Promela Models for Mutual Exclusion Algorithms

Jett Anderson, Andrew Chau, Justin Nguyen, George Tang
https://github.com/justin-nguyen-1996/Spin_Promela

Abstract

The purpose of this project was to verify the correctness of three different mutual exclusion algorithms by using the SPIN model checker and Promela modeling language. Our team chose to verify Peterson's Tournament algorithm, the N-Level Filter algorithm, and the Colored Bakery Algorithm. This paper gives a brief overview of SPIN and Promela, alternative model checkers we could have used, the details of how we implemented the three algorithms, and results showing that the algorithms were correctly implemented. The appendix at the end of the paper is a simple tutorial on how to use SPIN with Promela.

Introduction

SPIN (Simple Promela Interpreter) is an open-source software verification tool that can be used for model checking and verifying safety, liveness, and fairness properties of distributed systems. The tool supports a process modeling language called Promela (Process Meta Language), which was designed to dynamically create concurrent processes in order to model and verify the logic of parallel algorithms. Given a program written in Promela, SPIN can verify the model's correctness by either performing nondeterministic executions of the program or by generating a C program that exhaustively verifies the entire system's state space [1].

SPIN tests a program's validity by creating multiple threads with the same algorithm and running them concurrently. To test for safety, SPIN relies upon the user to asserts user-defined conditions. In our case, we asserted that the number of processes in the CS was always less than or equal to 1. To test for liveness and fairness, SPIN also keeps track of thread-specific variables called “timeout.” If a thread’s timeout variable throws an error, then one can assume that the thread has either starved or crashed. If the timeout variables for every thread throws an error, then SPIN assumes that deadlock has occurred [1].

Project Description

The sample mutual exclusion (mutex) algorithms specified in Promela and verified by our team included Peterson’s Tournament Algorithm, the N-Level Filter Algorithm, and the Colored Bakery Algorithm.

The first algorithm, Peterson’s Tournament Algorithm, expands upon Peterson’s original two process algorithm by having log(N) levels of binary contention, where each node in the “tournament bracket” allows two processes to compete using the two-way mutex algorithm. After competing, the process that won the exchange then proceeds to the next level to compete again, and so on and so forth until reaching the critical section (CS) and finally becoming the “winning” process. The tournament algorithm is implemented in Promela by using a two-dimensional array of request/want variables and turn variables for each “node of contention.” Upon exiting the CS, the process will set the values of the request variables it accesses on its path to victory to false [2]. Sample code is shown below in Figure B-1.

The N-Level Filter Algorithm uses a similar manner of having processes compete in various levels in order to eventually access the CS. There are N “waiting rooms,” where at most (N - room number) processes can advance, until there is a single process left that can access the CS. This scheme is accomplished in Promela by having two arrays of single-write, multiple-read variables, one to designate the current level a process is at and one to designate the last process to enter the current level, the “victim process.” When processes request to enter a level, they have to wait at that current level unless no other process has a higher level than it or until it is no longer the victim process. When a process is finished with the CS, it resets its level variable to -1 to signify it is no longer interested in the CS [3]. Sample code is shown below in Figure B-2.

The Colored Bakery Algorithm extends the original bakery algorithm, which involves processes receiving timestamped tickets to determine the order of requests for a system, but adds a global variable called the “color” variable. When a process requests the CS, its ticket is assigned the same color as the global color variable, and tickets that hold the same color as the global color variable have priority over the opposite colored tickets. When a process leaves the CS, the global color variable is flipped so that new requests become the opposite color. For example, if a process that has a white colored ticket leaves the CS, then new global color variables will become black [4]. Sample code is shown below in Figure B-3.

Various Design Alternatives

For our project, we chose to research Promela and SPIN in order to check the validity of various mutual exclusion algorithms. However, SPIN is not the only software capable of verifying the correctness of a mutex algorithm. For example, we could have chosen CBMC (Bounded Model Checker for C and C++). The main difference between SPIN and other model checkers is the specific language that the user would be writing the algorithm in.

SPIN uses Promela, a C-based language. One issue our team came across was that we had to workaround Promela’s restricted data types. In our case, the Peterson’s Tournament Algorithm needed to make use of floating point numbers, a data type that Promela does not have. As a result, we had to hard-coded the number of levels as an integer in the Tournament Algorithm.

Optionally, we could have used CBMC, a model checker built by Carnegie Mellon that uses C, C++, and/or Java. This would have reduced the amount of time needed to recreate various algorithms since we would have had access to the extensive libraries that come with the three languages. That being said, there are various reasons that we chose SPIN over CBMC. The primary reason is that SPIN has a larger user base and support base, which makes it easier to develop algorithms using SPIN and Promela. CBMC, on the other hand, does not have a large user base, or even a forum for that matter. For people who have no prior knowledge on model checkers, not having a reliable place to get answers for questions would definitely be detrimental when deciding which software to use [5].

Performance Results

The algorithms we tested all passed safety, liveness, and fairness checks. Figure B-4 shows an example of an intermediate stage program violating mutex and its corresponding SPIN output, while Figures B-5, B-6, and B-7 show examples of correct output of our three verified algorithms.

Conclusion

Our project used the SPIN model checker and Promela language to verify the correctness of Peterson's Tournament algorithm, the N-Level Filter algorithm, and the Colored Bakery algorithm. Although there are many different model checkers for mutex algorithms, we chose to use SPIN because of its large support base and maintained forum for user Q&As. The mutual exclusion algorithms we tested each satisfied safety, liveness, and fairness properties.

Appendix A: Spin tutorial Appendix

- Download spin here: http://spinroot.com/spin/Bin/index.html
- Unpack it on a linux system with
    - *tar -xzf spin.tar*
    - *cd Src*
    - *make*
    - *sudo make install*
- Using spin
    - Compile and run with *spin <filename>*
    - Use the *-n<seed#>* to seed your run *spin -n42 <filename>*
    - To get spin command line options use *spin --*
- Basic types in Promela (you can still use typedefs)
    - bit, bool, byte, short, int
    - Default value of basic types is zero
    - **NOTE:** there are no floating point types in Promela
- **NOTE:** Processes are basically synonymous with threads
    - Processes can be run at any time and will be interleaved with one another
    - To declare a process:
        - *processType <processName>(type paramName) { ...code... }*
    - The init process is special and is similar to the "main" function in a C++ program:
        - *init { ...code... }*
- To run a process
    - *init { int pid = run process(); }* */* all processes return their process ID */*
- Processes can also be created by adding active before the processType declaration
    - *active processType Foo(byte x) { printf("hello world"); }*
    - **NOTE:** These processes will start immediately
    - Do the following to have processes start running at the same time
        - *init { atomic { run P0(); run P1(); run P2(); } }*
        - Using the keyword atomic will force the expression to not become interleaved with other instructions

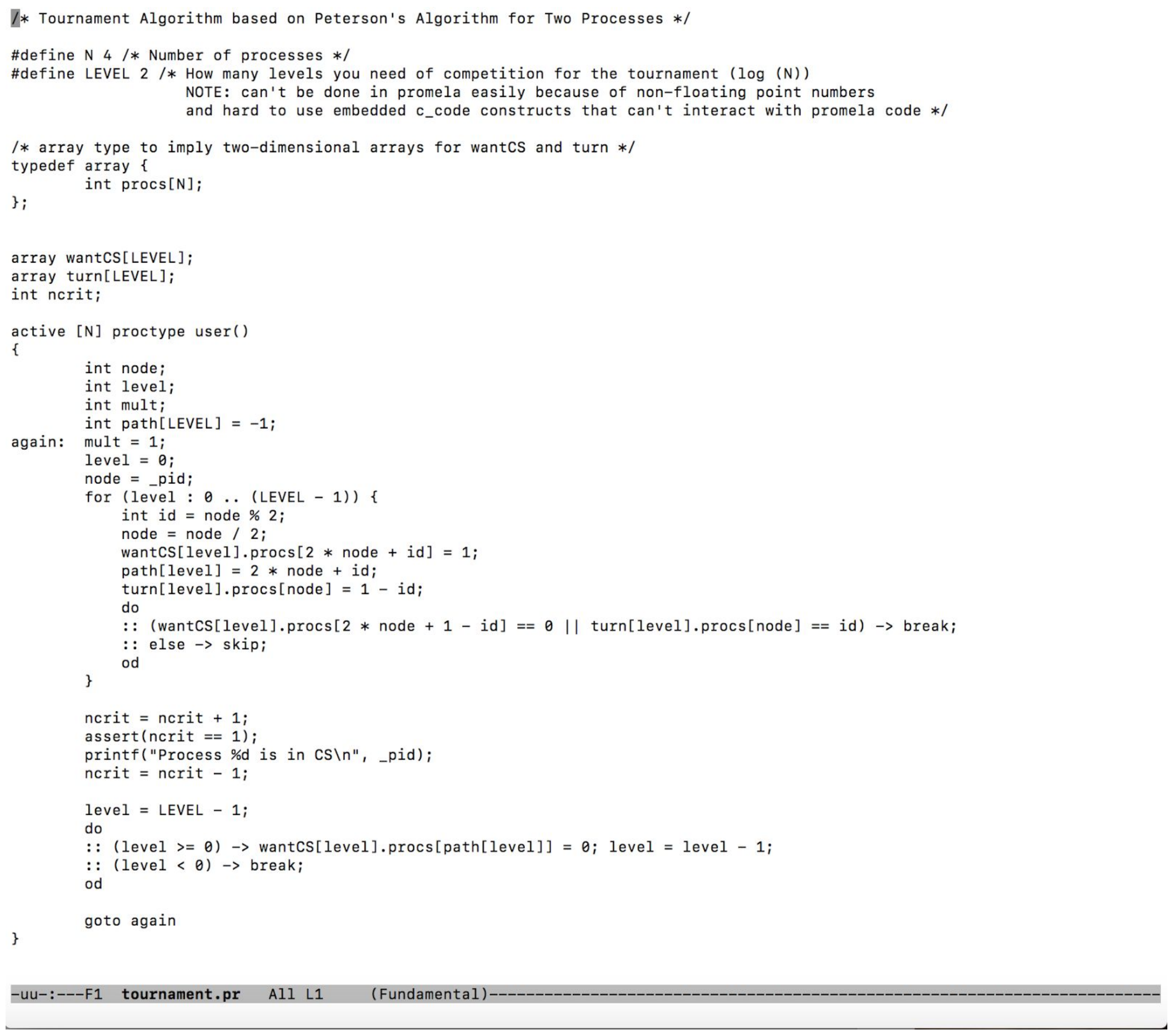
- **NOTE:** Statements are used differently in Promela than they are used in C
    - If a statement evaluates to true (i.e. non-zero) then the statement will "execute." Else, it will "block."
    - For example
        - *flag == 1;*   */* while (flag != 1) {} */*
        - Both will cause the process to block until flag becomes one
- If statements use the following syntax:
        - *if*
        - *:: choice1 -> doSomething1();*
        - *:: choice2 -> doSomething2();*
        - *:: choice3 -> doSomething3();*
        - *fi*
    - **NOTE:** If no choice is true, then the if statement will block until at least one of the conditions is satisfied. If more than one choice is true, then SPIN will *nondeterministically* choose one of the *doSomethings()* to run
- Do loops follow a similar structure (replace *if* with *do* and *fi* with *od*)
    - The concept of a do loop is similar to that of a while loop
- If you would like to ensure a variable is a certain value use assert:
    - *processType monitor (byte h) { assert(h <= 3); }*
- mtype is a keyword in Promela and is the same thing as an enum in C
    - *mtype = { RED, YELLOW, GREEN };*

- Basic Hello World in Promela

    -
      ```
      active proctype Hello() {     printf("Hello process, my pid is: %d\n", _pid);     }
      init {
              int lastpid;
              printf("init process, my pid is: %d\n", _pid);
              lastpid = run Hello();
              printf("last pid was: %d\n", lastpid);
      }
      ```

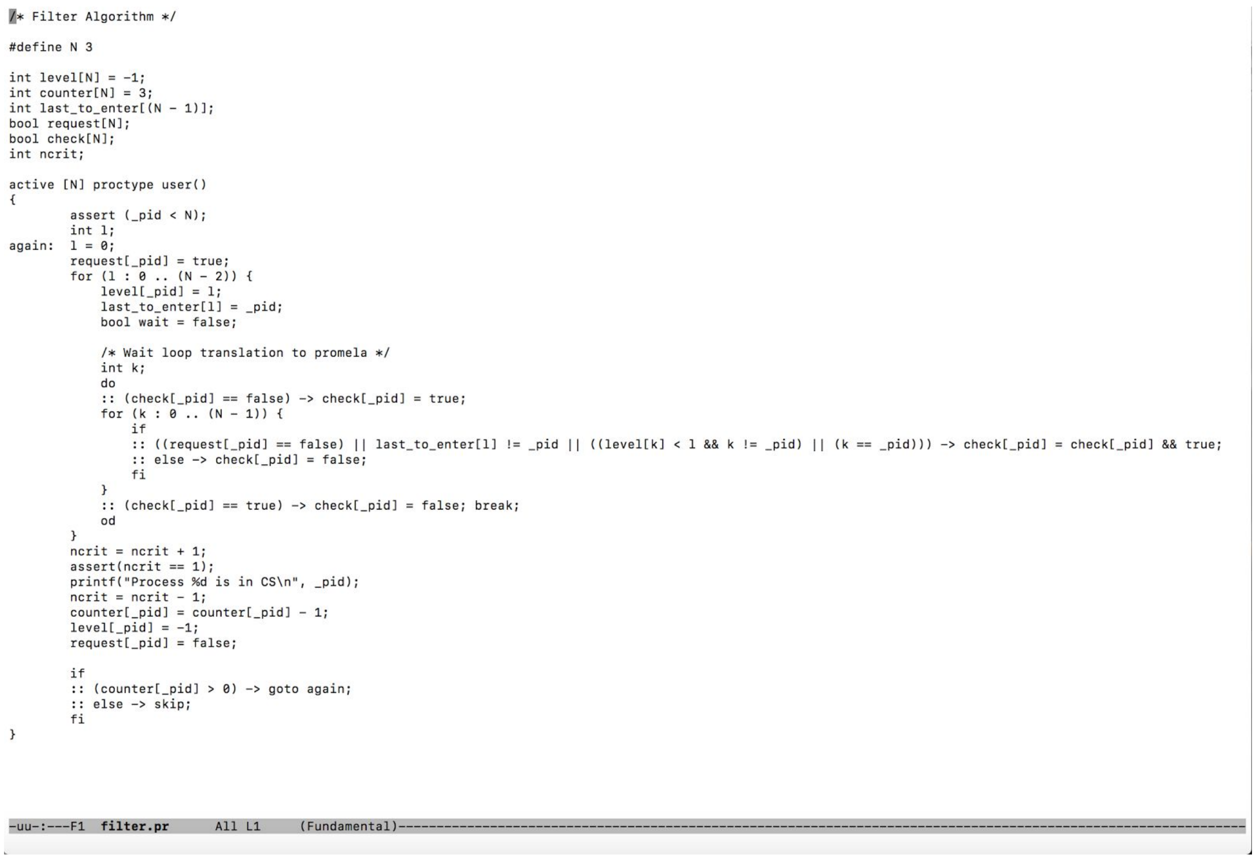
- For more information on promela, please see http://spinroot.com/spin/Doc/SpinTutorial.pdf

Appendix B: Figures (Code Samples and Output)



```
/* Tournament Algorithm based on Peterson's Algorithm for Two Processes */

#define N 4 /* Number of processes */
#define LEVEL 2 /* How many levels you need of competition for the tournament (log (N))
                   NOTE: can't be done in promela easily because of non-floating point numbers
                   and hard to use embedded c_code constructs that can't interact with promela code */

/* array type to imply two-dimensional arrays for wantCS and turn */
typedef array {
        int procs[N];
};


array wantCS[LEVEL];
array turn[LEVEL];
int ncrit;

active [N] proctype user()
{
        int node;
        int level;
        int mult;
        int path[LEVEL] = -1;
again:  mult = 1;
        level = 0;
        node = _pid;
        for (level : 0 .. (LEVEL - 1)) {
            int id = node % 2;
            node = node / 2;
            wantCS[level].procs[2 * node + id] = 1;
            path[level] = 2 * node + id;
            turn[level].procs[node] = 1 - id;
            do
            :: (wantCS[level].procs[2 * node + 1 - id] == 0 || turn[level].procs[node] == id) -> break;
            :: else -> skip;
            od
        }

        ncrit = ncrit + 1;
        assert(ncrit == 1);
        printf("Process %d is in CS\n", _pid);
        ncrit = ncrit - 1;

        level = LEVEL - 1;
        do
        :: (level >= 0) -> wantCS[level].procs[path[level]] = 0; level = level - 1;
        :: (level < 0) -> break;
        od

        goto again
}

-uu-:---F1  tournament.pr   All L1     (Fundamental)-----------------------------------------------
```

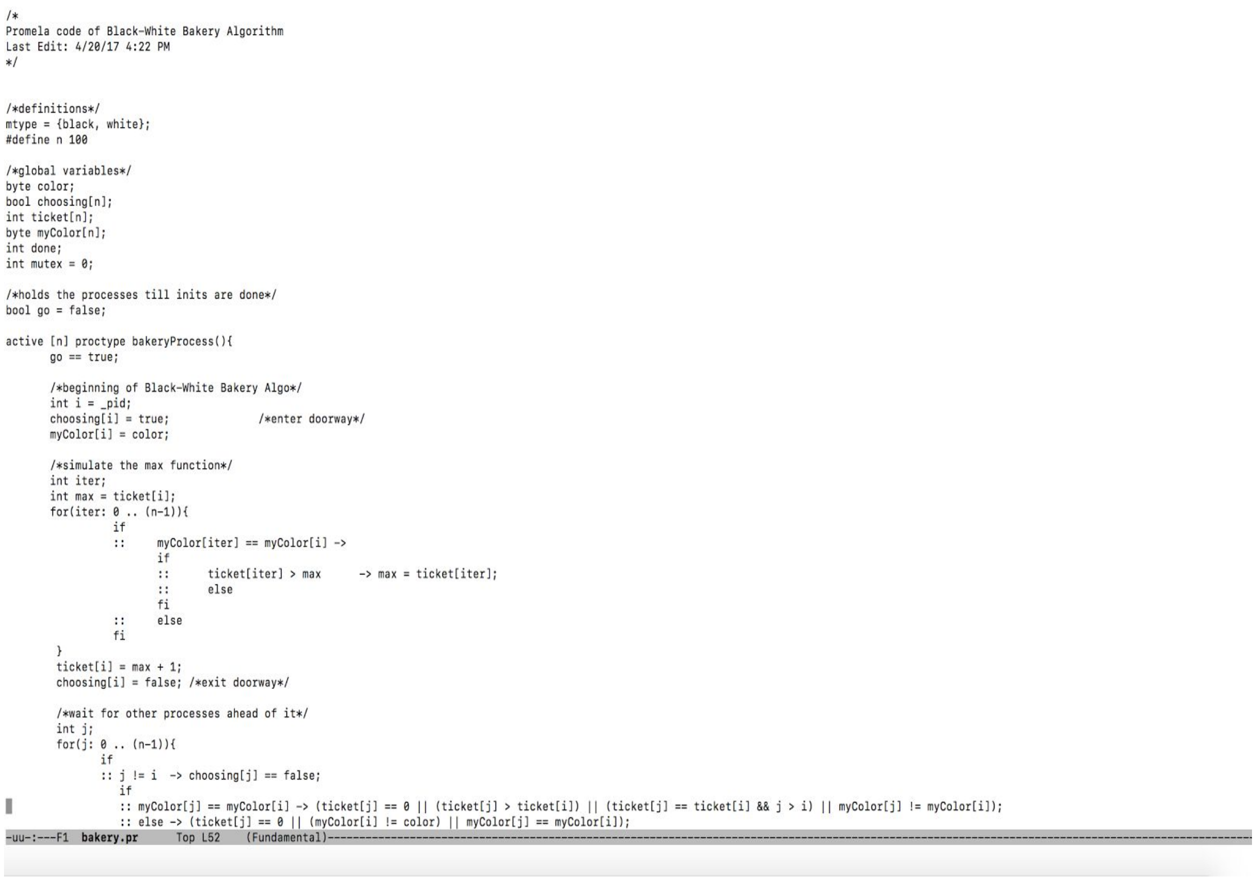
Figure B-1. Promela Code for Peterson’s Tournament Algorithm



```
/* Filter Algorithm */

#define N 3

int level[N] = -1;
int counter[N] = 3;
int last_to_enter[(N - 1)];
bool request[N];
bool check[N];
int ncrit;

active [N] proctype user()
{
        assert (_pid < N);
        int l;
again:  l = 0;
        request[_pid] = true;
        for (l : 0 .. (N - 2)) {
            level[_pid] = l;
            last_to_enter[l] = _pid;
            bool wait = false;

            /* Wait loop translation to promela */
            int k;
            do
            :: (check[_pid] == false) -> check[_pid] = true;
            for (k : 0 .. (N - 1)) {
                if
                :: ((request[_pid] == false) || last_to_enter[l] != _pid || ((level[k] < l && k != _pid) || (k == _pid))) -> check[_pid] = check[_pid] && true;
                :: else -> check[_pid] = false;
                fi
            }
            :: (check[_pid] == true) -> check[_pid] = false; break;
            od
        }
        ncrit = ncrit + 1;
        assert(ncrit == 1);
        printf("Process %d is in CS\n", _pid);
        ncrit = ncrit - 1;
        counter[_pid] = counter[_pid] - 1;
        level[_pid] = -1;
        request[_pid] = false;

        if
        :: (counter[_pid] > 0) -> goto again;
        :: else -> skip;
        fi
}
```

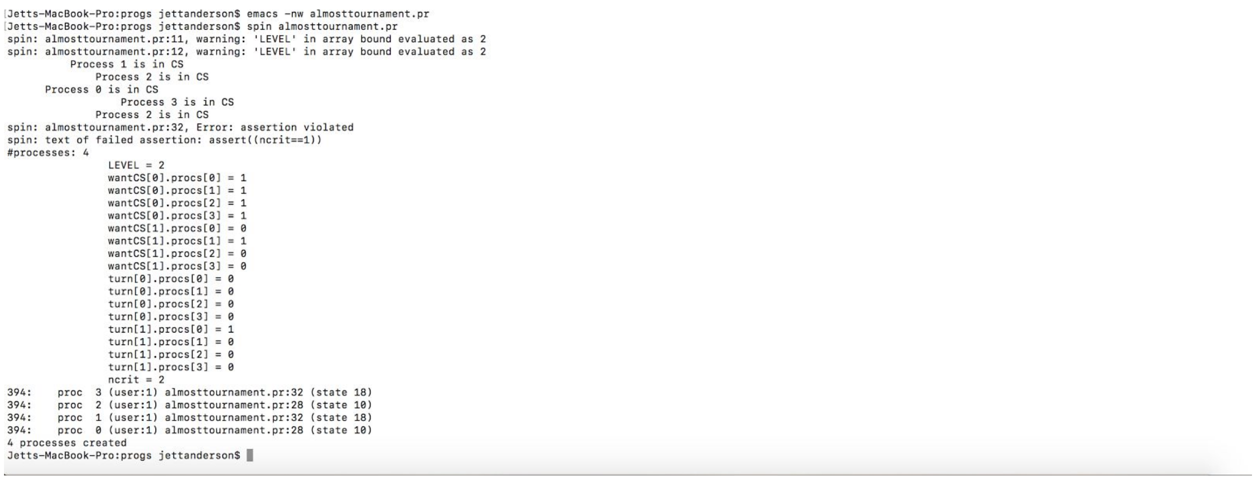
```
-uu-:---F1  filter.pr      All L1     (Fundamental)---------------------------------------------------------------------------------------------------------------
```

Figure B-2. Promela Code for N-Level Filter Algorithm



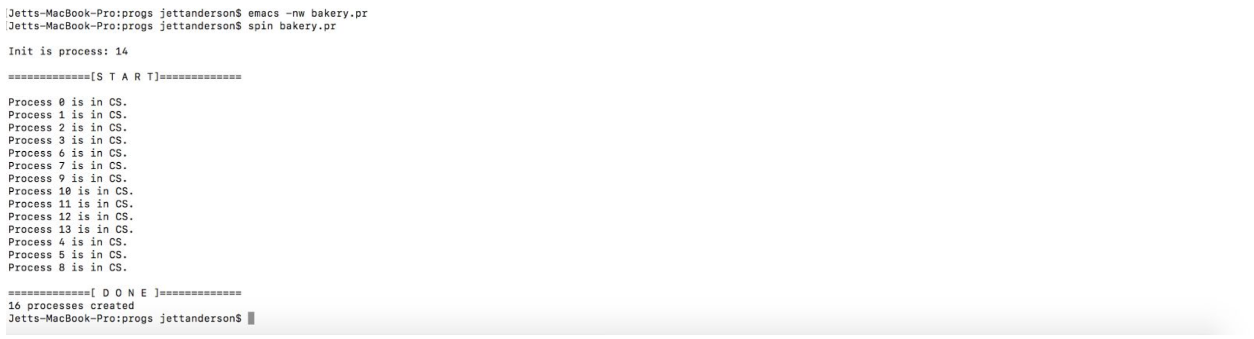
```
/*
Promela code of Black-White Bakery Algorithm
Last Edit: 4/20/17 4:22 PM
*/


/*definitions*/
mtype = {black, white};
#define n 100

/*global variables*/
byte color;
bool choosing[n];
int ticket[n];
byte myColor[n];
int done;
int mutex = 0;

/*holds the processes till inits are done*/
bool go = false;

active [n] proctype bakeryProcess(){
       go == true;

       /*beginning of Black-White Bakery Algo*/
       int i = _pid;
       choosing[i] = true;              /*enter doorway*/
       myColor[i] = color;

       /*simulate the max function*/
       int iter;
       int max = ticket[i];
       for(iter: 0 .. (n-1)){
                 if
                 ::     myColor[iter] == myColor[i] ->
                        if
                        ::      ticket[iter] > max      -> max = ticket[iter];
                        ::      else
                        fi
                 ::     else
                 fi
        }
        ticket[i] = max + 1;
        choosing[i] = false; /*exit doorway*/

        /*wait for other processes ahead of it*/
        int j;
        for(j: 0 .. (n-1)){
               if
               :: j != i  -> choosing[j] == false;
                  if
                  :: myColor[j] == myColor[i] -> (ticket[j] == 0 || (ticket[j] > ticket[i]) || (ticket[j] == ticket[i] && j > i) || myColor[j] != myColor[i]);
                  :: else -> (ticket[j] == 0 || (myColor[i] != color) || myColor[j] == myColor[i]);
-uu-:---F1  bakery.pr      Top L52    (Fundamental)----------------------------------------------------------------------
```

Figure B-3. Sample of Promela Code for Colored Bakery Algorithm



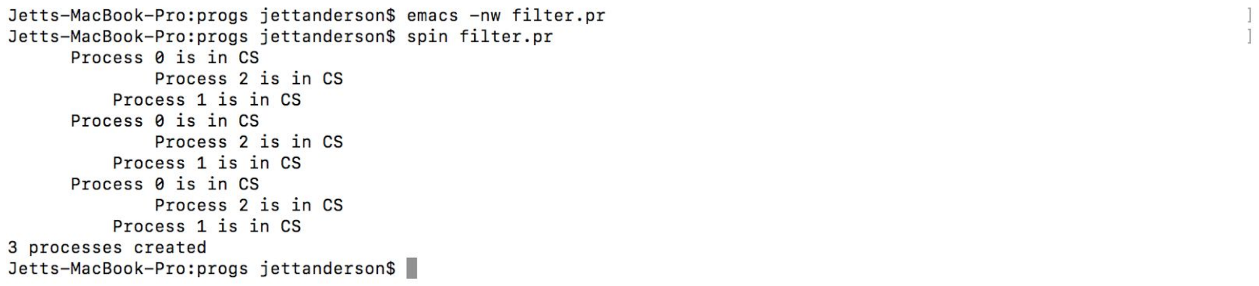
```
Jetts-MacBook-Pro:progs jettanderson$ emacs -nw almosttournament.pr
Jetts-MacBook-Pro:progs jettanderson$ spin almosttournament.pr
spin: almosttournament.pr:11, warning: 'LEVEL' in array bound evaluated as 2
spin: almosttournament.pr:12, warning: 'LEVEL' in array bound evaluated as 2
          Process 1 is in CS
              Process 2 is in CS
      Process 0 is in CS
                  Process 3 is in CS
              Process 2 is in CS
spin: almosttournament.pr:32, Error: assertion violated
spin: text of failed assertion: assert((ncrit==1))
#processes: 4
                LEVEL = 2
                wantCS[0].procs[0] = 1
                wantCS[0].procs[1] = 1
                wantCS[0].procs[2] = 1
                wantCS[0].procs[3] = 1
                wantCS[1].procs[0] = 0
                wantCS[1].procs[1] = 1
                wantCS[1].procs[2] = 0
                wantCS[1].procs[3] = 0
                turn[0].procs[0] = 0
                turn[0].procs[1] = 0
                turn[0].procs[2] = 0
                turn[0].procs[3] = 0
                turn[1].procs[0] = 1
                turn[1].procs[1] = 0
                turn[1].procs[2] = 0
                turn[1].procs[3] = 0
                ncrit = 2
394:    proc  3 (user:1) almosttournament.pr:32 (state 18)
394:    proc  2 (user:1) almosttournament.pr:28 (state 10)
394:    proc  1 (user:1) almosttournament.pr:32 (state 18)
394:    proc  0 (user:1) almosttournament.pr:28 (state 10)
4 processes created
Jetts-MacBook-Pro:progs jettanderson$
```

Figure B-4. Incorrect Output Showing Mutex Violation in Sample Broken Tournament Program



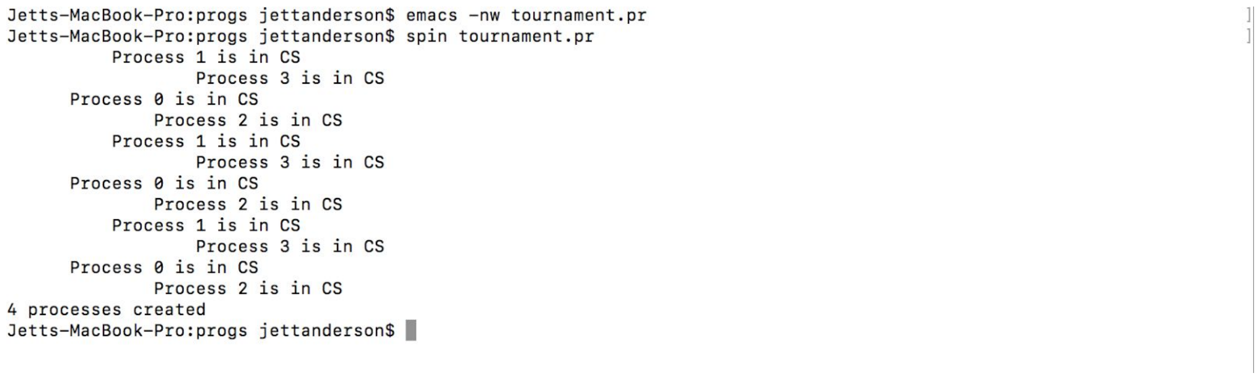
```
Jetts-MacBook-Pro:progs jettanderson$ emacs -nw bakery.pr
Jetts-MacBook-Pro:progs jettanderson$ spin bakery.pr

Init is process: 14

=============[S T A R T]=============

Process 0 is in CS.
Process 1 is in CS.
Process 2 is in CS.
Process 3 is in CS.
Process 6 is in CS.
Process 7 is in CS.
Process 9 is in CS.
Process 10 is in CS.
Process 11 is in CS.
Process 12 is in CS.
Process 13 is in CS.
Process 4 is in CS.
Process 5 is in CS.
Process 8 is in CS.

=============[ D O N E ]=============
16 processes created
Jetts-MacBook-Pro:progs jettanderson$ █
```

Figure B-5. Correct Output for Colored Bakery Algorithm with N = 14 Processes



```
Jetts-MacBook-Pro:progs jettanderson$ emacs -nw filter.pr
Jetts-MacBook-Pro:progs jettanderson$ spin filter.pr
      Process 0 is in CS
              Process 2 is in CS
          Process 1 is in CS
      Process 0 is in CS
              Process 2 is in CS
          Process 1 is in CS
      Process 0 is in CS
              Process 2 is in CS
          Process 1 is in CS
3 processes created
Jetts-MacBook-Pro:progs jettanderson$ █
```

Figure B-6. Correct Output for N-Level Filter Algorithm with N = 3 Processes



```
Jetts-MacBook-Pro:progs jettanderson$ emacs -nw tournament.pr
Jetts-MacBook-Pro:progs jettanderson$ spin tournament.pr
          Process 1 is in CS
                  Process 3 is in CS
      Process 0 is in CS
              Process 2 is in CS
          Process 1 is in CS
                  Process 3 is in CS
      Process 0 is in CS
              Process 2 is in CS
          Process 1 is in CS
                  Process 3 is in CS
      Process 0 is in CS
              Process 2 is in CS
4 processes created
Jetts-MacBook-Pro:progs jettanderson$ █
```

Figure B-7. Correct Output for Tournament Algorithm with N = 4 Processes

References

[1] Gerard J. Holzmann, “Verifying Multi-threaded Software with Spin”. [Online]. Available: http://spinroot.com/spin/what.html. [Accessd: 25-April-2017].

[2] Ted Herman, “N-Process Mutual Exclusion”. [Online]. Available: https://weblog.cs.uiowa.edu/cs5620f15/N-Process%20Mutual%20Exclusion. [Accessed: 25-April- 2017].

[3] H. Maurice, S. Nir, “The Art of Multiprocessor Programming,” 2012, p. 28–31

[4] Gadi Taubenfeld, “The Black-White Bakery Algorithm”, 2004, pp. 10-13.

[5] Daniel Kroening, “Bounded Model Checking for Software”. [Online]. Available: http://www.cprover.org/cbmc/. [Accessed: 25-April-2017].