EE219C HW3: Model Checking

Vighnesh Iyer

1 Interrupt Driven Program

(a) Describe the properties in the `Sys` module in English. Note the composition of `main` and `ISR` within the module `Sys` is incorrectly done (you will need to fix it later).

- `invariant main_ISR_mutex: (M_enable != I_enable);`
  Only the main module or the ISR module can be active at a given timestep.
- `property[LTL] one_step_ISR_return: G(return_ISR ==> X(!return_ISR));`
  If `return_ISR` is true in a given timestep, it should be false in the next timestep for any trace of the system. This should ensure that the ISR module can't be advanced through in less than 1 timestep.
- `property[LTL] main_after_ISR: G((I_enable && X(M_enable)) ==> return_ISR);`
  If on a particular timestep `I_enable` is true and on the next time step `M_enable` will be true, then `return_ISR` should also be true on this timestep.
- `property[LTL] ISR_after_main: G((M_enable && X(I_enable)) ==> (assert_intr));`
  The 'dual' of the previous property: if we are in the main module execution, and we are going to move into the ISR module next, the interrupt from the environment must also have been asserted right now.

(b) Run the file and interpret results.

The invariant passes a 20 cycle unrolling because `mode` isn't being updated, and `M_enable` and `I_enable` are mutually exclusive conditions.

The latter 2 LTL properties fail to check because `mode` is being set arbitrarily and the counter-example traces contain transitions between the main and ISR modules that don't match the havoc behavior of `mode` being arbitrarily set by the solver.

(c) Fix to correctly compose main and ISR in `Sys`, and eliminate the above CEXs. Change `update_mode`.

```
procedure update_mode() modifies mode; {
  case
    mode == main_t: {
      if (assert_intr) {
        mode = ISR_t;
      } else {
        mode = main_t;
```

```
      }
    }
    mode == ISR_t: {
      if (return_ISR) {
        mode = main_t;
      } else {
        mode = ISR_t;
      }
    }
  esac
}
```

Now all the assertions pass.

(d) Correctly compose `Sys` and `Env` (should it be async composition with interleaving semantics?). Explain why the `consec_main_pc_values` property may fail when the composition is corrected.

I modified the `init` and `next` blocks as such:

```
init { havoc turn; }

next {
  if (turn) {
    next (Sys_i);
  } else {
    next (Env_i);
  }
  if (assert_intr) {
    turn' = true;
  } else {
    havoc turn;
  }
}
```

I don't think it's acceptable for `Sys` and `Env` to be composed with async composition since once `Env` raises the interrupt line, it is required that `Sys` executes next to properly receive and interrupt. I think the decision as to which module to execute should be arbitrary, except with this one constraint.

With this modification the `consec_main_pc_values` property fails because the solver makes `turn` false all the time and prevents forward movement in `Sys` which causes the property to fail in limited time.

2 Smart Intersection

(a) Construct the `no_collision` invariant.

```
invariant no_collision: (car1_pos != car2_pos) && (car2_pos != car3_pos) &&
↪  (car1_pos != car3_pos);
```

(b) Define the `can_move` procedure to determine which car can move

This was a lot harder than I expected. I first tried to incorporate the next potential positions of each car and asserted that cars can only move if there are no 'conflicts' but this was hard to encode correctly as setting `move_p1` relied on knowing what `move_p2,p3` were *going* to be set to.

I then tried to enforce a car movement ordering like this:

```
move_p1 = (!at_sink(c1) && at_source(c2) && at_source(c3)) || (at_sink(c1)
↪  && at_sink(c2) && at_sink(c3));
move_p2 = (at_sink(c1) && !at_sink(c2) && at_source(c3)) || (at_source(c1)
↪  && at_sink(c2) && at_sink(c3));
move_p3 = (at_sink(c1) && at_sink(c2) && !at_sink(c3)) || (at_source(c1) &&
↪  at_source(c2) && at_sink(c3));
```

but this too had a counterexample where car1 lands in a sink, then car2 moves to that same sink before car1 had a chance to respawn.

A brute force method implements car1, car2, car3 ordering where each car must start from a source, finish its turn in the intersection, land in a sink and then respawn, before the subsequent car is allowed to move.

```
type turn_t = enum { move, respawn };
var turn1, turn2, turn3 : boolean;
var turn1_status, turn2_status, turn3_status: turn_t;
call (move1', move2', move3', turn1', turn2', turn3', turn1_status',
↪  turn2_status', turn3_status') = can_move(car1_pos, car2_pos, car3_pos,
↪  turn1, turn2, turn3, turn1_status, turn2_status, turn3_status);

procedure can_move(c1 : pos_t, c2 : pos_t, c3 : pos_t,
turn1: boolean, turn2: boolean, turn3: boolean,
turn1_status: turn_t, turn2_status: turn_t, turn3_status: turn_t)
  returns (move_p1 : boolean, move_p2 : boolean, move_p3 : boolean,
  turn1_nxt: boolean, turn2_nxt: boolean, turn3_nxt: boolean,
  turn1_status_nxt: turn_t, turn2_status_nxt: turn_t, turn3_status_nxt:
   ↪  turn_t)
{
    if (turn1) {
      move_p1 = true; move_p2 = false; move_p3 = false;
    }
    if (turn2) {
      move_p1 = false; move_p2 = true; move_p3 = false;
    }
    if (turn3) {
      move_p1 = false; move_p2 = false; move_p3 = true;
    }
    if (at_sink(c1)) {
      turn1_status_nxt = respawn;
    } else {
      turn1_status_nxt = move;
```

```
    }
    if (at_sink(c2)) {
      turn2_status_nxt = respawn;
    } else {
      turn2_status_nxt = move;
    }
    if (at_sink(c3)) {
      turn3_status_nxt = respawn;
    } else {
      turn3_status_nxt = move;
    }

    turn1_nxt = turn1; turn2_nxt = turn2; turn3_nxt = turn3;
    if (turn1_status_nxt == respawn) {
      turn1_nxt = false; turn2_nxt = true; turn3_nxt = false;
    }
    if (turn2_status_nxt == respawn) {
      turn1_nxt = false; turn2_nxt = false; turn3_nxt = true;
    }
    if (turn3_status_nxt == respawn) {
      turn1_nxt = true; turn2_nxt = false; turn3_nxt = false;
    }
}
```

This finally worked and passed a 16 cycle BMC. However it is needlessly complex; so I found another solution without adding new variables:

```
procedure can_move(c1 : pos_t, c2 : pos_t, c3 : pos_t)
  returns (move_p1 : boolean, move_p2 : boolean, move_p3 : boolean)
{
  move_p1 = (at_source(c1) && at_source(c2) && at_source(c3)) ||
  ↪  in_intersection(c1) || (at_sink(c1) && at_source(c2) &&
  ↪  at_source(c3));
  move_p2 = (at_sink(c1) && at_source(c2) && at_source(c3)) ||
  ↪  in_intersection(c2) || (at_source(c1) && at_sink(c2) &&
  ↪  at_source(c3));
  move_p3 = (at_source(c1) && at_sink(c2) && at_source(c3)) ||
  ↪  in_intersection(c3) || (at_source(c1) && at_source(c2) &&
  ↪  at_sink(c3));
}
```

The way this works is to let car1 go from source to sink and respawn, and at the same time it respawns, car2 is let go, and then car3 and the cycle repeats. This passes a 16 cycle BMC too.

(c) Define the `bounded_exit` invariant and verify it for 16 cycles with BMC

In the worst case, each car takes 6 cycles to move from source to sink and then respawns at a new source, but these cycles overlap by 1, so I expect the bound on `wait_cnt` to be 16.

```
invariant bounded_exit: (wait_cnt1 < 16) && (wait_cnt2 < 16) && (wait_cnt3 < 16);
property[LTL] bounded_1: G(F(wait_cnt1 == 0));
property[LTL] bounded_2: G(F(wait_cnt2 == 0));
property[LTL] bounded_3: G(F(wait_cnt3 == 0));
```

I extended the BMC to 17 cycles of unrolling just to be sure and these invariants and properties held.

(d) Copy over to the induction file and prove `no_collision` with induction for unbounded time. Add strenghtening invariants as necessary.

I spent a lot of time trying to prove the `bounded_exit` invariant with induction and only later realized that the problem didn't ask for that. Proving `no_collision` was much easier, only 1 invariant was needed:

```
invariant car_positions: (at_source(car1_pos) && at_source(car2_pos) &&
↪  at_source(car3_pos)) || (at_source(car1_pos) && at_source(car3_pos))
↪  || (at_source(car2_pos) && at_source(car1_pos)) ||
↪  (at_source(car3_pos) && at_source(car2_pos));
```

which just explicitly states the possible car locations according to the turn-taking algorithm.

I had a bunch of other invariants in a hopeless attempt to show `bounded_exit`, although now that I think about it, the BMC showed that already! There's no need for induction! Just for recording:

```
invariant valid_steps_src: (at_source(car1_pos) ==>
↪  valid_steps_at_src(steps1)) && (at_source(car2_pos) ==>
↪  valid_steps_at_src(steps2)) && (at_source(car3_pos) ==>
↪  valid_steps_at_src(steps3));
invariant valid_steps_sink: (at_sink(car1_pos) ==> steps1 == -1) &&
↪  (at_sink(car2_pos) ==> steps2 == -1) && (at_sink(car3_pos) ==> steps3 ==
↪  -1);
invariant valid_steps_sink_bi: (steps1 == -1 ==> at_sink(car1_pos)) &&
↪  (steps2 == -1 ==> at_sink(car2_pos)) && (steps3 == -1 ==>
↪  at_sink(car3_pos));
invariant valid_steps_all: valid_steps(steps1) && valid_steps(steps2) &&
↪  valid_steps(steps3);
invariant wait_above_zero: (wait_cnt1 >= 0) && (wait_cnt2 >= 0) &&
↪  (wait_cnt3 >= 0);
```

3 Linear Temporal Logic

(a) Labeled transition system that encodes the dynamics of the puzzle

Let the state variables of the system $S = \{g, w, c\}$, where $g \in \{G_L, G_R, G_B\}$, $w \in \{W_L, W_R, W_B\}$, $c \in \{C_L, C_R, C_B\}$, where g, w, c represent the locations of the goat, wolf, and cabbage on the left or right river banks, or on the boat.

The initial state $S_0 = \{(G_L, W_L, C_L)\}$ puts the goat, wolf, and cabbage on the left bank.

The transition function $\delta((g1, w1, c1), (g2, w2, c2))$ is true if only one of the state variables changes state and false otherwise. It can also be true if a boat swap is performed (i.e. $(G_B, W_L, C_L) \rightarrow (G_L, W_B, C_L)$) is permitted where the goat and wolf were swapped on the boat.

The labeling function $L(g, w, c)$ emits I (for illegal) if the state has the goat and cabbage on the same bank (without the wolf) or if the state has the wolf and goat on the same side (without the cabbage). States with 2 items on the boat are also labeled with I. Label the state (G_R, W_R, C_R) with G (for goal).

(b) LTL formula which encodes the task and the constraints on the goat/cabbage/wolf

The constraints are $\phi_C = G(!I)$. The task is $\phi_T = F(G)$. Final LTL formula: $\phi = \phi_C \wedge \phi_T$.

(c) Give a strategy to solve the river crossing puzzle

(a) Move the goat to the right bank and return
- $(G_L, W_L, C_L) \rightarrow (G_B, W_L, C_L) \rightarrow (G_R, W_L, C_L)$

(b) Then move the cabbage to the right bank, and bring the goat back to the left bank
- $(G_R, W_L, C_L) \rightarrow (G_R, W_L, C_B) \rightarrow (G_B, W_L, C_R) \rightarrow (G_L, W_B, C_R)$

(c) Move the wolf to the right bank and return
- $(G_L, W_B, C_R) \rightarrow (G_L, W_R, C_R)$

(d) Move the goat to the right bank
- $(G_L, W_R, C_R) \rightarrow (G_B, W_R, C_R) \rightarrow (G_R, W_R, C_R)$

The goal state is reached without encountering any illegal states.

(d) Encode the transition system and strategy as a Spin model

I very explicitly encoded the transition system with my particular strategy in spin:

```
#define left 0
#define right 1
#define boat 2
int goat_loc, cabbage_loc, wolf_loc;
bool illegal_state, goal_state;

init {
   goat_loc = left;
   cabbage_loc = left;
   wolf_loc = left;
   illegal_state = false;
   goal_state = false;
   gl_wl_cl:
      atomic{goat_loc = left; wolf_loc = left; cabbage_loc = left;}
      goto gb_wl_cl;
   gb_wl_cl:
      atomic{goat_loc = boat; wolf_loc = left; cabbage_loc = left;}
      goto gr_wl_cl;
   gr_wl_cl:
      atomic{goat_loc = right; wolf_loc = left; cabbage_loc = left;}
```

```
      goto gr_wl_cb;
   gr_wl_cb:
      atomic{goat_loc = right; wolf_loc = left; cabbage_loc = boat;}
      goto gb_wl_cr;
   gb_wl_cr:
      atomic{goat_loc = boat; wolf_loc = left; cabbage_loc = right;}
      goto gl_wb_cr;
   gl_wb_cr:
      atomic{goat_loc = left; wolf_loc = boat; cabbage_loc = right;}
      goto gl_wr_cr;
   gl_wr_cr:
      atomic{goat_loc = left; wolf_loc = right; cabbage_loc = right;}
      goto gb_wr_cr;
   gb_wr_cr:
      atomic{goat_loc = boat; wolf_loc = right; cabbage_loc = right;}
      goto gr_wr_cr;
   gr_wr_cr:
      atomic{goat_loc = right; wolf_loc = right; cabbage_loc = right;
      ↪  goal_state = true;}
      goto done;
   else_case:
      illegal_state = true;
   done:
   ltl goal {[] <> goal_state}
   ltl no_illegal {[] !illegal_state}
   ltl goat_cabbage_apart {[] !(goat_loc == left && cabbage_loc == left &&
   ↪  wolf_loc != left)}
   ltl goat_cabbage_apart_r {[] !(goat_loc == right && cabbage_loc == right
   ↪  && wolf_loc != right)}
   ltl goat_wolf_apart {[] !(goat_loc == left && wolf_loc == left &&
   ↪  cabbage_loc != left)}
   ltl goat_wolf_apart_r {[] !(goat_loc == right && wolf_loc == right &&
   ↪  cabbage_loc != right)}
}
```

(e) Use Spin to prove the strategy and the world model satisfies the LTL formula from (b)

Each run of spin only checks 1 LTL property at a time, so 6 runs were required:

```
spin -a puzzle.spin; and gcc -o pan pan.c; and ./pan -a -N goat_wolf_apart_r
```

and so forth for the 6 LTL properties. Spin exited with 0 errors in each run.

Note: couldn't we just model the problem dynamics in uclid instead and just ask the solver for a model (there should be 2 valid ones)?