Nicolas Desfeux Student No :110477367

A model Checking Exercise : Pablo's restaurant

Newcastle University

December 2, 2011

a) Simple promela design

i) Protocol implementation

Implementing this protocol requires some global variables. We used a mtype to define the different measl available :

```
mtype = {starter ,main, desert ,drink}
```

We decided to use two channels :

```
chan order_make = [20] of { mtype, int };
chan service_channel = [20] of { mtype, int };
```

Both channels take value of type {mtype, int}, in order to couple a customer and his order. By this add, we have a way to link a meal with a customer. The first channel is use to send order from the customer to the chief. The second is use to send meal from the chief to the client.

Chief proctype The chief proctype is actually quiet simple. We created a loop that check the order_make channel. As soon as something appears in this channel, the chief can start cooking. And as he cooks very fast, the meal is added to the service_channel channel directly. We associate to this meal the id we received from the order_make channel. The loop structure is done as the chief can only cook one meal at the time.

Customer proctype The customer proctype is a bit more complicate. The customer can go throw different states. The proctype we design consist on a series of switch which leads to different actions. As for the chief, the customer have a loop, as he can order several meal, or leave. The first switch we used check if the client already make an order.

- if the client haven't order yet : Thanks to labels and goto, we jump to another do-loop, which will allow a choose between the different meal or leave.The choice will be done arbitrarily. A choice to leave will kill the protype, an order will send you back to the main do-loop, after sending the order to the order_make channel.

- if the client already make an order : The proctype goes on a state where he is waiting for his order. to do that, we used a loop that check the service_channel :

```
pick_order:
//We check the service channel until we find the good id.
do
::service_channel?plat,id_in_queue;
  if
  :: id_in_queue==_pid ->
    printf("\nCustomer %d picked meal '%e' (associated id :
         %d)\n",_pid,plat,id_in_queue);
    wait_order=false;
    goto main;
  ::else -> service_channel!plat,id_in_queue;
  fi;
od;
```

  The principle is as follow : the customer check the first meal he found in the channel, look at the id. If it's his ID on the meal, he keep it and go back to the main loop. If it's not his ID on the meal, he put it back in the channel, and look at the next one. This is a waiting loop, the customer can't leave without his meal.

Runing the model To run the model, we created an init proctype, who launch one chief proctyp, and as many customer proctyp as we need. You can change the number of customer created by changing the variable nb_customer. There is no stop condition. We are waiting for every customer to leave.

ii) Check for deadlock, and model checking

We added a end label at the loop in the chief proctyp. That allow the chief not ending at the end of the execution. By this way, we avoid something seen as a mistake by Spin, but which is not in the real life (The chief only stop waiting at the close of the restaurant, that's include a time handling that we didn't implement here). At the end of the execution, each customers have to reach their end (we can't close the restaurant if some customers are still inside !).
To check the model, we run some simulations using the Simulate / Replay function of ispin (with random and interactive progress). We are checking the correctness of the model we design by priting informations. We also do a "visual" check : are every process stop normally at the end of execution ? You can find in appendix (figure 2 page 8).
Running verification confirm that the model we design have no dead lock. We run the safety verification, focusing on invalid end-states (deadlocks). You can the result of this verification in the appendix (figure 1 page 7).

b) Adding properties

We decided to introduce new variables and new conditions to satisfy those properties.

i) Maximum order and choices constraint

Number maximum of order We added a new local variable for each customer proctype. We increment this variable each time the customer place an order. We had a control on that variable before the customer try to make on order, checking if it's lower that the maximal number of order.

Choice constraint We added four integer for each customer process. Those integers count the number of each meal you order. Then we check those variables before the customer place an order.

We also introduce in this model a new channel, that save the list of what a customer ordered. This is simple channel of mtype that we fill each time the customer pick it order. It's usefull for checking that every customer take only one instance of each meal.

ii) Assertion

We used assertion to verify this correctness requirement. Here are the assertion we used :

```
//Verification for the number of order
assert(number_of_order<5
  && order_starter<2
  && order_main<2
  && order_desert<2
  && order_drink<2);
```

We placed this assertion at the beginning of the main loop of customer proctype. That mean that this assertion is verified :

- After each order from the customer;
- After each meal pick by the customer.

We also add some check on those variables, that should make the model respect the requirements :

```
do
:: (order_starter==0) && number_of_order<4 -> order_starter++;
     order_make!starter , _pid;goto makeOrder;
:: (order_main==0) && number_of_order<4 -> order_main++;
     order_make!main , _pid;goto makeOrder;
```

```
:: (order_desert==0) && number_of_order<4 -> order_desert++;
    order_make!desert,_pid;goto makeOrder;
:: (order_drink==0) && number_of_order<4 -> order_drink++;
    order_make!drink,_pid;goto makeOrder;
::!wait_order -> goto leave;
od;
```

We simply check those variable before make an order, and update them if an order is about to be done. We run a verification using Spin, focusing on assertion violations, and the result come back positive. The result of this verification can be find on the appendix (figure 3 page 9). To be sure of the assertions we make, we also make some test on slightly different models :

- Changing the assertion : We modify the values in the assertions, and watch the reaction of the system. In our case, the controls we set up seems effecient as value changes in the assertions leads to an assertion violation (if you reduce the value in the assertion, increasing them is meaningless)
- Changing variables control : We change values in the do loop (allow more order, or several times the same meal), and watch the reaction of the system. In our case, those modifications leads to an assertion violation.

c) Additional possibility

i) Thinking state

In our model, the add of a thinking state can be see as a new choice for the customer. After he choices a meal, we just have to had a new loop :

```
makeOrder:
think:
do
  ::goto think;
  ::goto makeOrder_or_leave;
  ::break;
od;
order_make!plat , _pid;
```

makeOrder is the label reached when the customer choose a meal to order. Here we offer three choices : stay in the think loop, change your mind, or keep your choice. No changes are needed on the channels, as the order is make only if the customer leave the think loop.

ii) Checking for deadlock

We check that model against deadlock, and no one was found. The result of the checking can be seen in the appendix (figure 4 page10). Simulations on this model also lead to a good respect of the new property.

Contents

List of Figures

Figure 1: Verification on simple model implementation

```
verification result:
spin -a  Pablo-a.pml
gcc-4 -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan
    pan.c
./pan -m10000  -A
Pid: 4596
error: max search depth too small

(Spin Version 6.1.0 -- 4 May 2011)
  + Partial Order Reduction

Full statespace search for:
  never claim           - (not selected)
  assertion violations  - (disabled by -A flag)
  cycle checks          - (disabled by -DSAFETY)
  invalid end states    +

State-vector 368 byte, depth reached 9999, errors: 0
    55727 states, stored
    44294 states, matched
   100021 transitions (= stored+matched)
        0 atomic steps
hash conflicts:      1431 (resolved)

   22.754 memory usage (Mbyte)

unreached in proctype chief
  Pablo-a.pml:25, state 9, "-end-"
  (1 of 9 states)
unreached in proctype customer
  (0 of 38 states)
unreached in init
  (0 of 10 states)

pan: elapsed time 0.096 seconds
No errors found -- did you verify all claims?
```

Figure 2: Sample of random simulation on simple model implementation

```
Chief proctype launched.
  3:  proc  1 (chief) Pablo-a.pml:16 (state 1)  [printf('\\nChief proctype launched. \\n')]
  5:  proc  0 (:init:) Pablo-a.pml:79 (state 7) [((nb_customer>0))]
Starting customer with pid 2
  6:  proc  0 (:init:) creates proc  2 (customer)
  6:  proc  0 (:init:) Pablo-a.pml:80 (state 3) [(run customer())]
  7:  proc  0 (:init:) Pablo-a.pml:80 (state 4) [nb_customer = (nb_customer-1)]
  9:  proc  0 (:init:) Pablo-a.pml:79 (state 7) [((nb_customer>0))]
Starting customer with pid 3
 10:  proc  0 (:init:) creates proc  3 (customer)
 10:  proc  0 (:init:) Pablo-a.pml:80 (state 3) [(run customer())]
 11:  proc  0 (:init:) Pablo-a.pml:80 (state 4) [nb_customer = (nb_customer-1)]
I'm the customer 2
 12:  proc  2 (customer) Pablo-a.pml:35 (state 1) [printf('\\nI'm the customer %d\\n\\n',_pid)]
I'm the customer 3
 13:  proc  3 (customer) Pablo-a.pml:35 (state 1) [printf('\\nI'm the customer %d\\n\\n',_pid)]
 16:  proc  2 (customer) Pablo-a.pml:39 (state 6) [else]
 18:  proc  2 (customer) Pablo-a.pml:41 (state 5) [goto makeOrder_or_leave]
I will maybe make an order or leave (customer 2)
 19:  proc  2 (customer) Pablo-a.pml:58 (state 21)  [printf('\\nI will maybe make an order or leave (customer %d)\\n',_pid)]
 20:  proc  3 (customer) Pablo-a.pml:39 (state 6) [else]
 22:  proc  3 (customer) Pablo-a.pml:41 (state 5) [goto makeOrder_or_leave]
 23:  proc  2 (customer) Pablo-a.pml:59 (state 32)  [order_make!drink,_pid]
 24:  proc  2 (customer) Pablo-a.pml:63 (state 29)  [goto makeOrder]
I will maybe make an order or leave (customer 3)
 25:  proc  3 (customer) Pablo-a.pml:58 (state 21)  [printf('\\nI will maybe make an order or leave (customer %d)\\n',_pid)]
 26:  proc  1 (chief) Pablo-a.pml:19 (state -)  [1?drink,2]
 26:  proc  1 (chief) Pablo-a.pml:18 (state 6)  [(order_make?[plat,id])]
 27:  proc  1 (chief) Pablo-a.pml:20 (state 3)  [order_make?plat,id]
 28:  proc  0 (:init:) Pablo-a.pml:79 (state 7) [((nb_customer==0))]
 30:  proc  2 (customer) Pablo-a.pml:67 (state 35)  [wait_order = 1]
 31:  proc  2 (customer) Pablo-a.pml:68 (state 36)  [goto main]
 32:  proc  3 (customer) Pablo-a.pml:59 (state 32)  [order_make!main,_pid]
 34:  proc  3 (customer) Pablo-a.pml:61 (state 25)  [goto makeOrder]
 35:  proc  2 (customer) Pablo-a.pml:39 (state 6) [(wait_order)]
New meal in the queue : drink for customer[2].
```

Figure 3: Verification with constraint choices and limited order

```
verification result:
spin -a  Pablo-b.pml
gcc-4 -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan
    pan.c
./pan -m10000
Pid: 4592
Depth=     3391 States=    1e+06 Transitions= 1.69e+06 Memory=
    462.890 t=      2.46 R=   4e+05
Depth=     3391 States=    2e+06 Transitions= 3.36e+06 Memory=
    922.656 t=      6.19 R=   3e+05
pan: resizing hashtable to -w21..  done
pan: reached -DMEMLIM bound
  1.07366e+09 bytes used
  102400 bytes more needed
  1.07374e+09 bytes limit
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
  -DMA=590   # better/slower compression, or
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation

(Spin Version 6.1.0 -- 4 May 2011)
Warning: Search not completed
  + Partial Order Reduction

Full statespace search for:
  never claim           - (not selected)
  assertion violations  +
  cycle checks          - (disabled by -DSAFETY)
  invalid end states  +

State-vector 590 byte, depth reached 3391, errors: 0
  2202866 states, stored
  1534198 states, matched
  3737064 transitions (= stored+matched)
        0 atomic steps
hash conflicts:   2923642 (resolved)

 1023.918 memory usage (Mbyte)


pan: elapsed time 7.38 seconds
No errors found -- did you verify all claims?
```

Figure 4: Verification for model we think state

```
spin -a  Pablo-c.pml
gcc-4 -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan
    pan.c
./pan -m10000
Pid: 4248
Depth=      264 States=    1e+06 Transitions= 1.39e+06 Memory=
    426.855 t=      2.12 R=   5e+05
Depth=      264 States=    2e+06 Transitions= 2.78e+06 Memory=
    851.465 t=      5.59 R=   4e+05
pan: resizing hashtable to -w21..  done
pan: reached -DMEMLIM bound
  1.07366e+09 bytes used
  102400 bytes more needed
  1.07374e+09 bytes limit
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
  -DMA=430   # better/slower compression, or
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation

(Spin Version 6.1.0 -- 4 May 2011)
Warning: Search not completed
  + Partial Order Reduction

Full statespace search for:
  never claim           - (not selected)
  assertion violations  +
  cycle checks          - (disabled by -DSAFETY)
  invalid end states    +

State-vector 430 byte, depth reached 264, errors: 0
  2388507 states, stored
   942934 states, matched
  3331441 transitions (= stored+matched)
        0 atomic steps
hash conflicts:   2484941 (resolved)

 1023.918 memory usage (Mbyte)


pan: elapsed time 7.32 seconds
No errors found -- did you verify all claims?
```