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.

So my final working code 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.

(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. So I expect the bound on `wait_cnt` to be 18.

```
invariant bounded_exit: (wait_cnt1 < 18) && (wait_cnt2 < 18) && (wait_cnt3 < 18);
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 20 cycles of unrolling and these invariants and properties held.

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