Table of Contents

PART 1

- 1.1.1

The deadlock free system with two communication channels 1,2. On the system below the producer communicates firstly using channel 1 while the consumer is waiting on the same channel till the data is received. After that the same procedure happens on channel 2 and the processes are repeating.

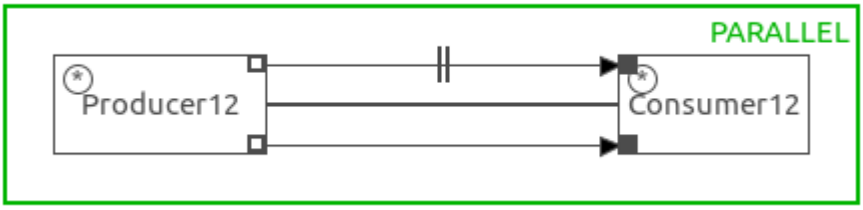

Figure 1. Deadlock free Producer-Consumer

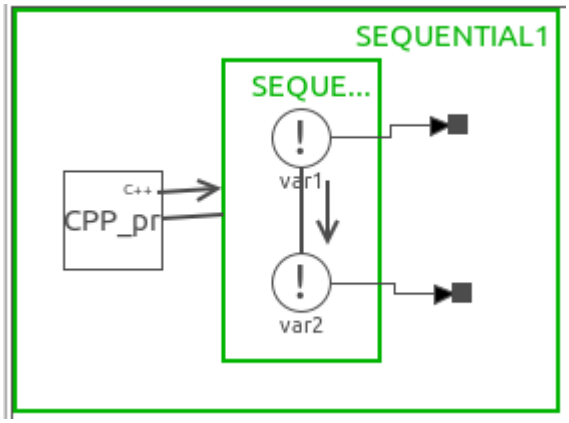

Figure 3.Producer

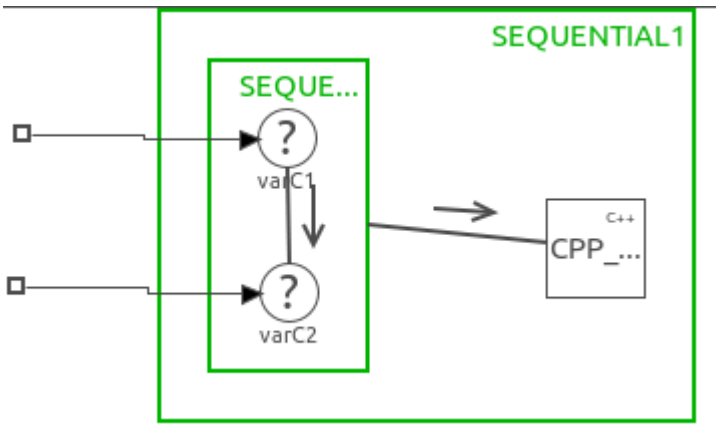

Figure 2.Consumer

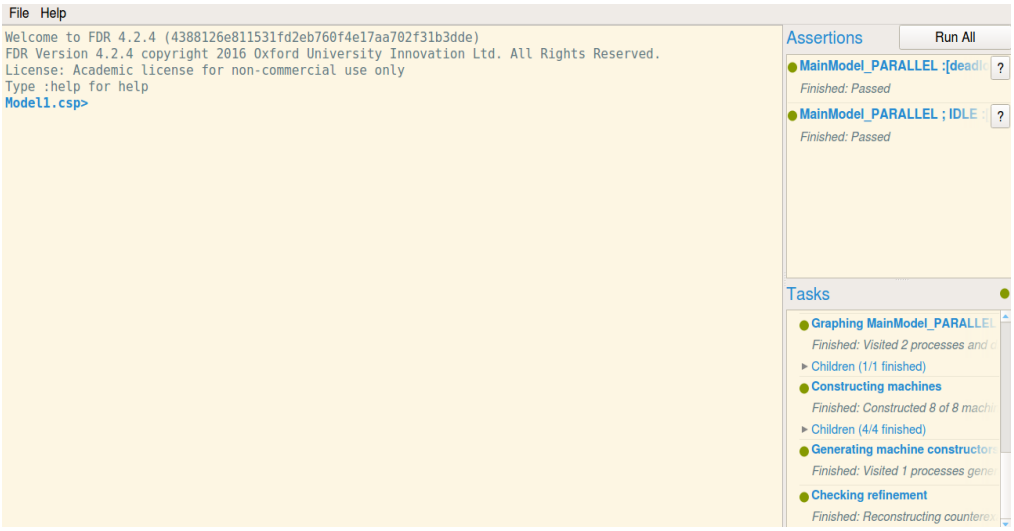

Figure 4.Testing the DF system with FDR

In the system that deadlocks the consumer waits on channel 2 while the producer is sending the data on channel 1 . This behavior makes both processes to wait each other , thus resulting to a deadlock. In order to create the deadlock we reversed the reading sequence on the consumer's side.

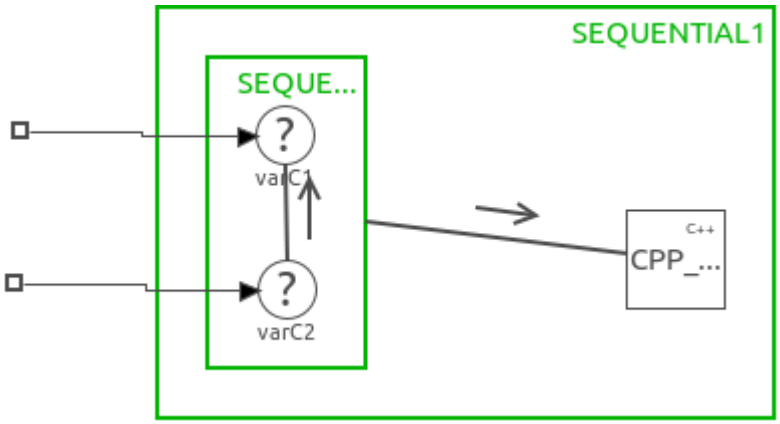

Figure 5.Consumer on the DC system

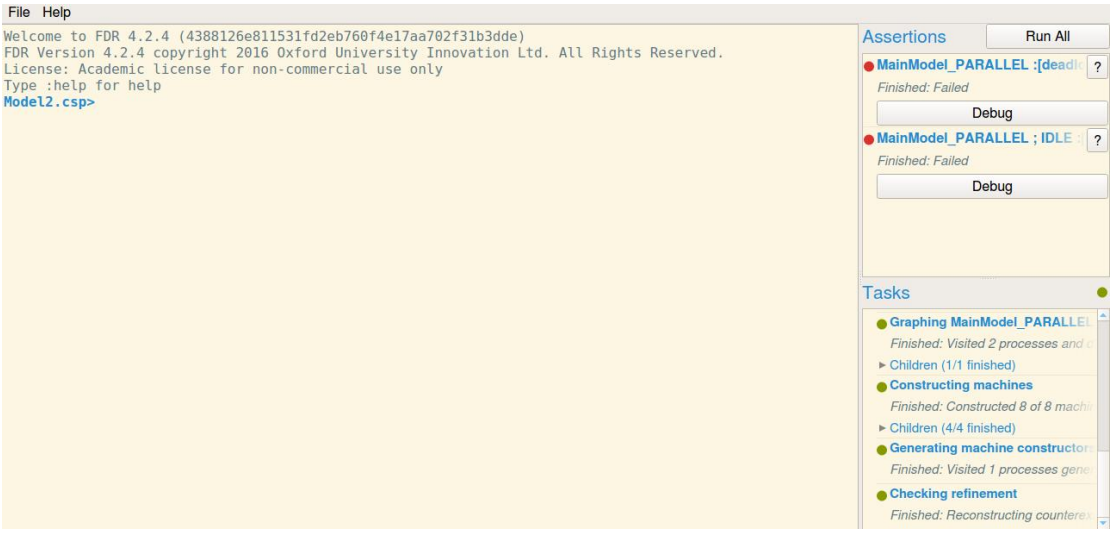

Figure 6.Testing that system using FDR

Through FDR simulation we see that this system deadlocks.

➢ 1.1.2

(For the model see figures 1,2,3). Using one communication variable for channel 1 and one for channel 2 the C++ block on the producer's side generates two pseudo-random numbers and loads them into those two variables indicating which number is for channel 1 and which for channel 2.

On the consumer's side when readers have both finished ,the C++ block indicates the numbers that were received from reader 1 (channel 1) and from reader 2 (channel 2) respectively.

What we expect to see on the terminal is 2 (or more) prints from the producer following by 2 (or more from the producer).

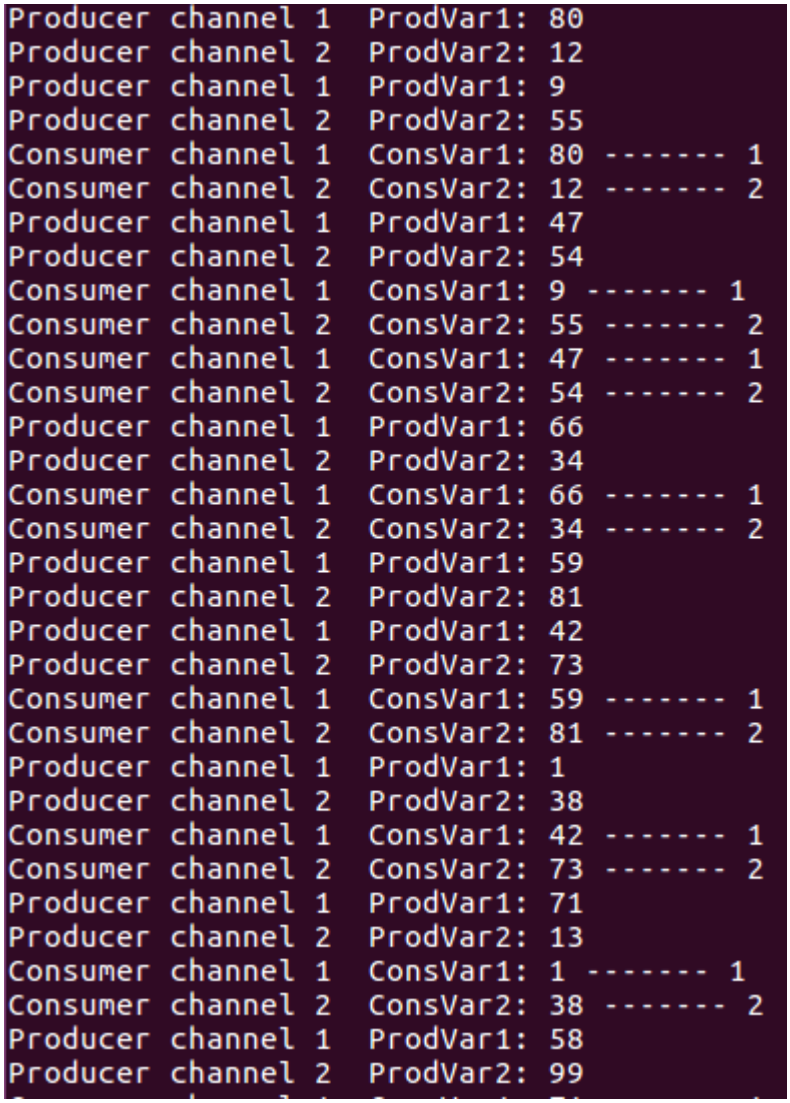

Figure 7.Producer-Consumer printed values

➢ 1.1.3

Any of the two processes (producer, consumer) can activated first as they are parallel. However, the consumer's code block cannot be executed before both readers have finished their execution. This means that producer's code block has already finished and both writers have run. What we see is producer making two prints and then consumer following with two more. This does make sense because of the above(consumer's code block cannot be executed before producer's). In some cases producer's or consumer's code block is executed twice (four prints) this is due to the parallelism of the system.

➢ 1.1.4

Probe interprets CSPm scripts and the user can check the process execution by seeing it on terms of CSP language (more abstract level). In case of a deadlock probe doesn't show anything. On the other hand the tree of the execution paths is infinite (because of the recursion). Probe is a tool for monitoring the logic behind the execution of the processes. FDR can recognize the infinite repletion (that happens because of the recursion not of a lifelock) of the executed processes and

decide whether the system is deadlock/lifelock free or not. For deadlock checking, FDR doesn't have to check all execution paths (as probe does) but by recognizing the finite or infinite behavior of the system it concludes if it deadlock free or not. (This is why a non recursive system cannot pass the FDR tests, because it is finite).

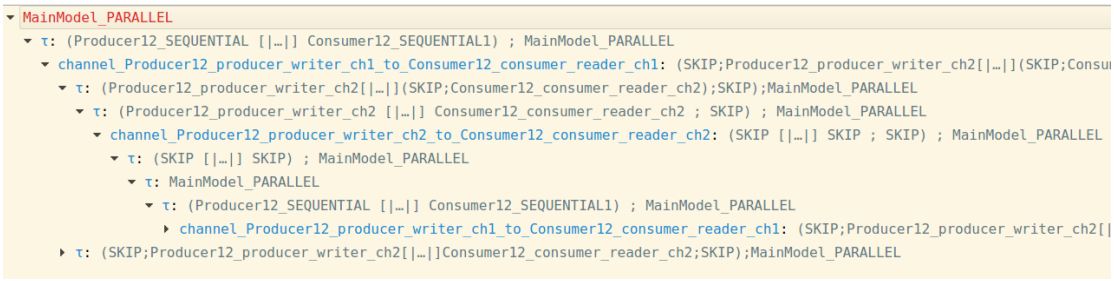

Figure 8.Probe of the DF system. We can spot the repetition and conclude that it is DF

➢ 1.1.5

Using only the print statements it is difficult to observe the parallelism on the producer-consumer system . Any of the two processes may activated first , if consumer starts then it has to wait for producer's code block to finish (at this point we see the producer printing on the terminal) and writer 1 to write the first value on channel 1 before actually running reader 1. This means that always we'll see the producer's code executed first without this meaning that producer was the first process that was activated. Sometimes we see producer(and consumer) printing four values that may happen because once the producer's process is finished it starts its execution over again without waiting for the other process to finish its execution. This behavior gives us a hint of the independent nature of the two processes but doesn't help us understanding which process was evoked first.

PART 2

➢ 1.2.1

No, it is not sufficient as mentioned in 1.1.5 using print statements we know when the code block sub-processes are executed and this order is more or less fixed (producer's first). This information doesn't give us any clue of the execution order of the other processes. What we can say about the order is that after having seen the two printfs from the producer then this process will move into activating the writer 1 , however this might not happen immediately because the consumer might run its own reader 1 first. Similar is the case and for the consumer's code block.

➢ 1.2.2

For that question we had to create a log event first and then use TERRA animation to monitor the execution order of the processes using the data that was acquired during the data logging session.

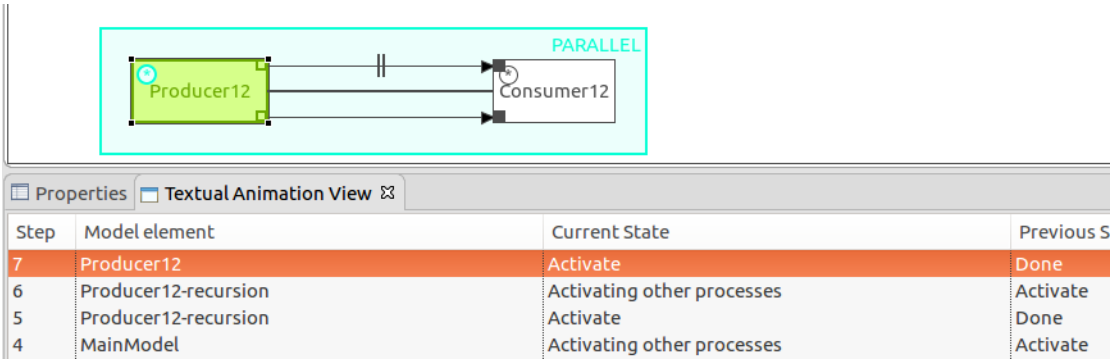

Figure 9.Producer is activated first

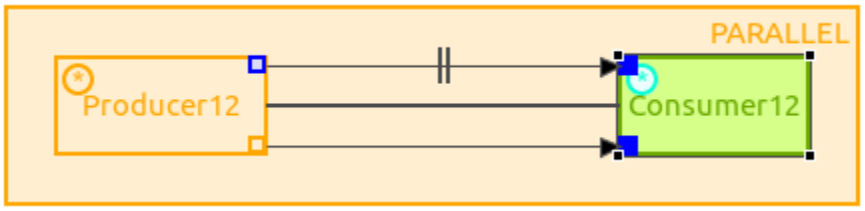

Figure 10.Consumer starts first

➢ 1.2.3

Using the TERRA animation we have a better insight of the execution order of the processes. We can see that there are times when the producer is executed first and then is executed again without waiting for the consumer. During this time consumer is waiting for a permission to start while it is "lagged" behind of the producer. This is the case when we actually see producer printing 4 times in a row. Also , we see some other cases when consumer has finished its execution and it starts again without having producer run its own part first. It is fair to say that both processes may start their execution at any order , something that wasn't visible previously with the print statements.

➢ 1.2.4

As it is also mentioned on question 1.1.4 probe interprets CSPm scripts and creates execution path trees on CSP language, it is a more abstract level of execution tracing. Using TERRA animation we trace the execution of the processes based on logged data that was acquired. We can change the input and see the difference on the animation. It is a non strict (language based) description of the processes execution. We can use TERRA animation for having a better understanding of how our system response in different cases and use it as a debugging tool. On the other hand, probe provides a general overview of the system's execution.

➢ 1.2.5

The readers/writes start from the *Done* state and remain there until they evoked for execution by transitioning into the *Activate* state . From there they will start running by moving into the *Running* state. Once they finished running their next state could be either *Done* or *Waiting*. If the reader/write from the other side the communicate with has already finished its running part and it is waiting then they will transition into the *Done* state. Otherwise they will move into the *Waiting* state and l remain there until the reader/writer from the other side finishes.

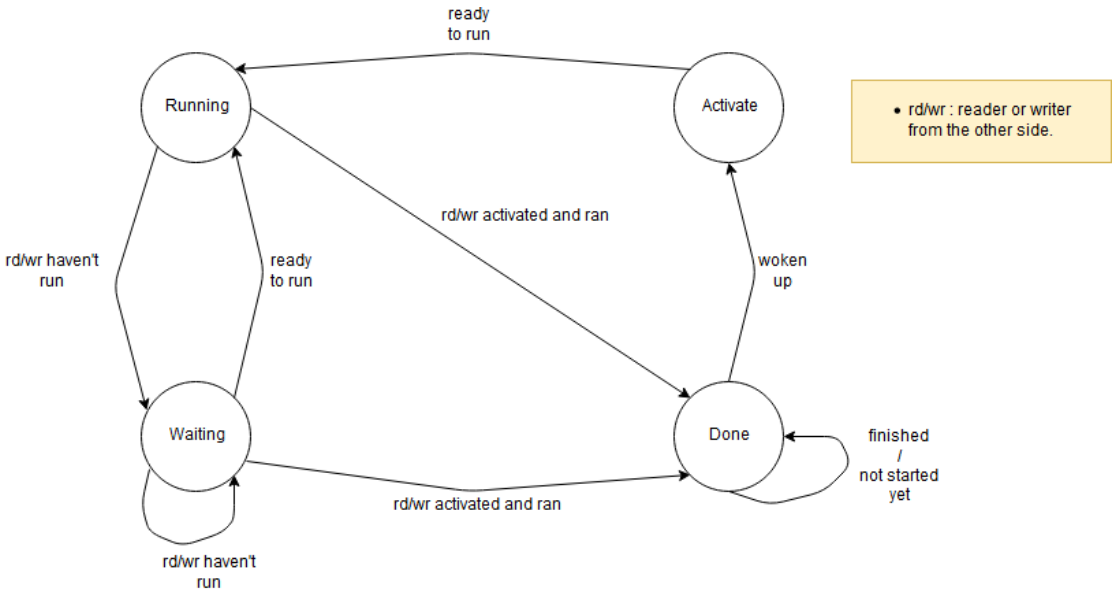

Figure 11.Transition diagram

➢ 1.2.6

Interleaving parallel processes are processes that are executed in parallel but they don't communicate. Completely independent processes that are parts of the same parallel system.

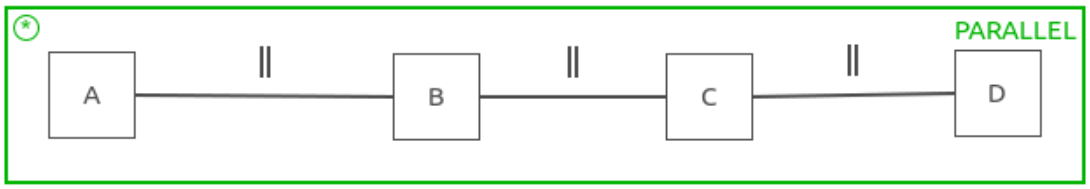

Figure 12.Four interleaving parallel processes

We expect such a system to be deadlock free. Having four processes that don't communicate with each other means that there is no dependency among them (like there was on the producer – consumer system) and thus no reason for a deadlock to occur.

Figure 13. Validating that the system above is DF using FDR

➢ 1.2.7

Using the TERRA animation we can see the execution order of the four processes. Due to the parallelism we expect the processes to be executed with a different order each time as all four should have equal possibilities of executed first.

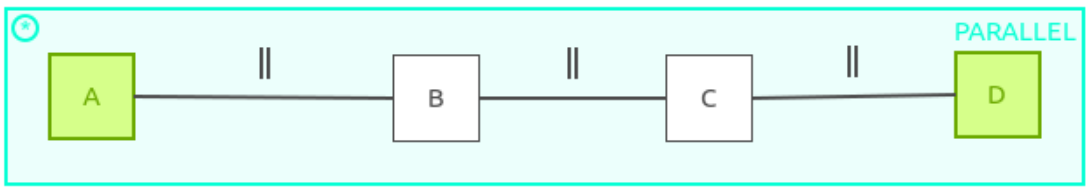

Figure 14. At this sequence A starts first and then is followed by D

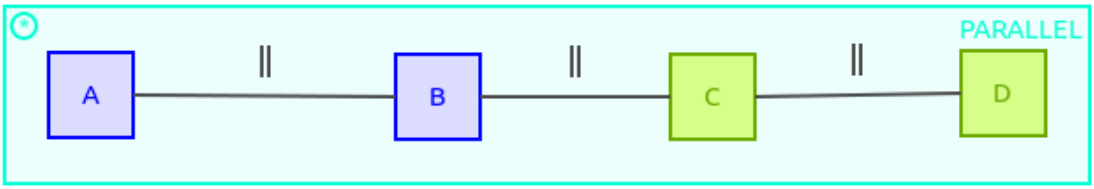

Figure 15. At this sequence C starts first and then is followed by D

By running the TERRA animation multiply times we see that the execution order seems to be decided “randomly”, something which in consistent with what we expected to happen.

PART 3

➢ 1.3.1

We created a producer-consumer system having three consumers that are executed in alternate.

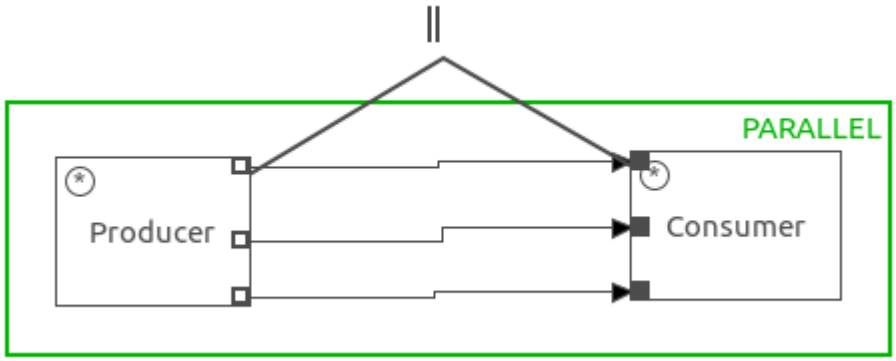

Figure 16. Top-level producer-consumer system

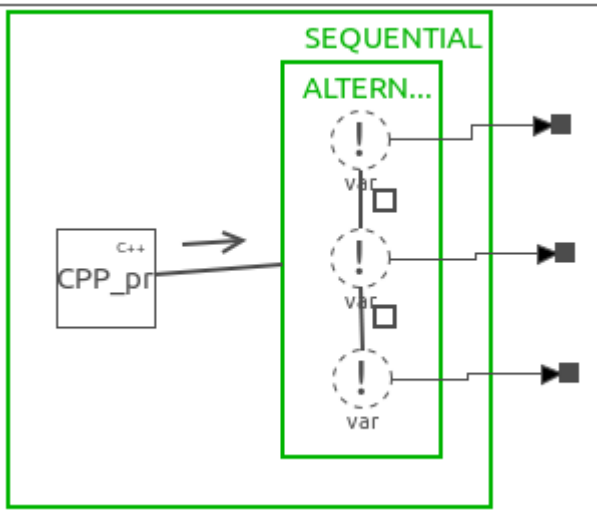

Figure 17. The producer with the 3 writers

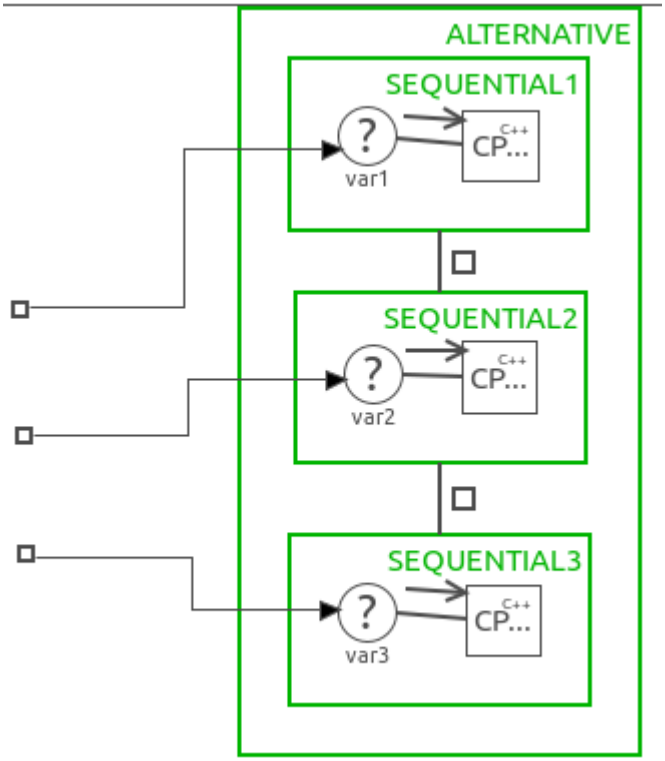

Figure 18. The three ALT consumers

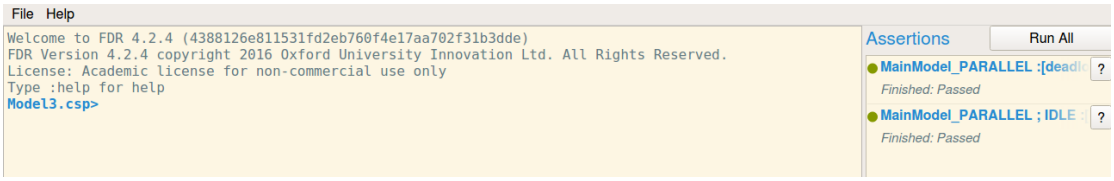

Figure 19. Using FDR to check the system

➢ 1.3.2

(For the model see the figures 16,17,18) . All three writers on the producer's side write on the same variable (var). In order to decide which value of var will go to channel 1 , which to channel 2 and which to channel 3 we applied the following conditions to the writer's guards.

Writer 1 : Accepts all values smaller that 10

Writer 2: Accepts only 20

Writer 3: Accepts all values greater than 30

Because of the guards when we insert a value smaller than 10 it goes through channel 1 to reader 1, when we insert 20 it goes through channel 2 to reader 2 and when we insert any value greater than 20 it goes through channel 3 to reader 3.

Due to the ALT structure only one writer and one reader is executed per execution cycle.

The user inserts three values upon the producer's request and by storing these values into an array we pass them in order to the consumer until all have been read.

```
void CPP_pr::execute()
{
  static int numbers = 3;
  static int values [3];
  if(numbers==3)
  {
      printf("Insert 3 numbers: \n");
      for(int i=0;i<numbers;i++)
      {
          scanf("%d",&values[i]);
          if(i==numbers-1)
              numbers=0;
      }
  }

  var = values[numbers];
  printf("sending %d \n",var);
  numbers++;
```

Figure 20. Producer's execution code block

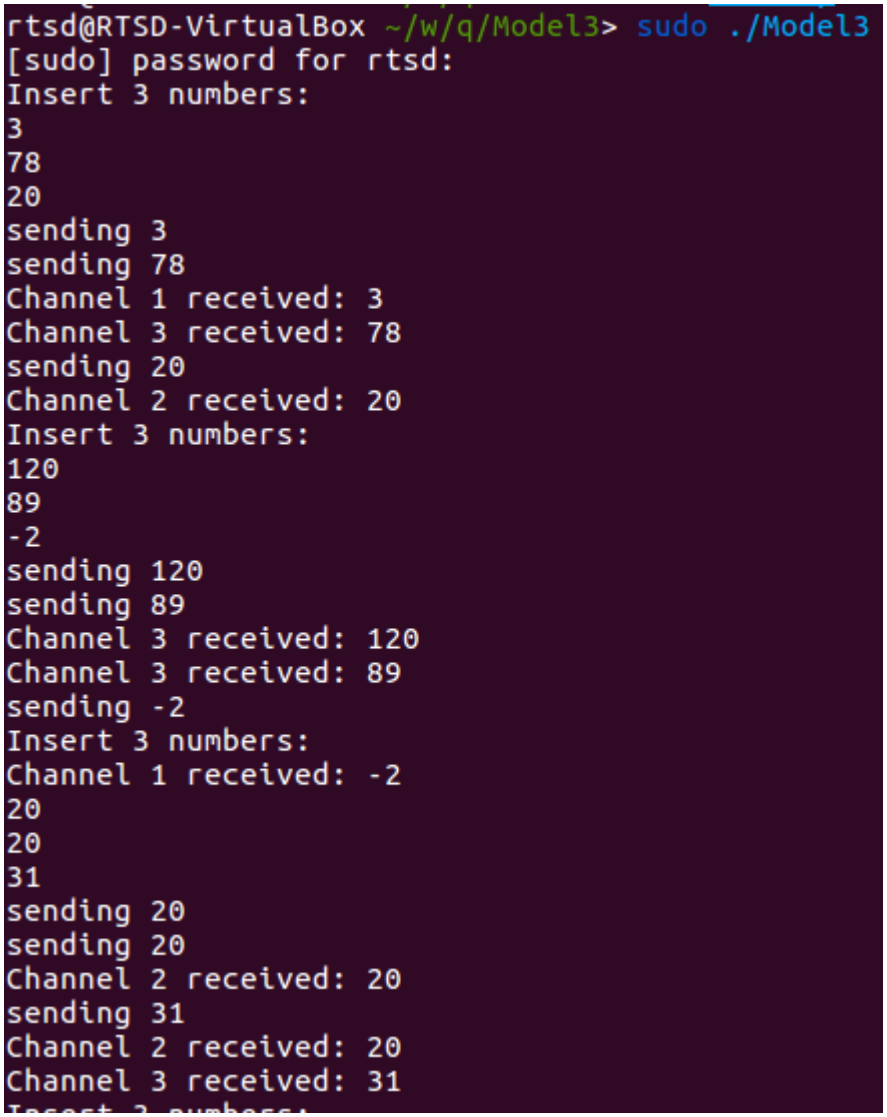

Figure 21. Printed values

On figure 21 we can see that the values that are entered be the user are going to the correct consumers according to the producers guards.

Because of the parallelism we see that the printed messages don't have a well defined order.

However, the above system is not robust and doesn't work nicely if the user enters any value between 10 and 20.

➢ 1.3.3

To tackle the issue with the values between 10 and 20 we applied a second code block on the producer's side called CPP_check in sequence to the first one.

The main idea is that the second code block will filter each value of the variable var that leaves the first C++ block before going to the writers.

If it encounters any value between 10 and 20 it prints a message and forces this value to become 0. In this way every value that is out of the guards range will become 0 and will pass through channel 1 to the first consumer.

➢ 1.3.4

Having a top-down design method starting from the process block structures and then going lower into their coding implementation is a way of designing structural a model and testing it on various levels before heading towards the coding. The multi-level testing process using FDR, probe, TERRA animation helps into creating a robust system which has a correct basic functionality before even implementing the code.

➢ 1.3.5

The idea that was proposed on 1.3.3 helps having a more robust system by redirecting all out of range values into channel 1. Another idea would be to remove entirely this CPP_check code block and adding its functionality into the first coding block. However, that would probably made the code less readable. Also, instead of redirecting the values we could remove them entirely from the system or even forcing the user to enter different values , but this implementation would seem to be more hard-coded than the proposed one. A disadvantage of the proposed method is that we have two different guarding systems , one on the writers and one on this extra coding block. Probably, by having this block in an ALT configuration alongside the other writers and putting a guard there would be an alternative. However, by doing so the whole system would become more complex.

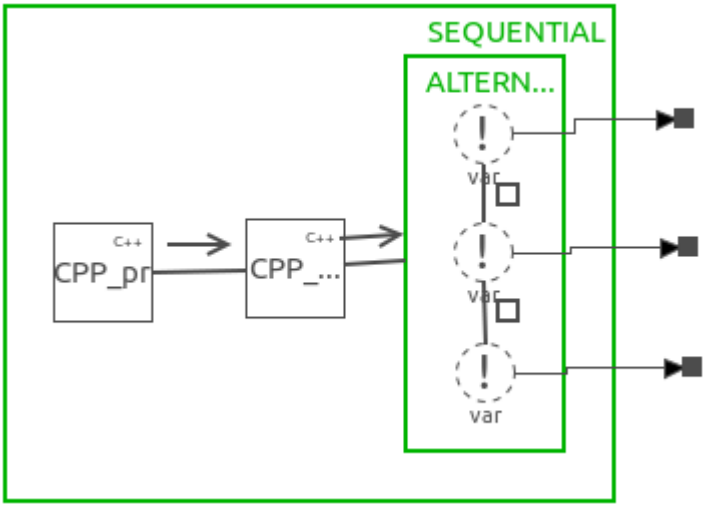

Figure 22. Proposed improvement on the producer's side

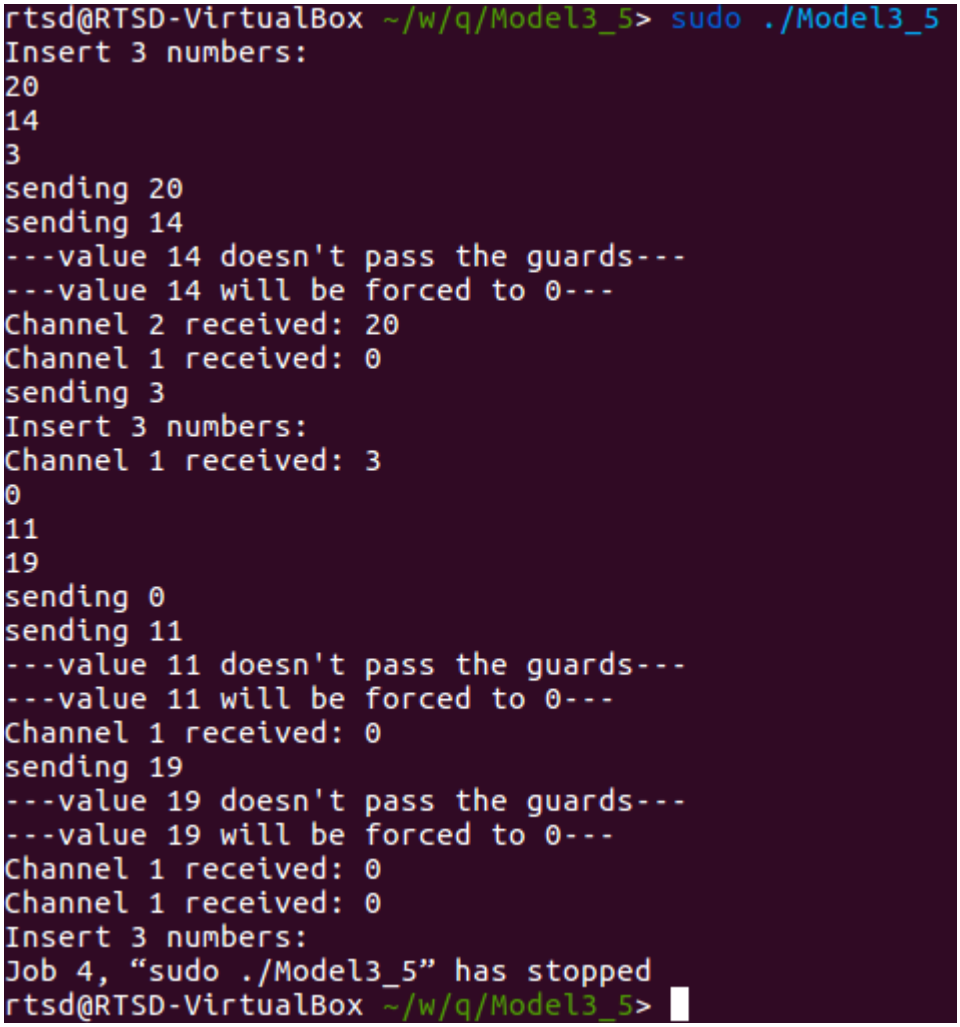

Figure 23. Applying the proposed method and seeing its results on the terminal