

Report – MRS Assessed Exercise 2465714a

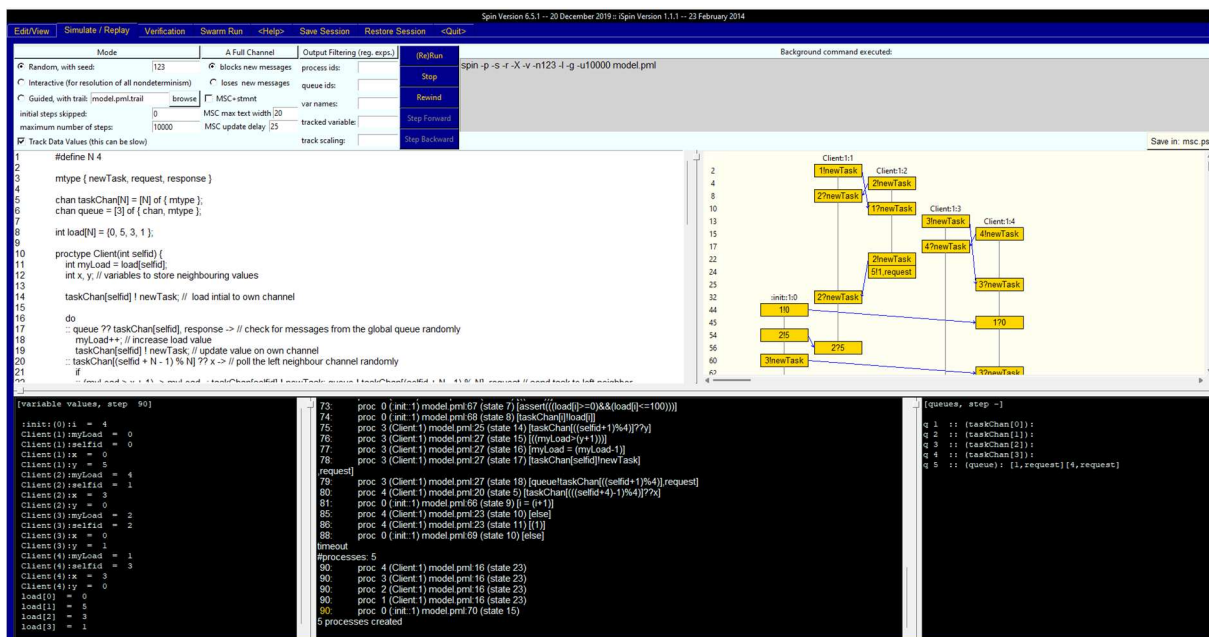
Problem:

The task in hand was to create a promela model which simulates a load balancer protocol which can deal with four client processes with their corresponding states. The balance is maintained by ensuring each client keeps a balanced load with their neighbours. It does this by comparing current load states then redistributing appropriately if there are differences greater than one. Each client process interacts with their adjacent clients or neighbours via channels, receiving load data.

Model:

When an imbalance is detected, a task is sent to the neighbour with the lower current load, in the comparison which detected the imbalance. To ensure balance the model must ensure that the difference between neighbour's loads remains within 1 and the global queue handles messages correctly. The model assumes messages between the client processes and the global queue are deterministic and are always received and sent. It is also assumed that load adjustments are made instantly and that loads values are integers within a certain range which can be adjusted.

Here are screenshots of the model being tests and verified. (Screenshots will also be attached if it is difficult to view here.)



Spin Version 6.5.1 -- 20 December 2019; Spin Version 5.1.3 -- 23 February 2014

Safety	Storage Mode	Search Mode
<input checked="" type="checkbox"/> safety <input checked="" type="checkbox"/> invalid endstates (deadlock) <input checked="" type="checkbox"/> assertion violations <input type="checkbox"/> x/x/s assertions	<input checked="" type="checkbox"/> exhaustive <input type="checkbox"/> minimized automata (slow) <input type="checkbox"/> collapse compression <input type="checkbox"/> hash-compact <input type="checkbox"/> bitstate/supertrace	<input checked="" type="checkbox"/> depth-first search <input type="checkbox"/> partial order reduction <input type="checkbox"/> bounded context switching <input type="checkbox"/> with bound: 0 <input type="checkbox"/> iterative search for short trail <input type="checkbox"/> breadth-first search <input type="checkbox"/> partial order reduction <input checked="" type="checkbox"/> report unreachable code
<input checked="" type="checkbox"/> non-progress cycles <input type="checkbox"/> acceptance cycles <input type="checkbox"/> enforce weak fairness constraint	<input type="checkbox"/> do not use a never claim or B property <input type="checkbox"/> use claim <input type="text"/> claim name (opt)	<input type="button" value="Show Error Trapping Options"/> <input type="button" value="Show Advanced Parameter Settings"/>
<input type="button" value="Run"/> <input type="button" value="Stop"/>	<input type="button" value="Save Results"/> <input type="text"/> pan.out	

```

1 #define N 4
2
3 ntype ( newTask, request, response )
4
5 chan taskChan[N] = [N] of ( ntype );
6 chan queue = [3] of ( chan, ntype );
7
8 int load[N] = {0, 5, 3, 1};
9
10 proctype Client(int selfid) {
11   int myLoad = load[selfid];
12   int x, y; // variables to store neighbouring values
13
14   taskChan[selfid] ! newTask; // load initial to own channel
15
16   do
17     queue ?? taskChan[selfid].response -> // check for messages from the global queue randomly
18     myLoad++; // increase load value
19     taskChan[selfid] ! newTask; // update value on own channel
20   taskChan[selfid + N - 1] % N ?? x -> // poll the left neighbour channel randomly
21   if
22     ! (myLoad > x + 1) -> myLoad--; taskChan[selfid] ! newTask; queue ! taskChan[selfid + N - 1] % N, request // send
23   else -> skip
24   fi
25   taskChan[selfid + 1] % N ?? y -> // poll the right neighbour channel randomly
26   if
27     ! (myLoad > y + 1) -> myLoad--; taskChan[selfid] ! newTask; queue ! taskChan[selfid + 1] % N, request // send to
28   else -> skip
29   fi
30 od
31 }
32
33
34 never {
35   // LTL property, to ensure the difference between adjacent loads remains within 1
36   // Iterate through each pair of neighbour clients and check the load difference
37   do
38     (1) ->
39     // Ensure that load values stay within their specified ranges

```

model pm1 67, state 9 "i = (i+1)"
model pm1 67, state 12 "j = (j+1)"
model pm1 67, state 12, "else"
model pm1 71, state 15, "-end"
(7 of 15 states)

pan: elapsed time 0 seconds
No errors found - did you verify all claims?
spin -a model.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNP -w -o pan pan.c
pan -m10000 -l
Pid: 23400
Depth= 125 States= 1e+06 Transitions= 1.97e+06 Memory= 208.905 t= 0.348 R= 3e+06
Depth= 125 States= 2e+06 Transitions= 3.95e+06 Memory= 288.081 t= 0.707 R= 3e+06
Depth= 125 States= 3e+06 Transitions= 6.3e+06 Memory= 398.065 t= 1.21 R= 2e+06

(Spin Version 6.5.1 -- 20 December 2019)
+ Partial Order Reduction

Full statespace search for:
never claim + (np...)
assertion violations + (if within scope of claim)
non-progress cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 164 byte, depth reached 125, errors: 0
2639795 states, stored (3.95969e+06 visited)
5191782 states, matched
9151484 transitions (= visited+matched)
0 atomic steps
hash conflicts: 208568 (resolved)

Stats on memory usage (in Megabytes):
483.221 equivalent memory usage for states (stored*(State-vector + overhead))
423.884 actual memory usage for states (compression 91.51%)
state-vector as stored = 148 byte + 20 byte overhead
128.000 memory used for hash table (-w24)
0.534 memory used for DFS stack (-m10000)
551.972 total actual memory usage

```

C:\Workspace\Promela\AE>spin -a model.pml

C:\Workspace\Promela\AE>spin model.pml
warning: never claim not used in random simulation
timeout
#processes: 5

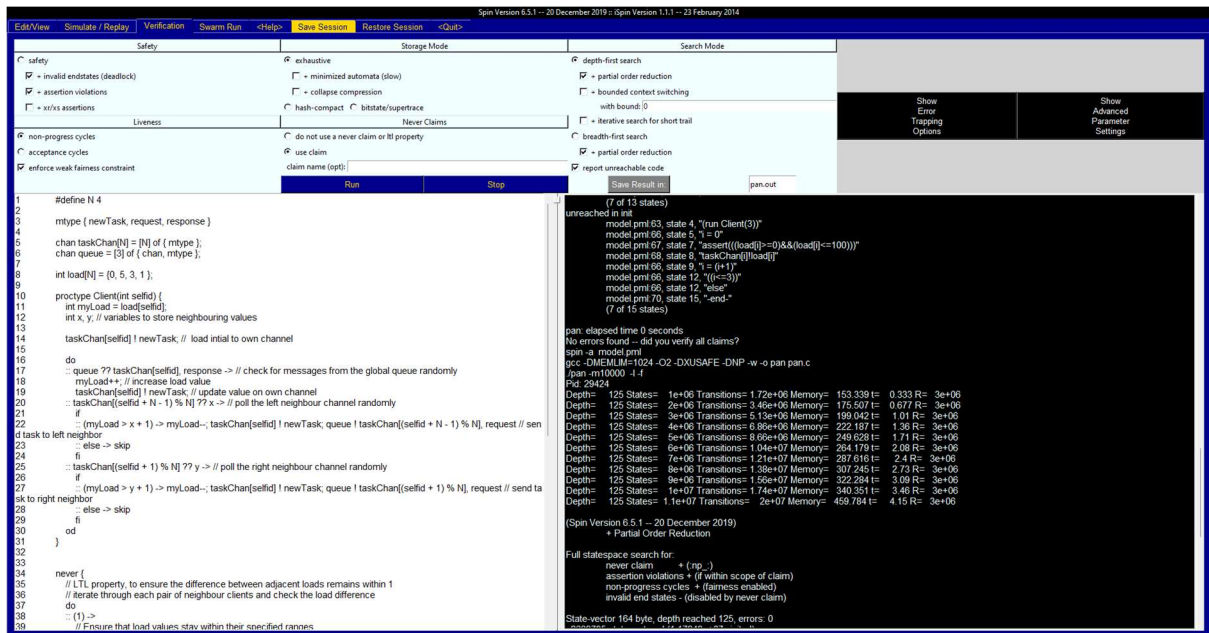
queue 1 (taskChan[0]):
queue 2 (taskChan[1]):
queue 3 (taskChan[2]):
queue 4 (taskChan[3]):
queue 5 (queue): [1,request]
load[0] = 0
load[1] = 5
load[2] = 3
load[3] = 1

83: proc 4 (Client:1) model.pml:16 (state 23)
83: proc 3 (Client:1) model.pml:16 (state 23)
83: proc 2 (Client:1) model.pml:16 (state 23)
83: proc 1 (Client:1) model.pml:16 (state 23)
83: proc 0 (:init::1) model.pml:71 (state 15) <valid end state>
5 processes created

C:\Workspace\Promela\AE>

```

Here you can see that my specification will satisfy the property ϕ when Weak Fairness is applied. (Screenshots will also be attached if it is difficult to view here.)



We will consider the following scenario/execution path. In order to show that ϕ will not hold when weak fairness is not applied.

Load[0] = 9, Load[0] = 5, Load[0] = 2, Load[0] = 1

Step 1: Client[0] is selected for checking and decreases its load by 1 as there is an imbalance

Step 2: As there is no weak fairness, Client[0] is selected again and its load decreases once more

Step 3: Client[0] will continue to be chosen for execution. This results in the neighbours not being given the chance to balance their load.

This leads to a persistent imbalance, and the load of Client 0 will become much lower than its neighbours as they increase, this violates the property ϕ (LTL) property. Thus, the system fails to achieve load balance as other clients remain at higher values. This example shows that without weak fairness a single high load client can dominate the execution of the protocol leading to a violation of ϕ , weak fairness ensures that each client will have a chance to be checked and balance maintained.