Coursework 1: PROMELA and SPIN

Overview

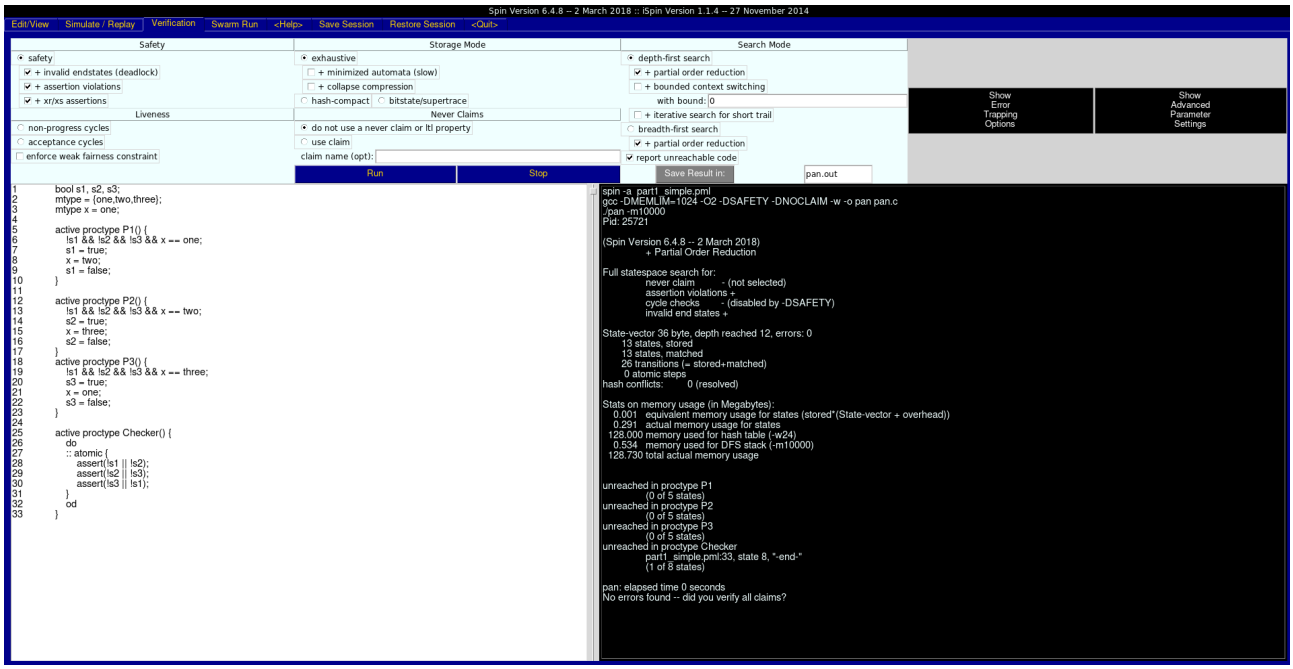
The assignment asks to write three PROMELA specifications that verify mutual exclusion and process fairness. I have completed all three parts. In addition, I have written several specifications for the first task, 2 specifications for the second. I have also looked into one of the points of part 4.

Part 1

The first task is about mutual exclusion between three processes that each have a ‘critical section’. No two processes should be in their critical sections at the same time. I have written four PROMELA specifications.
First, the one that satisfies the assignment to the letter is ‘part1_simple.pml’. It runs three different processes that each wait for their turn to enter their critical sections. An invariant ensures that no two processes are in their critical sections at the same time.
Second, ‘part1_simple_infinite.pml’ is the same, except it lets the processes run forever. When a process leaves its critical section, it goes back to waiting for its turn.
Third, ‘part1_array.pml’ defines an array of processes, rather than three separate ones. The array is of size 4. The process with index 0 terminates immediately. Thus, the three remaining processes are indexed 1, 2, and 3, as required by the assignment. Again, they each wait for their turn to enter their critical sections, and an invariant ensures no two processes enter their critical sections at the same time.
Finally, ‘part1_array_infinite.pml’ is the same as ‘part1_array.pml’, except the processes run forver. Process 0 terminates early, but processes 1, 2, and 3 return to waiting their turns after leaving their critical sections.

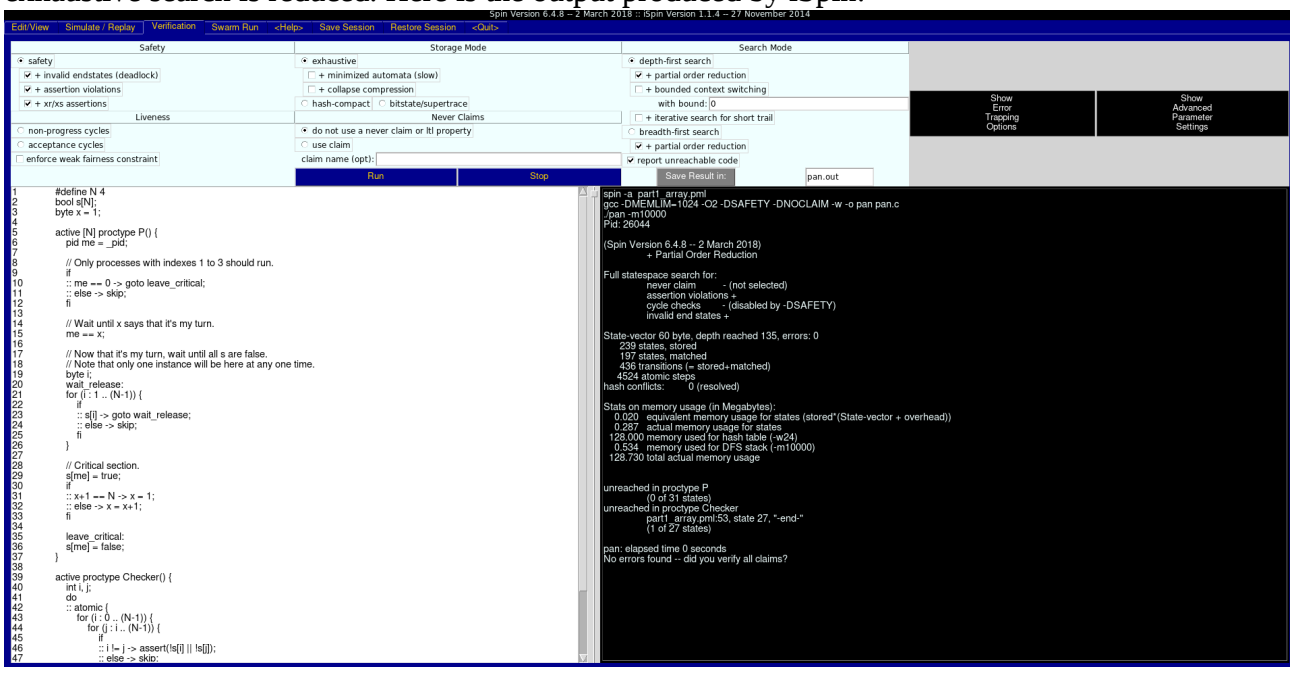
Both the ‘simple’ and the ‘array’ variations satisfy the mutual exclusion property.

The ‘simple’ variant does that by using PROMELA’s blocking on boolean statements. A process blocks until its boolean condition is met. The variable ‘x’ determines which process can have a go next, so only one of the conditions can be true at the same time. After a process passes the condition, it first changes its state variable (s_i) to true and then changes the value of ‘x’. Thus, all conditions become false (because one of the states is true). Changing the turn after that does not allow a process to enter its critical section. At the end of its critical section, a process changes its state back to false. Now all states are false, and ‘x’ determines which process can go next. The invariant ‘Checker’ has three asserts that are run in an infinite loop. They check that no two states are true at the same time. Running verification in Spin ensures that all asserts pass in all states of the program. Spin verification goes through each possible trace of execution. The asserts are put inside an atomic block, so that they are always executed together and thus reduce the search states. Here is the output produced by iSpin:



As can be seen, no errors are found, no assertions are violated.

The ‘array’ variant also uses PROMELA’s blocking on boolean statements to satisfy the mutual exclusion property. A process blocks until it is its turn to enter the critical section. After that it waits until all states are false. In the critical section, a process first changes the state, and then changes the value of ‘x’. Again, ‘x’ ensures that no two processes are waiting for the states to be false. At the end of its critical section, a process sets its state to false. In the ‘array’ variant, the ‘Checker’ invariant loops through all pairs of states, ensuring no two states are true at the same time. This is equivalent to having only one true state. Similar to the approach in ‘simple’, this PROMELA specification goes through all asserts in an atomic block. This way the search space for Spin’s exhaustive search is reduced. Here is the output produced by iSpin:



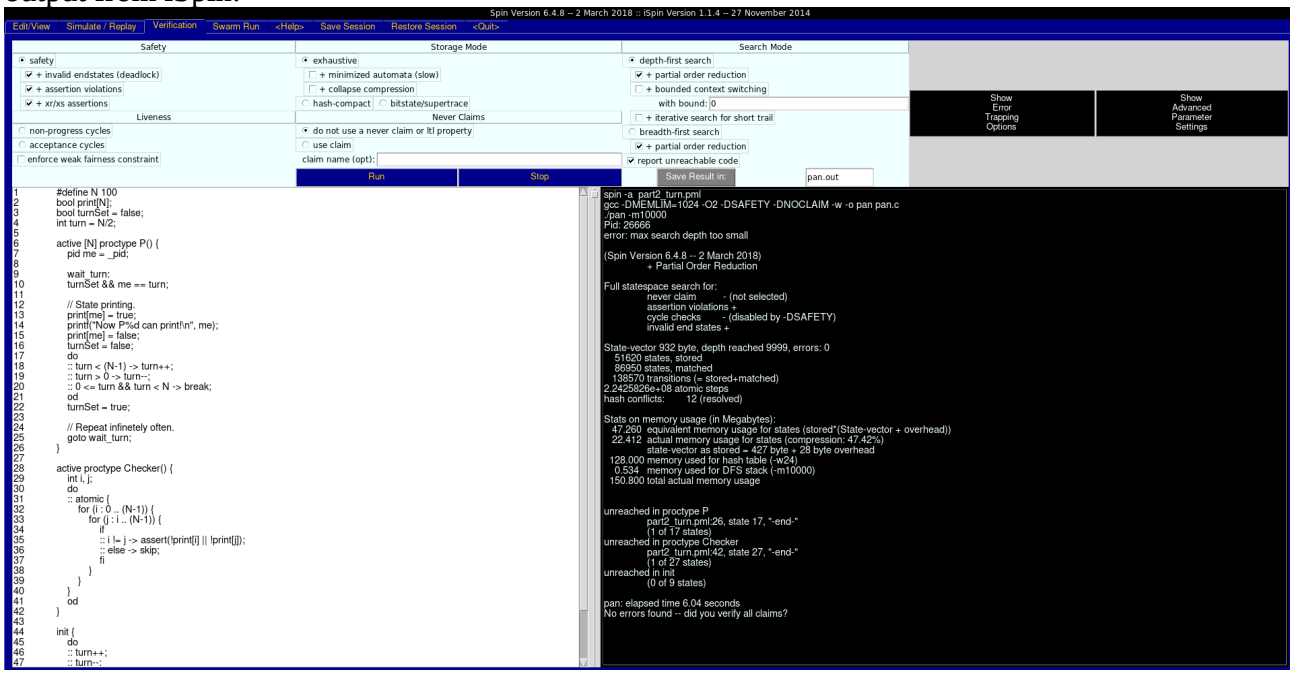
The ‘infinite’ variants differ only in the labels that they use to return to the start of the process. The output produced by iSpin for them is the same.

Part 2

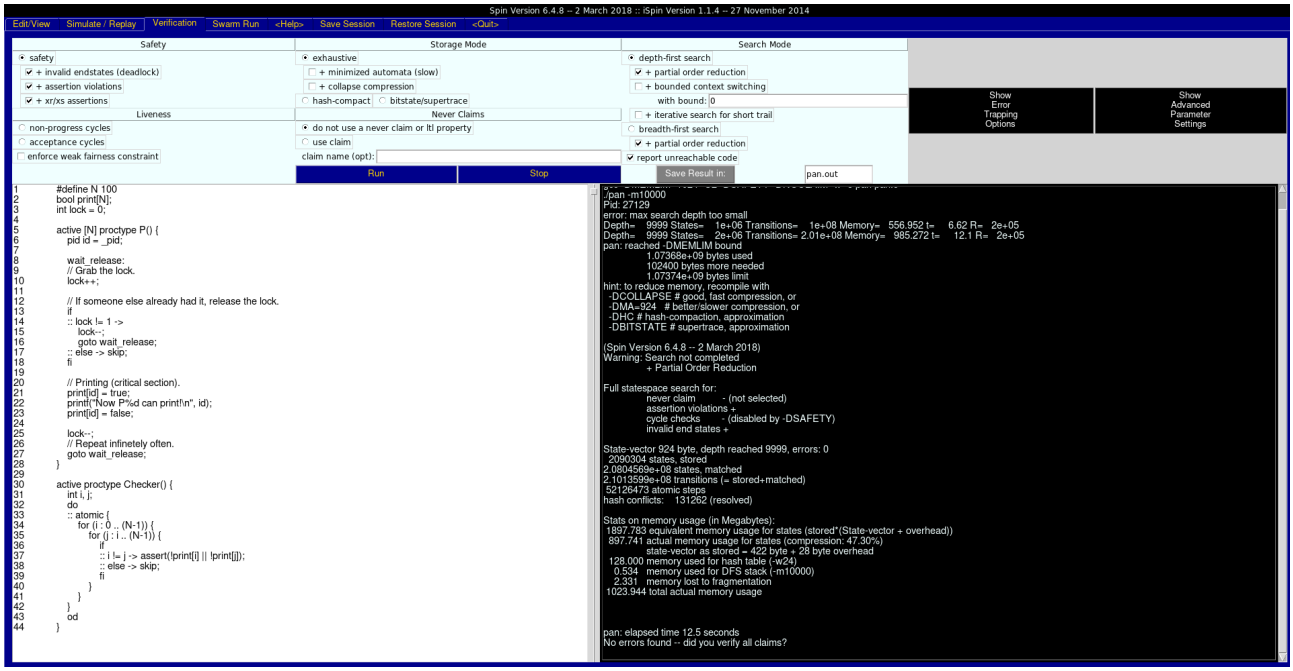
The second task is about mutual exclusion between an array of processes. No two processes should print at the same time. I have written two PROMELA specifications called ‘part2_lock.pml’ and ‘part2_turn.pml’.

In both of these, an invariant ensures that no two processes are printing at the same time. The invariant is almost the same as the one from the ‘array’ variant of part 1. The only difference is the iterated array. Again, an atomic block reduces the states for Spin’s exhaustive search verification. The verification will pass even without the atomic block, as Spin will exhaust the cases where the full ‘Checker’ is run after each statement of the other processes.

The ‘turn’ variant is similar to how part 1 utilized a variable to keep track of which process can enter the critical section. In this PROMELA specification, ‘turn’ is an integer that can take a random value between 0 and N (the number of processes that want to print). The value is set using PROMELA’s non-determinism in do-loops. The loop that changes ‘turn’ will increment it and decrement it a random number of times before moving on. A boolean variable ‘turnSet’ is used to determine whether or not ‘turn’ is currently being changed. The value of ‘turn’ is set once in the init process, and then every time a process finishes printing.
A process sets its respective ‘print’ value to true before printing, and then to false after printing but before changing the value of ‘turn’.
The processes that wish to print wait for their turn via PROMELA’s blocking on boolean statements. A process blocks until ‘turnSet’ is true and the value of turn is its id. This way no two processes can be printing at the same time, satisfying the mutual exclusion property. Here is the output from iSpin:



The ‘lock’ variant uses an integer variable called ‘lock’ to prevent more than one process from printing. When a process wishes to print, it increments the value of ‘lock’ by one. If it is the only process that has done that, it can then print. Similar to the ‘turn’ variant, a process will first set its respective value in the ‘print’ array to true, then print, and then set that value back to false. After that it will release the ‘lock’.
Race conditions (where multiple processes increment the ‘lock’ at the same time), are handled using a simple back-off procedure. If more than one process is holding the ‘lock’, than decrement the ‘lock’ and go back up. Spin can support up to 256 processes. This means that ‘lock’ will never overflow (the maximum integer value is 2^{31}-1). The ‘lock’ ensures that no two process can print at the same time and that is verified by the invariant. Here is the output from iSpin:

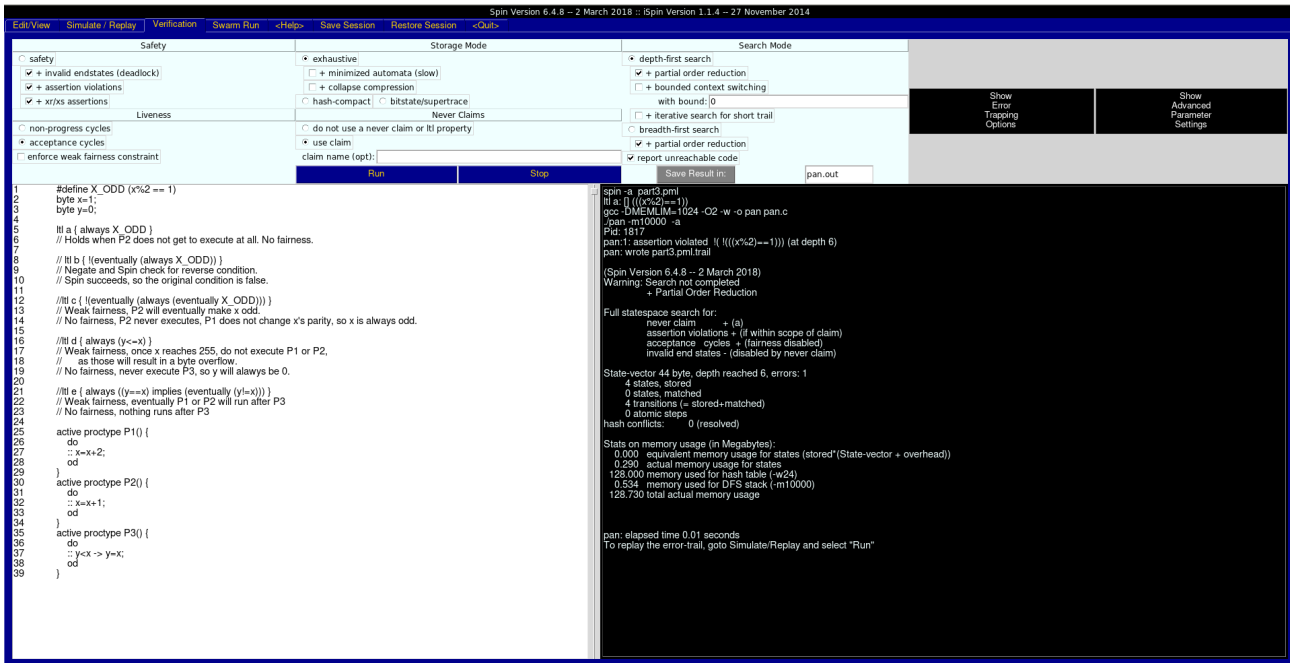


Due to the race condition, there are a lot more possible states for Spin to explore. The output above shows how Spin increased its memory usage and maximum depth for the iterations.

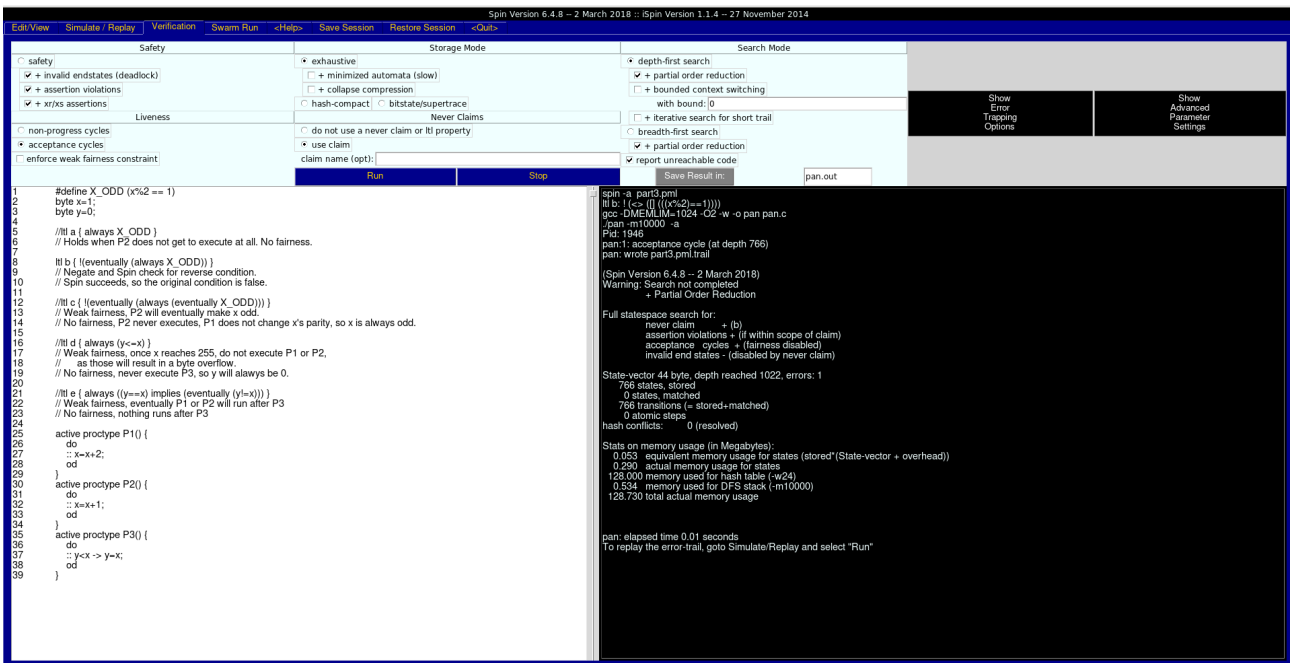
Part 3

The third task is about Linear Time Logic (LTL) and reflecting on when the formulae hold. The PROMELA specification has three processes. Two increment the value of ‘x’ and one sets the value of ‘y’ to that of ‘x’. Spin supports weak fairness. In this PROMELA specification, all three processes are enabled at all times. Thus, it follows that each process will get its turn infinitely often.

The first statement is “x is always odd”. In LTL, it is written as “□ (x%2 == 1)”. This statement holds only when P2 never gets to run. That is the case of no fairness. Spin supports weak fairness, therefore P2 is guaranteed to run infinitely often from a certain time onwards. When it executes, it will increment ‘x’ by 1 and thereby change its parity. So, this statement does not hold in weak fairness. Here’s the result of running Spin with this LTL:

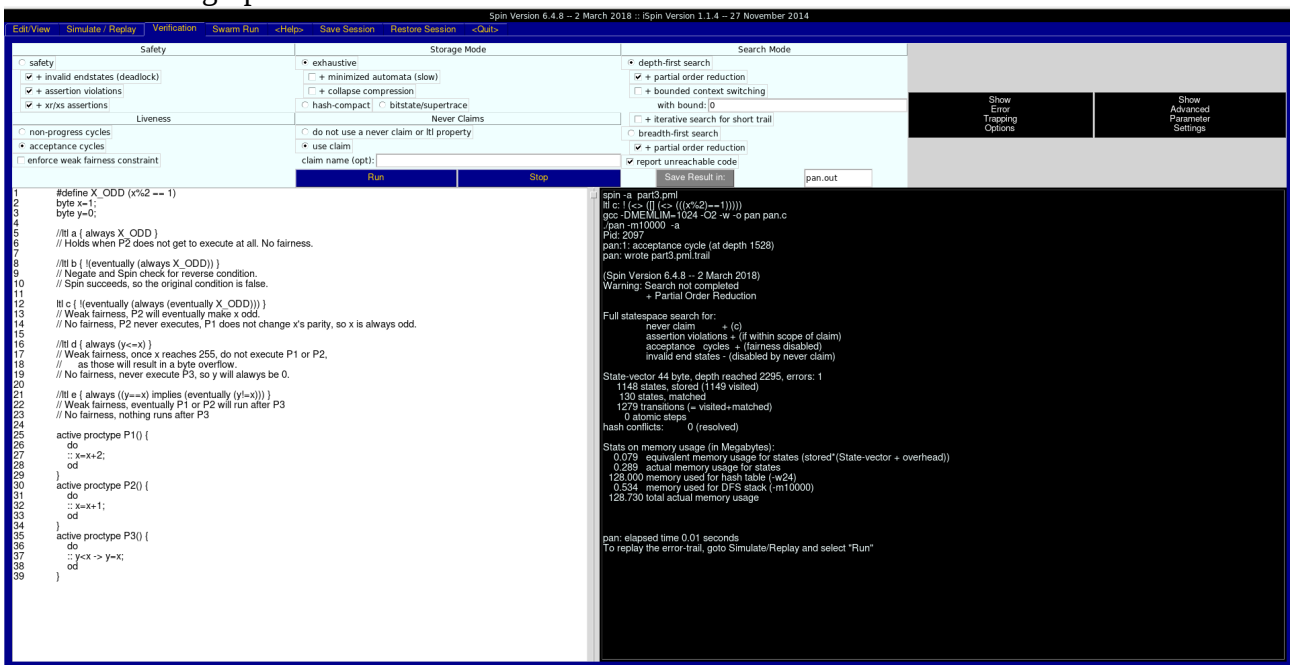


The second statement is “It is possible that from a certain point onwards x is always odd”. The way to check statements of the type “It is possible that A” is to negate A and verify that ¬A fails. If Spin finds a case where ¬A is false, then in that case A is true and thus the statement is true. If ¬A is true in all executions, than A is never possible. In out statement, A is “from a certain point onwards x is always odd”. In LTL, it is written as “◊□(x%2==1)”. Given weak fairness, P2 will eventually get a turn and thus change the parity of ‘x’. So, the parity of ‘x’ will never remain constant. Similar to the first statement, this one would be true if P2 does not get its turn (no fairness). Spin fails to verify the negation of A because it is not possible to simulate infinite execution. It simulates the case where

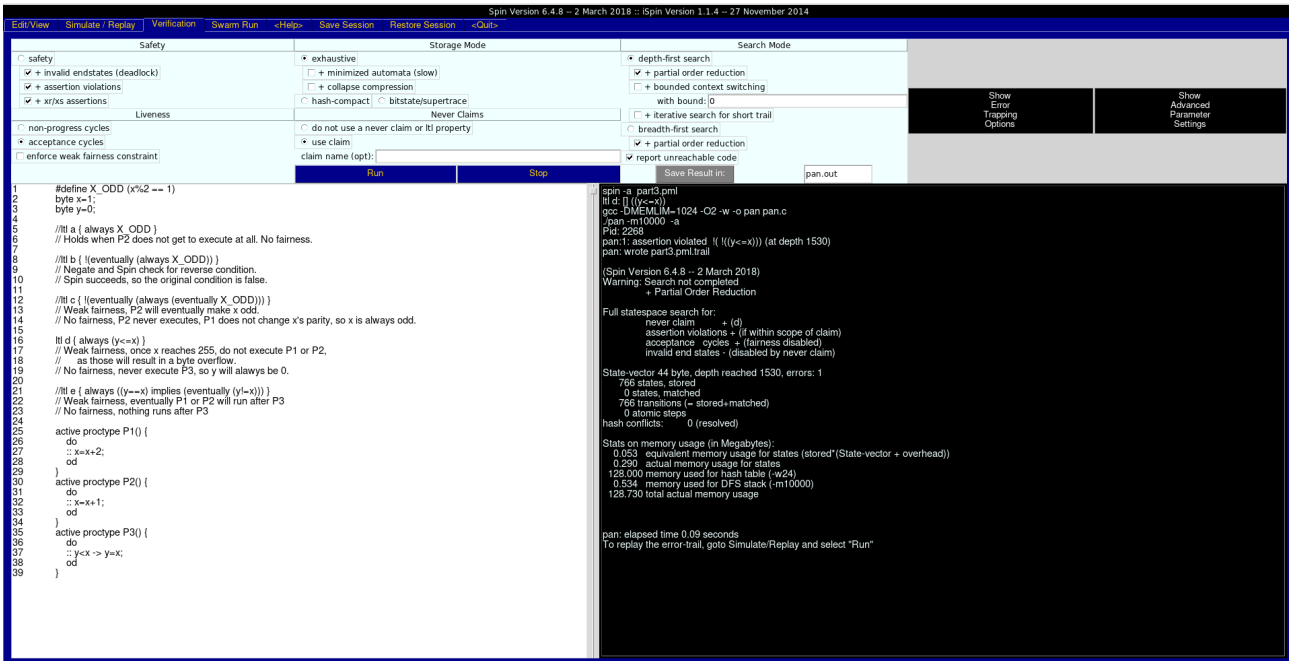


only P1 is executed (Spin trail and trail execution included in submission). Spin output above.

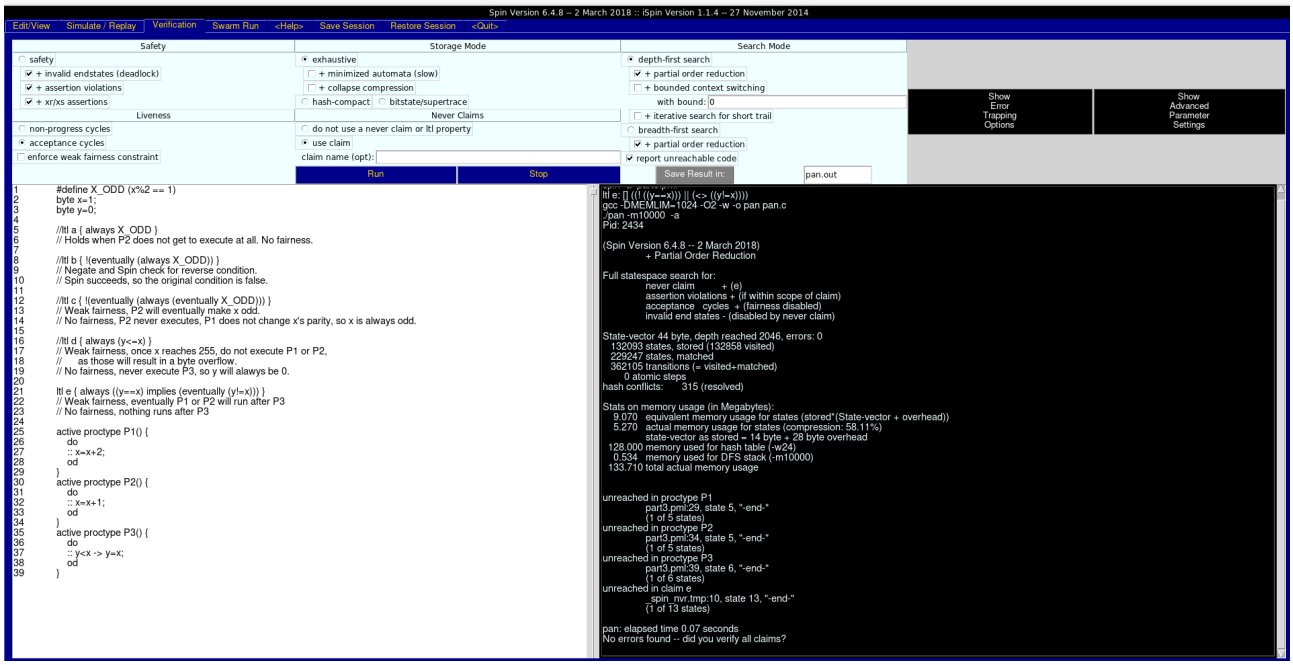
The third statement is “It is possible that from a certain point onwards x is indefinitely often odd”. Similar to the second statement, we need to verify the negation of A. In this case A is “from a certain point onwards x is indefinitely often odd”. In LTL, it is written as “◊□◊(x%2==1)”. Spin fails to verify the negation of A. This means that A is possible. Given weak fairness, P2 will eventually get a turn and thus change the parity of ‘x’. This will happen infinitely often as P2 is executed infinitely often. Thus ‘x’ will always eventually by odd. With no fairness, P2 will not get its turn and so the statement will be true if and only if ‘x’ is odd after P2’s last execution. Here’s the result of running Spin with this LTL:



The fourth statement is “It is always true that y ≤ x”. In LTL, it is written as “□ (y <= x)”. This statement does not hold in weak fairness. The variable ‘x’ is of type byte. The largest value it can have is 255. Processes P1 and P2 can both attempt to increase ‘x’ beyond that which will cause it to overflow to 0. In that case, the LTL can only hold if ‘y’ is never changed. The value of ‘y’ gets changed in P3. So, by weak fairness there will be a case where ‘y’ is set to a value greater than 0 and then ‘x’ will overflow to 0. With no fairness, P3 will never get a turn, and ‘y’ will remain 0, and then the property holds. Here’s the result of running Spin with this LTL:



The fifth statement is “It is always true that when y = x it follows that at some point y ≠ x.”. In LTL, it is written as “□ ((y=x) → (◊ y!=x))”. This statement holds in weak fairness. The process P3 sets the value of ‘y’ to that of ‘x’. Process P1 and P2 change the value of ‘x’ without affecting that of ‘y’. This will happen eventually, as weak fairness guaranties that P1 and P2 will get their turns. With no fairness, P1 and P2 will never get a turn, and then the property does not hold. Here’s the result of running Spin with this LTL:



Part 4

The fourth task is about detailing different aspects of Spin as a verification tool. I have focused on its current limitations. Several sources[1, 2, 3] have pointed out a couple of limitations that apply to most software verification tools, including Spin. The first one is “state space explosion”. Real systems often have unlimited number of states and huge ranges for variables’ values. In PROMELA, processes can be interleaved in any way, and Spin stores explicit state. It would, therefore, need infinite memory to explore the state space of an infinite system. The study [1] recommends looking into other formal verification methods, whereas [2] suggests simplifying the model. The latter leads to the second common problem of verification tools.

Creating an accurate PROMELA model of a real system is often impossible. Programs often have multiple variables of different types and PROMELA is limited to numbers of different ranges. Furthermore, there is no guarantee that a verified model will bring information about the real system[2].

A limitation specific to Spin is lack of real-time capabilities. It is pointed out in [1] and [2], that Spin does not natively support checking repetition cycles between events. In [3], it is mentioned that modifications and extensions of Spin can achieve this.

A final, small, Spin-specific limitation is that number of simultaneously running processes is limited to 255[4]. I have not been able to find out why the maximum is not 256. If we use 8 bits for process ids, we have a total of 256 possible ids. Process id 0 is not taken by a system process because creating 10 processes assigns ids from 0 to 9. I suppose the id 255 is used for a process that manages the others but I have not been able to find any confirmation.

On top of these limitations, I have also read a bit about how Spin handles LTL formulae. It first transforms them into never claims. Spin then represents the never claim as a Büchi automata[5]. A Büchi automata accepts an input sequence if it visits at least one of the final states infinitely often. Such an automata is designed to take an infinite input sequence. Thus, a never claim succeeds if it reaches the end of its input or if it passes through a final (aka ‘acceptance’) state infinitely often.

Bibliography

[1] A Survey of Tools for Model Checking and Model-Based Development
Elisabeth A. Strunk, M. Anthony Aiello, John C. Knight, Eds.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.436.1944&rep=rep1&type=pdf

[2] Applications for the Spin Model Checker
Ville R. Koskinen, Juha Plosila
http://www.tucs.fi/publications/attachment.php?fname=TR782.pdf
Note: this link initiates a PDF download.

[3] The Model Checker Spin
Gerard J. Holzmann
http://spinroot.com/spin/Doc/ieee97.pdf

[4] Spin Manual
http://spinroot.com/spin/Man/active.html

[5] Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker
Diego LatellaIstvan MajzikMieke Massink
https://link.springer.com/article/10.1007/s001659970003