1
 *
 * protected region document description on begin
 *
 * protected region document description end
 */

#include "Producer/PCode.h"
// protected region additional headers on begin
#include <stdio.h>
#include <iostream>
#include <algorithm>
// Each additional header should get a corresponding dependency in the Makefile
// protected region additional headers end

namespace MainModel { namespace Producer { namespace PCode {

PCode::PCode(int &peer) :
    CodeBlock(), peer(peer){
  SETNAME(this, "PCode");

  // protected region constructor on begin
  // protected region constructor end
}

PCode::~PCode()
{
  // protected region destructor on begin
  // protected region destructor end
}

void PCode::execute()
{
  // protected region execute code on begin
    static int index = 3;

    if (index > 2) {
        index = 0;
        while (!initialize()) {
            printf("Numbers not distinct.\n");
        }
    }
    printf("P1->Sending %d\n", vars[index]);
    this->peer = vars[index++];
  // protected region execute code end
}

// protected region additional functions on begin
bool PCode::initialize(void)
{
    printf("Enter 3 numbers\n");
    while (scanf("%i", &vars[0]) != 1) {}
    while (scanf("%i", &vars[1]) != 1) {}
    while (scanf("%i", &vars[2]) != 1) {}

    std::sort(vars, vars+3);

    printf("I got: %i, %i, %i\n", vars[0], vars[1], vars[2]);
    if (vars[0] == vars[1] || vars[0] == vars[2] || vars[1] == vars[2])
        return false;
    else
        return true;
}
```

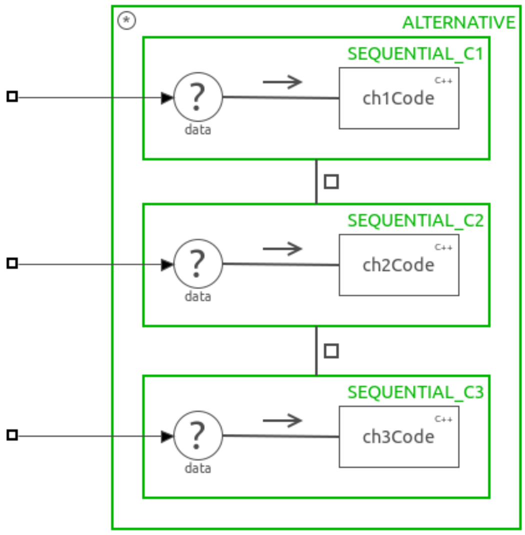
```
// protected region additional functions end

// Close namespace(s)
} } }
```



(a) Overview diagram of the multiple channel producer consumer system.



(b) The producer process.



(c) The consumer process.

Figure 7: The CSP model of exercise 1.2.

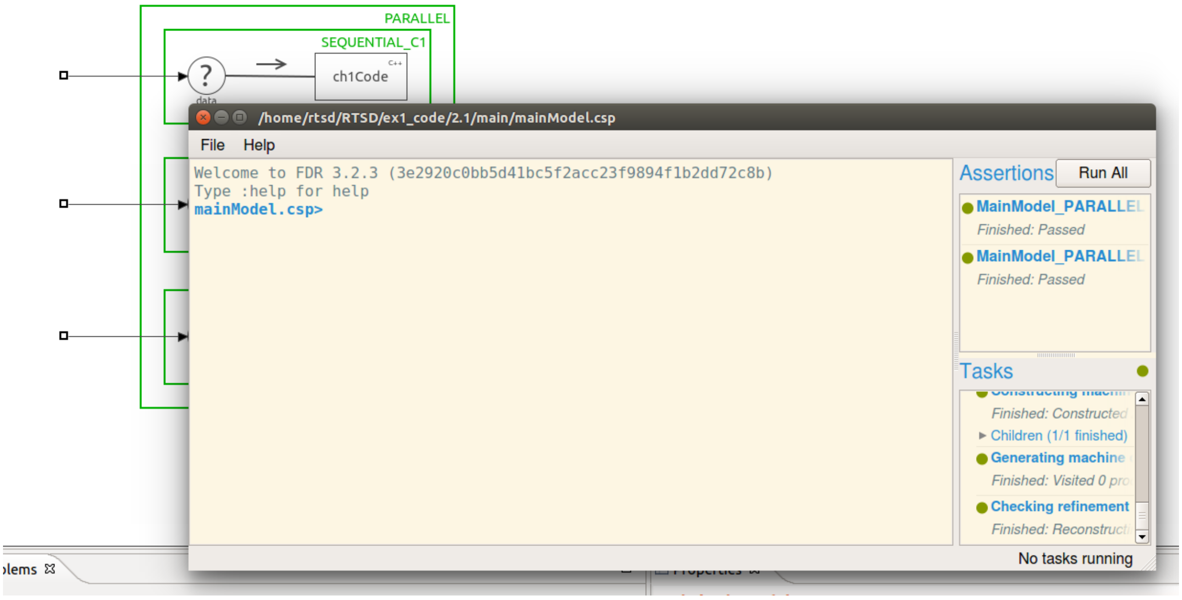


Figure 8: Deadlock free.

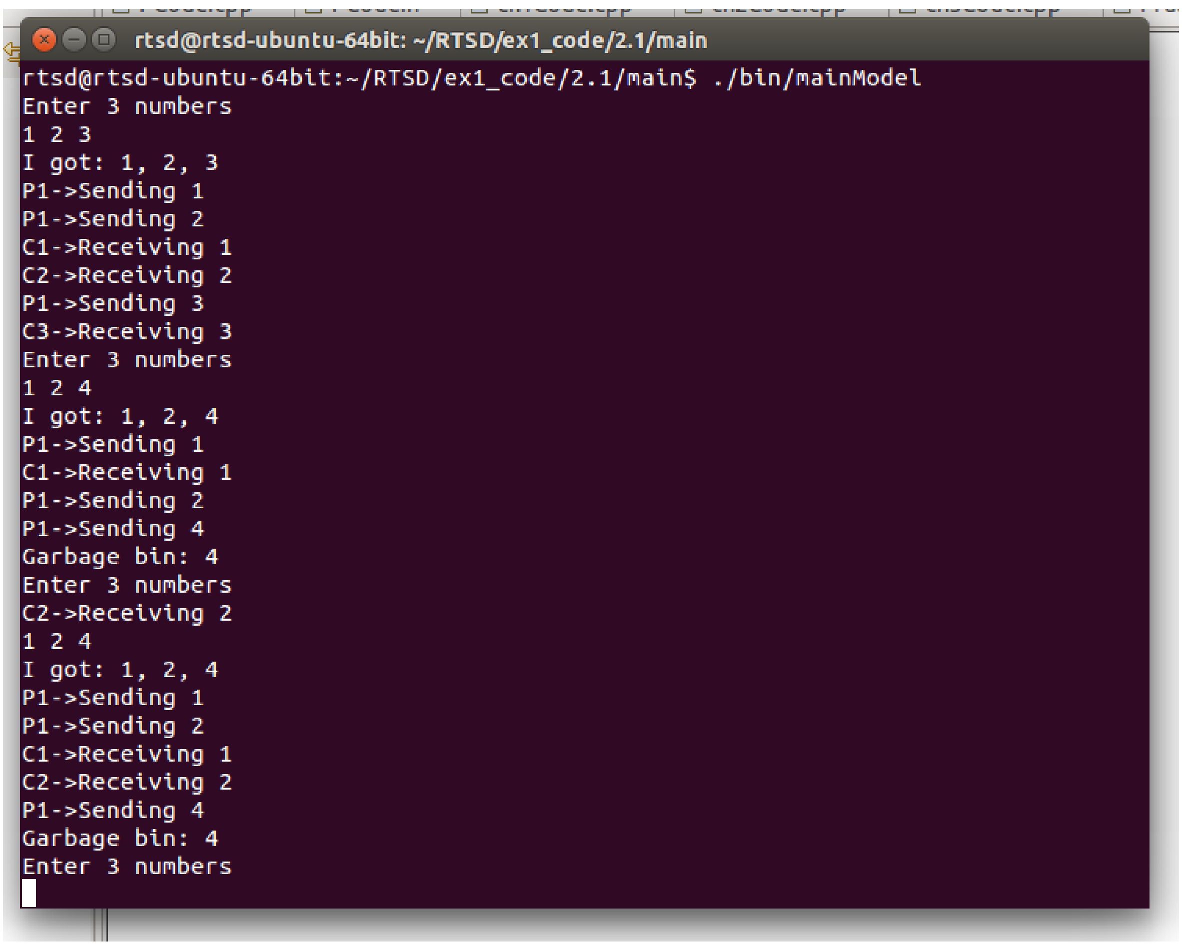


Figure 9: Output of the program exercise 1.2.

1.3

1.3.1

- We simulated the execution of the model in 20-sim, results shown in figure 11. As expected, after *stepTime* seconds, the input increases, and directly after x increases with the value of the input u. After that we see x increase again, but this time the increment is dampened by x having a non-zero value. As x increases, the increment decreases, until $u - 0.3x = 0 \Leftrightarrow x = \frac{10}{3}$. This equilibrium will, in theory, never be reached. In Figure 11 we see x reach a point between 3 and 3.33....
- We recreated the 20-sim model in TERRA just to check for deadlocks. See Figure 12 for the models, and Figure 13 for a screenshot of FDR that proofs there are no deadlocks.
- We implemented the 20sim project into TERRA by using the 20sim LUNA template, editing the xml and importing the 20sim models into the TERRA project after updating. See Figure 14 for the models of the TERRA project that includes `C++` code.
- Figure 15 shows the output of the control loop program and Figure 11 the output of the simulation.

  Note that the green line in the simulation (1.0) corresponds to the output of the setpoint value, and node that the blue line in the simulation (3.3) corresponds to the output of the steering value.

  The result of the program is the same as the simulation.

The configuration of the linear system is as follows:

$$x_n = 1 \cdot x + 1 \cdot u$$
$$y = 1 \cdot x + 0 \cdot u$$

where we use u to for dx, calculated in the controller (see figure 10d)

1.3.2

Figure 16 depicts the model after implementing the new display process and Figure 17 shows the output after running the TERRA `C++` program. Furthermore, Figure 18 proofs that the program does not deadlock.
We chose to have the display process to print only the steering value from the Controller process and the setpoint value from the Step process.

1.3.3

The e-mail of 28th of May stated that this exercise need not be implemented in TERRA.

1.3.4

The e-mail of 28th of May stated that this exercise is not applicable in the current form, because exercise 1.3.3 need not be implemented.

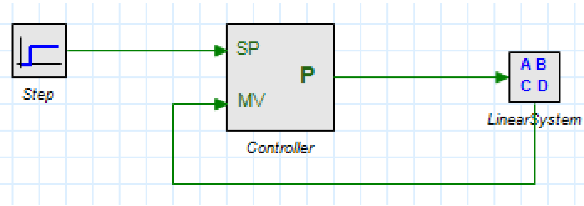
1.3.5

It is quite hard to truly test hard real-time constrains on a system that is not real time: the desktop PC that we use for executing TERRA programs is not real-time.
How do we know that hard-real-time constraints can always be met, and that the soft-real-time and hard-real-time parts are always correctly executed, if the system itself is not real time?

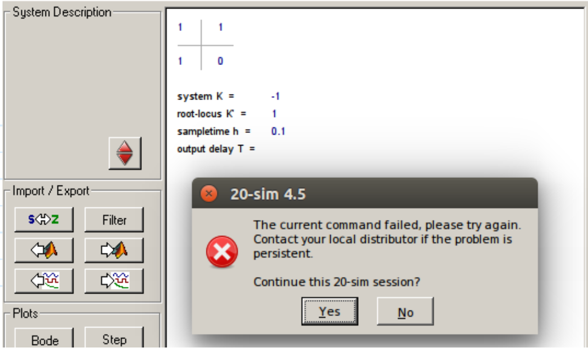
1.3.6

Yes, the approach is still a very "sophisticated" way of working. There are still several annoyances when using imported 20sim models in TERRA, these might make the development of a large system very much more complex.
To name a few:

- Imported 20sim models disable the use of CSP deadlock checking in FDR.
- Adding code to the `C++` files that are added by generated by imported 20sim models cannot be saved (no protected areas).
- External models (imported 20sim models). Have to be generated separately.



(a) Model in 20-sim.



(b) Configuration of the linear system. Could not be fully shown because of a persistent error that crashes 20-sim.

```
parameters
   real amplitude = 1;
   real start_time = 2.0 {s};
variables
   boolean hidden change;
equations
   "calculate at least at the start time"
   change = timeevent (start_time);

   "calculate the step signal"
   output = amplitude * step (start_time);
```

(c) The definition of the step function.

```
// Created by C. Kleijn, 22 Sep 1999
parameters
   real K = 1.0 {};         // Proportional gain
   real a = -0.3 {};
   real b = 1.0 {};
variables
   real dx;
equations
   dx = a*MV + b*SP;
   output = K * dx;
```

(d) The definition of the loop controller.

Figure 10: The model in 20 sim.

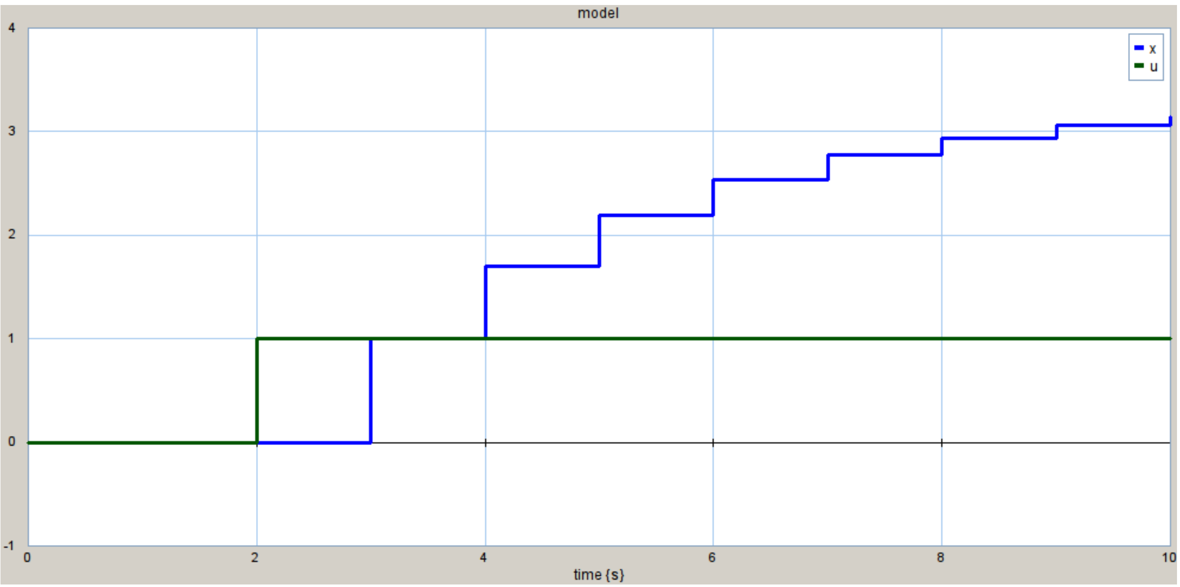
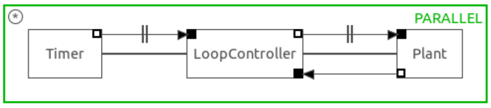
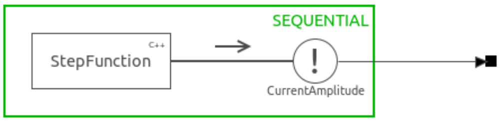


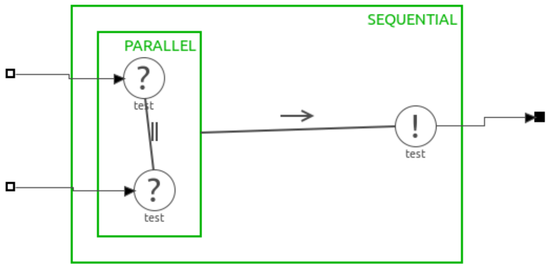
Figure 11: Simulation in 20-sim of the model shown in figure 10. Blue line: x, green line: u.



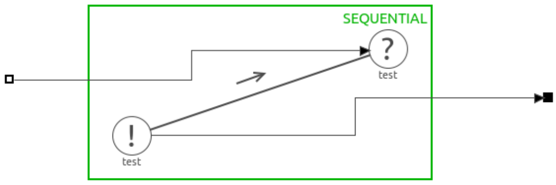
(a) Control model in TERRA.



(b) Step controller (Timer)



(c) Loopcontroller.



(d) LinearSystem (Plant)

Figure 12: Recreating the 20sim model to check for deadlocks.

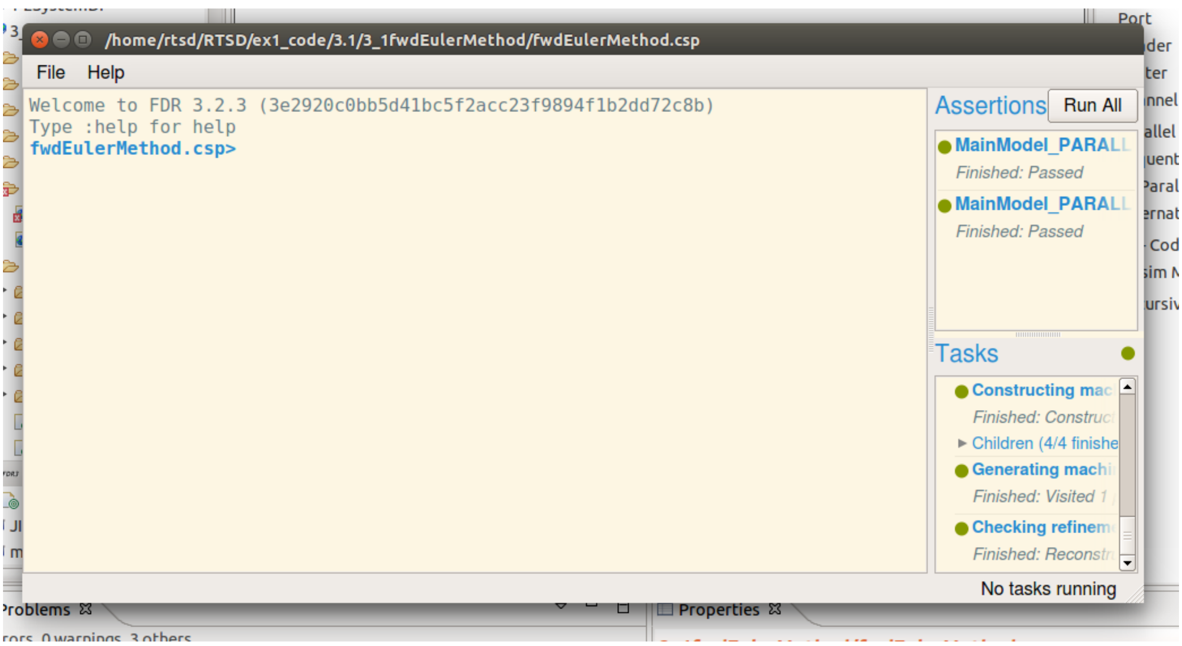
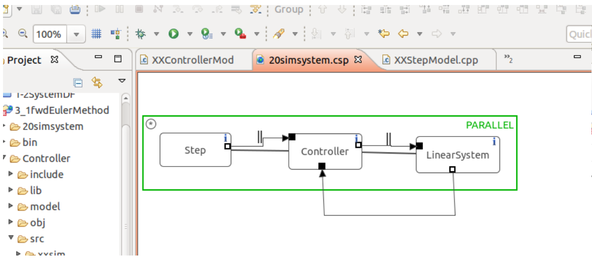
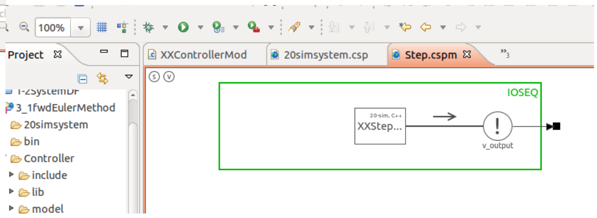


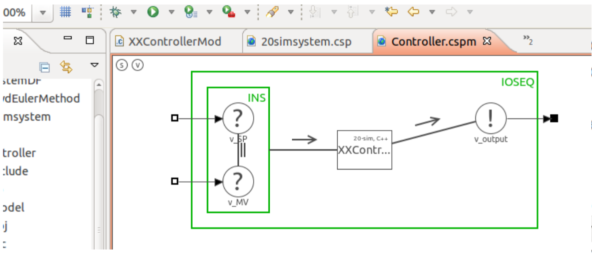
Figure 13: No deadlocks!



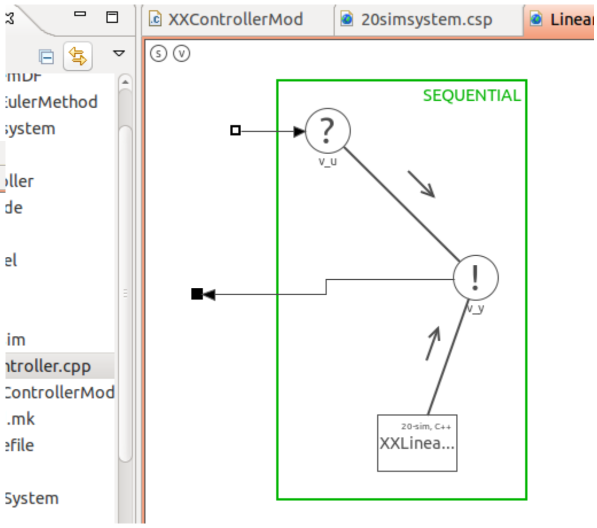
(a) Control model in TERRA.



(b) Step controller (Step)



(c) Loopcontroller (Controller).



(d) Plant (LinearSystem)

Figure 14: Implementation of the control loop model with C++ code from 20sim.

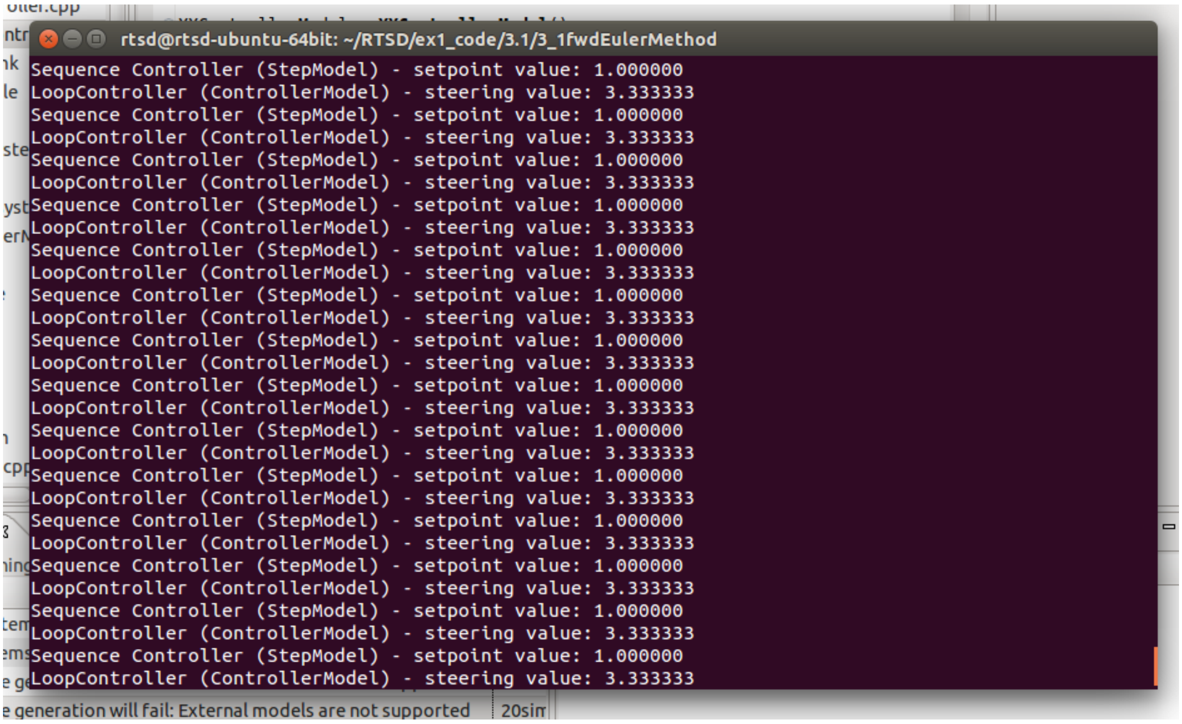
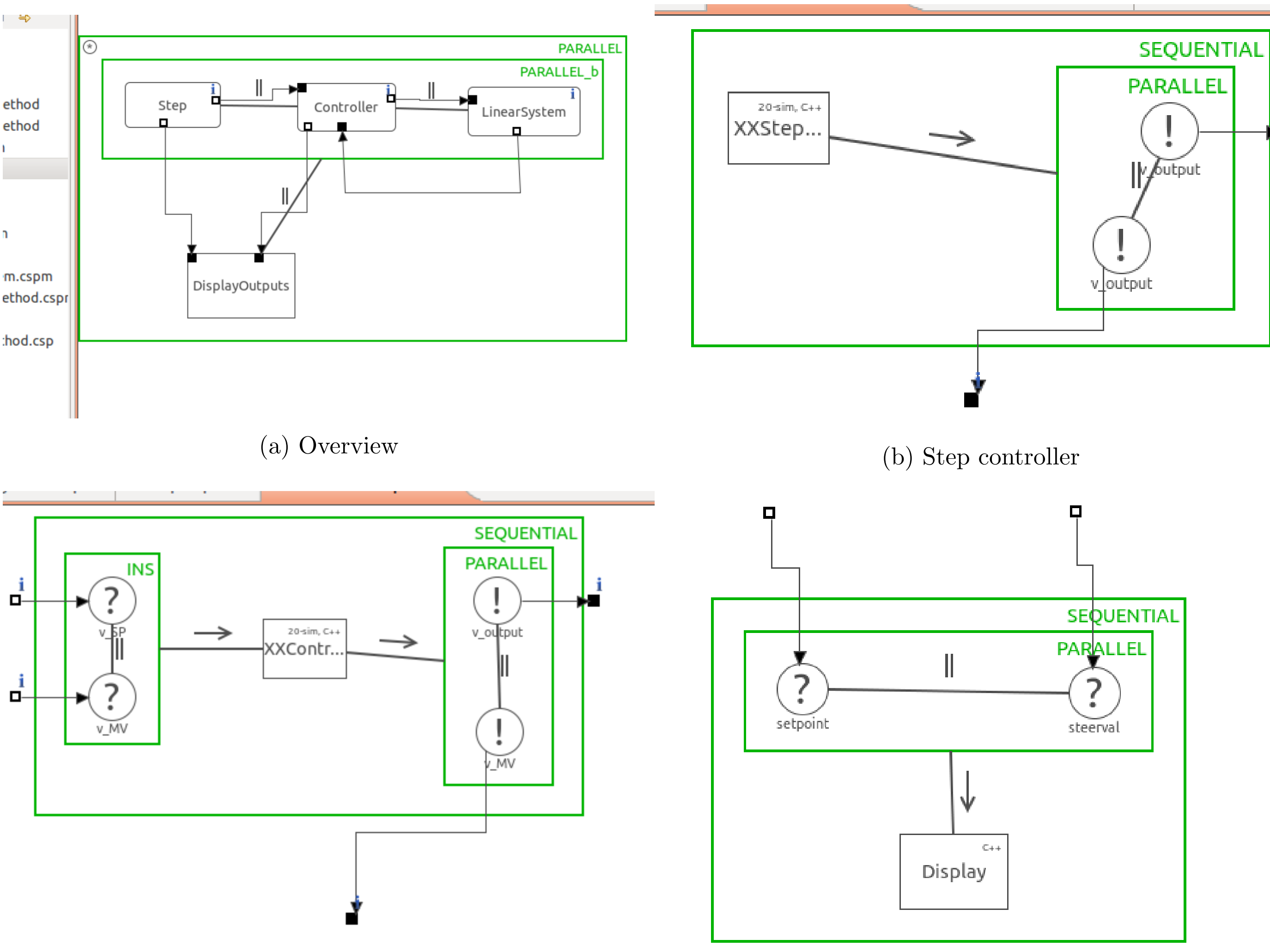


Figure 15: Output of the control loop model implemented in TERRA.



(a) Overview

(b) Step controller

(c) Inside Controller

(d) Inside the DisplayOutputs process.

Figure 16: Separate Display process for printing values.

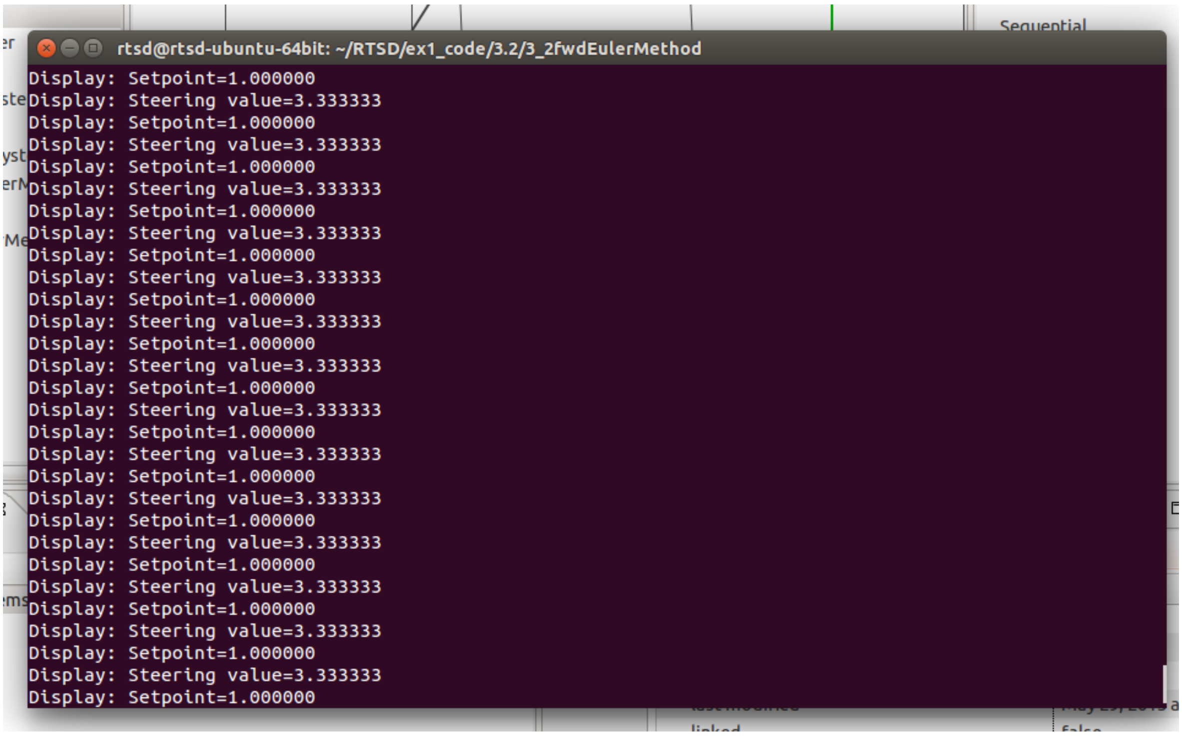


Figure 17: Output of the separate display process (1.3.2)

Figure 18: After adding the display process we do not get any deadlocks.

1.4

1.4.1

Figure 19 presents the JIWY controller model as presented in the slides, with the addition of extra readers and writers that: simulate the input of the joystick, the output to the robot and the feedback from the robot.
This is added because otherwise it was not possible to generate CSPm: one cannot leave input and output ports unconnected in the top model.
Figure 20 informs us that the model results in a deadlock in FDR.

1.4.2

Figure 21a shows the small fix to remove the deadlock and this is proven with a screenshot of FDR as seen in Figure 21b.
In the deadlock situation, observe the following:

- The Check process must read from the Vertical process first, before it can read from the Horizontal process.
- The Vertical process must read from the Horizontal process before it can write to the Check process.
- The Horizontal process must first write to the Check process before it can write to the Vertical process.

This results in the deadlock, we have a loop.

Making sure that the Check process has to read first from the Horizontal process instead of the Vertical process will fix the problem.
If the Check process first reads from the Horizontal process and then to the Vertical process; the Horizontal Process is first able to write to the Check process and then to the Vertical process (which is waiting on Horizontal), and this in turn allows the Vertical process to write to the Check process.
The Check process is able to receive the write from the Vertical process because it first reads from Horizontal and then from Vertical.

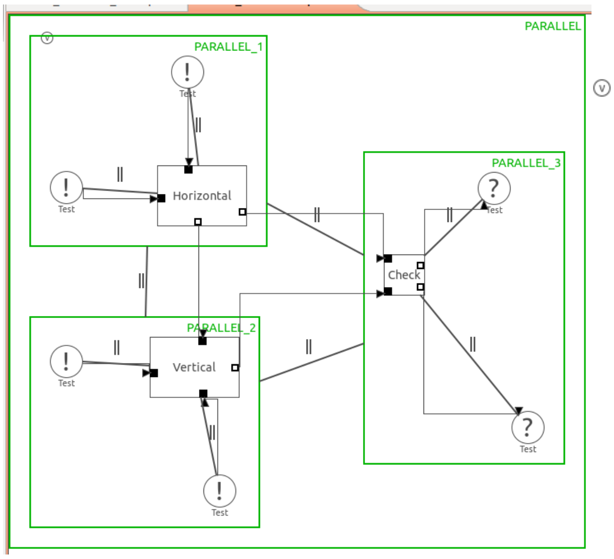
So this small change fixes the deadlock, see Figure 21a. Note the arrow is actually in the wrong direction, this is a visual bug in TERRA.

1.4.3

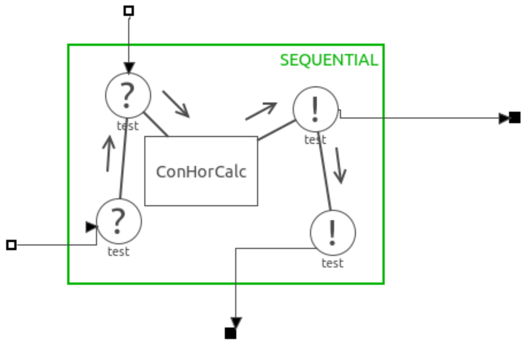
Figure 22 shows the IO SEQ pattern used on the JIWY model and Figure 23 displays the output of FDR, which does not deadlock.

1.4.4

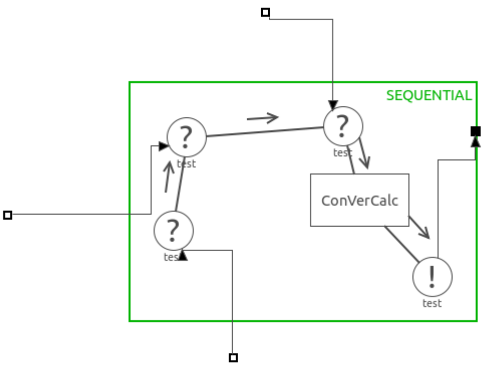
Figure 24 presents the TERRA system that we made to test the JIWY_Controller.
Figure 24b shows us the JIWY controller model with unconnected inputs and outputs that is being tested, and Figure 24a depicts a higher level of the system with JIWY_Controller model of Figure 24b as a process.
Figure 25 proofs that the model is deadlock free.



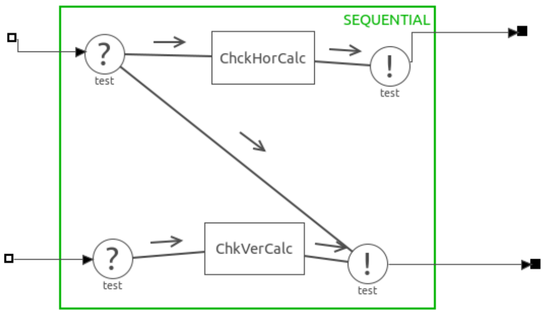
(a) Overview diagram of JIWY model.



(b) Inside the Horizontal process.



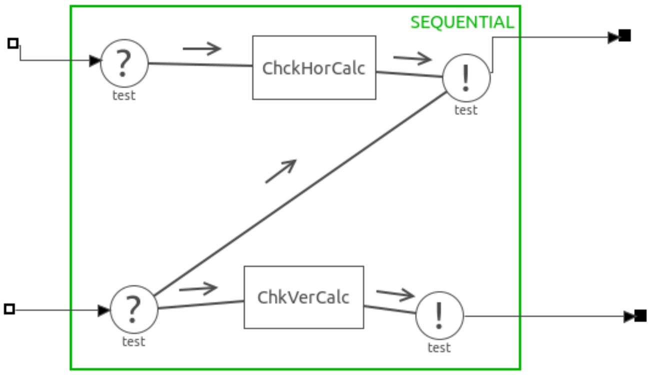
(c) Inside the Vertical process.



(d) Inside the Check process.

Figure 19: JIWY controller model (deadlock).

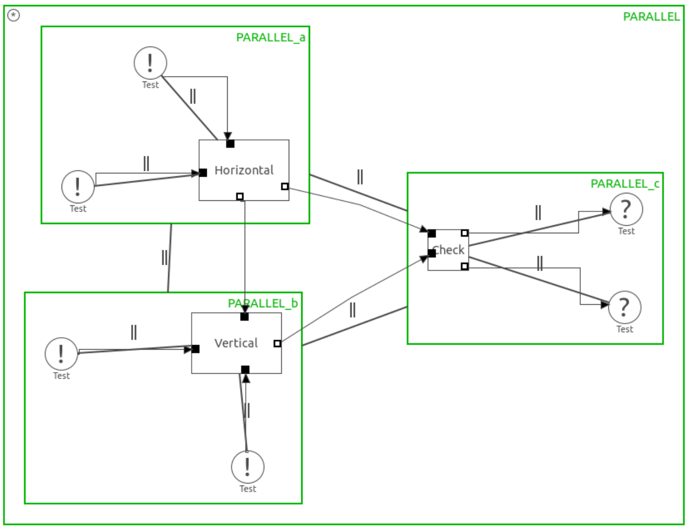
Figure 20: The model deadlocks.



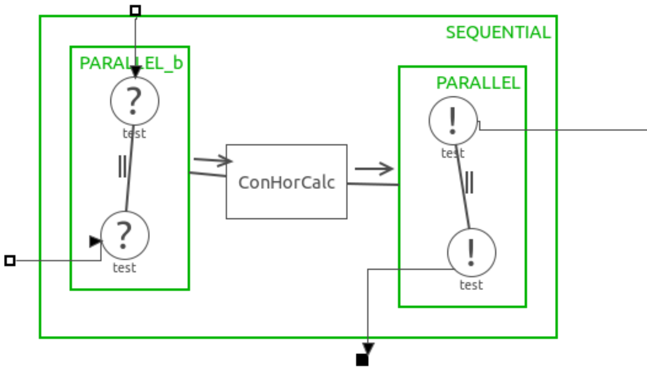
(a) Minimal deadlock fix.

(b) No deadlock according to FDR.

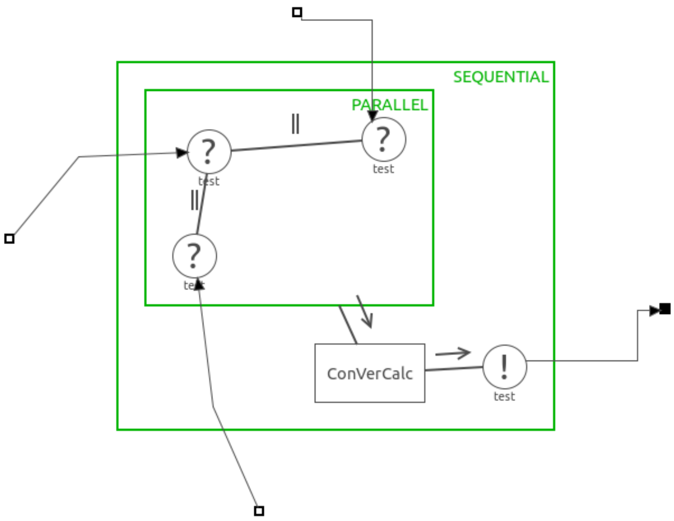
Figure 21: Fixing the deadlock.



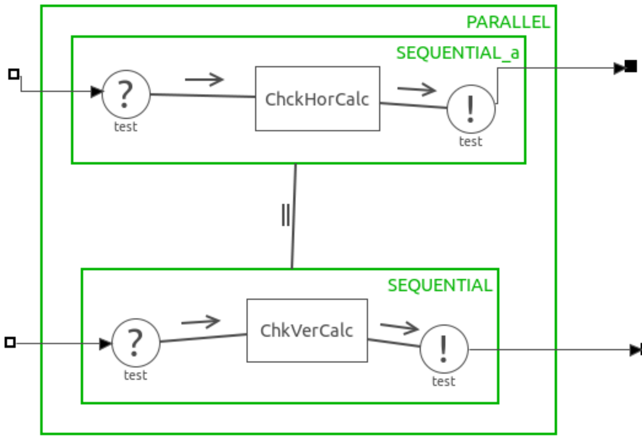
(a) Overview diagram of JIWY model.



(b) Inside the Horizontal process.



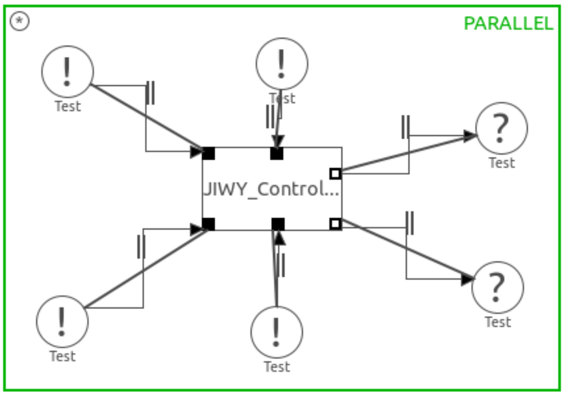
(c) Inside the Vertical process.



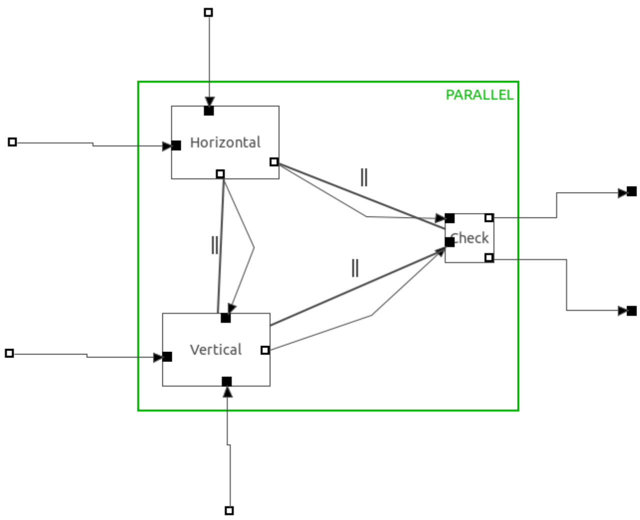
(d) Inside the Check process.

Figure 22: JIWY controller model IO SEQ pattern.

Figure 23: No deadlock with the IO SEQ pattern.



(a) JIWY_Controller is "under test".



(b) Inside the JIWY_Controller process.

Figure 24: "Submodel under test".

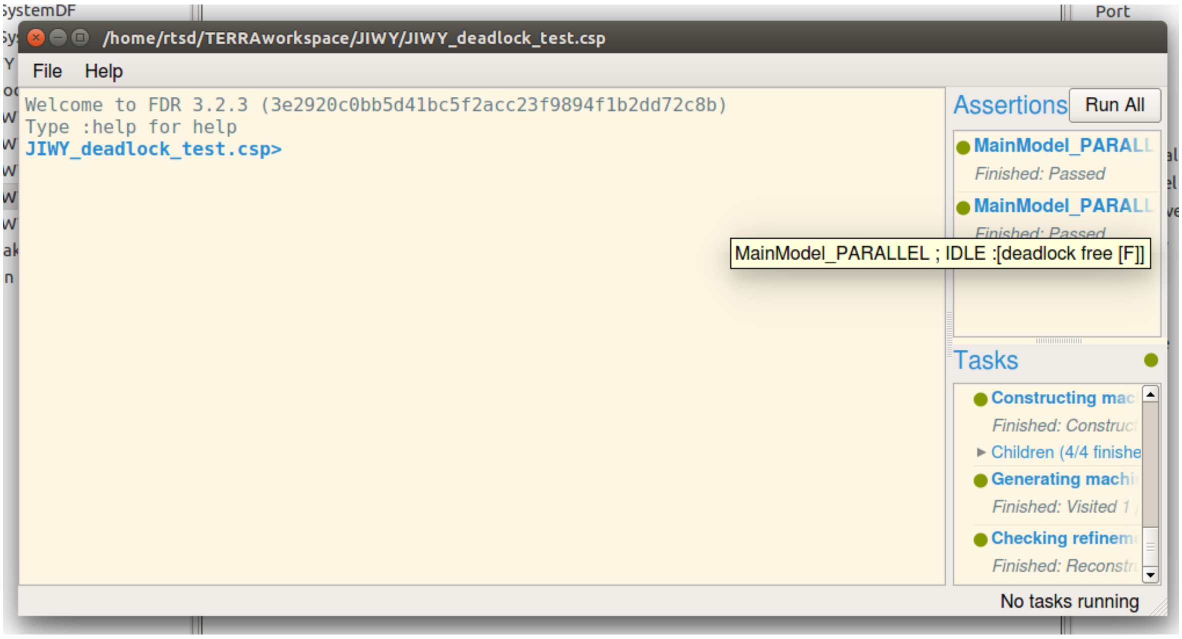


Figure 25: No deadlock with model under test TERRA System.

1.4.5

The solution of 1.4.2 expects that data will always be available from the inputs to the model, and if this is not the case then the system waits until data is written, because of the sequential reader-writer structure.
Solution 1.4.3 does not have this problem because the inputs and outputs are in parallel. So, having the inputs/outputs in parallel is the main advantage of 1.4.3.

1.4.6

One might choose to use the JIWY_Controller model in other TERRA systems, where only the input and outputs are different.
If you use the "submodel under test" approach you clearly separate the model and you make the interfaces to the model more clear.
In the case of large models with a lot of inputs and outputs, in can become very confusing if you need to change all the inputs/outputs from temporary test inputs/outputs (readers/writers) to say, `C++` code blocks.

1.4.7

The method of using "simple generator submodels" and "readers in the submodel that consumes the outputs" assumes that there is always incoming data to the model and always a possibility that output data of the model can be read.
In its realization, however, problems might occur if the system does not get input at all, or if the output buffers are full.
Implementing `C++` code blocks that simulate reading and writing, but with momentary (random) intervals of pausing, might more closely resemble a real feedback-control system.