Assignment 3:
Model checking with Spin

Software Analysis

Due date: 2025-05-30 at 23:00

1 The assignment

Your assignment in a nutshell:

1. Write a ProMeLa **finite-state model** of a program where multiple threads mark different positions in an array.
2. Write **LTL properties** that express correctness and other properties of the program's 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 **22%** of your overall grade in the course.

1.1 Workers and Tasks

The functionality in this assignment is a simplified synchronization model of multiple worker threads, each working on a task with a specific identifier. All threads work on an instance of class `Todos`, which has the following interface:

```
interface Todos
{
  // is ‘taskId‘ the next task to be done?
  public synchronized boolean isNext(int taskId);

  // is ‘taskId‘ already marked as done?
  public synchronized boolean isDone(int taskId);

  // if ‘isNext(taskId)‘, mark ‘taskId‘ as done, move to next task,
  // and return true; otherwise, return false
  public synchronized boolean doTask(int taskId);
}
```

Notice that all these methods are **`synchronized`**, which means that they are executed atomically, without interleaving between different threads. They are also non-blocking, in that they will always terminate in a finite amount of time.

Then, each worker thread will simply busy-wait on `doTask` until it returns **`true`**, and then terminate:

```
// ‘taskId‘ is the identifier of the worker’s assigned task
while (!todos.doTask(taskId)) /* skip */ ;
```

We’ll consider two implementations of `Todos`:

- In `OrderedTodos`, after a task with identifier `taskId` is done, the next task will be the task with identifier `(taskId + 1) % N_TASKS`, where `N_TASKS` is the total number of tasks.
- In `RandomTodos`, after a task with identifier `taskId` is done, the next task to go will be picked randomly among those that have not completed their task yet.

You can assume the following simplifying assumptions:

- The number of threads is always the same as the number of tasks.
- Tasks are numbered sequentially from `0` (included) to `N_TASKS` (excluded).

If you find it helpful, in *iCorsi* (under Assignment 3) you can download an example Java implementation of this functionality.

1.2 ProMeLa model

The first step of this assignment is writing a ProMeLa model that captures the behavior described above. Here are a few guidelines on how to build your model.

- Use a global array `done` of Booleans to record completion of each task.

- Similarly, use a global `int` variable `next_task` that stores the identifier of the next task that will be allowed to complete (variable `next_task` should satisfy the invariant `Todos.isNext(next_task) == true`).

- Define two variants of a `proctype worker(int task_id)` that correspond to the code execute by each worker. Thus, each `worker` process should:
  1. wait its turn;
  2. mark its task as done;
  3. update the identifier of the next task to execute,[1] according to the specific worker variant: one should follow the numeric order, whereas the other should assign the next task randomly.

Your ProMeLa model need not replicate the example Java implementation in every detail; the key aspect is the parallelization of workers, and the order in which the tasks are allowed to complete. 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, waiting conditions, ...

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, the synchronization should occur through global variable `next_task`, and not with an explicit locking mechanism.

The model should be parametric with respect to the number `N_TASKS` of tasks, and it should allow the user to select either ordered workers, or random workers. To this end, you can use the `#define` preprocessor directive to associate a value to `N_TASKS` to easily change it before each recompilation. You can also use `#ifdef` to selectively include one or the other variant of `proctype worker`:

```
#define N_TASKS 3
// Comment out the following #define to switch to random todos
#define ORDERED

/* Global variables */

#ifdef ORDERED
proctype worker(int task_id)
{ /* Definition of worker with ordered assignment of next task */ }
#else
proctype worker(int task_id)
{ /* Definition of worker with random assignment of next task */ }
#endif
```

[1]Note that, if all tasks have executed, the next task to execute is irrelevant, but you must ensure that all workers can terminate.

A somewhat delicate aspect is how to properly model the random assignment of the next task for the "random" worker. Naturally, ProMeLa offers nondeterministic assignment: `select(v: 0 .. R)` will assign to `v` any value from `0` to `R` included. The tricky bit is that you should check that the program does not go into an infinite loop in which it keeps on randomly picking an identifier corresponding to a task that has already completed.

Also remember to properly initialize all global variables (in particular, `done` should be initialized to all zeros, which you can do with just `done = 0` in ProMeLa).

1.3 LTL properties

Formalize the following properties in LTL:

1. eventually, all tasks complete
2. when all tasks have completed, `done[0]` is marked as done[2]

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)

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_TASKS` (number of tasks) and whether you are using ordered or random task order will affect the time it takes to run Spin. Start with small numbers (for example: `N_TASKS` between 1 and 3) 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). In the report, explain which property verifies in which amount of time, for each combination of model parameters.

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_TASKS` and whether ordered or random task order is followed?
- 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 so that it still captures the same aspect of program behavior but becomes verified?

These aspects can be discussed in the report.

[2]Obviously, we will always have at least one tasks; hence `done` cannot be empty.

2 Tool and documentation

2.1 How to use Spin

You can use Spin in a Docker container using the image `bugcounting/satools:y24`.[3] 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 re-playing 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:

[3] There is also another Docker image `bugcounting/satools:y24`that is built for the `linux/arm64` architecture, which runs without emulation on the new Macs.

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 that Spin can check and the corresponding errors see these slides by Bernhard Beckert and Matthias Ullbrich.

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 do changing the parameters `N_TASKS` and whether you're using ordered or random worker order affect Spin's running time?
- Describe the counterexample Spin found for the violated LTL property. 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 `Assignment3` in your assigned GitLab group for Software Analysis.[4]
   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.

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

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:y24`). 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 3*.