Software Fiavel - Assignement3

Santiago Benites fc54392

December 2022

1 Shared Memory

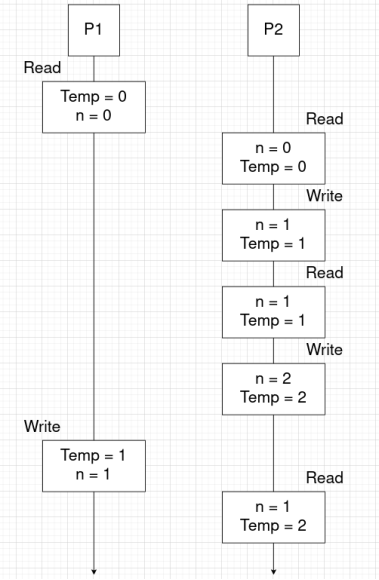
1.1 Use Spin to check if the system has the following properties;

1.1.1 Whenever a process is at reading, n is larger or equal to temp

Spin doesn't confirm this assertion.
To check this assertion we added the following code

```
reading :   assert(temp <= n);
            temp = n;
            temp ++;
}
```

We will now trying to explain the situation where this assert is violated. In the following execution we represent the value of n and temp at the time of the assert.
Both process start with a read, then P2 does 2 entire read write cycles. This makes it that when P1 does a write, it changes the value of n from 2 to 1. Making it so that when P2 makes another read, its internal temp is at 2 while the global variable n is at 1 because of P1. Violating the assertion.



1.1.2 The value of n is always smaller than 20.

Spin confirms this property
To check this assertion we added the following code

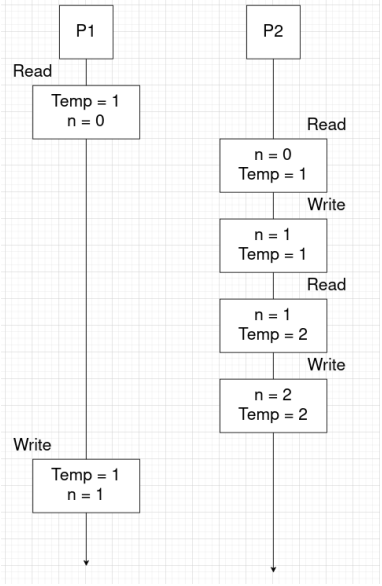
```
ltl smallerThan20{[] (n < 20)};
active [N] proctype P ( ) {
    byte temp , i = 1 ;
```

1.1.3 The value of n never decreases.

Spin doesn't confirm this assertion.
To check this assertion we added the following code

```
writing :   assert(temp >= n)
            n = temp;
            i ++;
}
```

We will now trying to explain the situation where this assert is violated. In the following execution we are representing the value of n and temp at the time of the assert.
The most crucial step of the execution, is the step between the second write of P2 to the first write of P1 where the value of n changes from 2 to 1, which means that the value of n decreases.
Therefore the assertion is violated



1.1.4 Once the value of n becomes larger than 0, it will continue so.

Spin confirms this property
To check this assertion we added the following code

```
ltl largerThanZerotheAlways {<> ([] (n > 0))}
active [N] proctype P ( ) {
    byte temp , i = 1 ;
}
```

1.1.5 It is not possible that the value of n becomes 2, then a value different from 2 and then 2 again

Spin doesn't confirm this assertion.
To check this assertion we added the following code

```
ltl twoNotTwoAgainTwo{n == 2 U ([] (n!=2)) };
active [N] proctype P ( ) {
    byte temp , i = 1 ;
}
```

The general idea for this is the same as 1.1.1 and 1.1.3, one process does a write that changes the value from 2 to 3 for example, and the other process changes it back with a dirty write.

1.2 What is the smallest possible value that the n can assume when both processes have terminated? Use Spin to justify your answer

When trying to solve this question we changed the code to the following

```
#define N 2
#define N_ITER 5
byte n = 0 ;
int finalizedProcesses = 0;
ltl minValue {[] ((finalizedProcesses == N) -> n >= 2)}

active [N] proctype P ( ) {
    byte temp , i = 1 ;
    do
        ...
```

```
11      od
12      finalizedProcesses++;
13  }
```

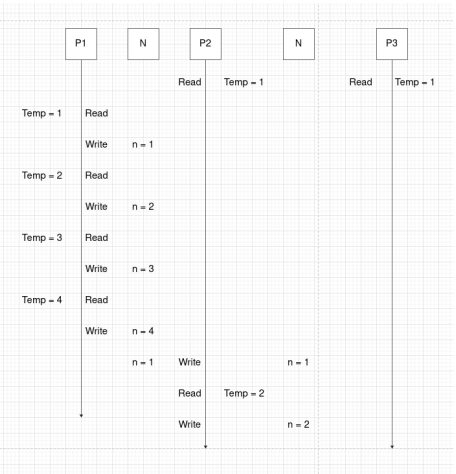
Considering we only want the assertion to be checked after both processes end, we use the variable $finalizedProcesses$ to make sure both processes are finalized before we checking for the assertion.
The assertion makes sure that after both processes are finished, the value of n is larger than 2, therefore being 2 the smallest value of n possible.
Finally to confirm that this is truly the case we checked in spin and the program passed without errors. In order to check that the smallest possible value is 2, we just need to make a small alteration to the code

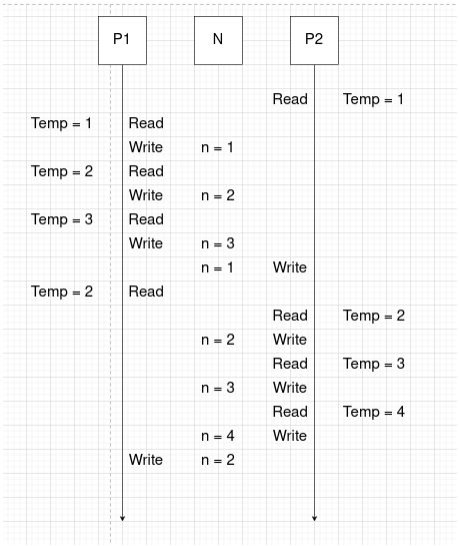
$$ltl\ minValue\{[]((finalizedProcesses == N) -> n >= 3)\}$$

And when running the program again in spin it checked for an error.
This because in some situations the value of n after the execution would be 2 and therefore would raise and error in the assertion.

1.3 Does the same minimum value holds for other N , say 3 or 4? Justify.

This value does hold for all values of N, this happens because in reality when we have more than 2 in N, we can simplify this problem to be similar to a problem with N = 2.
Take for example the execution on the left, as we can see after the entire execution of P1, the value of n can be returned to 1 because the process P2 kept the value of 1 in its local temp. Now we can start the execution of P3 and instead of having the original 3 processes we only have 2.
This way of keeping temp at 1 so that it can be passed to n, can be done any number of times as long as we have a processes without any operation.
Lastly we need to finalize the program's execution, this is done with the execution of the right. As we can see, we execute almost all of P1 except one write, then we do the entirety of P2, and then we use the last write of P1 to reset the value of n to temp of P1, but because of the read in P1, used to reset the value, temp of P1 is no longer 1 but now it is 2.
Therefore the smallest value of n is 2.





In order to justify this we used the same ltl, but we altered the value of N to 3 and 4.

$$ltl\ minValue\{[]((finalizedProcesses == N) -> n >= 2)\}$$

1.4 What is the largest possible value that the variable n can assume when both processes have terminated? Provide an expression for this value and define it with #define. Use Spin to justify your answer.

Just like in question 2 in order to solve this question we added some code to the program.

```
1
2  #define N 2
3  #define N_ITER 5
4  #define N_MAX (N * (N_ITER - 1))
5  byte n = 0 ;
6  int finalizedProcesses = 0;
7
8  ltl maxValue {[] ((finalizedProcesses == N) -> n <= N_MAX)}
9
10 active [N] proctype P ( ) {
11     ...
12     finalizedProcesses++;
13 }
```

In this case we changed the previous ltl statement to search for the largest value of n, this was done by changing the $\geq$ to $\leq$.
Moreover, as the prompt indicated we put the value that n can have max in a #define variable. This value is dependant on the N and the N_ITER. This because these 2 values in combination indicate the number of times P will run the write read cycle.
We run this new program thought spin, and with this specific value of N_MAX it passed without errors, any alteration to this value that makes it smaller makes spin indicate errors.

1.5 Does the model assume that two processes can be writing n at the same time? If so, what is assumed regarding the value that is wrote in two overlapping writes?

As we can see from the previous questions the model assumes that the 2 processes can be writing in n at the same time, this happens because the model doesn't implement any form of concurrency management.
Taking this into consideration, we can assume that the value when 2 overlapping writes occur, there will have been a dirty write by one of the processes.

1.6 Does the model assume that reads and writes of n are atomic actions or does it consider that a process can be reading a memory location while another process is writing it? If the latter is the case, what is assumed regarding the value that is read in a reading that overlaps a write?

The model doesn't assume that the reads and writes are atomic actions, this because there isn't concurrency management, therefore a process can write to n while another is reading from it.
If this situation happens, we can assume that the value read by the second program is a dirty read, this means the read of a value that isn't correct according to a linear execution of the program.

1.7 Develop a new model of the system, this time considering that reads and writes of n are atomic. What is the smallest and largest possible value that the variable n can assume when both processes have terminated? Use Spin to justify your answer

In order to make sure reads and writes are atomic we altered the code to the following

```
1  #define N 2
2  #define N_ITER 5
3  #define N_MAX (N * (N_ITER - 1))
4  byte n = 0 ;
5  int finalizedProcesses = 0;
6
7  bool wantC = false;
8
9  ltl maxValue {[] ((finalizedProcesses == N) -> n == N_MAX)}
```

```
10
11 active [N] proctype P ( ) {
12     byte temp , i = 1 ;
13     do
14     :: i < N_ITER ->
15
16         atomic{
17             !wantC
18             wantC = true
19         }
20
21
22         reading: temp = n ;
23                  temp ++;
24
25         writing: n = temp ;
26                  i ++
27
28         wantC = false;
29
30     :: else -> break
31     od
32     finalizedProcesses++;
33 }
```

This new code uses the variable $wantC$ to manage concurrency, and make sure only one process accesses the variable n at the time, each time a process accesses the variable n it executes and entire read write cycle. This was done because if it wasn't, we would still have processes writing local values of temp into n, these values of temp wouldn't be coordinated with writes and therefore would be the same as a linear execution.
Considering this new code, we completely removed the randomness that existed because of the processes, this lead to the value of n for all executions being the same N_MAX.
We can use the line

$$ltl\ maxValue\{[]((finalizedProcesses == N) -> n == N\_MAX)\}$$

to justify our answer.

2 Message Passing

2.1 Client - LocalController

In this step of the process, we are going to implement a Call service, which will consist of a client and a phone. This service, will have a LocalController which will represent an old telephone, this being said we assumed somethings about this phone service. The phone service just like all old phones will have 4 main operations

1. pick up
2. return
3. click

and lastly the possibility of introducing numbers to call the phones of other people.
The first operation, pick up does what it says picks up the phone, allowing the client to be able to interact with it in other ways. Such as, inputting a phone number,clicking the phone button and return the phone to the stand.
The second operation, return allows the client to return the phone to the stand, which disables it from serving its purpose, and makes it so that the client cant interact with it anymore, other than picking it back up.
The third operation, click is a feature characteristic from old phones, in which we can click a button (usually on the stand) to reset the phone number we are trying to call, when the phone number is reset it will erase all previously inputted numbers.
The last operation, is probably the most important, and allows the client to input a single number, this if done multiple times makes the client able to input an entire phone number, which will be called automatically as soon as 9 digits are inputted.
The process by which we implemented all these features is multi-fold.

2.1.1 Client

The client, interacts with the outside world by using a channel in which we insert multiple int values which represent different operations.

```
#define pickup  11
#define click   12
#define ret     13
```

Each operation is then read by the client and interpreted, after this the client acts in an according manner. We should also note that, the client does check only client side to make sure that no illegal movements are made, such as picking up the phone 2 times in sequence or trying to input numbers with the phone on the stand.

2.1.2 LocalController

When the client is sure the movements are valid it sends the operation to the LocalController, our phone, for they're execution. The client send the operations through a channel which is shared between the client and the LocalController, and which is passed to both of them on initialization.
Depending on which operations are made the LocalController can operate in different ways.
The LocalController keeps, local variables about the phone number we are trying to reach, specifically one with the complete phone number and another with its length. These are used for when the phone number reaches a size of 9, the phone automatically "call" the phone in question.
Considering this implementation doesn't have a way to communicate with other clients, our calls random choose between a Busy tone and a Ring Tone, and present this output to the client.

2.2 Client - LocalController - Remote Controller

In this step of the process we are going to finally implement the call function of the telephone. For this purpose we will need an additional controller, the remoteController.
This remote controller, serves as a way to mange the connections between localControllers, and it has 4 main functions

1. isValid
2. start
3. release

Which make use of multiple defines in order to operate, these being said defines:

```
#define start         21
#define complete      22
#define confirmation  23
#define denial        24
#define accept        25
#define release       26
#define terminate     27
```

The first operation, is called when a localController asks to start a call with a specific number, it serves the purpose of analysing the number passed by the local controller and seeing if this number is valid according to our model. In our implementation valid numbers are stored in a global var called $knownNumbers$. This is done in order to remove some of the randomness in the remoteController and make it easier to test.
The second operation, is the real start of the call procedure. It will simulate a separate localController and send back to the localController whether or not the simulated localController accepts the connection started in the previous operation.
The last operation, is used to terminate a connection, and it is called when the caller returns the phone to the stand.
Our implementation doesn't actually communicate with a second localController, it just pretends to do so, and simulates the output of said localController randomly. This is done, by adding randomness in the way the remote-Controller handles responses and requests.
We also added a timeout to the loop in order to prevent promella timing out every time we run the code

```
  ::timeout -> printf("Connection has timed out. Disconnecting...\n")
            break
```

There are 2 main local variables we need to address

```
proctype Client(chan request;chan operations){

  byte i;
  bool onHook = true;
```

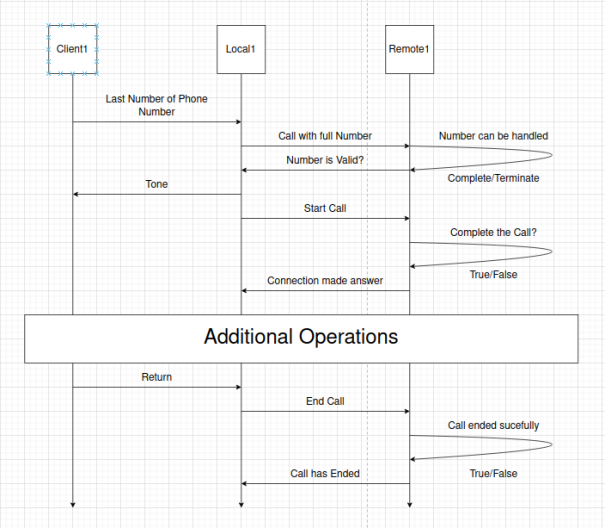
The variable *onhook* from the client, is used to keep whether the phone is on the hook or in the stand, its main purpose is to make sure that the client doesn't do illegal operations, such as picking up the phone twice or inputting numbers while the phone is on the stand.

```
proctype LocalController(chan request;chan remote){

    int operation;
    int number;
    int numberSize = 0;
    bool onCall = false;
```

The variable *onCall* from the localController, is used for checks in the localController, and to keep whether or not the localController is on a call. Its main purpose is to make sure that the localController doesn't do any illegal operations, like inputting numbers in the middle of call.
We must also denote that we made some alterations to the call function in the localController, it now takes care of the entire "handshake" process between the localController and the remoteController.
Lastly we must also be aware that all the randomness was implemented in the remoteController and that the localController for the most part is "deterministic".

2.2.1 Model diagram



This model shows how we interpreted the given prompt. It represents the process by which client1 asks Local1 for the connection. How Local1 asks for the connection to Remote1, and how Remote1 proceeds in giving answers to local1 by simulating local2.
Lastly we can see at the bottom the process of ending the connection. Which consists of client1 sending a return to local1, local1 sending an end call to remote, and remote simulating the Local2 to respond validly to Local1.

2.2.2 Problems

The main problems with this project were of 2 main categories
On one hand, problems relative to design, where we had to think of the best way to implement a given feature, without losing too much of the characteristics of the design. An example of this type of problem, was when we decided that the call would be automatic after 9 digits were inputted, that the process of ending the call would be done by putting the phone back on the stand, asking the receiving party if they wanted to receive the call, among others...
On the other hand, problems relative to implementing common programming concepts using promela, take for example the way we inputted operations into the client. Originally we thought about using arrays, but this approach

wasn't fruitful because of the static size of arrays, we then thought about sending all operations in one single line, but this approach was inconvenient, at last we decided that the best solution would be to just use channels and,even thought they made the *init* much bigger,this approach was the best one and the most simple to use.