# 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, an effect that is exploited by the main model body implementation to wait for it. 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 */
    run SequentialReverser();
    /* parallel reverser algorithm */
    run ParallelReverser();

    (done[0] == true && done[N] == true);

    { /* 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.

As requested by the assignment, the sequential and parallel reverser are implemented in a `proctype` and spawned in parallel in the model. The two *ProMeLa* processes join before the congruence check thanks to an “expression” statement waiting on the termination boolean array to signal that both reversers have finished doing their job.

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 `ParallelReverser proctype`:

```
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

In this section I cover the LTL property definitions I included in the model.

```
ltl seq_eq_parallel {
    [] (seq_eq_to_parallel == true)
}
```

This LTL property definition checks that once both reversers have terminated, the content of the respective reversed arrays they produce is the same. As discussed in the previous section, this variable can only turn to false during the execution of the congruence check and only if a pair of array elements of same index is indeed different. Therefore, if the program is correct, the value of the variable will always be true.

Note that this property does not ensure termination of the program, at it relies on the congruence check to eventually run at the end of the program. To ensure termination, I define the following LTL property:

```
ltl termination {
    <> (done[0] == true && done[N] == true)
}
```

This mirrors the wait statement introduced in the model code before the congruence check block, and relies exactly in the same way on the termination boolean array. Note that the elements of the array can only turn from `false` to `true` or not change at all, thus the property in the "eventually" operator is actually always true after it becomes indeed true (i.e. the program cannot un-terminate according to the model).

I then define other two custom properties showcasing the powers of the M4 macro processor when compared to the built-in *ProMeLa* one.

```
// ifelse(LTL, correctness_seq, `
ltl correctness_seq {
    [] (done[0] == true -> (true for(`k', 0, LENGTH-1, ` &&
        r_s[eval(LENGTH - k - 1)] == a[k]')))
}
// ', `')
```

This property checks if the array produced by the sequential reverser is indeed the reverse of the input array. Note that the "polyglot" M4 sugar allows for the property to be arbitrairly unraveled based on the value of `LENGTH`. Notice that to simplify the *ProMeLa* source code to compile for long array lengths, thanks to the `ifelse` macro the property is omitted by M4 when the property is not actually checked (because long LTL properties actually make the model fail to parse). Here is an example of the unravelled property for `LENGTH = 10`:

```
ltl correctness_seq {
    [] (done[0] == true -> (true  &&
        r_s[9] == a[0] &&
        r_s[8] == a[1] &&
        r_s[7] == a[2] &&
        r_s[6] == a[3] &&
        r_s[5] == a[4] &&
        r_s[4] == a[5] &&
```

```
        r_s[3] == a[6] &&
        r_s[2] == a[7] &&
        r_s[1] == a[8] &&
        r_s[0] == a[9]))
}
```

Note that the use of true as the first condition after the implication is simply to allow for && to be simply appended at the start of each condition.

In a similar fashion, I also define a property that is not generally true. The following LTL formula specifies that when the second thread of the parallel reverser terminates, the section to be reversed by the first thread is already reversed.

```
// ifelse(LTL, correctness_par, `
ltl correctness_par {
    [] (done[2] == true -> (true for(`k', 0, LENGTH / N - 1, ` &&
        r_p[eval(LENGTH - k - 1)] == a[k]')))
}
// ', `')
```

This property is clearly not generally true as the first thread may not complete the reversal process before the second thread terminates due to concurrency.

Here is the counterexample Spin provides to show that the property does not hold for some executions with $N = 2$, LENGTH $= 3$ and $R = 2$:

```
program start
a[0]: 1
a[1]: 0
a[2]: 0
sequential start
parallel start
r_s[2] = a[0]
r_s[1] = a[1]
r_s[0] = a[2]
proc[1]: started from=0 to=1
proc[2]: started from=1 to=3
proc[2]: r_p[1] = a[1]
proc[2]: r_p[0] = a[2]
proc[2]: ended
```

Indeed, the output shows that the second thread can terminate before the first does. Here, the first thread (`proc[1]`) is just started but does not manage to execute `r_p[2] = a[0]` before the second thread terminates. Indeed, this is a realistic counterexample that could be reproduced in the Java program.

# 3 Model Execution

To compile and check all LTL properties included in the model execute the command:

```
./ReversalModel/run.sh all $N $LENGTH $R
```

from the root git directory of the assignment, where `$N` is the value of the variable $N$, `$LENGTH` is the length of the array and `$R` is the value of $R$. The parameter `all` can be replaced by the name of an LTL property to only check the model for that property.

### 3.1 Execution limits

To better gauge the efficiency of the model checker algorithm, I run the model to check the LTL property `seq_eq_parallel` for all possible combinations of the values of $N \in [2, 10]$, LENGTH $\in [3, 10]$ and $R \in [2, 10]$. The script `ReversalModel/grid-search.sh` implements this procedure with the following GNU Parallel [3] command, allowing for parallel execution of up to 4 model instances:

```
parallel --jobs 4 "$SCRIPT_DIR/test-property.sh" ::: "$prop" ::: "$(seq 2 10)" :::
↪  "$(seq 3 10)" ::: "$(seq 2 10)" ::: "$SCRIPT_DIR/$time_out"
```

The `test-property.sh` script times out the model execution after 5 minutes of `real` execution time due to time constraints. Thus, for each combination of parameters, I measure the process time (`user` and `sys` time) it takes to verify the LTL property and, if the execution times out, whether this happens or not. I then summarize the execution times for successful runs and the frequency of timeouts grouping the execution by each variable in figure 1.

It is apparent that even small increases in the array length significantly hinder the performance of the model checking algorithm and increasee the number of timeouts. Increasing the number of processes $N$ instead shows a tamer effect on execution time, and avoids a significantly greater number of timeouts for the values 8, 9, and 10. Finally, increasing upper bound on the values of the array ($R$) has the least significant effect, albeit being a close second to the distribution for $N$.

## 4 Conclusions

I find the Spin model checker a relatively straightforward tool to use. The *ProMeLa* model implementation was easy to carry out due to syntax similarities with C, and no tweaks were required after writing it to make the model work.

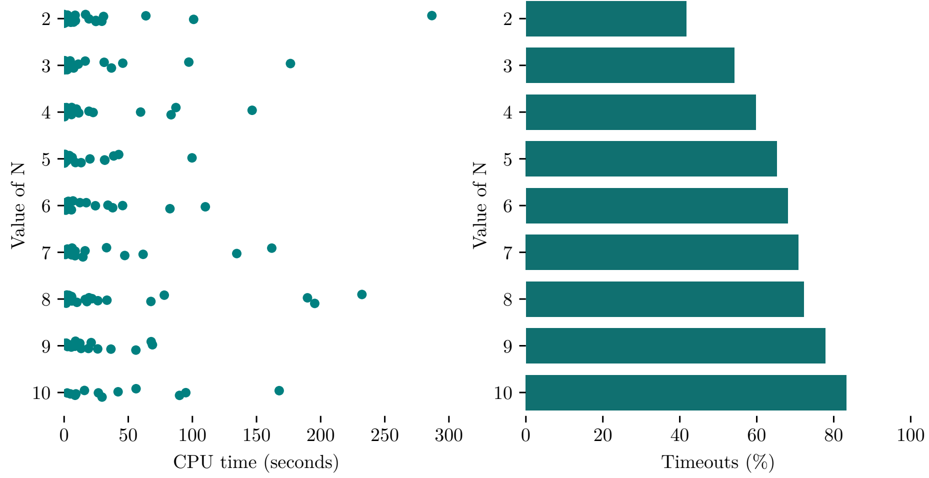
The model manages to realistic model program behaviour of the Java reverser implementations, the moder checker did not flag any surprising behaviour w.r.t. my understanding of the original code. It is however interesting that the `correctness_par` property was invaidated with such an effective counterexample, and it shows the value of the tool for analysis of (albeit simple) concurrent program behaviour.

## References

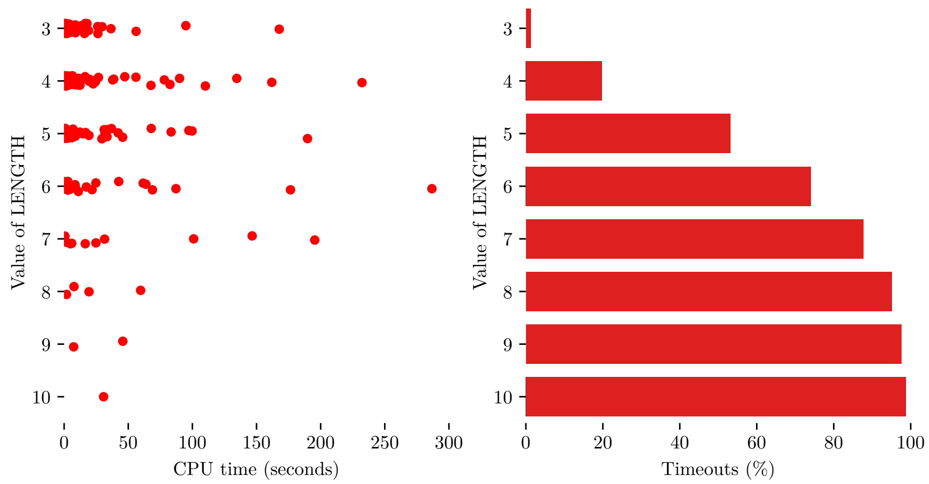

[1] Free Software Foundation. *GNU M4 - GNU macro processor*. Version 1.4.6. May 29, 2021. URL: `https://www.gnu.org/software/m4/manual/index.html`.

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

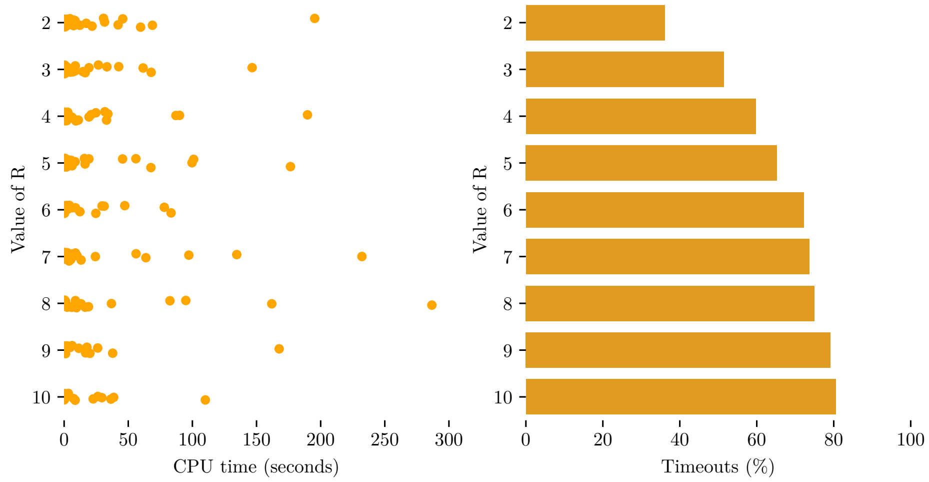
[3] Ole Tange. *GNU Parallel 20230422 ('Grand Jury')*. GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them. Apr. 2023. DOI: `10.5281/zenodo.7855617`. URL: `https://doi.org/10.5281/zenodo.7855617`.




(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.