Exercise 1

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

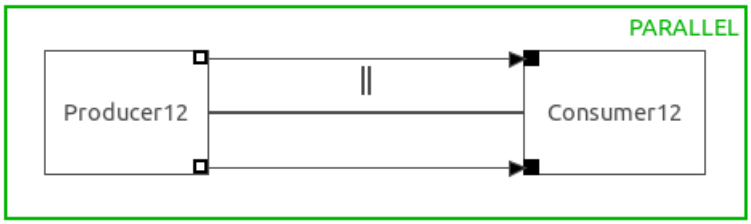
May 17, 2015

1

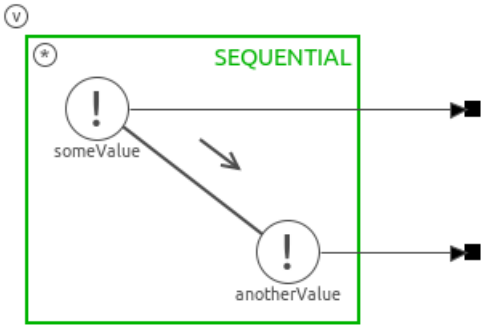
1.1

1.1.1

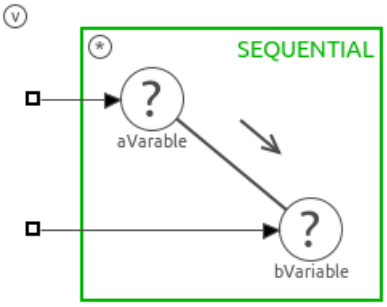
We modelled the deadlock free system as show in figure 1, and the deadlocked system as shown in figure 3. As expected, FDR accepts the deadlock free system (firgure 2), but not the deadlocked system (figure 4).



(a) Composition of the Producer and Consumer processes.



(b) The producer process.



(c) The consumer process.

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

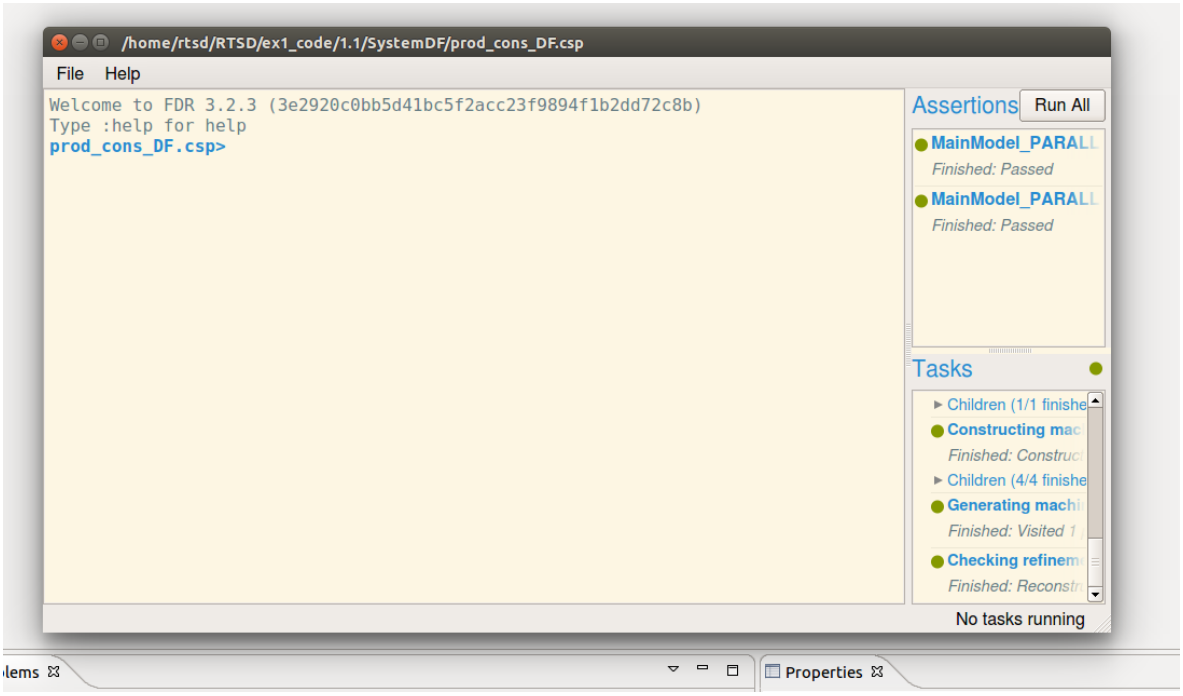
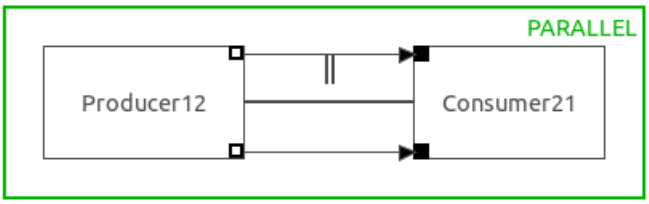
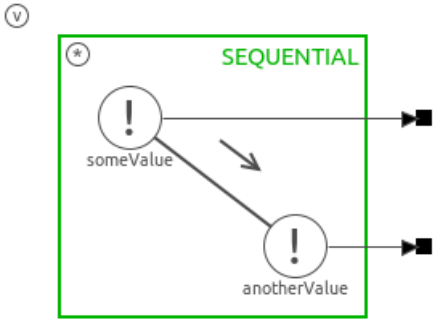


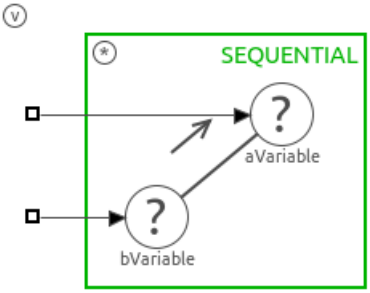
Figure 2: Succesfully passes tests in FDR.



(a) Composition of Producer and Consumer processes.



(b) The producer process.



(c) The consumer process.

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

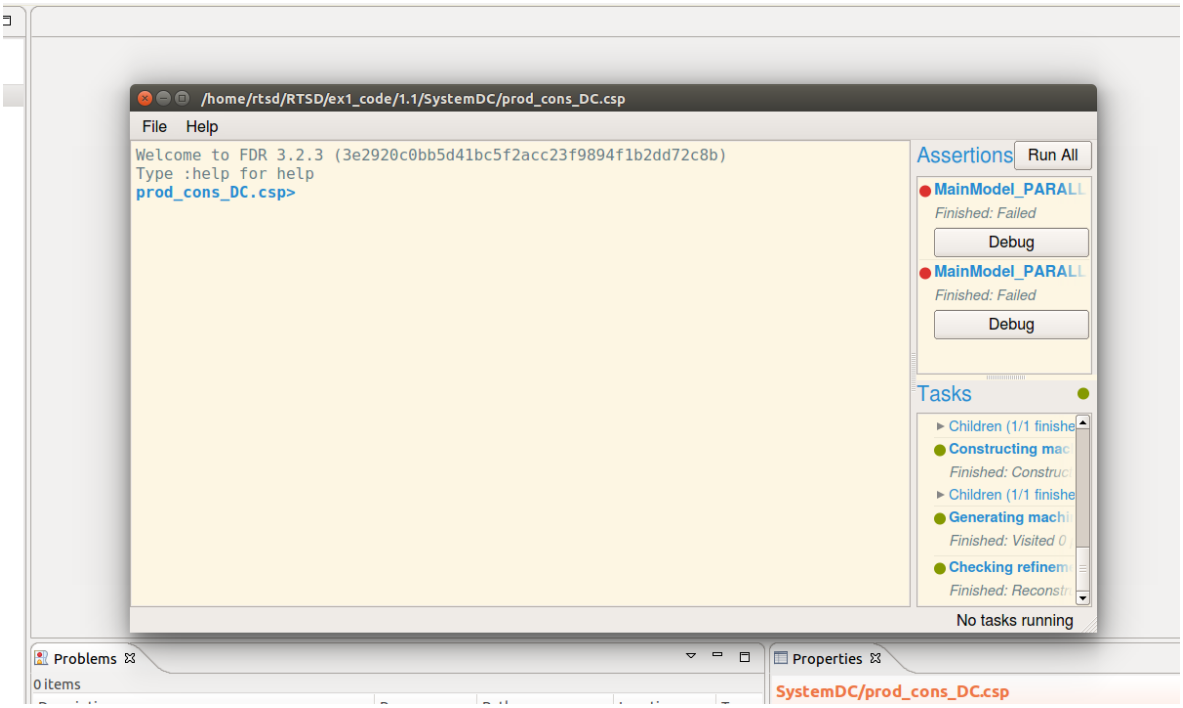


Figure 4: Fails the tests in FDR.

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 6 and the code in included in blocks 1 and 2. The initial few lines of the output is shown in figure 7.

Codeblock 1: Consumer12/CCode.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

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

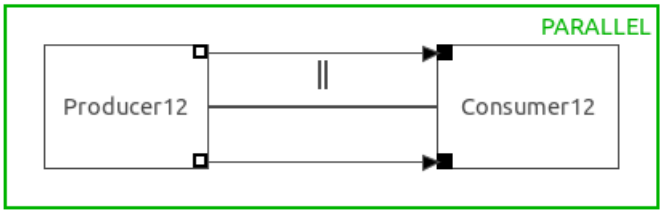
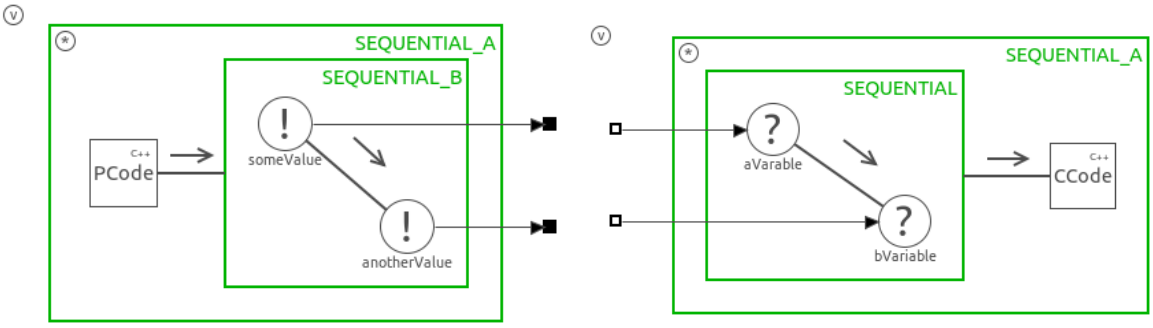


Figure 5: Overview diagram of the producer consumer system.



(a) The producer process.

(b) The consumer process.

Figure 6: The producer-consuder (DF) model extended with code.

```
3   * Generated by the TERRA CSPm2LUNA generator version 1.1.1
4   *
5   * protected region document description on begin
6   *
7   * protected region document description end
8   */
9
10  #include "Producer12/PCode.h"
11  #include "string.h"
12  // protected region additional headers on begin
13  // Each additional header should get a corresponding dependency in the Makefile
14  // protected region additional headers end
15
16  namespace MainModel { namespace Producer12 { namespace PCode {
17
18  PCode::PCode(int &PCode_anotherValue, int &PCode_someValue) :
19      CodeBlock(), PCode_anotherValue(PCode_anotherValue), PCode_someValue(↩
            PCode_someValue){
20    SETNAME(this, "PCode");
21
22    // protected region constructor on begin
23    // protected region constructor end
24  }
25
26  PCode::~PCode()
27  {
28    // protected region destructor on begin
29    // protected region destructor end
30  }
31
32  void PCode::execute()
33  {
```

```
34    // protected region execute code on begin
35
36      static int index = 0;
37
38      char *stuff = "Appelflap";
39
40      if (index == strlen(stuff)) {
41          this->PCode_someValue = -1;
42          this->PCode_anotherValue = -1;
43      } else {
44          this->PCode_someValue = stuff[index];
45          this->PCode_anotherValue = stuff[index++];
46
47          printf("Sending: PCode_someValue: \t'%c'\n", this->PCode_someValue);
48          printf("Sending: PCode_anotherValue: \t'%c'\n", this->PCode_anotherValue);
49      }
50
51
52    // protected region execute code end
53  }
54
55  // protected region additional functions on begin
56  // protected region additional functions end
57
58  // Close namespace(s)
59  } } }
```

```
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 7: 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 addapt 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.
Give proof that it is fair using the tools

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.