Assignment 4 – Software Analysis

Model checking with Spin

Claudio Maggioni

1 Introduction

This assignment consists in using model checking tecniques to verify the correctness of the algorithm implemented in an existing program. In particular, a sequential and a multi-threaded implementation of a array-reversing Java utility class implementation are verified to check correctness of both reversal procedures, consistency between the results they produce and for absence of race conditions.

To achieve this I use the Spin model checker [2] to write an equivalent finite state automaton implementation of the algorithm using the *ProMeLa* specification and define linear temporal logic (LTL) properties to be automatically verified.

This report covers the definition of the model to check and the necessary LTL properties to verify correctness of the implementation, and additionally presents a brief analysis on the performance of the automated model checker.

2 Model definition

In this section I define the *ProMeLa* code which implements a FSA model of the Java implementation. The model I define does not match the exact provided Java implementation, but aims to replicate the salient algorithmic and concurrent behaviour of the program.

Due to the way I implement the LTL properties in the following section, I decide to implement the model as a GNU M4 macro processor [1] template file. Therefore, the complete model can be found in the path `ReverseModel/reversal.pml.m4` in the assignment repository

usi-si-teaching/msde/2022-2023/software-analysis/maggioni/assignment-4

on *gitlab.com*.

As suggested by the assignment description, I define some preprocessor constants to allow for altering some parameters. As mentioned above, I use GNU M4 instead of the regular *ProMeLa* preprocessor to implement these definitions. Specifically, I define the following properties:

N, which represents the number of parallel threads spawned by the parallel reverser;

LENGTH, which represents the length of the array to reverse;

R, which represents the upper bound for the random values used to fill the array to reverse, the lower bound of them being 0.

The variable values are injected as parameters of the `m4` command, so no definition is required in the model code.

Then by using these values the model specification declares the following global variables:

```
int to_reverse[LENGTH];
int reversed_seq[LENGTH];
int reversed_par[LENGTH];
bool done[N + 1];
bool seq_eq_to_parallel = true;
```

`to_reverse` is the array to reverse, and `reverse_seq` and `reverse_par` are respectively where the sequential and parallel reverser store the reversed array. The `done` array stores an array of boolean values: `done[0]` stores whether the sequential reverser has terminated, and each `done[i]` for $1 \leq i \leq N$ stores whether the i-th spawned thread of the parallel reverser has terminated (consequently, since threads are joined in order, when `done[N] == true` the parallel reverser terminates). Finally `seq_eq_to_parallel` is set to `false` when an incongruence between `reversed_seq` and `reversed_par` is found after termination of both reversers.

The body of the model is structured in the following way:

```
init {
    { /* array initialization */ }
    { /* sequential reverser algorithm */ }
    { /* parallel reverser algorithm */ }
    { /* congruence check between reversers */ }
}
```

Each of the enumerated sections is surrounded by curly braces to emulate the effect of locally scoped variables in procedures, which do not exist in *ProMeLa* aside the concurrency emulating `proctype` construct.

The array initialization is carried out as follows:

```
int i;
for (i in to_reverse) {
    int value;
    select(value: 0 .. R);
    to_reverse[i] = value;
    printf("to_reverse[%d]: %d\n", i, value);
}
```

As specified above, the array is initialized with values in $[0, R]$. Specifically, values are generated using a nondeterministic `select` statement to allow the model checker to try all possible values efficiently.

The sequential reversed algorithm is implemented with the following code:

```
int k;
for (k: 0 .. (LENGTH - 1)) {
    reversed_seq[LENGTH - k - 1] = to_reverse[k];
    printf("reversed_seq[%d] = to_reverse[%d]\n", LENGTH - k - 1, k);
}
done[0] = true;
```

which is a direct translation of the Java implementation to verify.

The sequential reverser is used to implement each thread of the parallel reverser through the *ThreadedReverser* class. In the model, the class is translated in a Spin process through the `proctype` construct with the following implementation:

```
proctype ThreadedReverser(int from; int to; int n) {
    printf("proc[%d]: started from=%d to=%d\n", n, from, to);
    int k;
    for (k: from .. (to - 1)) {
        printf("reversed_par[%d] = to_reverse[%d]\n", LENGTH - k - 1, k);
        reversed_par[LENGTH - k - 1] = to_reverse[k];
    }
    printf("proc[%d]: ended\n", n);
    done[n] = true;
}
```

The implementation is closely related to the sequential one, as it differs only in `reversed_par` being used as the destination array and limiting the reversal between `from` and `to`. The argument `n` is used to identify the thread in the `done` array to store the termination state. Following the indexing rules of `done` given earlier, the i-th spawned thread corresponds to a `proctype` call with $n = i$, so that at termination `done[i]` is set to `true`.

The actual thread-spawning part of the parallel reverser, i.e. class *ParallelReverser* itself, is represented by the following *ProMeLa* code placed in the `init` section of the model:

```
int n;
int s = LENGTH / N;
for (n: 0 .. (N - 1)) { // fork loop
    int from = n * s;
    int to;
    if
    :: (n == N - 1) -> to = LENGTH;
    :: else -> to = n * s + s;
    fi
    run ThreadedReverser(from, to, n + 1); // fork here
}
for (n: 1 .. N) { // join loop
    (done[n] == true); // join n-th thread here
    printf("[%d] joined\n", n);
}
```

Here the values of `n`, `s`, `from` and `to` replicate exactly the values used in the Java implementation. The `n + 1` parameter identifies maps each `proctype` invocation to its place in the invocation order (e.g. for $n = 0$, `ThreadedReverser` is called with $n + 1 = 1$, since this is the 1st invocation of the process).

The second "join" loop waits for each process to complete in order of invocation, replication the thread joining behaviour of the parallel reverser implementation in the Java program. Note that the *ProMeLa* statement `(done[n] == true);` will "wait" for the value of `done[n]` to be `true` before executing the following statement, thus being an adequate analogy for `Thread.join()`.

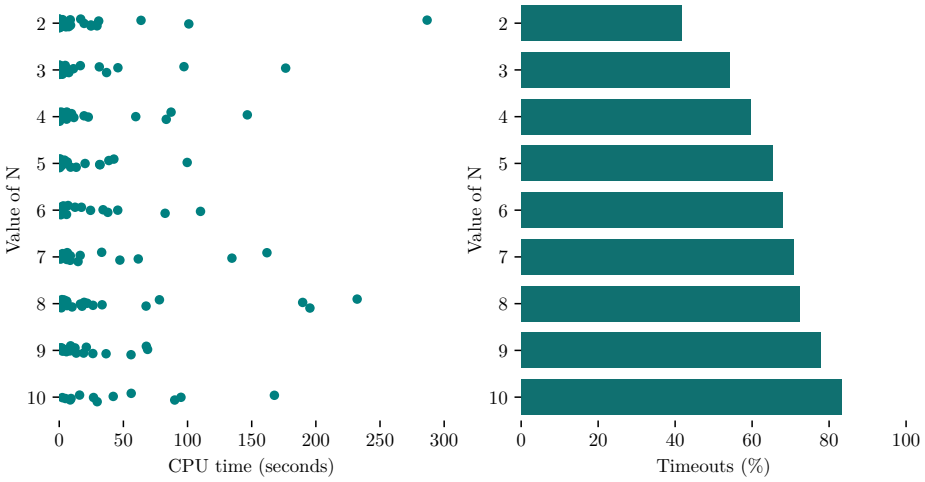
Finally, the congruence check between the arrays produced by both implementation is implemented with the following code:

```
int i;
for (i: 0 .. (LENGTH - 1)) {
    if
    :: (reversed_seq[i] != reversed_par[i]) -> seq_eq_to_parallel = false;
    fi
}
```

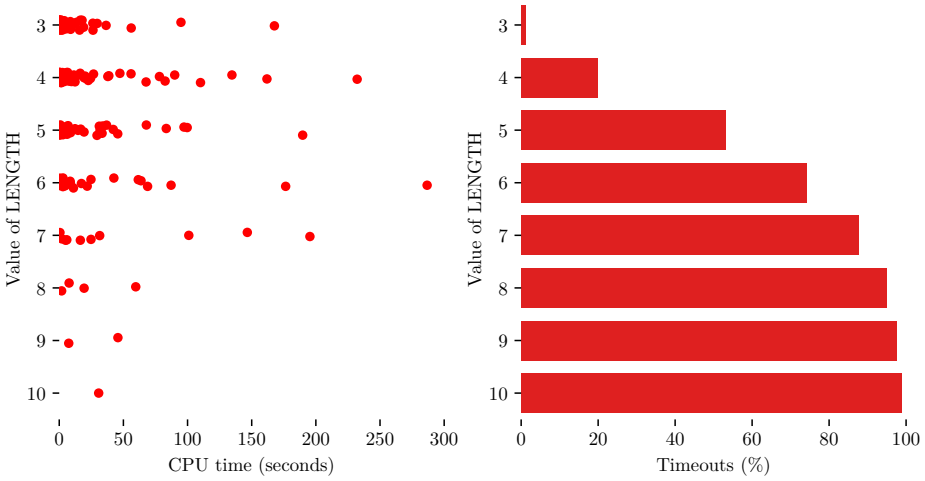
Should any matching pair of elements be different, `seq_eq_to_parallel` will be set to `false`. Note that this boolean variable is used to implement one of the LTL properties, hence why it is declared and set to a meaningful value in this block of the model.

2.1 LTL properties

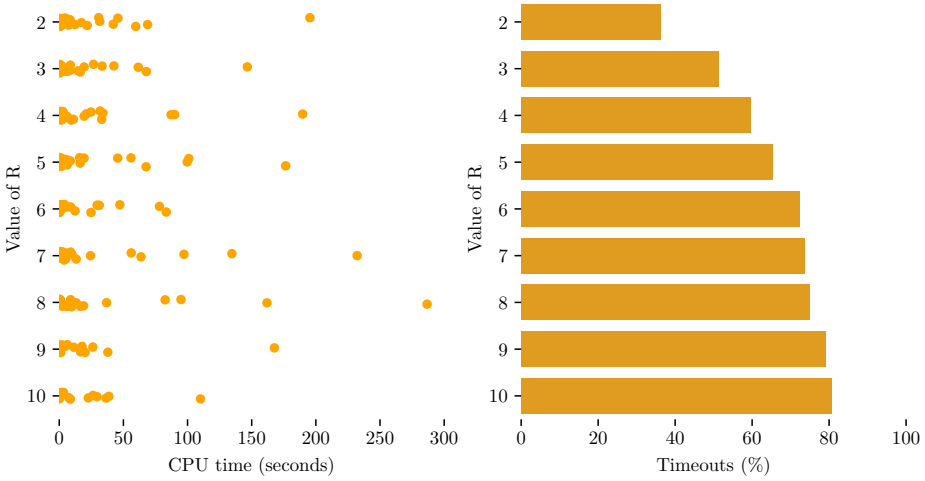
The citation [3]. The citation [2].



(a) Variable `N`



(b) Variable `LENGTH`



(c) Variable `R`

Figure 1: Distribution of CPU time and percentage of timeouts (i.e. executions with a real execution time greater than 5 minutes, discarded for sake of time) for different executions of the model checker for different parameters of `N`, `LENGTH` and `R`.

References

[1] Free Software Foundation. *GNU M4 - GNU macro processor*. Version 1.4.6. May 29, 2021. URL: https://www.gnu.org/software/m4/manual/index.html.

[2] Gerard J. Holzmann. *Spin model checker*. Version 6.5.2. Dec. 6, 2019. URL: https://spinroot.com/spin/whatispin.html.

[3] Ole Tange. *GNU Parallel 20230422 ('Grand Jury')*. GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them. Apr. 2023. DOI: 10.5281/zenodo.7855617. URL: https://doi.org/10.5281/zenodo.7855617.