EECS 219C: Formal Methods — Assignment 2

Parker Ziegler

March 28, 2022

1. Interrupt-Driven Program

a. Describing properties of the `Sys` module

We can describe the properties of the `Sys` module as follows:

1. `invariant main_ISR_mutex` — this property requires that execution of `main` and `ISR` is mutually exclusive. That is, if `main` is executing, `ISR` cannot be executing at the same time (or vice versa).
2. `property[LTL] one_step_ISR_return` — this property requires that, globally, if `ISR` has just returned then, in the next state, `ISR` will not return.
3. `property[LTL] main_after_ISR` — this property requires that, globally, if `ISR` is currently enabled to run and, in the next state, `main` is enabled to run, this implies that `ISR` has just returned.
4. `property[LTL] ISR_after_main` — this property requires that, globally, if `main` is enabled and, in the next state, `ISR` is enabled, this implies that an interrupt has occurred.

b. Interpreting counterexamples from the verifier

Running `uclid` with all properties commented out *except* for `main_after_ISR` results in the following counterexample:

```
CEX for vobj [Step #3] property main_after_ISR:safety @ IntSW.ucl, line 105
=================================
Step #0
  mode : main_t
  M_enable : true
  I_enable : false
  return_ISR : false
  assert_intr : initial_1570_assert_intr
=================================
=================================
Step #1
  mode : ISR_t
  M_enable : true
  I_enable : false
  return_ISR : false
  assert_intr : false
=================================
=================================
Step #2
  mode : main_t
  M_enable : false
  I_enable : true
  return_ISR : false
  assert_intr : false
==================================
==================================
Step #3
  mode : main_t
  M_enable : true
  I_enable : false
  return_ISR : false
  assert_intr : false
==================================

// Steps 4 and step 5 elided.

Finished execution for module: Sys.
```

In this counterexample, the violation of `main_after_ISR` occurs between step 2 and step 3 of the transition system. At step 2, we see that `I_enable` is set to *true* and `M_enable` is set to $false$, indicating that `ISR` is enabled to run. Additionally, the value of `return_ISR` is *false*, indicating that `ISR` has not yet returned. In the next step, `I_enable` is $false$ and `M_enable` is *true*; this indicates that `ISR` should have completed running and `main` can safely be enabled. However, `return_ISR` is still $false$. In this instance, we have a case where `ISR` *was enabled*, but we have no indication that it ever returned before `main` was enabled. This violates the property that if `ISR` was enabled and, in the next state, `main` is enabled, then `return_ISR` *must* be *true*.

Running `uclid` with all properties commented out *except* for `ISR_after_main` results in the following counterexample:

```
CEX for vobj [Step #2] property ISR_after_main:safety @ IntSW.ucl, line 106
==================================
Step #0
  mode : main_t
  M_enable : true
  I_enable : false
  return_ISR : false
  assert_intr : false
==================================
==================================
Step #1
  mode : ISR_t
  M_enable : true
  I_enable : false
  return_ISR : false
  assert_intr : false
==================================
==================================
Step #2
  mode : main_t
  M_enable : false
  I_enable : true
  return_ISR : false
  assert_intr : false

// Steps 3, 4, and 5 elided.
```

```
Finished execution for module: Sys.
```

In this counterexample, the violation of `ISR_after_main` occurs between step 1 and step 2 of the transition system. At step 1, we see that `M_enable` is set to *true* while `I_enable` is set to $false$, indicating we are in the `ISR` function. In step 2, `I_enable` is *true*, `M_enable` is $false$, indicating we moved execution to the `main` function. This second state would imply that an interrupt has been issued; however, `assert_intr` is $false$. This indicates that we transitioned into enabling `ISR` without an interrupt ever occurring. This violates the property that, if `main` is enabled in the current state and, in the next state, `ISR` is enabled, then `assert_intr` *must* be *true*.

c. Correctly composing `main` and `ISR`

See my updated definition of the `update_mode` procedure.

d. Composing `Sys` and `Env` using asynchronous composition with interleaving semantics

See my updates to the `init` and `next` blocks of the `main` module.

The key change introduced in my updates is ensuring that `turn` is non-deterministically updated in the composition of `Sys` and `Env` using `havoc`. In an asynchronous composition of `Sys` and `Env` with interleaving semantics, either `Sys` can transition and `Env` stutters, or `Env` can transition and `Sys` stutters:

$$\begin{aligned}&(\delta((Sys, Env), (Sys', Env')) = \delta(Sys, Sys') \wedge Env = Env)\vee\\&(\delta((Sys, Env), (Sys', Env')) = Sys = Sys \wedge \delta(Env, Env'))\end{aligned}$$

Importantly, however, *which* of these entities transitions in a given step is non-deterministic. For example, we could have a model where only `Env` ever transitions.

This model does lead to a violation of the `consec_main_pc_values` LTL property. `uclid` provides a counterexample in which only `Env` ever transitions. In this case, the `main` function is always at its initial program counter A but, because `Sys` never runs, it never transitions to program counter B. This violates the property that, at some point in the future, `main` will transition from program counter A to program counter B.

2. Smart Intersection

a. Encoding the `no_collision` invariant

In our modeled intersection, we encode collisions as situations in which two of our autonomous vehicles are in the same location in the intersection. To do this, we use pairwise location comparisons between all vehicles. These can be enumerated as follows:

1. If `car1_pos` and `car2_pos` are in the intersection, this implies that `car1_pos` is not equal to `car2_pos`.
2. If `car2_pos` and `car3_pos` are in the intersection, this implies that `car2_pos` is not equal to `car3_pos`.
3. If `car1_pos` and `car3_pos` are in the intersection, this implies that `car1_pos` is not equal to `car3_pos`.

The formal encoding of our `no_collision` invariant is:

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

This invariant is stronger than just doing pairwise comparisons of car locations. A "collision" should be restricted to only occurring within an intersection.

b. Defining the `can_move` procedure

To determine whether or not a car can move, we simply check whether a given car is at a source location, sink location, or in the intersection. If a given car is at a source location or in the intersection, we return *true* for that car. Our reasoning is that:

1. From a source location, a car can always move into the intersection.
2. From an intersection location, a car can always move into either a different intersection location or a sink location.

Conversely, if a car is at a sink location, we return $false$. Our reasoning is that being at a sink location requires a respawn, and therefore the car cannot move in the current transition.

This definition of the `can_move` procedure takes advantage of the fact that the `next_location` procedure already enumerates the exact locations all cars can enter in the current transition. Additionally, our `no_collision` invariant provides a set of constraints guiding where our cars can move while avoiding collisions. All `can_move` needs to encode, then, is a set of constraints that bounds a given car's movement based on its location at a source, sink, or intersection location.

c. Encoding the `bounded_exit` invariant

In order to ensure that each car exits the intersection in a bounded number of steps, we want to ensure that the number of steps it takes combined with the number of times it waits is always less than or equal to 4, the maximum number of steps it can take to move through the intersection. We encode this invariant as follows:

```
invariant bounded_exit: steps1 + wait_cnt1 <= 4 && steps2 + wait_cnt2 <= 4 && steps3 + w
```

The reasoning behind this invariant is based on the relationship between steps and waits for each car. A car's step count is decremented if the car is both able to move and *does* move to a potential next location. Conversely, its wait count is incremented on every transition as long as it hasn't reached a sink location. If a car's step count plus its wait count exceeds 4, this would imply that the car has either:

1. Entered the intersection and has been waiting such that the number of steps its taken plus the number of times its waited exceeds the maximum number of steps it can take to exit the intersection. This would correspond to a situation in which the car is "idling" for an indefinite period of time in the intersection.
2. Not entered the intersection and is "idling" at a source location.

With this invariant, we ensure both that all cars enter the intersection *and* that they exit the intersection in the specified number of steps.

3. Linear Temporal Logic (LTL)

a. A labeled transition system for Sisyphus' task

To define a labeled transition system for Sisyphus' task, we must define the constituent parts of our transition system. Recall that a labeled transition system, or Kripke structure, takes the form:

$$(S, S_0, \delta, L)$$

In order to define our set of states S and initial state S_0, we need to define the variables that make up our state. We'll use four Boolean valued variables to represent our four entities, each with a value of 0 or 1 indicating whether they are on the western bank of the river (start) or the eastern bank of the river (end):

1. `s` — Sisyphus' location
2. `g` — the goat's location
3. `w` — the wolf's location
4. `c` — the cabbage's location

Our initial state S_0 can then be described by the set $\{s = 0, g = 0, w = 0, c = 0\}$. Our set of states S is the set of all Boolean combinations of these states, such that two invariants are not violated:

1. $w = g \land g \neq s$ — the wolf and goat are on the same side and Sisyphus is not on that side.
2. $c = g \land g \neq s$ — the cabbage and goat are on the same side and Sisyphus is not on that side.

In this system, a transition δ involves flipping the values of at most two entities, one of which must be s. This reflects the constraint that the boat only has room for at most two entities, one of which must be Sisyphus himself.

Finally, our labeling function defines all possible

b. LTL formula for Sisyphus' task

To define LTL formula for Sisyphus' task and the constraints that the goat / wolf and goat / cabbage not be left alone on the same bank, we use the same variables as above.

Let us now describe the reasoning for our LTL formula. First, we know that if the cabbage and the goat are on the same side ($c = g$), then Sisyphus must be on that side as well ($c = g \implies c = g = s$). Likewise, if the goat and the wolf are on the same side ($g = w$), then Sisyphus must be on that side as well ($g = w \implies g = w = s$). Notice that the goat, g, appears twice in these constraints. Combining these together, we can arrive at the following LTL formula:

$$G(c = g \lor g = w \implies g = s)$$

Concretely, this formula states that, globally, if the cabbage and the goat are on the same bank or if the goat and the wolf are on the same bank, then Sisyphus *must* be on the same bank as the goat.

c. A solution to Sisyphus' task

Uncovering a solution to Sisyphus' task relies on making one key observation: there is no restriction stating that Sisyphus cannot *return* items to the opposing bank during the transport of all three items from one bank to the other. With this observation, the following procedure solves Sisyphus' task:

1. Assume all items are on the western bank to start — $s = 0, g = 0, w = 0, c = 0$.
2. Bring the goat to the eastern bank — $s = 1, g = 1, w = 0, c = 0$.
3. Return to the western bank — $s = 0, g = 1, w = 0, c = 0$.
4. Bring the wolf to the eastern bank — $s = 1, g = 1, w = 1, c = 0$.
5. Return to the western bank *with the goat* — $s = 0, g = 0, w = 1, c = 0$.
6. Bring the cabbage to the eastern bank — $s = 1, g = 0, w = 1, c = 1$.
7. Return to the eastern bank — $s = 0, g = 0, w = 1, c = 1$.
8. Bring the goat to the western bank — $s = 1, g = 1, w = 1, c = 1$.