Promela and Spin

160004048

October 2019

1 Introduction

2 Part 1

The task requires to write a Promela specification for the given mutual exclusion algorithm. We were also asked to prove whether this algorithm satisfies the mutual exclusion property.

2.1 Implementation

I decided to write 3 separate processes for P1, P2 and P3. The problem states that 'Pi can only enter the critical section if x = i and all other variables are false'. To achieve this, 3 processes contain a loop which checks all 4 conditions (3 states to be false and the value of x to be i for Pi).

From the program description, it seems like first, the value of x has to be changed and only then the value of the state is changed. In this case, the algorithm is not mutually exclusive. Proof of this can be seen in 1.

The end state is reached in 'never' claim. The 'never' claim checks if at every step, none of the pars of two states are both true. Therefore, only one or none state can be true at a time. The Promela specification in this case is written in the file called 'part1_not_mutually_exclusive.pml'. Logically, it is seen that the algorithm is not mutually exclusive e.g. when the program enters it's critical state in P1 and x is changed to be 2, then P2 can enter it's critical state since all the states are still 'false' and x is now equal to 2. Then if state1 and state2 are both made to be true, the *never* claim will break.

On the other hand, if the state i is changed to true in Pi before changing the value of x, then the algorithm is mutually exclusive. Logically, given the value of x, only one of the statements in P1, P2 and P3 is true. Therefore the value of one of the states is changed to 'true' straight away. Since one state is now true, the only line that can be executed next is the next line in the same process. Therefore, x value is changed. Since one state is still 'true', the only line that can be executed next is the line in the same process which changes the state value to 'false' again. The proof of this can be seen when looking at the outcome of spin in 2. The Promela specification in this case is written in the file called *part1_mutually_exclusive.pml*.

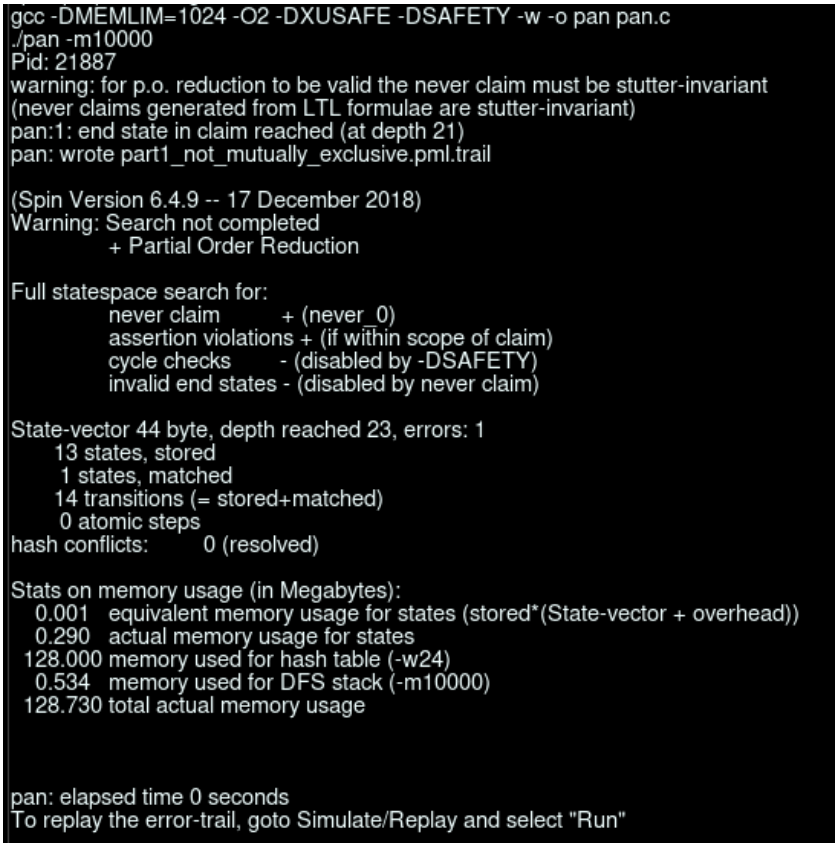

Figure 1: Part 1, *never* claim reaches the end

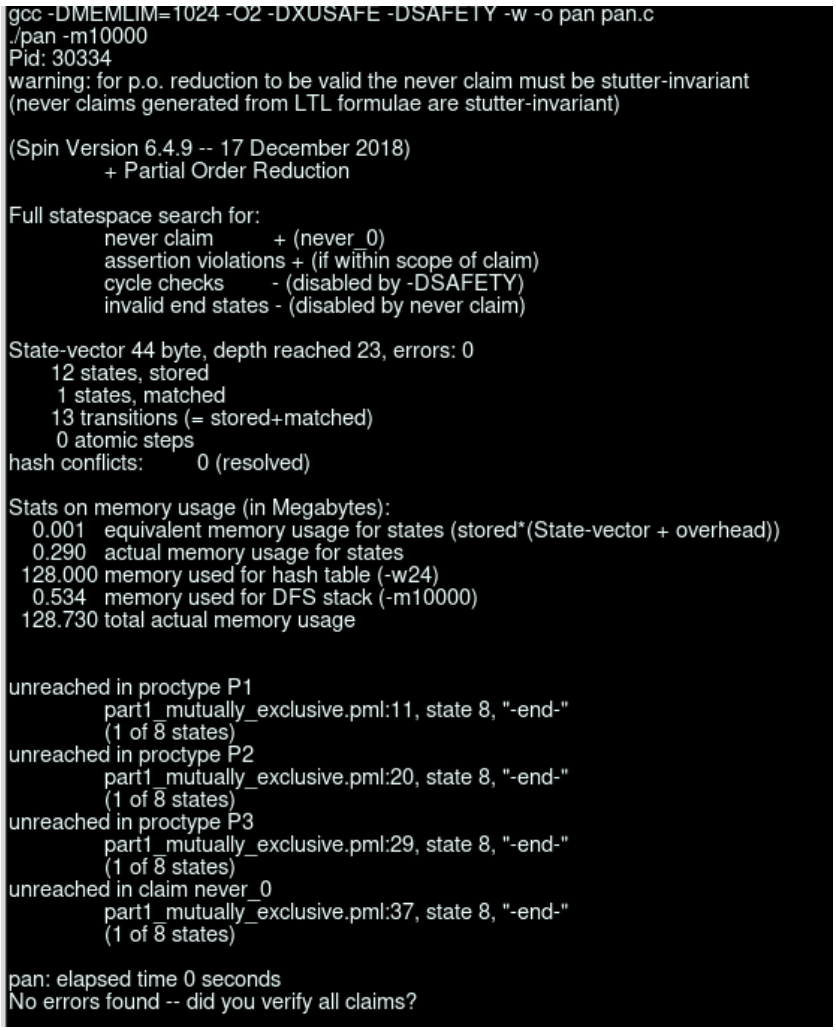

Figure 2: Mutual exclusion case for Part 1

3 Part 2

We were asked to write a specification in Promela for N mutually exclusive processes. To implement this, I used Peterson's **algorithm** for N processes. In 3, *level* is an array of N integers and *last_to_enter* is an array of N-1 integers.

We can imagine all processes to be put in a queue which shows the order they have to be executed. Process i thinks that it is in position l in the queue. This process is also the last one to think that they are in a position l. The process i has to wait till the condition in line 4 in 3 becomes false. Therefore either $last\_to\_enter[l]$ is not i or for all k, $level[k] < l$.

```
for ℓ from 0 to N−1 exclusive
    level[i] ← ℓ
    last_to_enter[ℓ] ← i
    while last_to_enter[ℓ] = i and there exists k ≠ i, such that level[k] ≥ ℓ
        wait
```

Figure 3: Peterson's algorithm for N processes

The Promela code is implemented in the file called 'part2.pml'. The implementation uses similar approach to the one above. The *move* array is equivalent to the *level* array and the *critical* array is equivalent to *last_to_enter*. In Promela, *_pid* stores the number of of the current process. For each process k we iterate through all the elements in *critical* and check if every element that is not in *_pid* is smaller than k. When the process *_pid* is in its critical section, we change the value of *print[_pid]* to true and then to false. It is useful to have a variable which counts the total number of critical states. The integer variable is incremented by one whenever an element in *print* is made to be true and decremented by one whenever an element in *print* is made to be false. An *assert* statement checks that the amount of critical sections is correct. In addition, I wrote a *never* claim which makes sure that number of critical sections is never more than 1. When running the Promela code, the output shown in 4 is printed. You can see that the end of the *never* claim and the end of the *P()* are never reached. In addition, I am using a *progress* label to check for non-progress cycles. As seen in the output, there are no violations. Therefore the program is working correctly and all the processes run infinitely often.

4 Part 3

This part asked to formulate the given requirements as an LTL formulae. In addition, each case has to be run with and without weak fairness. Each part of this section has a separate file containing different LTL formulas.

4.1 Part a

The given requirement: **x is eventually not 10**.

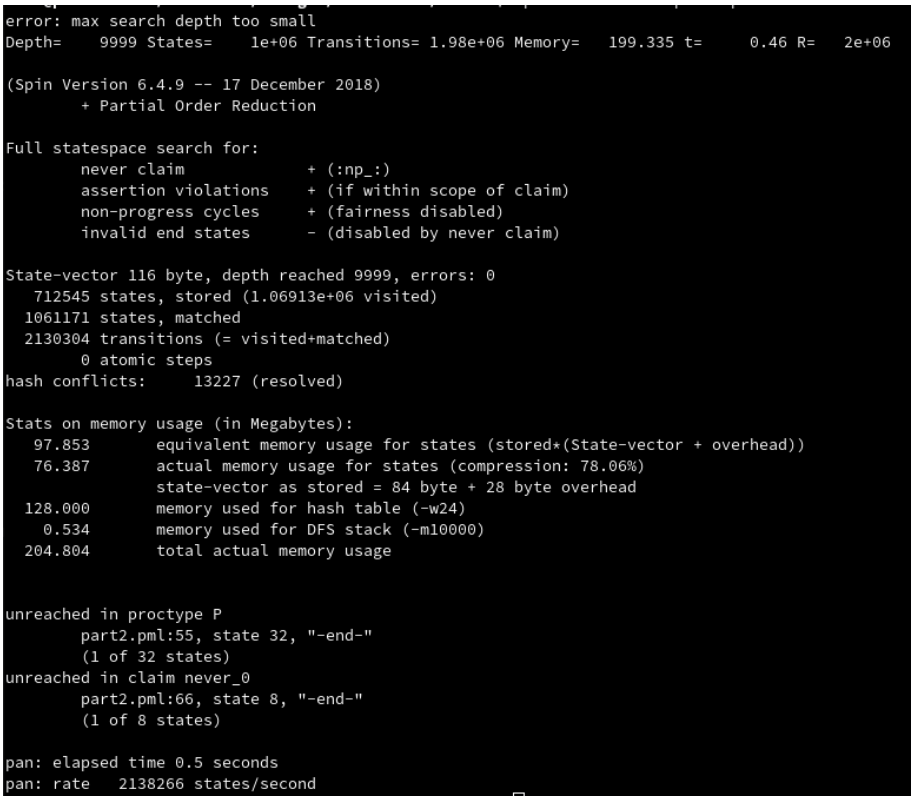

Figure 4: Output of Part 2

The LTL formula is quite straightforward: $<>$ **(x != 10)**

The Promela specification is written in a file called 'part3_a.pml' When running the program with no fairness and weak fairness, the output is the same and is shown in 5. It is seen that none of the processes reach the end. The LTL formula named p also never reaches the end. Therefore, the statement is always true.

```
unreached in proctype P1
        part3.pml:8, state 5, "-end-"
        (1 of 5 states)
unreached in proctype P2
        part3.pml:14, state 5, "-end-"
        (1 of 5 states)
unreached in proctype P3
        part3.pml:18, state 2, "y = x"
        part3.pml:20, state 6, "-end-"
        (2 of 6 states)
unreached in claim p
        _spin_nvr.tmp:6, state 6, "-end-"
        (1 of 6 states)

pan: elapsed time 0 seconds
```

Figure 5: Output of weak and no fairness for part a

4.2 Part b

The given requirement: **It is possible that from a certain point onwards x is always odd**.

The statement says that eventually x is always odd. To represent the negation of this we can write a LTL statement equivalent to 'never eventually x is always odd'. Then if the end of the statement is never reached, it is true and the initial statement is false. A number is odd if it gives a remainder 1 when dividing by 2.

The LTL formula: **!** ($<>$ [ ] **(x % 2 == 1)**)

The Promela specification is written in a file called 'part3_b.pml' When running the program with weak fairness, the program produces an output shown in 6. It is seen that none of the processes reach the end. The LTL formula named p also never reaches the end. Therefore, the statement executed LTL statement is true and the initial statement is false with weak fairness. The results when running it in no fairness mode are shown in 7. It is seen that an error is found. Therefore the executed statement is incorrect and the initial requirement is true when running in no fairness. It makes sense since when fairness is disabled, if only the process P1 is executed when x is odd, then x is always odd.

```
Full statespace search for:
        never claim         + (p)
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness enabled)
        invalid end states - (disabled by never claim)

State-vector 44 byte, depth reached 2043, errors: 0
   261632 states, stored (375808 visited)
   489859 states, matched
   865667 transitions (= visited+matched)
        0 atomic steps
hash conflicts:      3476 (resolved)

Stats on memory usage (in Megabytes):
    17.965  equivalent memory usage for states (stored*(State-vector + overhead))
    10.251  actual memory usage for states (compression: 57.06%)
            state-vector as stored = 13 byte + 28 byte overhead
   128.000 memory used for hash table (-w24)
     0.534  memory used for DFS stack (-m10000)
   138.690 total actual memory usage


unreached in proctype P1
        part3_b.pml:8, state 5, "-end-"
        (1 of 5 states)
unreached in proctype P2
        part3_b.pml:14, state 5, "-end-"
        (1 of 5 states)
unreached in proctype P3
        part3_b.pml:20, state 6, "-end-"
        (1 of 6 states)
unreached in claim p
        _spin_nvr.tmp:10, state 13, "-end-"
        (1 of 13 states)

pan: elapsed time 0.11 seconds
No errors found -- did you verify all claims?
```

Figure 6: Output of weak fairness for part b

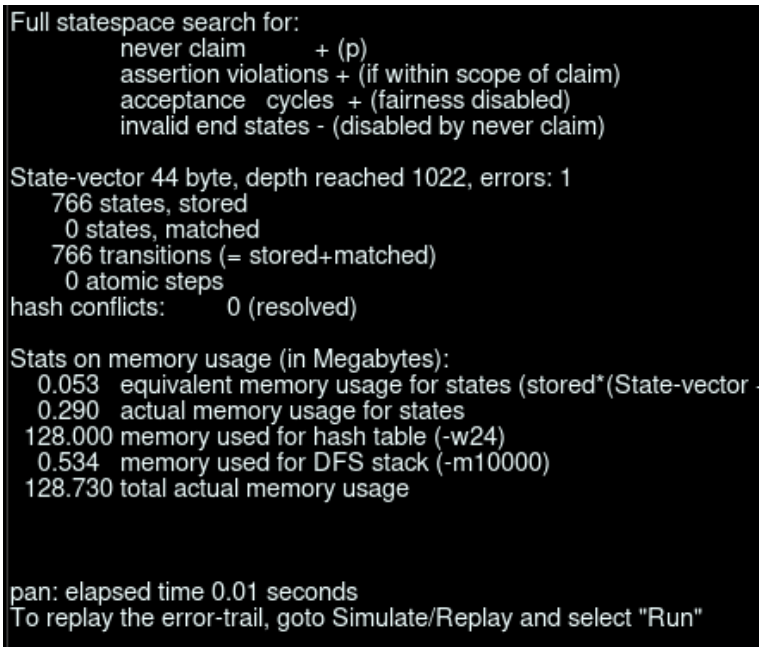

Figure 7: Output of no fairness for part b

4.3 Part c

The given requirement: **It is possible that from a certain point onwards x is infinitely often odd**.

The statement is equivalent to saying that x is always eventually odd. Similarly to Part b, we can write a LTL statement to negate this. Then if the end of the statement is never reached, the initial statement is false.

The LTL formula: **! ([ ] (< > (x % 2 == 1)))**

When running the LTL statement with weak fairness enabled, the end of the LTL statement is never reached which means that the executed statement is true and the initial statement is false. The output of the program can be seen in 8.

When running the program with no fairness, the program has an error which means the executed statement is false and the initial requirement is true. The output is seen in 9.

4.4 Part d

The given requirement: **It is always true that y $\leq$ x**.

The LTL formula is quite straightforward: **[ ] (y $\leq$ x)**.

It seems like this should always be true as y is either less than x or equal to x. However, since x and y are bytes, there is a case where x=0 and y=255. In

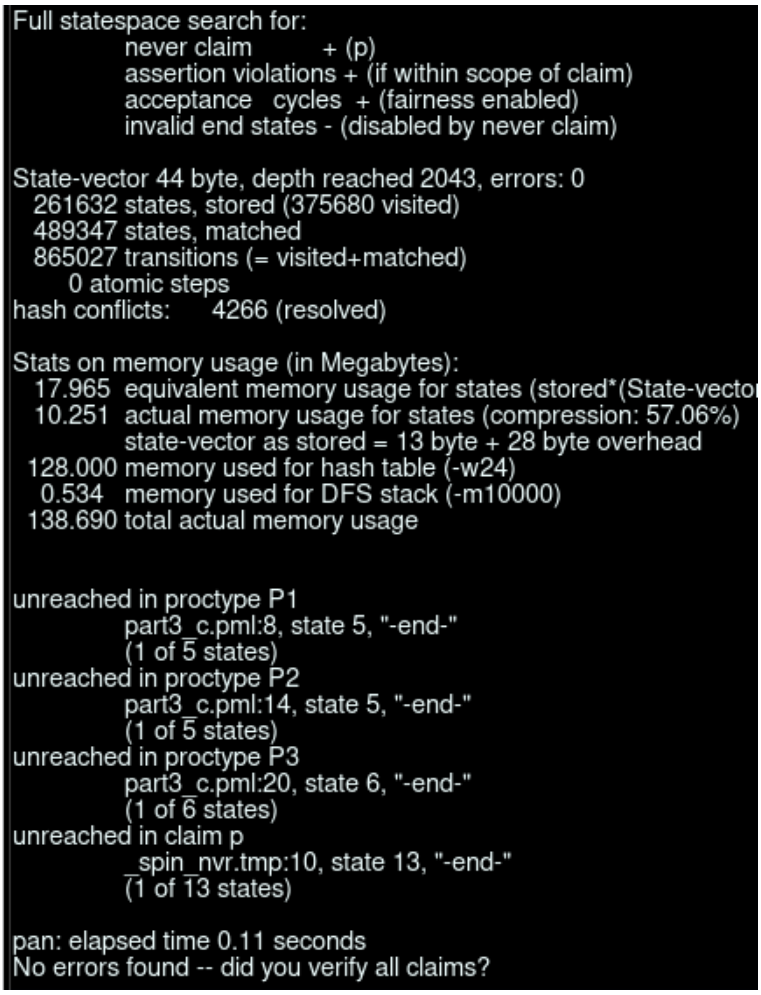

Figure 8: Output of weak fairness for part c

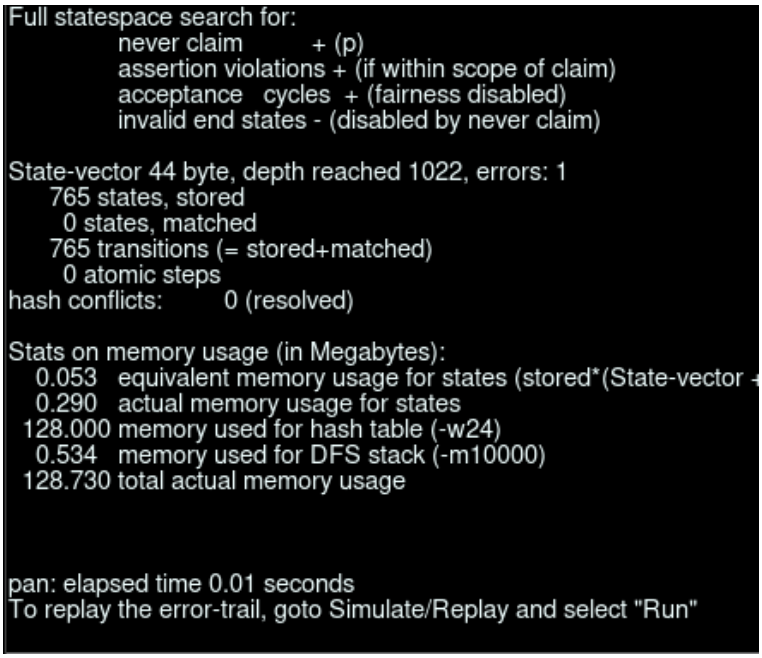

Figure 9: Output of no fairness for part c

this case an error is thrown since the LTL formula is not satisfied. The output of weak fairness is shown in 10. Initially, I thought that running the LTL formula with no fairness might not give an error if x is never being increased and only P3 is being executed. However, this is not true since when $y < x$, then y is changed to be $y = x$ and P3 can not be executed next as $y = x$ and not $y < x$. Therefore, either P1 or P2 will have to be executed and at some point $y < x$ will not be true. The same error is found. The output of the program is shown in 11.

4.5 Part e

The given requirement: **It is always true that when $y \neq x$ it follows that at some point $y = x$.**

The statement is equivalent to saying that always $y \neq x$ implies that eventually $y = x$.

The LTL formula for this is: $[\,]\,(\,(\,y \neq x\,) -> (\,<>\,(\,y == x\,)\,)\,)$

After some thinking, it is clear that this is not true. For example, when x is 255 and y is less than 255, P3 can be executed and now $y = 255$. Since 255 is the maximum possible value, P3 will never be executed after this point. Let's say x is 0 and we always execute only P1 (x is always even). Therefore x is never equal to 255 again. It is possible to execute P1 and P2 in a way such that x always skips 255. An error is found when executing LTL statement with

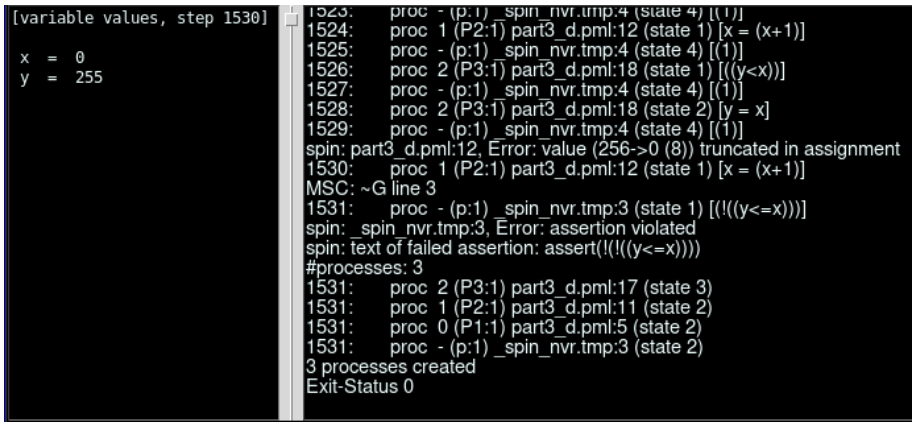

Figure 10: Output of weak fairness for part d

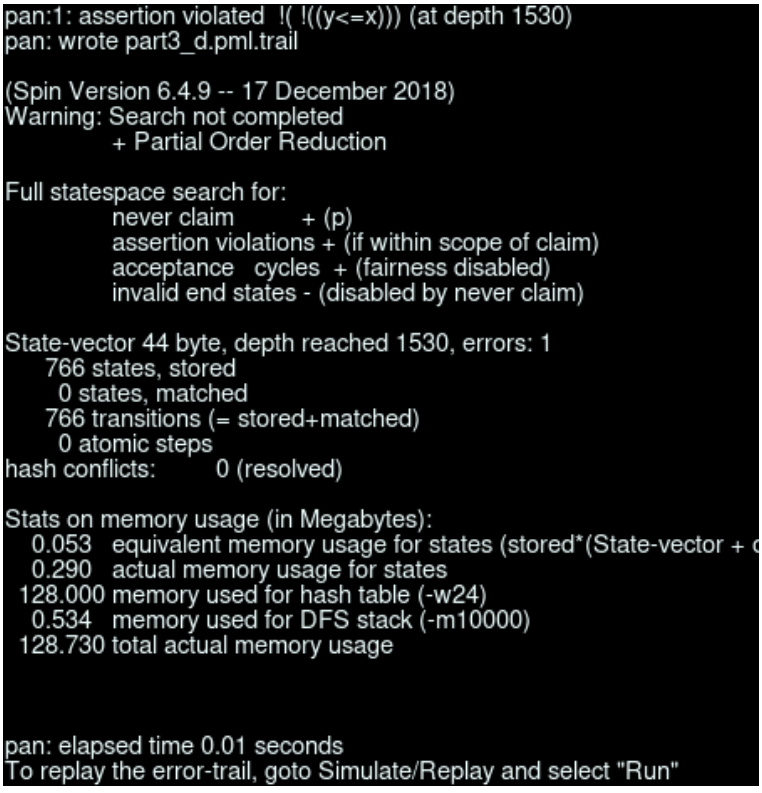

Figure 11: Output of no fairness for part d

weak fairness and no fairness. The output of weak fairness is shown in 12. The

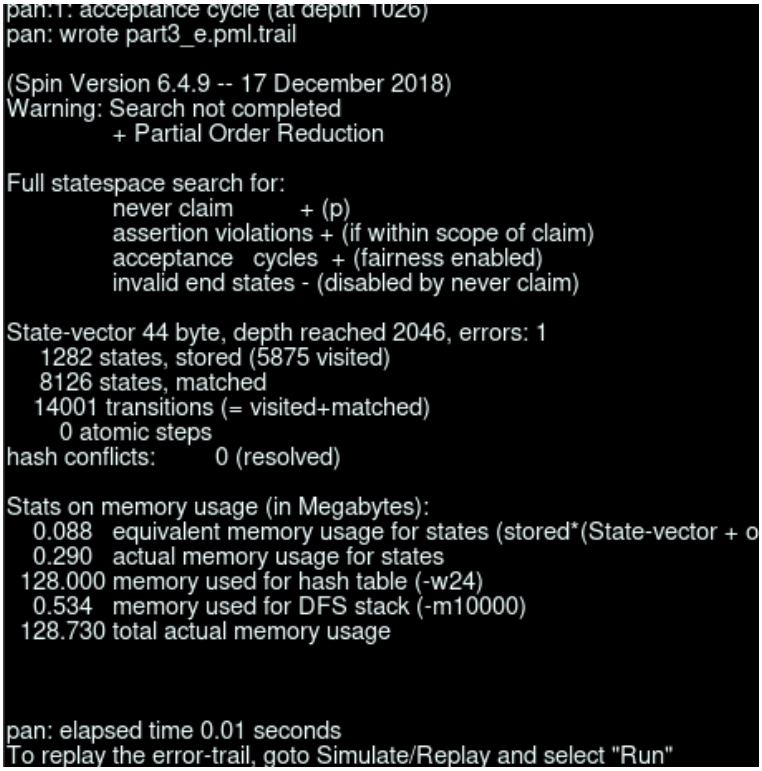

Figure 12: Output of weak fairness for part e

output of no fairness is shown in 13. Both executions find an error.

5 Part 4

The part requires to write an in detail description of how formal verification with Spin works. The resaurces to answer this question were taken from 'The Model Checker Spin' paper written by Gerard J. Holzmann and Wikipedia 'Spin model checker'. • underlying representation of PROMELA models; • LTL model checking algorithms; • current limitations; • mechanisms for improving scalability and/or efficiency.

5.1 Overview of Spin

SPIN generates C sources to check the model of specific problems. Therefore, it does not actually perform model-checking itself. in this way, SPIN saves some memory and achieves better performance. It also allows direct insertion of chunks of C code into the model.

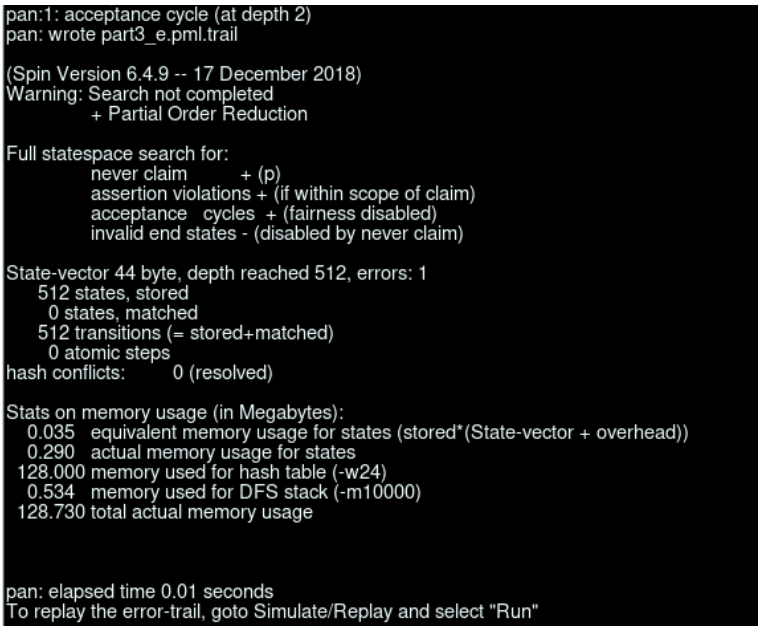

Figure 13: Output of no fairness for part e

5.2 Underlying representation of Promela models

Each process is translated into a finite automaton in Spin. We compute asynchronous interleaving products of automata and form a concurrent system. The resulting behaviour of such global system is itself an automaton. Thew interleaving product is usually called a *state space* of the system. The state space can easily be represented as a graph. To perform verification, Spin takes a correctness claim that is specified as a temporal logic formula, converts that formula into a Büchi automaton, and computes the synchronous product of this claim and the automaton representing the global state space. The result is again an automaton. If the language accepted by this automaton is empty, this means that the original claim is not satisfied for the given system. If the language is nonempty, it contains exactly the behaviors that satisfy the original formula. In Spin, we use the correctness claims (and temporal logic formulae) to formalize behaviors that are undesirable. The verification process then either proves that such behaviors are impossible or it provides detailed examples of behaviors that match. In the worst case, the global reachability graph has the size of the Cartesian product of all component systems. Promela is defined in such a way that each component always has a finite range. A number of complexity management techniques have been developed to perform verification in a less expensive way.

5.3 LTL model checking algorithms

Spin's verification procedure is based on an optimized depth-first graph traversal method. The cycle detection method used in SPIN is of central importance. The method is required to be compatible with all modes of verification, including exhaustive search, bit-state hashing, and partial order reduction techniques. The most common algorithm for finding cycles in a graph is Tarjan's depth-first search algorithm. It forms the strongly connected components in linear time by adding two integer numbers to every state reached. If a strongly connected component in the reachability graph contains at least one accepting state, a reachable acceptance cycle has been shown to exist.

5.4 Current Limitations of Spin

LTL model checking is automatic, efficient and can produce counterexamples. However, there are also some limitations. LTL only works in finite-state systems therefore there is always a bound on the number of possible processes to be executed. In addition, only the properties which hold in all runs can be checked. The complexity of the problem grows rapidly due to how the combinatorics of the problem is affected by the input, constraints, and bounds of the problem. However, there are many techniques which can speed up the process. They are discussed in the next section.

5.5 Mechanisms for improving scalability and efficiency

There are many algorithms that are used in order to optimize the search. Spin uses a partial order reduction method to reduce the number of reachable states that must be explored to complete a verification. The reduction is based on the observation that the validity of an LTL formula is often insensitive to the order in which concurrent and independently executed events are interleaved in the depth-first search. Instead of generating an exhaustive state space that includes all execution sequences as paths, the verifier can generate a reduced state space, with only representatives of classes of execution sequences that are indistinguishable for a given correctness property. 14 shows a measurement of the number of reachable states that has to be generated to complete the verification for a model.

Another important factor is memory management. The size of the interleaving product that SPIN computes can, in the worst case, grow exponentially with the number of processes. State compression and bit-state hashing are two main techniques to economize the memory requirements. Instead of storing the complete concatenation of all local state descriptors for variables, channels, and processes, the compression algorithm now stores each separable element alone, and uses unique indices to the local descriptors in the global state vector. The amount of reduction achieved depends on the relative size of indices and local states, which can be arbitrarily large. In addition, Spin includes an implementation of the bit-state hashing technique. With this algorithm, two bits of memory

are used to store a reachable state. The bit-addresses are computed with two statistically independent hash functions.

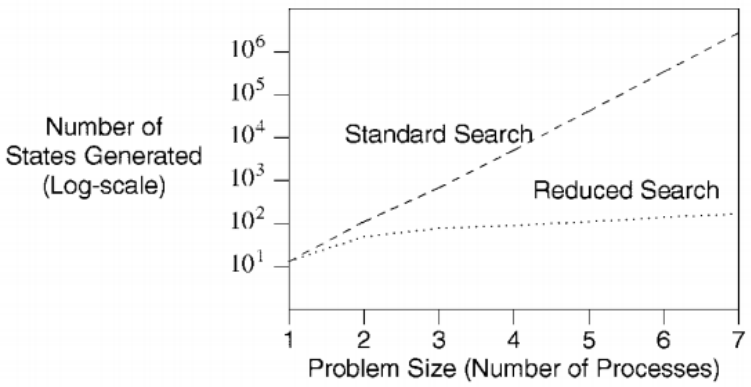

Figure 14: Effect of partial order reduction

6 Conclusion

I attempted to complete all 4 parts of the practical. Screenshots of the program outputs are provided and the thought process is explained. Part 4 helped to understand how Promela actually works. Overall, the tasks helped to understand the material of the course much better. I learned to write Promela code and LTL statements as well as *never* claims. Parts 2 and 3 were the most challenging.