Report on Assignment 3: Model Checking with Spin

Airaghi Luca
Student ID: 24-995-813

May 2025

1 Introduction

This assignment focuses on modeling and verifying a concurrent program where multiple worker threads perform tasks on shared data. The goal is to use the Spin model checker and its specification language ProMeLa to capture the synchronization and execution order of these tasks, ensuring correctness properties through formal verification.

Specifically, the assignment requires the development of a ProMeLa model that represents a set of threads working on distinct tasks, with two variants controlling task order: an ordered approach where tasks proceed sequentially, and a random approach where the next task is chosen non-deterministically among unfinished tasks.

Key correctness properties are formalized in Linear Temporal Logic (LTL), and Spin is used to verify whether these properties hold. For properties that fail, counterexamples are analyzed to understand possible errors or overly restrictive specifications. This process helps in identifying subtle concurrency issues and deepening the understanding of parallel program behavior.

2 ProMeLa Model

2.1 Model Structure

The ProMeLa model simulates a system of multiple worker processes, each assigned a unique task identifier. The workers operate concurrently, attempting to complete their respective tasks. The main structure involves a global boolean array `done` which records the completion status of each task, and a shared integer variable `next_task` indicating which task is allowed to execute next.

Each worker process loops until it successfully completes its assigned task by checking if it is their turn to execute (i.e., whether their `task_id` matches `next_task`). Once allowed, the worker marks its task as done and updates the `next_task` according to the variant of the model. This design captures the synchronization and mutual exclusion of task execution without explicit locking mechanisms, closely modeling the behavior of the original Java implementation.

2.2 Ordered and Random Task Runners

The core of the model is the `task_runner` process, which represents a concurrent worker assigned to execute a specific task. Each task runner is instantiated with a unique task

identifier (`task_id`) and is responsible for completing that task according to a defined scheduling strategy. Two distinct execution variants are implemented, controlled via preprocessor directives: `ORDERED` for deterministic round-robin scheduling, and its absence for nondeterministic execution.

- **Ordered task runners:** When the `ORDERED` flag is defined, the system enforces a strict round-robin order among task runners. Each `task_runner` repeatedly checks whether it is its turn to run, based on a shared global variable `next_task`. Only the runner whose `task_id` matches `next_task` proceeds. Upon completion, it marks its task as done and updates `next_task` to point to the next task in sequence using modulo arithmetic:

  ```
  next_task = (task_id + 1) % N_TASKS;
  ```

  This guarantees that tasks are completed in a predictable, cyclic order, avoiding race conditions and ensuring fairness by construction.

- **Randomized (unordered) task runners:** Without the `ORDERED` directive, task runners still coordinate through the shared `next_task` variable, but the selection of the next task is nondeterministic among all unfinished tasks rather than strictly sequential.

  Each runner proceeds only if its task is not done and its `task_id` matches `next_task`. After marking its task as done, it scans all tasks to check if any remain unfinished. If none remain, the runner terminates.

  If unfinished tasks remain, the runner nondeterministically selects the next task to run from the entire set of unfinished tasks using a nondeterministic choice construct. This means that in the verification model, all possible unfinished tasks are considered as potential next steps, enabling exploration of all possible execution orders.

  The loop executed by each runner thus performs the following steps:

  1. Wait until `task_id == next_task` and `done[task_id] == 0`.
  2. Mark the current task as done by setting `done[task_id] = 1`.
  3. Check if any tasks remain unfinished by scanning the `done[]` array.
  4. If tasks remain, nondeterministically select one unfinished task as the new `next_task`.
  5. Exit when all tasks are completed.

  This design models an unordered scheduler that can pick any unfinished task next, ensuring that the model checker explores all possible scheduling orders and uncovers potential concurrency issues that could be missed with deterministic ordering.

All these operations are wrapped in atomic blocks to prevent interleaving that could corrupt shared state. This structure ensures safe and synchronized task progression under both deterministic and nondeterministic scheduling strategies.

2.3 Key Variables and Synchronization

The main global variables are:

- `done`: A boolean array indexed by task ID indicating whether each task has been completed.
- `next_task`: An integer indicating the task ID of the task currently allowed to proceed.

Synchronization occurs implicitly through the `next_task` variable: a worker can only proceed if its task ID matches `next_task`. The model uses atomic blocks to ensure that marking a task as done and updating `next_task` happen atomically, preventing interleaving that could violate correctness.

This approach avoids explicit locks or message passing, relying on shared state and atomic updates to model thread coordination accurately within ProMeLa's capabilities.

3 LTL Properties

3.1 Formalized Properties

The following four Linear Temporal Logic (LTL) properties were formalized to verify the correctness and behavioral aspects of the ProMeLa model:

1. **Eventually all tasks complete:**

$$\texttt{p1}: \quad \Diamond(done[0] \land done[1] \land done[2])$$

   This property ensures that, regardless of execution order, all tasks will eventually be marked as completed.

2. **When all tasks complete, `done[0]` is done:**

$$\texttt{p2}: \quad (done[0] \land done[1] \land done[2]) \rightarrow done[0]$$

   This property checks a trivial but necessary condition that if all tasks are done, then specifically the first task (index 0) must be done as well.

3. **Once a task is done, it remains done:**

$$\texttt{p3}: \quad \Box(done[0] \rightarrow \Box done[0]) \land \Box(done[1] \rightarrow \Box done[1]) \land \Box(done[2] \rightarrow \Box done[2])$$

   This safety property guarantees that task completion is irreversible: once a task is marked done, it cannot revert to undone.

4. **Tasks 0 and 1 are never both done (expected to fail):**

$$\texttt{p4}: \quad \Box\neg(done[0] \land done[1])$$

   This deliberately incorrect property states that tasks 0 and 1 cannot both be done simultaneously. It is expected to fail, serving as a test case to verify that Spin correctly identifies violations and produces counterexamples.

4 Verification Results with Spin

4.1 Verification Setup

The Promela model was verified using the SPIN model checker. Verification was conducted under two configurations described before: **ordered** and **unordered**.

For both configurations, the number of tasks (`N_TASKS`) was varied: I started verifying my program for 3, 4, 5 and 10 tasks. In each case, the SPIN verifier was run on the model using the following procedure:

1. The desired value of `N_TASKS` was set using a preprocessor directive in the Promela file.

2. The verifier was generated and compiled using the following script:

```
#!/bin/sh
# generate the verifier
spin -a model.pml

# compile the verifier
gcc -Wno-format-overflow -o analyzer pan.c

# run each LTL property by name
./analyzer -a -N p1
./analyzer -a -N p2
./analyzer -a -N p3
./analyzer -a -N p4
```

3. Linear Temporal Logic (LTL) properties `p1` through `p4` were defined in the model file. While SPIN only verifies one LTL formula at a time, the script executed the verifier sequentially for each named property, effectively automating the verification of all properties in a single testing session.

4.2 Results Summary

I evaluated both ordered and unordered executions of the task system with varying values of `N_TASKS`. Below is a summary of the verification results.

Property	N=3	N=4	N=5	N=10
		Ordered Execution		
p1	Pass	Pass	Pass	Pass
p2	Pass	Pass	Pass	Pass
p3	Pass	Pass	Pass	Pass
p4	Fail	Fail	Fail	Fail
		Unordered Execution		
p1	Pass	Pass	Pass	Pass
p2	Pass	Pass	Pass	Pass
p3	Pass	Pass	Pass	Pass
p4	Fail	Fail	Fail	Fail

Table 1: Verification results across increasing N_TASKS for ordered and unordered execution

Discussion: In both ordered and unordered executions, properties p1 through p3 consistently hold. This confirms that all tasks eventually complete (p1), task 0 is completed whenever all tasks are done (p2), and once a task is completed, it remains completed (p3).

However, property p4 fails in all configurations. This property asserts that tasks 0 and 1 are never both completed simultaneously, effectively requiring mutual exclusion between their completions. Since the system's design expects all tasks to eventually complete, this mutual exclusion is inherently violated, causing p4 to fail by design. Thus, p4 serves as a sanity check to confirm that tasks are indeed completing concurrently rather than exclusively.

4.3 Counterexample Analysis

Spin produces a counterexample trace for property p4, which states that *"at no point should both done[0] and done[1] be true simultaneously"*, i.e., `ltl p4: [] (!(done[0] && done[1]))`. This safety property was expected to hold under the assumption that tasks are executed in isolation or not concurrently marked as done. However, the counterexample reveals a valid execution in which both done[0] and done[1] become true sequentially, violating the property. Below is an excerpt from the generated trail:

```
done[0] = 1
done[1] = 1
...
assert(!(!(!((done[0] && done[1])))))
```

This behavior results from the ordered execution logic implemented in the model, where tasks complete strictly in sequence:

```
next_task = (task_id + 1) % N_TASKS;
```

Since task 0 sets `done[0] = 1` and then immediately allows task 1 to proceed and set `done[1] = 1`, it is expected and correct that both flags are true at some point. Therefore,

the counterexample does not indicate a fault in the model's logic but rather shows that the property `p4` is too restrictive in this context. This illustrates the importance of aligning LTL properties with the intended semantics of the modeled system.

5 Model Tweaks and Fairness Handling

In earlier versions of the model using unordered execution, fairness and progress were concerns due to the potential for some tasks to be indefinitely postponed, leading to starvation. Unlike the ordered execution, which cycles deterministically through tasks, the unordered variant relies on nondeterministic selection of the next unfinished task.

5.1 Fairness via Nondeterministic Task Selection

To mitigate starvation without additional global counters, the model employs a nondeterministic scan over all tasks when selecting the next task to run. Specifically, after a task completes, the system chooses the next unfinished task by checking each task in order and selecting one nondeterministically from those not yet done:

- This ensures that all unfinished tasks remain eligible for selection.
- The nondeterminism in the selection reflects the interleaving semantics of the model checker rather than purely random choice.
- By explicitly excluding completed tasks, the model guarantees progress by eventually scheduling every pending task.

5.2 Consequences of the Chosen Design

Without explicit progress variables, the fairness guarantee depends on the model checker's nondeterministic scheduler exploring all possible interleavings. While this approach avoids complexity, it requires thorough verification to ensure no unfair infinite postponements occur.

This design choice trades off explicit fairness enforcement for simplicity and leverages the model checker's exploration capabilities to validate progress and completion properties such as `p1`.

Memory Usage Summary

Notes:

- DFS stack memory usage remained consistent across runs, approximately 0.534 MB.
- Total Memory remain constant too, approximately 128.730 MB
- Overall memory usage is low, indicating efficient verification performance for the tested model sizes.

Run Type	Tasks	States Stored	States Matched	Transitions	DFS Stack Mem (MB)	Total Mem (MB)
ORDERED	3	25	20	70	0.534	128.730
ORDERED	4	88	76	164	0.534	128.730
ORDERED	5	228	215	443	0.534	128.730
ORDERED	10	681	679	1360	0.534	128.730
UNORDERED	3	37	30	104	0.534	128.730
UNORDERED	4	113	98	224	0.534	128.730
UNORDERED	5	257	242	515	0.534	128.730
UNORDERED	10	613	598	1254	0.534	128.730

Table 2: Summary of memory and state statistics for Spin verification runs across different task counts and run types.

6 Conclusion

This assignment offered valuable hands-on experience in modeling and verifying concurrent systems using Spin and Promela. By developing a model of multiple worker threads interacting with shared data, we addressed fundamental concurrency challenges including task synchronization, execution ordering, and guarantees of progress.

The comparison between ordered and unordered execution modes underscored the critical impact of scheduling on system liveness. In the ordered mode, key properties, such as eventual task completion, state stability, and correctness, were successfully verified. Conversely, the unordered mode revealed potential starvation scenarios, resulting in violations of liveness properties like `p1`. This contrast highlights the importance of fairness assumptions in both the design and verification of concurrent systems.

Additionally, the analysis of counterexamples, particularly the deliberate failure of property `p4`, validated the robustness of the model checker and the synchronization logic, demonstrating how incorrect properties can aid in debugging and refinement.

In summary, this assignment enhanced our understanding of model checking in concurrent environments and showcased the effectiveness of formal verification for uncovering subtle bugs and specification mismatches. It reinforces the essential role of precise modeling and careful property formulation in the analysis and assurance of parallel system correctness.