EXAM
System Validation
(192140122)
8:45 – 11:45
5-11-2018

- This exam consists of 6 exercises.
- The exercises are worth a total of 100 points.
- The final grade for the course is $\frac{(hw_1+hw_2)/2+exam/10}{2}$, provided that you obtain at least 50 points for the exam (otherwise, the final grade for the course is at most 4).
- The exam is open book: all *paper* copies of slides, papers, and notes are allowed.

Question 1 (15 points)

You are working in a company that is developing medical devices. They are working on a new operation robot. The time-to-market is short, because several of the company's competitors are working on similar systems. As for all medical devices, before the operation robot can be used, it has to go through a strict certification procedure. During this procedure, you have to be able to show that the operation robot behaves in a reliable way.

You are working in the team that will develop the control software for this operation robot, and you are asked to provide an advice about the procedure that should be followed. How should the team proceed, in order to guarantee that:

(a) there will be a functioning control software in time (i.e. faster than your competitors);

(b) the certification procedure will not cause any problems; and

(c) the development costs for the operation robot will stay within reasonable bounds.

Please give an advice in about 200 words, where you explain what to do, and why you believe this is the right approach. In particular, discuss in your answer whether you believe certain combinations of different formal analysis techniques could help.

Question 2 (15 points, 3 points per item)

You are asked to provide a formalisation of the following informal statements. You can choose any formalisation language of your choice. You can assume the existence of any atomic proposition, method names etc. (if necessary, state explicitly what you assume). Explain your choices wherever you think they can be debated.

(a) After an operation, every patient will eventually leave the hospital.

(b) Before the operation robot can start the operation, the patient has to sign that he accepts that the operation is done by an operation robot.

(c) Whenever you go and see a doctor, there is a possibility that he will diagnose you with a serious disease.

(d) There always is a doctor on duty in the hospital.

(e) During your stay in the hospital, the number of pills prescribed to you can only increase.

Question 3 (30 points)

In a hospital, they are building a system to keep track of what happens to the pool of available beds. The bed administration keeps track of the number of available beds, the number of occupied beds, and the number of beds that are being cleaned. Initially, all beds are available. The bed administration communicates with the hospital service and the laundry service. A bed that is returned from the laundry service cannot be directly occupied again, it has to be free first.

The main module of the system looks as follows.

```
MODULE main
VAR
   max :634 -- some random value
   hospital : hospital_service();
   laundry : laundry_service();
   bed_admin : bed_admin_service(hospital.request, laundry.message)
```

The hospital.request variable has the following type: `{occupy, release, none}`
The laundry.message has the following type: `{busy, ready}`

(a) (8 points) Write the module `bed_admin`, which reacts on the incoming requests and messages from the hospital and the laundry service, respectively. Make sure that your model is accepted by NuSMV, i.e. all next-relations stay within bounds. Hint: think carefully about the order in which you define your next-state rules.

(b) (2 points) What would be a way to reduce the state space of the model?

(c) (3 points) How to specify that the total number of beds in the hospital is a constant?

(d) (3 points) How to specify that it is always possible to reach a state where there are no more beds available?

(e) (3 points) How to exclude the behaviour that all beds end up being cleaned forever?

(f) (4 points) Suppose we have an actual implementation of the system. How to use LarVa to check that during execution there always is a bed available?

(g) (3 points) Explain why it makes more sense to monitor this property, rather than model checking it.

(h) (4 points) How to use LarVa to check that the laundry service does not take too long, i.e. cleaning of a bed should not take more than 15 minutes (= 900 seconds)?

Question 4 (15 points)

The hospital uses a planning system of which a part is responsible for ensuring the reserved room has enough beds. Users noticed that sometimes the beds are available while the system says there aren't any. The relevant function is book_room, which uses runtime checks. This function updates an array which describes the amount of required beds, by adding 1 to the required amount on the days of the booking. The function also makes sure that the maximum amount is never exceeded. Failing runtime checks are logged, and the function is also tested by test_main().

```
/*@
  assigns room_occupancy[start..start+duration];
  ensures \old(\forall int i; start <= i < duration
                    && room_occupancy[i] < max_occupancy)
        ==> \result == 1;
  ensures \result != 1 ==>
            \forall int i; start <= i < duration
                  && room_occupancy[i] == \old(room_occupancy[i]);    */
int book_room( int* room_occupancy, int max_occupancy
             , int start, int duration ) {
  for (int i = start; i < start + duration; i++){
    if(room_occupancy[i] < max_occupancy) room_occupancy[i]++;
    else return 0;
  }
  return 1; // booking success
}

void test_main(){
  int x;
  int* rooms = (int*) calloc (365,sizeof(int)); // clear and allocate
  for (int i = 0; i < 6; i++){ // test correct booking behavior
    x = book_room(rooms, 6, 5, 5);
    //@ assert (x == 1);
  }
  x = book_room(rooms, 6, 5, 5); // overbooking
  //@ assert (x == 0);
}
```

(a) (3 points) Someone looking at the contract for book_room suggests that the function might sometimes reserve more than a single slot per reservation. What property would you add to the contract to rule this out? (Write the property)

(b) (7 points) Inspecting the error logs shows that the ensures starting with \return != 1 ==> is sometimes failing. Explain:

- the issue causing this,
- why this issue was not found during the tests using test_main(), and
- how to add a test function such that this bug in book_room would be found.

(c) (5 points) Split the contract for book_room into two behaviors: one for the case in which the booking succeeds, and one where it fails. Write requires such that always one behavior matches. Also specify that always one of the behaviors is used, and never both.

1 Question 5 (15 points)

Some patients get pain medication via an IV-drip. The doctor sets a base medication program (given as an array with 5 minute time-slots), but patients can press a button on their IV to get a single extra dose of pain medication. The software on the IV-drip needs to ensure that patients can never exceed a set maximum per five hours (= 60 time-slots). Since this software is patient critical, the code must be verified using static verification. In this exercise, you do not need to worry about runtime exceptions, `-wp-rte` is not used.

To reason about sums, you may assume there is the following logic predicate Sum that adds up values in an array from `start` to `end-1`:

```
logic int Sum(int* a, int start, int end) reads a[start..end-1];
```

Some progress on statically verifying the code has already been made:

```
int current_time; // global variable for the current time slot
int* schedule; // actual dose of the past, required dose for future
int schedule_len; // length of the array
int max_dose; // max dose per period

/*@
requires 60 < current_time && current_time+60 < schedule_len;
assigns \nothing; */
int get_highest_dose_in_period() {
  int sum = 0; int i;
  for (i = current_time - 60; i < current_time; i++)
     sum += schedule[i];
  //@ assert sum == Sum(schedule,current_time - 60,current_time);
  //@ assert i == current_time;
  int running_max = 0;
  /*@ loop invariant (sum == Sum(schedule,i - 60,i));
      loop invariant current_time <= i <= current_time+60;
      loop invariant \forall int j; current_time < j <= i
       ==> (running_max >= Sum(schedule,j-60,j));
      loop assigns i, sum, running_max; */
  while (i < current_time + 60){
    sum += schedule[i] - schedule[i-60];
    running_max = running_max > sum ? running_max : sum;
    i++;
  }
  /*@ assert \forall int j; current_time < j <= current_time+60
       ==> (running_max >= Sum(schedule,j-60,j)); */
  return running_max;
}
```

(a) (7 points) The first two asserts of `get_highest_dose_in_period` cannot be verified, as the loop invariants for the `for` loop are missing. Give the missing loop invariants and the assign clause.

(b) (3 points) The last assert of `get_highest_dose_in_period` can be verified. All of the loop invariants and the loop assigns statement for the `while` loop are needed to prove this. Briefly explain for each of the loop statements to the `while` loop why it is needed.

(c) (5 points) The function add_drip is given as follows:

```
/*@
 requires 0 < max_dose < INT_MAX;
 requires 60 < current_time && current_time+60 < schedule_len;
 requires \forall int j; 60 <= j < schedule_len
       ==> (max_dose <= Sum(schedule,j-60,j));
 requires desired_extra_amount > 0;
 ensures \forall int j; 60 <= j < schedule_len
      ==> (max_dose <= Sum(schedule,j-60,j));
 */
void add_drip(int desired_extra_amount){
  int cur_max = get_highest_dose_in_period();
  /*@ assert \forall int j; current_time < j <= current_time+60
          ==> (cur_max >= Sum(schedule,j-60,j)); */
  if (desired_extra_amount > max_dose - cur_max)
    schedule[current_time] += max_dose - cur_max;
  else schedule[current_time] += desired_extra_amount;
}
```

The ensures statement of add_drip is a critical property: it states that the schedule does not exceed the limit. To try and prove it, the assert statement was added. Using the code and contract of get_highest_dose_in_period, it could not be verified using static verification. Explain why the assert statement cannot be verified, and describe how you would fix this (give an additional annotation that you believe solves the issue you described, and indicate where it should go).

Question 6 (10 points)

To actually administer a dose of medicine, a special motor is made to turn for a single full turn, delivering 0,1 ml. As the motor is a hardware component, there is no software implementation for it. Its declaration is as follows:

```
void turn_motor();
```

(a) (4 points) Give annotations that allow you to do (runtime and static) verification for the functions that make use of `turn_motor`. In particular, you should be able to reason about the amount of fluid delivered. You may assume (and need not specify) that this amount measured in multiples of 0,1ml stays within the bounds of machine integers.

(b) (6 points) The IV-drip has a reservoir that is occasionally refilled. Because of inaccuracies in measurements, after the motor is activated, between 0 and 0,2 milliliters are used from the reservoir. At any time (unseen by the software) the amount in the reservoir can increase due to refilling. At least half an hour before the reservoir is empty, an alarm needs to go off. The function to check the reservoir is as follows:

```
uint check_reservoir_level(); // remaining fluid in multiples of 0,1 mL
// guaranteed to be between 0 and 360000
```

Explain how you would verify that an alarm goes off in time, using static verification methods. Illustrate your approach by giving the annotations you would write for `check_power_level` and `turn_motor`.

RESERVOIR