Model checking with SPIN

Software Analysis, Spring 2024

Luca Di Bello

May 15, 2024

Contents

1 Introduction 1

2 ProMeLa Model 2

2.1 Sequential Process 3

2.2 Parallel Process 3

2.2.1 Worker Process 4

2.3 Parameters: MAX and LENGTH 5

3 LTL Properties 6

3.1 Verify completition of both sequential and parallel processes 6

3.2 Sum of frequencies 6

3.3 Invalid LTL formula: partial result 6

4 Verification with SPIN 8

1 Introduction

In this assignments examines the implementation of model checking using the SPIN tool to verify the correctness of two versions of a frequency counter program: one sequential and the other parallel. Model checking is a technique that allows to verify the correctness certain properties of a system described in a finite-state model. In the following sections will be discussed how the program has been

modeled using ProMeLa language, which Linear Temporal Logic (LTL) properties have been defined to verify the correctness of the program, and how the verification has been performed using SPIN.

2 ProMeLa Model

The ProMeLa model consists of two main processes: the first process handles the sequential computation of frequency counts, storing results in an array `sequential_counts`. The second process on the other hand, initiates parallel computation, spawning worker processes for each possible value in the input array. These workers update an array `parallel_counts` concurrently. Race conditions are avoided as each worker updates a unique position in the array.

As explicitly stated in the assignment, the ProMeLa model presents two constrants:

1. `MAX`: It represents the maximum value that can be assigned to an element in the array. Used while filling the input array with random values.
2. `LENGTH`: the length of the input array.

The model presents an `init` block that initializes the input array with random values between 0 and `MAX` and starts both the sequential and parallel processes. The code is available in listing 1.

```
// Define the maximum number of elements in the array
#define MAX 2
#define LENGTH 2

// Define the variables
int a[LENGTH];

// Keep track of result of both versions of the program
int sequential_counts[MAX + 1];
int parallel_counts[MAX + 1];

// Entry point of the program
init {
  // Initialize the array non-deterministically
  printf("Random state:\n")
  int i;
  for (i : 0 .. LENGTH - 1) {
    // Select a random value for the array
    int v;
    select(v : 0 .. MAX);
    // Assign the value to the array
    a[i] = v;

    // Print the value
    printf("\ta[%d] = %d\n", i, v);
  }

  // Run the sequential version of the program
  printf("Running sequential version...\n");
  run sequentialCounter();

  // Run the parallel version of the program
  printf("Running parallel version...\n");
  run parallelCounter();
}
```

Listing 1: ProMeLa array initialization and start of sequential and parallel processes

As is possible to see from the code above, both versions of the program have been started using the `run` command inside the same ProMela model. This is different from the Java implementation, where the two versions were implemented in separate classes. This design choice was essential to allow the verification of both versions of the program in the same model as otherwise, it would be impossible to verify by confronting data yielded by two different simulations.

In the abstraction, the sequential and parallel processes are implemented as separate processes. An in depth analysis of each process is provided in the following sections.

2.1 Sequential Process

The sequential version of the frequency counter program is implemented in the `sequentialCounter` process. This process iterates over the input array and increments the corresponding position in the `sequential_counts` array.

The ProMeLa implementation of this version of the program is almost identical to the Java implementation. This was expected, as both languages uses a C-like syntax, and the logic of the program is simple. The code is available in listing 3.

```java
public int mostFrequent() {
    int mostFrequent = -1;
    int maxFrequency = -1;
    int[] frequencies = new int[max + 1];
    for (int k = 0; k < a.length; k++) {
        int value = a[k];
        frequencies[value] += 1;
        int frequency = frequencies[value];
        if (frequency > maxFrequency) {
            mostFrequent = value;
            maxFrequency = frequency;
        }
    }
    return mostFrequent;
}


```

Listing 2: Java sequential version of the frequency counter program

```
proctype sequentialCounter() {
    int maxFrequency = -1;
    int k;
    for (k : 0 .. LENGTH - 1) {
        int value = a[k];
        sequential_counts[value] = sequential_counts[value] + 1;
        if
        :: sequential_counts[value] > maxFrequency ->
            maxFrequency = sequential_counts[value];
            sequential_result = value;
        :: else -> skip;
        fi
    }
    // Signal that the sequential version is done
    sequentialDone = 1;
}

```

Listing 3: ProMeLa sequential version of the frequency counter program

The main difference between the Java and ProMeLa implementation is the way the program yields the result: in the Java implementation, the result is simply returned by the method, while in the ProMeLa implementation, the result is stored in a global variable `sequential_result`. This is necessary as the process cannot return a value.

2.2 Parallel Process

The parallel version of the frequency counter program is implemented in the `parallelCounter` process. This process spawns `MAX + 1` worker processes that concurrently count the frequency of each value in the input array. Each worker process is started with a unique value to count, and by iterating over the input array, increments the corres from the channelponding position in the `parallel_counts` array.

After all worker processes have completed, the main parallel counter process iterates throgh the results and saves the value with the highest frequency inside the `parallel_result` variable.

To be able to detect when all worker processes have completed, I decided to use a *channel* to synchronize the main process with the workers. The channel created in the main process and passed as an argument to each worker process. As soon as a worker process completes, it sends its PID through passed channel. As we start `MAX + 1` worker processes, we expect to read `MAX + 1` values from the channel. By leveraging this, the *parallelCounter* process is able to detect when all worker processes have completed. The code is available in listing 4.

```
proctype parallelCounter() {
  // Create channel to wait for workers to finish
  chan joinCh = [MAX + 1] of { pid };
  // Create array of PID's for workers
  pid workers[MAX + 1];
  // Create a worker for each possible value
  int i;
  for (i : 0 .. MAX) {
    workers[i] = run parallelWorker(i, joinCh);
  }
  // Wait for all workers to finish by reading from the channel
  for (i : 0 .. MAX) {
    int done;
    joinCh ? done;
  }
  // If we are here, all workers are done as we read MAX+1 values from the channel
  printf("\t [!] all workers are done\n");
  ...
}
```

Listing 4: ProMeLa parallel frequency counter - worker synchronization with channel

This implementation differs from the Java implementation, where it does not wait for all threads to complete but instead starts them all together and then, waits for each thread singularly to complete, to save its result. In ProMeLa, we start the worker processes right away and use a channel to wait for all of them to complete before saving the result.

2.2.1 Worker Process

The worker process has been implemented in the `parallelWorker` process. As cited in the previous section, each worker process is started with a unique value to count and increments the corresponding position in the `parallel_counts` array. The code is available in listing 6.

```java
protected int frequencyOf(int n) {
    int frequency = 0;
    for (int value: a) {
      if (value == n)
        frequency += 1;
    }
    return frequency;
}

class ThreadedCounter extends SequentialCounter implements Runnable
{
   private int frequency;
   private int n;

   ThreadedCounter(int[] a, int n, int max) {
      super(a, max);
      this.n = n;
   }

   public void run() {
      frequency = frequencyOf(n);
   }

   public int frequency() {
      return frequency;
   }
}

```

Listing 5: Java parallel worker thread implementation leveraging Threads

```
// The worker process represents a thread that looks for the count of a specific value in the array
proctype parallelWorker(int value; chan out) {
  // Look for the value in the array
  int frequency = 0;
  int i;
  for (i : 0 .. LENGTH - 1) {
    if
      :: a[i] == value -> frequency = frequency + 1;
      :: else -> skip;
    fi
  }
  // Update the value in the parallel counts array
  parallel_counts[value] = frequency;
  printf("Worker for value %d is done\n", value);
  out ! _pid;
}

```

Listing 6: ProMeLa parallel frequency counter - worker process

As mentioned above, each worker process counts the frequency of a specific value in the input array. The worker process is implemented in the `parallelWorker` process. The code is available in listing 6.

As also described in the caption of the listings above, the main difference between the two implementations is how the worker process is started. In the Java implementation, this required the creation of a specific class that implements the `Runnable` interface, while in the ProMeLa implementation, this is simply a process that is started by the main `parallelCounter` process.

A small implemetation detail is that the Java worker, leveraging an external function named *frequencyOf*, only computes the frequency of a specific value, while the ProMeLa worker process needs to handle both this logic, but also the saving of the result in the `parallel_counts` and the synchronization with the parent process using the channel (as explained in detail in section 2.2).

2.3 Parameters: MAX and LENGTH

If MAX and LENGTH are too high, the model checking process can take a long time to complete, or even run out of memory. To avoid this, I decided to set the values of MAX and LENGTH to 2. This way, the model checking process is fast and efficient, and it is possible to verify the correctness of the program in a reasonable amount of time.

3 LTL Properties

3.1 Verify completition of both sequential and parallel processes

To be able to verify the completition of both the sequential and parallel processes, I decided to use two global variables: `sequentialDone` and `parallelDone`. These variables are set to 0 by default, and set to 1 when the respective process has completed. To verify the completition of both processes, I defined the following LTL property:

```
ltl termination { <> (sequentialdone == 1 && paralleldone == 1) }
```

Listing 7: LTL properties to verify the completition of both sequential and parallel processes

3.2 Sum of frequencies

As we are interested in verifying the correctness of the frequency counter program, I decided to add a new LTL properety that ensures that the sum of the frequencies of all values in the `sequential_counts` and `parallel_counts` arrays is equal to the length of the input array. This property is defined as follows:

```
ltl sumCounts { [] (sequentialDone == 1 && parallelDone == 1) -> (sumCountsSequential
    == LENGTH && sumCountsParallel == LENGTH) }
```

Listing 8: LTL properties to verify the sum of frequencies in the sequential and parallel arrays

As also possible to see from the code above, to verify this property I had to add two counters, `sumCountsSequential` and `sumCountsParallel`, that are computed by summing the frequencies of all values in the respective arrays right before completing the respective processes. This way, I can verify that the sum of the frequencies is equal to the length of the input array.

3.3 Invalid LTL formula: partial result

As requesed by the assignment, I also added and invalid LTL formula that SPIN will not be able to verify. The property I decided to add is the following:

```
ltl alwaysSameResult { [](sequential_result == parallel_result) }
```

Listing 9: Invalid LTL property that SPIN will not be able to verify

This LTL is wrong, as the two results are not expected to be the same while the processes are still running. This is because the two versions of the program are executed concurrently, and, while the sequential frequency counter keeps tracks of the most frequent value, the parallel counter does not. In fact, the latter will update the `parallel_result` variable only after all worker processes have completed. This means that the two results will always be different while the processes are still running.

SPIN gives th following error when trying to verify this property:

```
ltl notSameResult: [] ((sequential_result==parallel_result))
Never claim moves to line 62    [(1)]
          Random state:
                a[0] = 0
                a[1] = 0
          Running sequential version...
          Running parallel version...
spin: _spin_nvr.tmp:61, Error: assertion violated
spin: text of failed assertion: assert(!(!((sequential_result==parallel_result))))
Never claim moves to line 61    [assert(!(!((sequential_result==parallel_result))))]
spin: trail ends after 41 steps
```

```
#processes: 2
                a[0] = 0
                a[1] = 0
                sequential_counts[0] = 1
                sequential_counts[1] = 0
                sequential_counts[2] = 0
                parallel_counts[0] = 0
                parallel_counts[1] = 0
                parallel_counts[2] = 0
                sequential_result = 0
                parallel_result = -1
                sequentialDone = 0
                parallelDone = 0
                sumCountsSequential = 0
                sumCountsParallel = 0
 41:    proc  1 (sequentialCounter:1) ../src/promela/model.pml:51 (state 12)
 41:    proc  0 (:init::1) ../src/promela/model.pml:42 (state 22)
 41:    proc  - (notSameResult:1) _spin_nvr.tmp:60 (state 6)
```

4 Verification with SPIN