Software Specification Report 2

Sjoerd van de Goor (1557815), Panagiotis Panagiotou (1815423)

March 2022

1 Introduction

For this second report of this course, we are tasked to create a formal specification model in the SPIN/Promela language, which can verify correctness of our system. We are provided a template lock and ship model already implemented in the language, for which both a single lock implementation, and a multi-lock implementation were required to be implemented. Both used Promela files are sent with this report, named *lock_env_single.pml* and *lock_env_multi.pml* for the single and multi-lock system respectively.

2 Design Choices & Motivation - Single Lock

For the single lock we have seven requirements. The first five of which were definable by constantly running assertion checks. These checks were implemented using the monitor as follows:

```
proctype monitor() {

        // Asserts the lower pair of doors and the
        // higher pair of doors are never simultaneously open.
        assert(doors_status.higher == closed
        || doors_status.lower  == closed)

        // Asserts that when the lower pair of doors is open,
        // the higher slide is closed.
        assert(doors_status.lower  == closed
        || slide_status.higher == closed)

        // Asserts that When the higher pair of doors is open,
        // the lower slide is closed.
        assert(doors_status.higher == closed
        || slide_status.lower  == closed)

        // Asserts the lower pair of doors is only open
        // when the water level in the lock is low.
        assert(doors_status.lower  == closed
        || lock_water_level    == low_level)

        // Asserts the higher pair of doors is only open
        // when the water level in the lock is high.
        assert(doors_status.higher == closed
        || lock_water_level    == high_level)

        // Provided assertion
        assert(0 <= ship_pos[0] && ship_pos[0] <= N);
}
```

We decided to use assertions because these requirements have to be met at any given time, and have no temporal aspect to them. The d1 and d2 requirements, however, are more challenging since they cannot be checked in just one moment. Instead of using assertions, we chose to store for each ship that the ship requested entry into the lock, and ensure that the checker can only terminate successfully if all these requests were handled. To do this, we use a LTL formula:

```
ltl p1 { []<> (ship_did_request_but_not_fulfilled[0] == false) }
```

ship_did_request_but_not_fulfilled is an array that stores for every ship whether or not they have made a request but not entered the lock yet after.

3 Controller implementation

First, we checked for events, like *request_low* and *request_high*:

```
:: request_low?true ->
```

Then we check if the doors on the other side are open and closed them if we need to.

```
if
:: doors_status.higher == open ->
        change_doors_pos!high; doors_pos_changed?true;
:: doors_status.higher == closed -> skip;
fi;
```

After, we check if the doors on the side of the request are open and if they are not open we check to see if the water is at the right height. If the water is not at the right height, then we open the slides and we wait for the water height to change. Lastly, we close the slides and open the doors.

```
if
:: doors_status.lower == closed ->
        if
        :: lock_water_level == high_level ->
                if
                :: slide_status.higher == open
                        change_slide_pos!high; slide_pos_changed?true;
                :: slide_status.higher == closed -> skip;
                fi;
                change_slide_pos!low; slide_pos_changed?true;
        :: lock_water_level == low_level -> skip;
        fi;
        change_doors_pos!low; doors_pos_changed?true;
:: doors_status.lower == open -> skip;
fi;
```

The last step is informing the boat that the lock doors have opened on that side.

```
observed_low[0]?true;
```

The same is done in the other direction, with all high and low doors, slides and water level swapped.

4 Design Choices & Motivation - Multi-lock

The second half of this assignment is about multiple locks with multiple ships. The challenge is to modify the initialization and controller such that multiple locks and ships are supported, and creating formal specifications to verify functionality.

4.1 Verify run-time analysis and scalability modifications

After implementing multiple locks and ships (3 locks and 2 ships) and being able to run error-free, the run-time increased to 25.5 seconds, at 176853 states per second. This time-frame is certainly too long for repeat-testing, and hence it was decided to remove the assertions ran every iteration in the Monitor. After this change, run-time went to 0.69 seconds, which is far more manageable, and it was decided to not spend further time looking into scalability optimizations.

4.2 Ship and lock numbers leading to deadlocks

After some trial-and-error tests, it was found that a combination of two locks and three ships causes a deadlock. This deadlock can be explained by the fact that ships do not check if they can leave a lock, before entering one. Because of the absence of such a check, it is possible that a ship enters a lock, only to find two ships waiting on the other side of it, preventing it from leaving the lock. Since the ship in the lock cannot go back either, after the top doors open, the three ships reach a stalemate and cannot move ahead or return, causing the deadlock. To expedite the occurrence of the deadlock, initialize the first ship in position 0, going upstream and two other ships to start at position 1, as shown here:

```
ship_status[0] = go_up;
ship_status[1] = go_down;
ship_status[2] = go_down;
ship_pos[0] = 0;
ship_pos[1] = 1;
ship_pos[2] = 1;
run ship(0);
run ship(1);
run ship(2);
```

Attached is a file called *lock_env_multi_deadlock.pml.trail*, which contains the counterexample for this initial configuration.

4.3 Requirement formalisation

For the requirements d and e, it was decided LTL formulas would be most suitable considering the existence of the <> (eventually) marking, which is required for both requirements. Checking this with assertions is possible, but requires significantly more effort.

```
ltl e1l1 = [] (request_low[0]?[true] ->
    (<> doors_status[0].lower == open))
ltl e1l2 = [] (request_low[1]?[true] ->
    (<> doors_status[1].lower == open))
ltl e1l3 = [] (request_low[2]?[true] ->
    (<> doors_status[2].lower == open))
ltl e2l1 = [] (request_high[0]?[true] ->
    (<> doors_status[0].higher == open))
ltl e2l2 = [] (request_high[1]?[true] ->
    (<> doors_status[1].higher == open))
ltl e2l3 = [] (request_high[2]?[true] ->
    (<> doors_status[2].higher == open))

ltl f1 = []<> (request_high[N-1]?[true])
ltl f2 = []<> (request_low[0]?[true])
```

As to achieve functionality with all locks, and with the absence of looping in LTL, simply repeat the LTLs for each lock.

4.4 Code iterations through formal property failure

During development, it was observed (early on) that many states were not reached, despite no errors occurring. Because of this, a second iteration of the code was created, which resulted in (almost all) states being reached. Because of this, and perhaps some luck, the LTL formulas above passed on their first run.

~~Now to answer why this is to be expected, the answer is quite simple: my code is perfect.~~

As for the analysis of why this is be the case, take a look at the behaviour of the ships traversing the lock system. In the *ship* proctype, observe there are exclusively options to take a new step. Every possible state the ship may have, at any given position, will lead to a move that traverses the path between the lower side of the first, and higher side of the last lock, without the possibility to stray from this path. The first four clauses in the outermost *do* block allow the ship to proceed into or out of a lock, in the direction it is travelling, and the last two turn the ship when reaching the end of the path. Since each ship can only turn when at the end of its path, and must proceed in every other case, the ships will traverse the path up and down, ensuring $f1$ and $f2$ are satisfied (since passing through a lock requires making a request for passage).

Then, observe the *lock* proctype. In the outermost *do* block, upper and lower requests are received. Observe the second if block in both event handler. In the event the lock door is not already open (if the door is already open, the LTL requirement is satisfied), there is no possibility of successfully executing without opening the door with *change_door_pos*, which means the LTL $e1$ and $e2$ requirements are satisfied.