Software Specification (2IX20) 2021–2022 Assignment 2: Formal specification in PROMELA

Introduction

This second assignment is about creating a formal model of the control software of a lock system, formalising some requirements of this system, and proving that the model satisfies these requirements. Two models will be constructed expressed in the PROMELA language. The correctness criteria will be specified as formulas in linear temporal logic (LTL) and/or assertions. The SPIN model checker will be used to formally verify the correctness of the PROMELA models.

This assignment is divided into three tasks:

1. Studying the SPIN model checker and the given template for a lock system model;
2. Modelling and verifying the controller for a system with one lock;
3. Extending and verifying this system for multiple locks.

Note 1. Tasks 2 and 3 may require several iterations. The properties that need to be verified may lead you to modify your model. Make sure that all properties are expressed in the corresponding model. In other words, the properties mentioned in Section 2.2 must be expressed in the model of the single lock system, and the properties mentioned in Section 3.2 must be expressed in the model of the multiple locks system.

Note 2. This assignment contains the requirements for a particular model of the control software, as opposed to requiring you to use the informal specification you created when you worked on assignment 1. This is to ensure that your success for this assignment does not depend on your result for the previous assignment.

1 A lock system model for SPIN

1.1 The SPIN model checker

The SPIN model checker accepts as input formal models written in the PROMELA language (SPIN is an acronym for 'Simple PROMELA INterpreter').

The first task of this assignment is to install SPIN and start using SPIN via the JSPIN graphical user interface (GUI). The version of SPIN we use is the latest, 6.5.1. It is recommended to download the Virtual Machine image available

on Canvas, install VirtualBox, and set up the Virtual Machine containing a ready for use installation of Spin and jSpin.

Documentation and tutorials can be found on the Spin website (http://www.spinroot.com) here: http://spinroot.com/spin/Man/. The reader of the course provides explanation about the Promela language and the Spin model checker, see Chapter 11. In addition, it is highly recommended to use the book *Principles of the SPIN Model Checker* by Mordechai Ben-Ari, which is available as an e-book via the TU/e library. It provides an explanation of Spin from a user-perspective, and refers to the jSpin GUI, which has been developed by the same author.

1.2 Modelling a lock system

The second and third tasks of this assignment are focussed on creating and verifying two Promela models of the control software of a lock system.

You will perform the modelling in two steps:

1. The first step is to model a main control process that correctly controls a single lock for a configurable number of ships. You need to verify, using Spin, that this control process correctly manages the system. See Section 2 for more details.

2. The second step is to generalise the model to account for multiple locks. This generalisation mainly requires an adaptation of the main control process, but the lock and ship process types are also mildly affected. The lock process definition specifies how a lock reacts to requests and the ship process definition specifies how ships act and react. See Section 3 for more details.

In order to build full, verifiable models of your controller, a model of the environment of the control unit must be taken into account. To help you focus your modelling effort on the control unit, Section 1.3 presents a model of the environment. The file containing the Promela model of the environment, `lock_env.pml`, can be found in the 'Documents' folder when running the Virtual Machine. Furthermore, the model can be downloaded from Canvas, along with the assignment description.

1.3 Model of the environment

To help you focus the modelling on the main control process of the lock system, yet obtain a verifiable model of the system as a whole, we provide a model of the environment of the system. The environment is composed of the following components:

- Lock processes.
- Ship processes.

In addition, an initialisation process is given, to set up the initial state of the system. The model has been set up for one lock and a configurable number of ships, indicated by the constant M, which has been preset to 1.

```
// Lock process type. It reacts to requests to open its doors and slides.
proctype lock(byte lockid) {
        do
        :: change_doors_pos?low ->
                if
                :: doors_status.lower == closed -> doors_status.lower = open;
                        lock_water_level = low_level;
                :: doors_status.lower == open -> doors_status.lower = closed;
                fi;
                doors_pos_changed!true;
        :: change_doors_pos?high ->
                if
                :: doors_status.higher == closed -> doors_status.higher = open;
                        if
                        :: doors_status.lower == closed && slide_status.lower == closed ->
                                lock_water_level = high_level;
                        :: doors_status.lower == open || slide_status.lower == open -> skip;
                        fi;
                :: doors_status.higher == open -> doors_status.higher = closed;
                fi;
                doors_pos_changed!true;
        :: change_slide_pos?low ->
                if
                :: slide_status.lower == closed -> slide_status.lower = open;
                        lock_water_level = low_level;
                :: slide_status.lower == open -> slide_status.lower = closed;
                fi;
                slide_pos_changed!true;
        :: change_slide_pos?high ->
                if
                :: slide_status.higher == closed -> slide_status.higher = open;
                        if
                        :: doors_status.lower == closed && slide_status.lower == closed ->
                                lock_water_level = high_level;
                        :: doors_status.lower == open || slide_status.lower == open -> skip;
                        fi;
                :: slide_status.higher == open -> slide_status.higher = closed;
                fi;
                slide_pos_changed!true;
        od;
}
```

Figure 1: PROMELA model of a lock.

1.3.1 The lock process type

The behaviour of a lock is specified by the lock process type. This process type is modelled in PROMELA as given in Figure 1. A lock receives messages via the synchronous channels `change_door_pos` and `change_slide_pos`.

When the message `low` is received via `change_door_pos`, the lower pair of doors of the lock is requested to change state. In this case, if the doors are closed, they will be opened, which may cause the water level inside the lock to drop, if it was not already at the appropriate level. On the other hand, if the doors are open, they will be closed. Via the synchronous channel `door_pos_changed`, it is confirmed that the doors have changed state.

For a lock, the status of both pairs of doors (low and high) is stored in a global variable `doors_status` of type `doorpairs_t`. It has a field `doors_status.lower` and a field `doors_status.higher` to store the status of the lower and higher pair of doors, respectively. An `mtype` called `mtype:pos` is defined in the system specification to refer to the two possible situations `closed` and `open`. The water level inside a lock is stored in the global variable `lock_water_level`, and can have the value `low_level` or `high_level`.

When the message `high` is received via the channel `change_door_pos`, the

higher pair of doors is requested to change state. The procedure to do this is very similar to the procedure for the lower pair of doors, but the effect may be different: if the doors are opened, the water level may increase, but only if both the lower doors and the lower slide are not open.

Via the channel `change_slide_pos`, a slide can be requested to change state. If the message `low` is received via this channel, the lower slide is requested to change state. If it is closed, it will be opened, after which the water level in the lock becomes low. If the slide is open, it will be closed. Confirmation that the slide has changed state is sent via the synchronous channel `slide_pos_changed`. The case that the higher slide has to change state is very similar, but opening it has the same effect as opening the higher pair of doors: the water level may rise, but only if both the lower pair of doors and the lower slide are closed.

Note 3. To keep the system model relatively simple, we do not consider the water level changing outside of the lock. For the case with one lock, it can still be argued that as the higher side has a virtually infinite supply of water and the lower side can drain a virtually infinite amount of water, opening the doors on either side of the lock has practically no effect on the water level outside of the lock. For multiple locks, we also consider this to not have any effect on the water level between locks. This may be seen as unrealistic, but it simplifies the model while keeping the core functionality of the system. We therefore assume the following: to lower the water level inside a lock, its lower slide should be opened, and to raise the water level inside a lock, its higher slide should be opened.

Note 4. When an instance of the lock process is created (see Section 1.3.3), this instance is given an ID, which is referred to as `lockid`. Note that in Figure 1, this ID is not yet used, but it will be needed when the system is extended to multiple locks, see Section 3.

1.3.2 The ship process type

In the formal model of the lock system, the lock keeper is not present. Instead, requests to open doors actively come from ships. This is different from the informal lock system description of assignment 1, and it requires the system to directly react to the behaviour of ships.

Figures 2 and 3 present the definition of the ship process type, in two parts. A ship has an ID called `shipid`, which it uses to access its status, its position, and information about that position. All this information is stored in global variables.

We refer to the position of a ship using the scheme presented in Figure 4. The position outside the lower pair of doors of lock i is identified as position i, while the position outside the higher pair of doors of lock i is identified as position $i+1$. In other words, a flight consisting of N locks has $N+1$ positions for ships.

Initially, a ship is given a position and a direction to travel in. This information is stored in the arrays `ship_pos` and `ship_status`, respectively, at position `shipid`. The initialisation is done in the init process (see Section 1.3.3).

A ship repeatedly checks its current situation, and acts on it, inside the big `do`-statement of Figures 2 and 3. Figure 2 describes what it should do when its direction is either `go_down` or `go_down_in_lock`. When its direction

```
// Ship process type. Based on its direction and position, it makes requests to open doors,
// and moves when possible.
proctype ship(byte shipid) {
        do
        :: ship_status[shipid] == go_down && ship_pos[shipid] != 0 ->
                do
                :: doors_status.higher == closed ->
                        request_high!true;
                        atomic { doors_status.higher == open ->
                                if
                                :: !lock_is_occupied ->
                                                ship_status[shipid] = go_down_in_lock;
                                                lock_is_occupied = true;
                                                nr_of_ships_at_pos[ship_pos[shipid]]--;
                                                observed_high[0]!true;
                                                break;
                                :: lock_is_occupied ->
                                                observed_high[0]!true;
                                fi; }
                :: atomic { doors_status.higher == open &&
                        !lock_is_occupied ->
                                ship_status[shipid] = go_down_in_lock;
                                lock_is_occupied = true;
                                nr_of_ships_at_pos[ship_pos[shipid]]--;
                                break; }
                od;
        :: ship_status[shipid] == go_down_in_lock ->
                do
                :: doors_status.lower == closed ->
                        request_low!true;
                        atomic { doors_status.lower == open ->
                                if
                                :: (nr_of_ships_at_pos[ship_pos[shipid]-1] < MAX
                                        || ship_pos[shipid]-1 == 0) ->
                                                ship_status[shipid] = go_down;
                                                lock_is_occupied = false;
                                                ship_pos[shipid]--;
                                                nr_of_ships_at_pos[ship_pos[shipid]]++;
                                                observed_low[0]!true;
                                                break;
                                :: (nr_of_ships_at_pos[ship_pos[shipid]-1] == MAX
                                        && ship_pos[shipid]-1 != 0) ->
                                                observed_low[0]!true;
                                fi; }
                :: atomic { doors_status.lower == open &&
                        (nr_of_ships_at_pos[ship_pos[shipid]-1] < MAX
                        || ship_pos[shipid]-1 == 0) ->
                                ship_status[shipid] = go_down;
                                lock_is_occupied = false;
                                ship_pos[shipid]--;
                                nr_of_ships_at_pos[ship_pos[shipid]]++;
                                break; }
                od;
                ...
```

Figure 2: PROMELA model of a ship, part 1.

is `go_down`, and it has not yet reached the final position for that direction, i.e., 0, it enters a nested `do`-statement, in which it checks the status of the higher pair of doors of the lock (remember that we consider a system here with a single lock). If those doors are closed, it makes a request via the asynchronous channel `request_high`. After that, it starts waiting for the higher pair of doors to open. Note that the next statement is placed inside an `atomic` statement, to make the entire procedure a single, atomic step of the process. No other process is allowed to execute behaviour when this sequence of statements is executed.

When the higher pair of doors is open, the ship process acts on this, based on whether or not the lock is occupied by another ship, which is indicated by the

```
        :: ship_status[shipid] == go_up && ship_pos[shipid] != N ->
                do
                :: doors_status.lower == closed ->
                        request_low!true;
                        atomic { doors_status.lower == open ->
                                if
                                :: !lock_is_occupied ->
                                                ship_status[shipid] = go_up_in_lock;
                                                lock_is_occupied = true;
                                                nr_of_ships_at_pos[ship_pos[shipid]]--;
                                                observed_low[0]!true;
                                                break;
                                :: lock_is_occupied ->
                                                observed_low[0]!true;
                                fi; }
                :: atomic { doors_status.lower == open &&
                        !lock_is_occupied ->
                                ship_status[shipid] = go_up_in_lock;
                                lock_is_occupied = true;
                                nr_of_ships_at_pos[ship_pos[shipid]]--;
                                break; }
                od;
        :: ship_status[shipid] == go_up_in_lock ->
                do
                :: doors_status.higher == closed ->
                        request_high!true;
                        atomic { doors_status.higher == open ->
                                if
                                :: (nr_of_ships_at_pos[ship_pos[shipid]+1] < MAX
                                        || ship_pos[shipid]+1 == N) ->
                                                ship_status[shipid] = go_up;
                                                lock_is_occupied = false;
                                                ship_pos[shipid]++;
                                                nr_of_ships_at_pos[ship_pos[shipid]]++;
                                                observed_high[0]!true;
                                                break;
                                :: (nr_of_ships_at_pos[ship_pos[shipid]+1] == MAX
                                        && ship_pos[shipid]+1 != N) ->
                                                observed_high[0]!true;
                                fi; }
                :: atomic { doors_status.higher == open &&
                        (nr_of_ships_at_pos[ship_pos[shipid]+1] < MAX
                        || ship_pos[shipid]+1 == N) ->
                                ship_status[shipid] = go_up;
                                lock_is_occupied = false;
                                ship_pos[shipid]++;
                                nr_of_ships_at_pos[ship_pos[shipid]]++;
                                break; }
                od;
        :: ship_status[shipid] == go_down && ship_pos[shipid] == 0 ->
                ship_status[shipid] = goal_reached; ship_status[shipid] = go_up;
        :: ship_status[shipid] == go_up && ship_pos[shipid] == N ->
                ship_status[shipid] = goal_reached; ship_status[shipid] = go_down;
        od;
}
```

Figure 3: PROMELA model of a ship, part 2.

global boolean variable `lock_is_occupied`. If there is no ship present, the ship process can change its state to `go_down_in_lock`, to record that it is moving downwards, and is currently placed inside a lock. When this is done, it sets `lock_is_occupied`, updates `nr_of_ships_at_pos` to record that the ship has moved from its position, and notifies the main controller that it has seen that the higher pair of doors has been opened. This is done via the synchronous channel `observed_high` that corresponds with lock 0. Note that `observed_high` is actually an array of channels. This will prove useful when extending the model to multiple locks for Section 3!

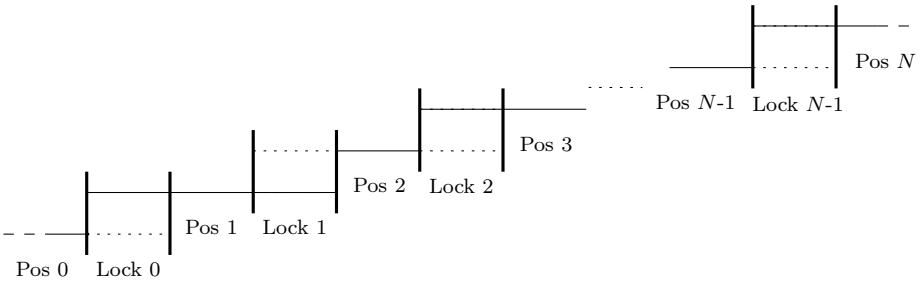

Figure 4: A flight of locks

The `nr_of_ships_at_pos` array stores per position (the `Pos i` positions in Figure 4) the number of ships located there.

It is crucial that the procedure to enter a lock is placed in one `atomic` statement, to exclude the possibility that multiple ships notice that a lock is not occupied, and then simultaneously decide to enter it.

Once the notification is sent to the main controller, the ship process breaks out of the inner `do`-statement, leading back to the outer `do`-statement, in which the status of the ship is checked again. Alternatively, if the lock is occupied, the notification is still sent without the ship entering the lock. In that case, the ship can send another request to open the doors at a later time.

If the ship encounters open higher doors and the lock is not yet occupied by a ship, it does not need to send a request via `request_high`, but can immediately occupy the lock.

When a ship inside a lock desires to move down, the ship process checks the lower doors of that lock. If those doors are closed, it makes a request to open the lower doors over the asynchronous channel `request_low`. After that, the ship process waits for the lower doors to open, and when they do, the ship process checks how many ships are currently located outside the lower doors of the lock. The relevant position for this is position `shippos[shipid]-1`, corresponding to the position to which the ship with ID `shipid` intends to move. If this number of ships is currently smaller than the maximum number allowed per position, indicated by the constant `MAX`, then the ship can move. An exception is position 0, which allows any number of ships to be placed there. If the ship can move, it changes its status to `go_down` again, the lock is no longer occupied, it changes its position, and a notification is sent to the main controller via the synchronous channel `observed_low` for lock 0. This notification is also sent if the ship cannot move outside the lock.

If the ship encounters open lower doors, and the maximum number of ships below the lock has not been reached, the ship can move outside the lock without first sending a request via `request_low`.

The two cases for `go_up` and `go_up_in_lock` are presented in Figure 3. These are very similar to the cases for `go_down` and `go_down_in_lock`, respectively, and address the situation that a ship moves from low to high. We do not discuss them due to their similarity to the other two cases.

The final two cases express what should happen when a ship has reached its goal: in that case, its status should change to `goal_reached`. Once this has been done, the ship starts moving in the opposite direction.

```
// Initial process that instantiates all other processes and creates
// the initial lock and ship situation.
init {
        byte proc;
        atomic {
                run main_control();
                // In the code below, the individual locks are initialised.
                // The assumption here is that N == 1. When generalising the model for
                // multiple locks, make sure that the do-statement is altered!
                proc = 0;
                do
                :: proc < N ->
                        doors_status.lower = closed;
                        doors_status.higher = closed;
                        slide_status.lower = closed;
                        slide_status.higher = closed;
                        lock_water_level = high_level;
                        lock_is_occupied = false;
                        run lock(proc);
                        proc++;
                :: proc == N -> break;
                od;
                // In the code below, the individual ship positions and directions
                // are initialised. Expand this when more ships should be added.
                proc = 0;
                do
                :: proc == 0 -> ship_status[proc] = go_up; ship_pos[proc] = 0;
                        run ship(proc); proc++;
                :: proc > 0 && proc < M -> proc++;
                :: proc == M -> break;
                od;
                // Do not change the code below! It derives the number of ships per
                // position from the positions of the individual ships.
                proc = 0;
                do
                :: proc < M -> nr_of_ships_at_pos[ship_pos[proc]]++; proc++;
                :: else -> break;
                od;
        }
}
```

Figure 5: PROMELA model of the init process.

1.3.3 The init process

The init process is used to initialise the system, i.e., to provide its initial configuration. This process is presented in Figure 5. All this is done in a single `atomic` statement. First, the main control process is started, after which N locks are started inside a `do`-statement. The initial values of the relevant variables for the single lock are set inside this `do`-statement. When developing the multiple locks system, this statement must be altered. Following this, the M ships are instantiated. Inside this `do`-statement, you can set the initial direction and position of each ship. In its current form, only a single ship process is effectively created. Extend this code if you want to use more than one ship!

In the final part of the process, the `nr_of_ships_at_pos` array is initialised. This code does not need to be changed, as this information is automatically derived from the initial positions of the ships.

```
// DUMMY main control process type. Remodel it to control the lock system and handle
// requests of ships!
proctype main_control() {
        do
        :: request_low?true ->
                if
                :: doors_status.lower == closed ->
                        change_doors_pos!low; doors_pos_changed?true;
                :: doors_status.lower == open -> skip;
                fi;
                observed_low[0]?true;
        :: request_high?true ->
                if
                :: doors_status.higher == closed ->
                        change_doors_pos!high; doors_pos_changed?true;
                :: doors_status.higher == open -> skip;
                fi;
                observed_high[0]?true;
        od;
}
```

Figure 6: Dummy PROMELA model of a lock system main control process type.

2 A single lock system

2.1 Modelling

The first modelling step consists of modelling the main control of a single lock system. This controller needs to be able to receive requests from ships, and control the lock to handle the requests, in other words, to open and close its doors in a safe way.

For the main control, a dummy process type is given. It should be replaced by a fully functioning main control process type. Next, we describe the process type as it exists in the given model. It is presented in Figure 6. It receives requests via the asynchronous channels `request_low` and `request_high`, and acts on these by instructing the appropriate pair of doors to change state, if needed, after which it waits for confirmation that the doors have indeed changed state. Once a request has been processed, the controller waits for confirmation from a ship that the open doors have been observed.

The main control process as it is now specified is clearly not providing the required functionality. For instance, it does not consider the slides. You have to rewrite it, so that it controls all components of the lock system in a correct way.

Note 5. Note that since `request_low` and `request_high` are *asynchronous* channels, any requests made by ships are added to a queue. Processing these requests is not immediately needed. When constructing and investigating the behaviour of the main controller, you will therefore notice that ship requests are not immediately followed by a reaction of the main controller.

Note 6. When you use SPIN to explore the state space of the system, SPIN will report messages such as this:

```
unreached in proctype lock
        lock_env_1.pml:104, state 50, "-end-"
        (1 of 50 states)
unreached in proctype ship
        lock_env_1.pml:156, state 72, "-end-"
        (1 of 72 states)
```

These messages indicate that the lock and ship processes in the model never actually terminate; they do not reach their end state. However, this is not a problem, since they are not supposed to ever terminate. Therefore, do not be alarmed by these messages! In addition, some parts of the model, particularly of the lock process type, may not be reached, depending on how you design your main controller. This is also not necessarily a problem!

2.2 Property verification

Besides the modelling, you need to formalise system requirements in the form of assertions and LTL formulas. Assertions can be used to formalise safety properties, and LTL formulas are typically used in SPIN to express liveness properties. The `lock_env.pml` file contains one example for each type of property.

1. When no LTL formulas or assertions are present in a PROMELA model, SPIN checks for the presence of deadlocks (called 'invalid end states' in SPIN) when you press the 'Verify' button of JSPIN. The first step is to make sure that your model can be checked for deadlocks. If SPIN does not report that invalid end states exist, then there are no deadlocks. The model that you submit to us should be deadlock-free! Discuss the result of this check in your report. How many states and transitions were identified by SPIN?

2. In addition, the following properties need to be formalised to check the correctness of the single lock system:

   (a) The lower pair of doors and the higher pair of doors are never simultaneously open.

   (b1) When the lower pair of doors is open, the higher slide is closed.

   (b2) When the higher pair of doors is open, the lower slide is closed.

   (c1) The lower pair of doors is only open when the water level in the lock is low.

   (c2) The higher pair of doors is only open when the water level in the lock is high.

   (d1) Always if a ship requests the lower pair of doors to open and its status is `go_up`, the ship will eventually be inside the lock.

   (d2) Always if a ship requests the higher pair of doors to open and its status is `go_down`, the ship will eventually be inside the lock.

   Provide formalisations of these properties in your report, and motivate your choices, i.e. why you have used assertions and/or LTL formulas. You may assume that $M = 1$ in your formalisations.

Note 7. In assertions and LTL formulas, you can only refer to the values of variables. It is not possible to directly refer to execution steps taken by the PROMELA processes. If you feel the need to, you may add variables to the model to keep track of information you wish to refer to in the assertions and LTL formulas.

Note 8. Assertions can be used to verify safety properties, but it may sometimes not be very clear where these assertions should be placed in the model. One elegant way to address this is by creating a new *monitor* process type. Consider the following process description:

```
proctype monitor() {
        assert(B);
}
```

This process type never alters the system state, but only checks whether the assertion `B` holds. When SPIN explores the state space, this results in checking whether `B` holds in every reachable system state, and hence, this mechanism can be used to check the safety property that `B` is never `false`. You never need multiple monitor process types in a model, as multiple assertions can be checked in a single process type. If you use a monitor process type, do not forget to create an instance of it in the init process! Monitor processes are also used in the SPIN video instructions.

Note that the `lock_env.pml` template also contains a monitor process type. The assertion in it expresses that the position of ship 0 always refers to a valid position, i.e., it is never smaller than 0 and never larger than N.

3. Once the properties have been formalised, they can be verified using the SPIN model checker. Describe in your report the outcome of verifying whether the above properties hold in your single lock model. This means that:

   - When a property is true, you explain why this is to be expected.
   - When a property is false, you give a counter-example (trace of the system demonstrating that the property does not hold), and explain the erroneous behaviour of the system. If the property is a liveness property, does it hold under weak fairness?

   In the final version of your single lock model, all properties should hold, and the formulas to check this should be present in the model. It is allowed that a property only holds under weak fairness.

   Note 9. When verifying, be careful to set the maximum search depth sufficiently high (in the JSPIN menu under 'Settings'). If SPIN reports `error: max search depth too small`, the verification is actually not performed completely, and the absence of deadlocks or validity of a property is not guaranteed!

   Note 10. When inspecting a counter-example, SPIN may report that the maximum number of steps has been reached. To view the entire counter-example, you can increase this maximum by selecting the menu option 'Max steps' under 'Settings'.

3 A multiple locks system

3.1 Modelling

The next step in the modelling task is to generalise the lock system by allowing the presence of multiple locks. Similar to how the given model is configurable w.r.t. the number of ships (that is, only the constant M needs to be changed and the init process must be adapted in order to change the number of ships in the model), you should make sure that in your model, the number of locks is configurable as well.

The introduction of multiple locks has a number of consequences for the model:

1. As each lock has its own pairs of doors and slides, water level and occupancy state, not only the right number of processes must be instantiated, given the number of locks, but also a sufficient number of global variables must be instantiated to store the states of the locks and positions outside the locks.

2. Rethink how ships can request that doors are opened. The channels `request_low` and `request_high` are used to request the opening of the lower and higher pair of doors, respectively, of a single lock. Now, with multiple locks present, when a request is made, it should be clear which lock needs to open its doors. Requests still need to be made via channels, though; global variables are not allowed for this.

3. Make sure that the lock and ship process types are adapted to the new way of channel communication.

4. Make sure that your main control process can control all locks in the system, and handle all requests of the ships.

3.2 Property verification

1. When increasing the number of locks and ships in the system, you will notice that the verification time of Spin rapidly increases. This phenomenon is known as the *state space explosion problem*. Try to make the model verifiable when it is configured for three locks and two ships, and four locks and two ships. For these configurations, the multiple locks system should be free of deadlocks. The complexity of the main control process has considerable impact on the verification time. In your report, discuss the time needed to verify the absence of deadlocks of your model with the two configurations (reported in the output of Spin as `pan: elapsed time ...`), and if relevant, which modifications you made to improve the scalability of the model.

2. You may notice that as the number of locks and ships increases, Spin starts to report that there are deadlocks (invalid end states) present. What is the minimum number of locks and ships that leads to this result, and which initial configuration is used? Provide a counter-example, as given by Spin, and explain why this execution leads to a deadlock.

3. In addition, the following properties need to be formalised, to check the correctness of the multiple locks system:

   (e1) When a request is made to open the lower doors of lock i, eventually the lower doors of lock i are open.

   (e2) When a request is made to open the higher doors of lock i, eventually the higher doors of lock i are open.

   (f1) Always eventually a request is made to open the higher doors of lock $N - 1$.

   (f2) Always eventually a request is made to open the lower doors of lock 0.

   Provide formalisations of these properties in your report, and motivate your choices, i.e., why you have used assertions and/or LTL formulas.

   Note 11. In an LTL formula, you cannot express that for all N locks, or for all M ships, a property φ holds. If there is a need to do so, you may express a weaker version of the property, in which you state that φ holds for a particular lock or ship. When verifying the properties, checking different versions of that property for each lock or ship (given a concrete N or M, respectively) is then sufficient to verify the original property.

   Hint. Checking whether a particular message is currently at the head of the buffer of an asynchronous channel can be done via *polling*. For instance, `c?[5]` evaluates to **`true`** if and only if the current message at the head of the channel `c` (of type `byte` or `int`) has the value `5`. You can use polling in models, assertions and LTL formulas.

4. Once the properties have been formalised, they can be verified using SPIN. Describe in your report the outcome of verifying the above properties for your multiple locks system with the three locks, two ships configuration. This means that:

   - When a property is true, you explain why this is to be expected.
   - When a property is false, you give a counter-example (trace of the system demonstrating that the property does not hold), and explain the erroneous behaviour of the system. If the property is a liveness property, does it hold under weak fairness?

   In the final version of your multiple locks system model, all properties should hold. It is allowed that a property only holds under weak fairness.

5. **Bonus question:** Finally, the model may contain deadlocks when the number of locks and ships is sufficiently high, but you can actually use SPIN to construct a *schedule* for the main controller that avoids deadlocks in those cases, and allows all ships to reach their goal. For the minimal configuration with deadlocks that you identified earlier, formalise the property that it is not possible for all ships to reach the status `goal_reached`. Verify this property, and explain what SPIN reports. Make sure that you add `-E` to the options for 'Pan' (in the 'Options' menu), to tell SPIN to ignore any deadlocks it encounters along the search. If everything works

fine, SPIN should report a counter-example that demonstrates how deadlocks can be avoided. How long is the counter-example, in steps? (It can be very long!). Reason about why this execution avoids deadlocks. To inspect the counter-example, you can select which details to show using the 'Common' menu item under 'Options'.

4 Deliverables

4.1 Modelling and verification

The deliverable for this assignment is a report in which you describe and motivate your modelling choices, give formalisations of the requirements, and discuss the verification outcomes. This report should be a **single PDF file**. The assignment is to be handed in using the submission page in Canvas. In addition, hand in two PROMELA files, one for the single lock system and one for the multiple locks system. Both files should include the formalised properties for the corresponding model. The files are to be submitted in a ZIP file via Canvas.

Files not included in the report or without reference from the report will not be considered when grading. Make sure you give a textual explanation for both models in the report, specifically focussing on what you added to the given template.

4.2 Reflection

At the end of your report, include a section in which you reflect on the assignment. This should consist of the following:

1. Description of difficulties you encountered, or any questions you still have.
2. Brief description of the contributions of both team members. Should it turn out that there are large differences between the contributions of the team members, this may affect the individual grades.