Tyler Marenger

James Helm

**CS4710: Model-Driven Software Development, Spring 2020**

**Computer Science Department Michigan Technological University**

# SPIN Group Assignment #1: Parallel Swap

**Deadline: February 23, 2020 at mid-night**

## The Model:

Our model was specifically designed to meet the requirements of the Parallel Swap Group Assignment. It utilizes a shared memory array which is declared in the global scope of the model. The size of the array is defined by a macro N. Such that, the value of N determines the length of the array and the values which will populate it. The array is initialized with distinct non-negative integer values. To handle this requirement, we put the values 1 - N into the array. This means if N=5, then the array will hold [ 1, 2, 3, 4, 5 ]. Additionally within the init of the model, we declare an array called arrayIndexLocks and procCount which are both size N and map to the corresponding position in the array. The arrayIndexLocks array holds the index locks which was our method of avoiding race conditions. This array is initialized with 1's which represent an available lock. The procCount array is initialized with 0's. These represent the number of processes that are manipulating that index of the array. This is used in the ltl conditions to ensure mutual exclusion. All of this occurs within a for loop which is defined by the for.h header file provided in class. After the initialization occurs, our proctype begins running using the run command.

Our init method creates N threads of our proctype. It sets i to the PID of the process that generates the random number j. Following this step, the process tries to acquire the locks which allow it to swap the values. In order for the process. The following occurs when this happens:

1) The process enters an atomic do loop to ensure that the if condition is checked in one instruction to avoid interleaving/context switching while it is checking the conditions and switching the values.
2) The process loops while another process is altering the values of the locks.
3) Once a process is not editing the values of the locks, it checks:
    a) Is the array index of i available? (arrayIndexLocks[i] == 1)

    b) Is the array index of j available? (arrayIndexLocks[j] == 1)
    c) Is there 0 processes manipulating the index of i? (procCount[i] == 0)
    d) Is there 0 processes manipulating the index of j? (procCount[j] == 0)
    e) Does i != j?
4) If all of these conditions evaluate to true, then that process takes it turn by setting turn to 1. It then locks the array index by setting the i and j index of arrayIndexLocks to 0. Additionally, it increments the procCount of these indices (procCount[i]++ and procCount[j]++).
5) If this evaluates to false, they either:
    a) There is another process manipulating one of these values
    b) Or i == j
6) In the case that it evaluates false, the model will print and evaluate a new value for j which will guarantee it to be different from i. It cannot be equal to i again because it will be equal to i + 1 or i - 1.

The steps above show the entry method for entering the critical section. Once the model has entered the critical section, it will swap the values outside of an atomic block like the requirements specified. After the swap has completed another atomic action is run where: the arrayIndexLocks are unlocked by setting the values to 1 (arrayIndexLocks[i]=1; arrayIndexLocks[j]=1;). Additionally, the procCounts of the indices are decreased (procCount[i]--; procCount[j]--;).

**Model Safety, Liveness, Concurrency & Mutual Exclusion:**

Throughout the process the assertion (procCount[i] <=1 && procCount[j]) is tested to ensure mutual exclusion. Since these assertions are held during verification, this model is mutually exclusive. Additionally, to check our model, we have multiple LTL expressions. The first LTL ensures safety; in other words, it is data race free because no two processes can simultaneously access the same array cell with a write operation at the same time. It is worth noting that this also means that there will not be any duplicate values after execution. Here are our safety LTLs:

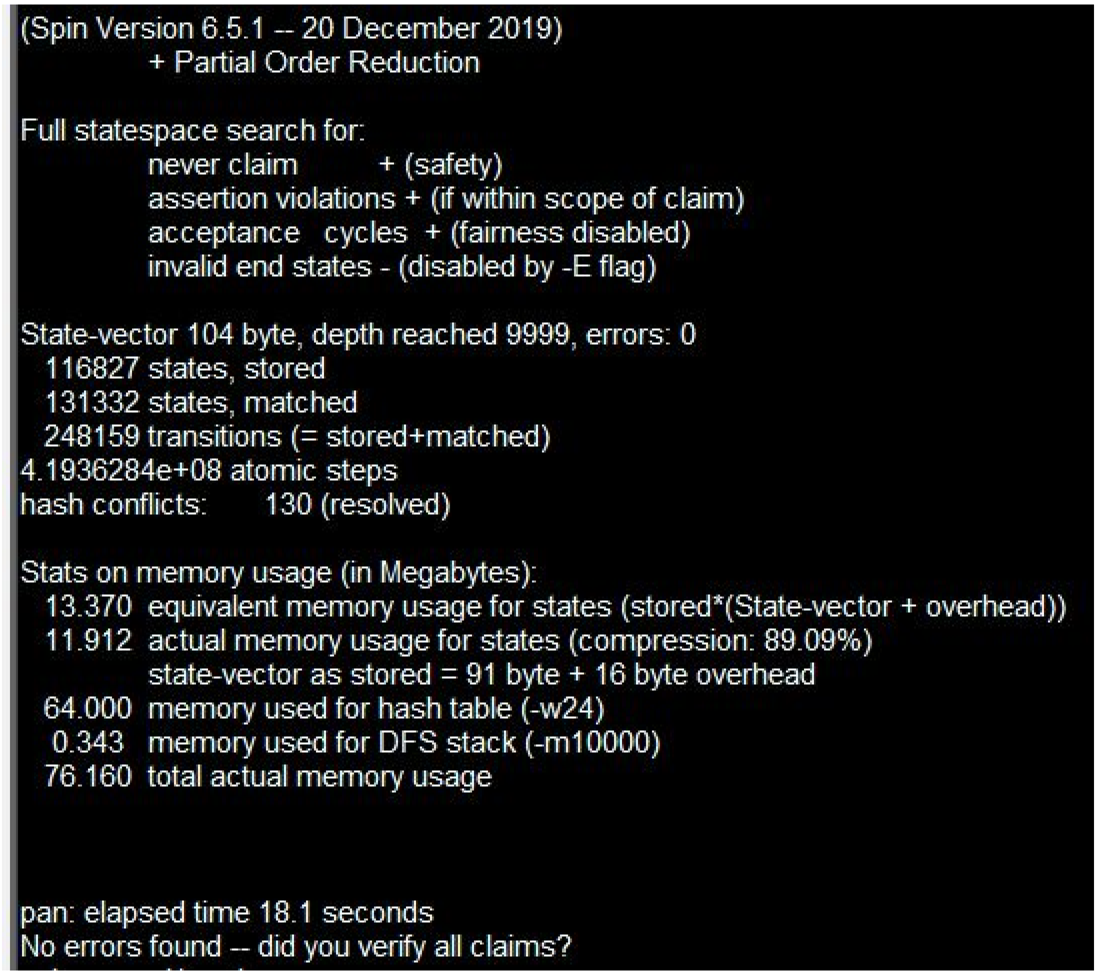
1) #define mutex (((procCount[0] <= 1) && (procCount[1] <= 1) && (procCount[2] <= 1) && (procCount[3] <= 1)) )

   ltl safety { [] mutex }

2) #define distinctNumbers (array[0] != array[1] && array[1] != array[2] && array[0] != array[2] )

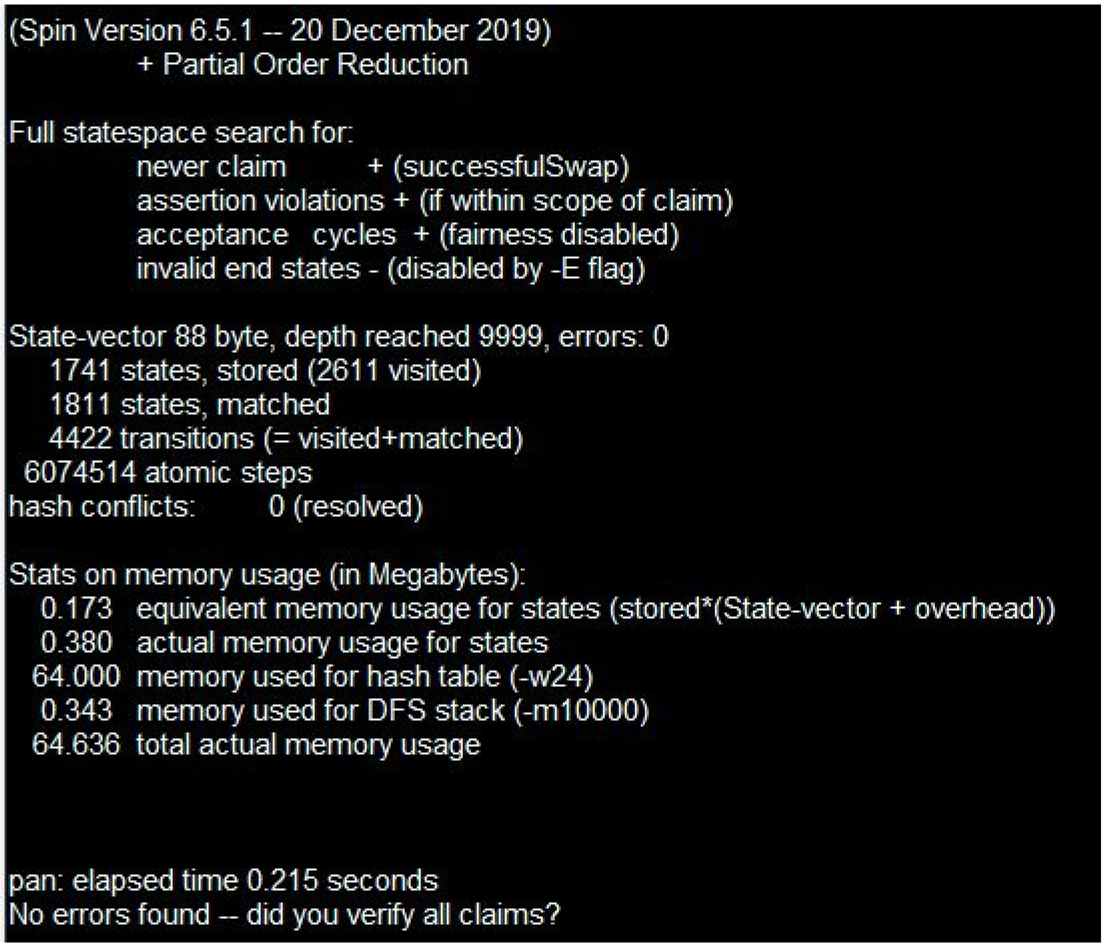
   ltl successfulSwap { [] (Terminated -> distinctNumbers) }

The first statement is checking that no more than one process is manipulating an index of the array at the same time. Here is a snip of the verification running without error (this took a long time to run):



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

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

State-vector 104 byte, depth reached 9999, errors: 0
  116827 states, stored
  131332 states, matched
  248159 transitions (= stored+matched)
4.1936284e+08 atomic steps
hash conflicts:       130 (resolved)

Stats on memory usage (in Megabytes):
  13.370  equivalent memory usage for states (stored*(State-vector + overhead))
  11.912  actual memory usage for states (compression: 89.09%)
          state-vector as stored = 91 byte + 16 byte overhead
  64.000  memory used for hash table (-w24)
   0.343  memory used for DFS stack (-m10000)
  76.160  total actual memory usage



pan: elapsed time 18.1 seconds
No errors found -- did you verify all claims?
```

The second statement is used to ensure that the values are always distinct. That is, no two values in the array should be equal. Here is a snip of the verification running without error:

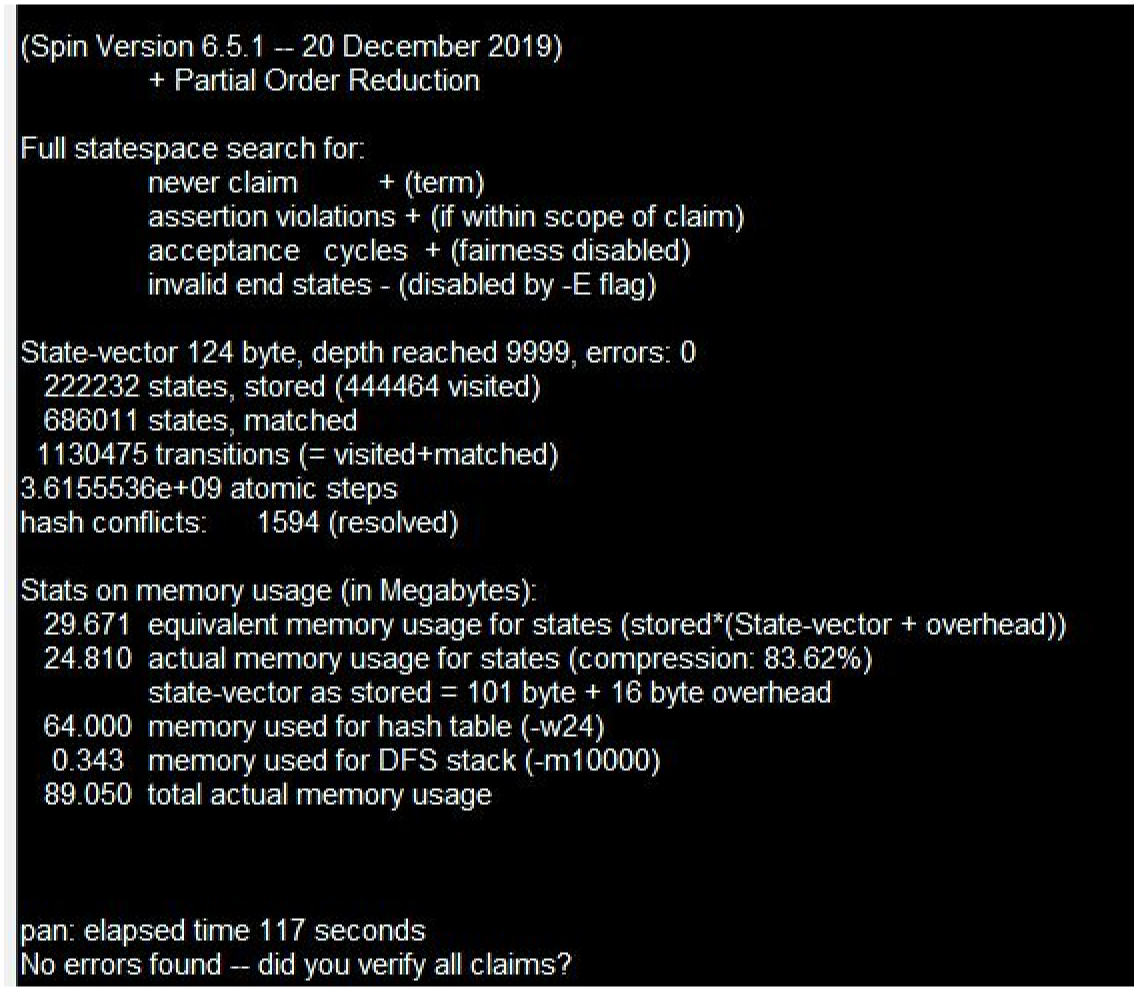


```
(Spin Version 6.5.1 -- 20 December 2019)
            + Partial Order Reduction

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

State-vector 88 byte, depth reached 9999, errors: 0
    1741 states, stored (2611 visited)
    1811 states, matched
    4422 transitions (= visited+matched)
  6074514 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.173   equivalent memory usage for states (stored*(State-vector + overhead))
    0.380   actual memory usage for states
   64.000  memory used for hash table (-w24)
    0.343   memory used for DFS stack (-m10000)
   64.636  total actual memory usage



pan: elapsed time 0.215 seconds
No errors found -- did you verify all claims?
```

The second set of LTL expressions which can be found in our model are used to ensure liveness. That is, per the requirements, that all processes eventually terminate. To ensure this we used the following LTL:

```
#define Terminated (np_ == 0)

ltl term { <> Terminated }
```

When this was verified, it ran without error. Here is a snip of the verification:

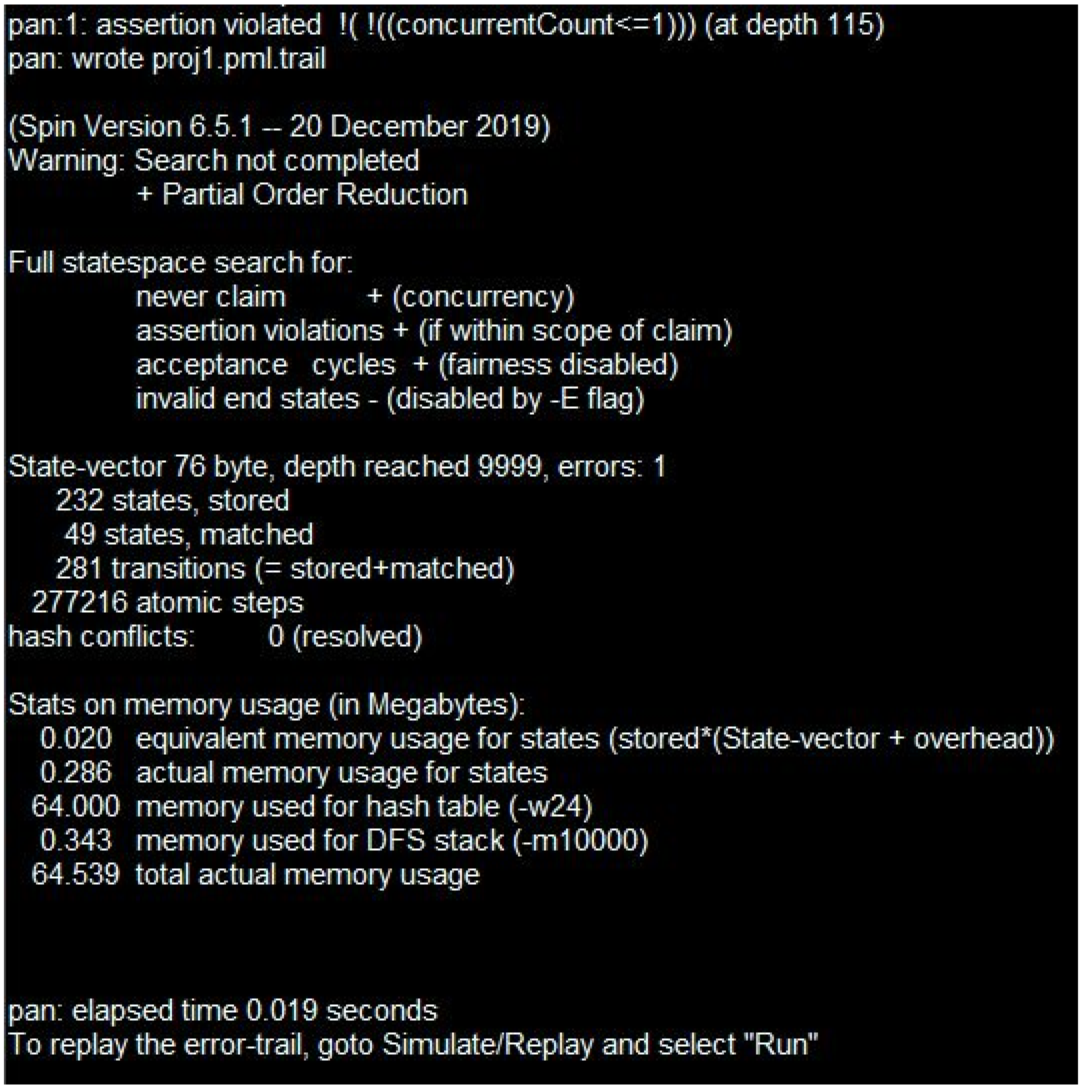


```
(Spin Version 6.5.1 -- 20 December 2019)
            + Partial Order Reduction

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

State-vector 124 byte, depth reached 9999, errors: 0
   222232 states, stored (444464 visited)
   686011 states, matched
  1130475 transitions (= visited+matched)
3.6155536e+09 atomic steps
hash conflicts:      1594 (resolved)

Stats on memory usage (in Megabytes):
   29.671  equivalent memory usage for states (stored*(State-vector + overhead))
   24.810  actual memory usage for states (compression: 83.62%)
           state-vector as stored = 101 byte + 16 byte overhead
   64.000  memory used for hash table (-w24)
    0.343  memory used for DFS stack (-m10000)
   89.050  total actual memory usage



pan: elapsed time 117 seconds
No errors found -- did you verify all claims?
```

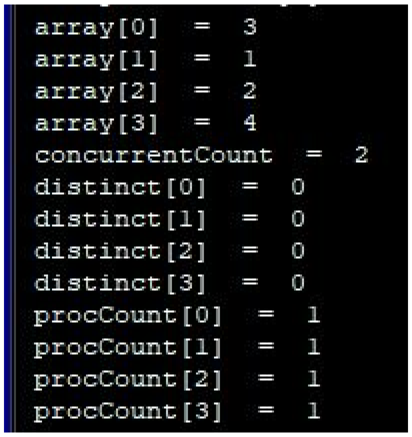
Finally, the requirements stated that the code must not serialize the execution process. That is, if two pairs of processes have no intersections in terms of the array cells they want to swap, then they should be able to execute concurrently. For instance, the swapping that processes 1 and 3 do can be done concurrently with the swapping that processes 2 and 5 do because the set of memory cells the pairs of processes access are disjointed. Here is the LTL expression we used to test this:

```
#define swapOccursConcur (concurrentCount <= 1)

ltl concurrency { [] swapOccursConcur} /* Want to fail */
```

We have a counter that is incremented at the start of the critical section and is decremented at the end of the critical section. The above LTL is checking if there is ever a time where more than one process is in the critical section. As you can see, it is checking that it is always the case concurrentCount <= 1. This means if two processes are in the critical section it will fail and provide a counterexample. Here is a snip of the error occuring and the counterexample:



```
pan:1: assertion violated  !( !((concurrentCount<=1))) (at depth 115)
pan: wrote proj1.pml.trail

(Spin Version 6.5.1 -- 20 December 2019)
Warning: Search not completed
            + Partial Order Reduction

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

State-vector 76 byte, depth reached 9999, errors: 1
      232 states, stored
       49 states, matched
      281 transitions (= stored+matched)
   277216 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.020   equivalent memory usage for states (stored*(State-vector + overhead))
    0.286   actual memory usage for states
   64.000  memory used for hash table (-w24)
    0.343   memory used for DFS stack (-m10000)
   64.539  total actual memory usage



pan: elapsed time 0.019 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```



```
array[0]  =  3
array[1]  =  1
array[2]  =  2
array[3]  =  4
concurrentCount  =  2
distinct[0]  =  0
distinct[1]  =  0
distinct[2]  =  0
distinct[3]  =  0
procCount[0]  =  1
procCount[1]  =  1
procCount[2]  =  1
procCount[3]  =  1
```

As you can see directly above, the procCount for index 0, 1, 2, and 3 are all one. This means that there is a swap occurring between two sets of indices at the same time. This ensures our desired qualities.

It is worth noting that since all of these LTLs are held during verification, we can be certain that our model holds the requirements that the assignment requested. Additionally, you can see that we listed these LTLs with a certain N. However, when we were ensuring the correctness of our program we tested a variety of N values (and edited these LTLs accordingly) to ensure correctness within our model. You can also see that we created an LTL called combinedLTLs. This is just all of the LTLs '&&' together aside from the concurrency LTL which is tested separately so that we can view the counterexample. Testing the combined LTLs does succeed.

**Extra Credit:** After you verify the safety and liveness properties, revise your Promela code such that each proctype performs the swapping for an unbounded number of times. Note that your code must be non-terminating.

• Are the LTL properties still satisfied? Explain what you experience.

Yes, most of the LTL properties are still satisfied. However, no, not all of them are still satisfied. The reason that I say that they are not is because, by design, they cannot all still be satisfied. The liveness LTL checks to see that all processes must eventually terminate. However, now the program is supposed to perform the swapping for an unbounded number of times. That being said, since the code is supposed to be *non-terminating*, it cannot satisfy the termination LTL. Additionally, the distinct values LTL does not hold because this depends on the termination of the program. Creating an LTL for this scenario seemed impossible as due to no atomicity on swapping there is always an instance where the swap is only partially complete and the distinct check would fail during this state. However, the model still holds the properties of mutual exclusion and concurrency which are found in the other LTLs. This is to be expected since it runs just like it would if it ran once. If you let this run and run until it times out you will see that that array continues to hold distinct values but the values will keep randomly swapping since the value of j is being constantly changed.

• Should the proctypes be synchronized after each round of swapping?

Yes, the data is contained within a shared memory array. Thus, after each round of swapping all of the processes are synchronized due to the position locking in the array performed. You can ensure this by clicking ‘stop’ any time that the program is outside of swapping. Using the step forward and step backward we can see that the array always contains distinct values within simulations. As explained in the previous answer, you can let it continue until it times out and it will still contain distinct values. The program will keep swapping around the elements of the array because the value of j continues to be randomly selected.