EECS 219C: Formal Methods — Assignment 3

Parker Ziegler

March 31, 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 in `Int_SW.ucl`.

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

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

The key change introduced in my updates is ensuring that `turn` is non-deterministically updated in the `init` and `next` blocks of the `main` module 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:

$$(\delta((Sys, Env), (Sys', Env')) = \delta(Sys, Sys') \wedge Env = Env) \vee$$
$$(\delta((Sys, Env), (Sys', Env')) = Sys = Sys \wedge \delta(Env, Env'))$$

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

This composition 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. Since cars are respawned at source locations non-deterministically, we don’t consider two cars at the same source as a ”collision”.

b. Defining the `can_move` procedure

To determine whether or not a car can move, we use an algorithm that gives priority to cars that are currently in the intersection over cars that are at a source location. The algorithm begins by first checking if a car is at a sink location or if it’s currently in the intersection. If the car is at a sink location, we immediately set its `move` flag to *true*, because it must move to (respawn at) a source location in the next step. If the car is in the intersection, we also set its `move` flag to *true*, giving it ”priority” over cars still at source locations.

We then handle cars at source locations. We only allow a car at a source location to move if its next location (computed by calling the `next_location` procedure with a *steps* argument of 1) is not equal to the next locations of the other two cars. In other words, we only allow a car at source locations to move if cars currently in the intersection will not occupy the same location in the next step.

c. Encoding the `bounded_exit` invariant

In order to ensure that each car exits the intersection in a bounded number of steps, we simply check that its `wait_count` does not exceed the total number of steps taken in the bounded model checking process.

```
invariant bounded_exit: wait_cnt1 < 16 && wait_cnt2 < 16 && wait_cnt3 < 16;
```

A car’s wait count is incremented on every transition of the system as long as it hasn’t reached a sink location. Therefore, if a car’s wait count reached the number of steps of the bounded model checking process, this would imply that the car was either stuck at a source location or in the intersection.

d. Induction proof

`uclid` was able to construct an inductive proof using our definition of the `no_collision` invariant and implementation of the `can_move` procedure without any additional strengthening invariants.

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 hold:

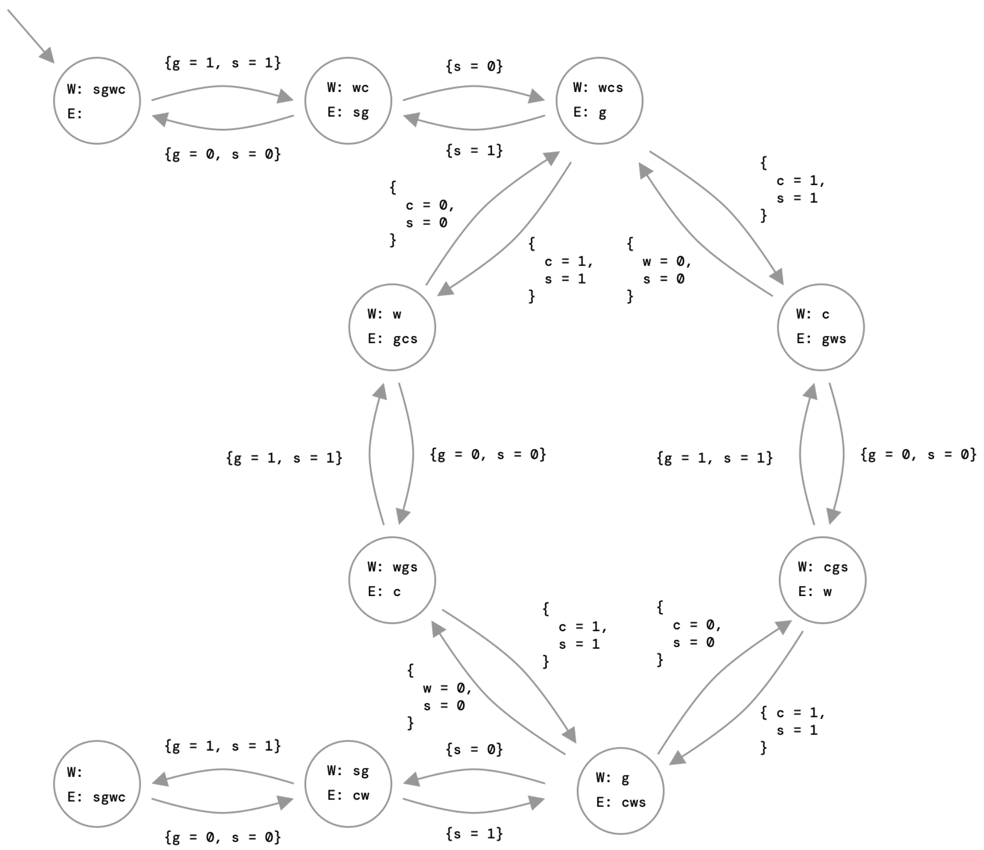

Figure 1: The labeled transition system encoding the river crossing problem.

1. $w = g \implies w = g = s$ — if the wolf and goat are on the same side, then Sisyphus must also be on that side.
2. $c = g \implies c = g = s$ — if the cabbage and goat are on the same side, then 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 transitions of this system. We represent this using a labeled finite state automata in Figure 1 In this automata, we use the notation $W :< s|g|w|c >$ and $E :< s|g|w|c >$ to label the state at each node. For example, the state in which the wolf w is on the western bank and Sisyphus, the goat, and the cabbage are on the eastern bank is notated as $W : w, E : sgc$. We also label the transition relation δ as a set of updates that take place at an edge between two states.

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 defined 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:

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

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.

We could additionally add the formula specifying that, eventually, all entities will be on the opposite side of the river. Since we start with all entities on the western bank ($s = g = w = c = 0$), we want to verify that all entities eventually reach the eastern bank. We can encode this using the future operator F as follows:

$$F(s = g = w = c = 1)$$

Conjuncting all of these together, we arrive at:

$$G(c = g \vee w = g \implies g = s) \wedge F(s = g = w = c = 1)$$

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 western bank — $s = 0, g = 0, w = 1, c = 1$.
8. Bring the goat to the eastern bank — $s = 1, g = 1, w = 1, c = 1$.

d. Creating a Spin model for the river crossing problem

My encoding of a Spin model for the river crossing problem can be found in `river_crossing.pml`.

e. Showing model satisfaction of LTL formula for the river crossing problem

My encoding of LTL formula for the river crossing problem can be found in `river_crossing.pml`.