Assignment 4: Model checking with Spin

Software Analysis

Due date: 2023-05-16 at 23:00

1 The assignment

Your assignment in a nutshell:

1. Write ProMeLa **finite-state models** of a program that reverses the elements of an array `a` in parallel, and of a program that reverses them sequentially.
2. Write **LTL properties** that express correctness and other properties of the programs' behavior.
3. Run **Spin** on the model to verify which properties hold and which don't. For the properties that *don't* verify, explain the counterexample found by Spin and what scenario it represents at a high level.
4. Write a short **report** discussing your work.
   Maximum length of the report: 8 pages (A4 with readable formatting).

The assignment must be done *individually*.
This assignment contributes to **25%** of your overall grade in the course.

1.1 Reversing: sequentially and in parallel

In this assignment, we are modeling a basic piece of functionality: reversing the content of an integer array `a` while copying it into another array `r`.[1] In *sequential* programming, this is done simply with a loop:

[1]We are assuming that the length of `a` is the same as the length of `r`.

```
void reverse(int[] a, int[] r) {
   for (int k = 0; k < a.length; k++)
      r[a.length - k - 1] = a[k];
}
```

On the other hand, if we have N *threads* $T_0, T_1, \ldots, T_{N-1}$ that can work in parallel, we can ideally[2] speed-up the computation by assigning to different threads different portions of arrays `a` and `r`. To distribute the computation evenly, each thread processes $S =$ `a.length`$/N$ contiguous elements of the array. Precisely, for each $0 \leq k < N$, thread T_k processes the elements of `a` from index $k \cdot S$ included to index $k \cdot S + S$ excluded. The last thread T_{N-1} may have to take care of all remaining elements; if `a.length` is not exactly divisible by N, these will be more than S. For example, if `a.length` is 13 and N is 4, threads T_0, T_1, T_2 get $3 = 13/4$ elements each, which leaves T_3 with the trailing 4 elements.

Once all N threads have processed their array portions, `r`'s content should be the same as after calling the sequential method `reverse` shown above on the same initial array.

If you find it helpful, in *iCorsi* (under Assignment 4) you can download an example Java implementation of sequential and parallel reversal methods, which fully implement the informal description of this section.

1.2 ProMeLa model

The first step of this assignment is writing a ProMeLa model that captures the behavior described above. Precisely, there should be a process that reverses an array `a` sequentially and writes the reversed array into an array `r_s`, and another, separate, one that spawn N worker threads to reverse the same array `a` in parallel and writes into a separate array `r_p`. This way, since the sequential and parallel processes write to two different arrays `a_s` and `a_p`, and the parallel subprocesses modify disjoint portions of `a_p`, we don't have to worry about race conditions.

Your ProMeLa model need not replicate the example Java implementation in every detail; the key aspect is the allocation of work to threads, which has to follow the description above. The details of how threads map to ProMeLa processes, and how processes synchronize, can be equivalently modeled in different ways – using message passing and channels, shared global variables, or a combination of both.

It is important that the ProMeLa model allows a level of inter-process concurrency that accurately models the actual Java threaded execution (see the example Java implementation available in *iCorsi*). In particular, since different Java threads work on disjoint sections of the output arrays, the corresponding ProMeLa processes should be allowed to interleave freely with the other processes; they should not use unnecessary "locks" (such as `atomic` blocks in ProMeLa) within their main body or be forced to execute in a fixed order.

The model should be parametric with respect to the number N of threads and the size `LENGTH` of arrays `a`, `a_s`, and `a_p`. To this end, you can use the `#define` preprocessor directive to associate a value to N and `LENGTH` and to easily change it before each recompilation.

[2]Ideally, because the synchronization overhead may actually dominate the overall execution time.

An important aspect to make the model realistic is the initialization of input array `a`'s content. In ProMeLa, an array `int a[LENGTH]` is initialized to all zeros by default. If we analyzed a model where `a` is not modified after this default initialization, we would just verify the case of reversing an array with all zeros, which is a very narrow verification result.

To have a meaningful model, we have to *explicitly* initialize the array `a` to *nondeterministic* integer values. ProMeLa offers the `select` statement to do this: `select(v: 0 .. R)` will assign to `v` any value from `0` to `R` included. In this case Spin will verify all cases: one for each value of `v` in that range. Make sure your ProMeLa model includes an explicit nondeterministic initialization of `a`'s content using `select` *before* the sequential and parallel processes execute. In contrast, you do not need to explicitly initialize the output arrays `a_s` and `a_p`, since they will be entirely overwritten by the sequential and parallel processes.

1.3 LTL properties

Formalize the following properties in LTL:

1. once the sequential and parallel computations have terminated, the content of `a_s` and `a_p` is the same

2. the sequential and parallel computations eventually terminate

In addition, formalize another *two* LTL properties of your choice:

3. one property should be *verified* by the ProMeLa model

4. one property should be *violated* by the ProMeLa model (that is, Spin should find a counterexample)

To fully express property 1. above, you will have to add an element-by-element check to your ProMeLa model, which runs after the sequential and parallel computations have terminated and compares `a_s` to `a_p` element by element. Alternatively, you could express a *partial* correctness property, which only compares some elements of `a_s` and `a_p` (this would be a weaker property, which may be reflected in your grade for the assignment).

1.4 Verification with Spin

Once you have built the ProMeLa model and formalized the four LTL properties, run Spin to verify the model against each property in turn.

The model's parameters `N` (number of threads), `LENGTH` (length of array `a`), and `R` (range of integer values that are stored in `a`) will greatly affect the time it takes to run Spin. Start with small numbers (for example: `N = 2`, `LENGTH = 3`, and `R = 2`) to ensure that everything works as expected. Once it does, you can increment the numbers gradually and see how far you can push them before you run out of memory and/or time (and with consistent verification results).

For the properties that don't verify, analyze the counterexample trace produced by Spin and explain it in terms of program behavior:

- At what point of the computation does the counterexample violate the property?
- Does the property violation depend on the values of parameters `N`, `LENGTH`, and `R`?
- Does the property violation indicate some genuine issues of the modeled program, or is the property just too restrictive?
- When the violated property is too restrictive, can you modify it (relax it) so that it still captures the same aspect of program behavior but becomes verified?

These aspects can be discussed in the report.

2 Tool and documentation

2.1 How to use Spin

You can use Spin in a Docker container using the image `bugcounting/satools:y23`. A simple GUI is available by calling `ispin`. Otherwise, use the following basic sequence of commands to run Spin on ProMeLa model `model.pml` with LTL property `prop`:

```
# build the analyzer from the model
$ spin -a model.pml
# compile the analyzer
$ gcc -Wno-format-overflow -o analyzer pan.c
# run the analyzer, trying to verify property 'prop'
$ ./analyzer -a -N prop
```

When verification fails, the counterexample trace will be stored in `model.pml.trail`, and can be analyzed with:

```
# build the analyzer from the model
$ spin -k model.pml.trail model.pml
```

To make counterexample traces more readable, you may add `printf` statements at various places in the ProMeLa model where it's useful to keep track of the program's evolution. Spin ignores `printf` statements when performing verification, but it will execute them when replaying a single trace from a `.trail` file.

2.2 Documentation about Spin and ProMeLa

More information about Spin is available from the project's website:

http://spinroot.com/

In addition to the examples that we have seen during the Spin tutorial in class (which are included in the Docker image under `examples/spin/`), Spin's basic manual is a good place to become familiar with ProMeLa's syntax (you can skip section *Advanced Usage*):

http://spinroot.com/spin/Man/Manual.html

2.3 Spin's output

Spin's command-line output contains a lot of information and can be a bit overwhelming at first. In this assignment, we are mainly interested in these kinds of *errors* that Spin may report:

assertion violated means that Spin found an execution (trace) of the ProMeLa model that violates an assertion. A violated assertion can be either an explicit statement `assert (exp)` in the code, or an implicit assertion generated by Spin to check an LTL property P (declared in the ProMeLa code using an `ltl` block, or passed in negated form on the command line with option `-f`). In the case of an LTL property, Spin sometimes refers to the violated property as a *never claim*, which is Spin's name for what we called the *monitor* of the *negated property*.

acceptance cycle means that Spin found an execution (trace) of the ProMeLa model that continues indefinitely but never satisfies the property we're trying to check. In this case, the property is usually an LTL formula using the *eventually* or *until* operators. For example, if we're trying to verify `<> p` but there are executions where `p` never occurs, Spin's counterexample trace will show a cycle (loop) where `p` doesn't happen and that can repeat forever.

invalid end state means that Spin found an execution that *deadlocks*, that is where all processes are stuck waiting for one other. In this case, you typically have to revise how processes *synchronize* to ensure they can always make progress.

unreached states are locations of the ProMeLa model that Spin never executed. The presence of some unreached states is not necessarily an error, but if you find out that fundamental portions of your code don't run at all, it probably means that process synchronization is incorrect, and some processes are prevented from running as intended.

For a more detailed overview of properties and errors see these slides by Gerardo Schneider.

2.4 Plagiarism policy

You are allowed to learn from any examples that you find useful; however, you are required to:

1. write down the solution completely on your own; and,
2. if there is a publicly available example that you especially drew inspiration from, credit it in the report (explaining what is similar and how your solution differ).

Failure to do so will be considered plagiarism. (If you have doubts about the application of these rules, ask the instructors *before submitting* your solution.)

2.4.1 ChatGPT & Co.

The plagiarism policy also applies to AI tools such as ChatGPT or CoPilot:

1. You are allowed to use the help of such tools; however, you remain entirely responsible for the solution that you submit.
2. If you use any such tools, you must add a section to the report that mentions which tools you used and for what tasks, how you checked the correctness and completeness of their suggestions, and what modifications (if any) you introduced on top of the tool's output.
3. If you use a text-based tool such as ChatGPT, also show a couple of examples of prompts that you provided, with a summary of the tool's response.

Failure to abide by these rules, including failing to disclose using AI tools, will be considered plagiarism.

3 What to write in the report

Topics that can be discussed in the report include:

- A presentation of your ProMeLa model, with a discussion of how it relates to the "real" implementation.
- A discussion of the two properties you chose, and the formalization in LTL of all four LTL properties.
- Did you have to tweak the model to make it work as expected?
- How much could you increase the parameters `N`, `LENGTH`, and `R` without Spin blowing up?
- Describe the counterexample Spin found for the violated LTL property 4. Is this counterexample feasible in the "real" implementation?
- Were there any unexpected aspects of the program behavior that you discovered thanks to Spin?

4 How and what to turn in

Turn in:

1. The following artifacts in a project named `Assignment4` in your assigned GitLab project for Software Analysis.[3]

[3]The same project you used for the previous assignments; see details in Assignment 1's description.

   a) Your **ProMeLa model**, including the four LTL properties described in Section 1.3.

   b) A shell **script** file that *runs Spin* on the ProMeLa model and checks the four properties.

   The script can assume that the executable `spin` and a C compiler (such as `gcc`) are reachable within the path where the script is executed (as in the environment provided by the Docker image `bugcounting/satools:y23`). Make sure the script works without problems: if it does not run effortlessly, your submission may not be accepted or lose points.

2. The **report** in PDF format as a single file using *iCorsi* under *Assignment 4*.