EECS 219C: Formal Methods **S. A. Seshia, K. Cheang**

Homework 3: Model Checking

Assigned: March 22, 2019 *Due in class: April 10, 2019*

Note: This homework involves experimentation with multiple tools. Please install the tools and start working on the assignment early, as it will take some time for you to learn how to work effectively with each software tool. You may work in teams of up to 3, but clearly state the names of your collaborators. See the webpage for further rules on collaboration. If your uses any material from published papers or elsewhere, please cite the material you use appropriately.

The first two problems involve using the UCLID5 modeling and verification system. You can download cross-platform binaries for UCLID5 from: `https://github.com/uclid-org/uclid/releases`. The github homepage also has a link to a tutorial: `https://github.com/uclid-org/uclid/`.

1. **Interrupt-Driven Program** (Modeling / Model Checking) *(20 points)*

   In this assignment you will complete/correct a model of the interrupt-driven program covered in the lecture on Modeling for Verification (Mar. 4). Recall that the program contained a `main` function and an interrupt service routine `ISR`. In class, we discussed how neither purely synchronous composition nor purely asynchronous composition works for creating a model that faithfully represents timing semantics.

   A unit of concurrency in this model is the `module`. Modules can be synchronously composed (default in UCLID5) or asynchronously composed (you will need to encode this manually). As discussed in class, this program reqires a mix of asynchronous and synchronous composition. In this question, you are tasked with coming up with the right kind of composition and encode it in your UCLID5 model.

   Download the skeleton code from the Homeworks/HW3 folder on bCourses: the file you need is `IntSW.ucl`. This file contains a partial and slightly buggy model of the system that you will need to modify only in a few prescribed places indicated by comments.

   Complete the following steps:

   (i) Read the file `IntSW.ucl`. Read the properties listed in the Sys module and describe each of them in English. Note that the composition of main and ISR within the module Sys is incorectly done (you will need to fix it in subsequent questions).

(ii) Run the file as

```
uclid IntSW.ucl -m Sys
```

Interpret the results from the verifier. If you obtain counterexamples to certain properties, explain why those counterexamples arise.

(iii) Fix the model to correctly compose main and ISR in module Sys and eliminate the above counterexamples. You will only need to change the procedure `update_mode`.

(iv) Now consider the top-level main module. The composition of Sys and Env must be asynchronous composition with interleaving semantics. Ascertain whether this is the case. If not, correct the model to correctly compose Sys and Env. You will only need to change the `init` and `next` blocks by changing the way the variable `turn` is updated.

The property `consec_main_pc_values` should hold on the skeleton code we have provided but may not hold on your corrected asynchronous composition. In your write-up, explain what this property is checking. If it does not hold on the asynchronous composition, explain why.

Note: you need to use the command `uclid IntSW.ucl` (without the "`-m Sys`" for this part of the question since you are analyzing the main module.

2. **Smart Intersection** (Synchronous Model Checking) *(30 points)*

In this problem, you will build a model of autonomous cars navigating the four-way intersection shown in Figure 1. In our simplified model, cars will be "spawned" at the four *source* zones labelled R_{WS}, R_{SE}, R_{EN} and R_{NW}. Their goal is to successfully navigate the intersection to one of the *sink* zones labelled R_{WN}, R_{SW}, R_{ES} and R_{NE}. The intersection itself is modeled as consisting of four zones: I_{SW}, I_{SE}, I_{NE} and I_{NW}.

We will model three cars in this intersection as a synchronous system. Each step of the system will consist of one or more cars moving from one zone to an adjacent zone. We want to ensure that two cars are never in the same zone, and that each car that enters the intersection will exit the intersection within a bounded number of steps. In our model, when the cars reach the one of the sink zones (R_{WN}, R_{SW}, R_{ES} and R_{NE}), they are "respawned" at one of the source zones (R_{WS}, R_{SE}, R_{EN} and R_{NW}).

Download the skeleton code from the Homeworks/HW3 folder on bCourses: the two files you need are `Intersection_BMC.ucl` and `Intersection_Induction.ucl`. These files contains a partial model of the system – cars moving through the intersection are modeled, but "collisions" are not prevented.

(a) In your write-up, provide an assertion stating that none of the cars "collide." State this assertion as the `no_collision` invariant in the UCLID5 model in `Intersection_BMC.ucl`.

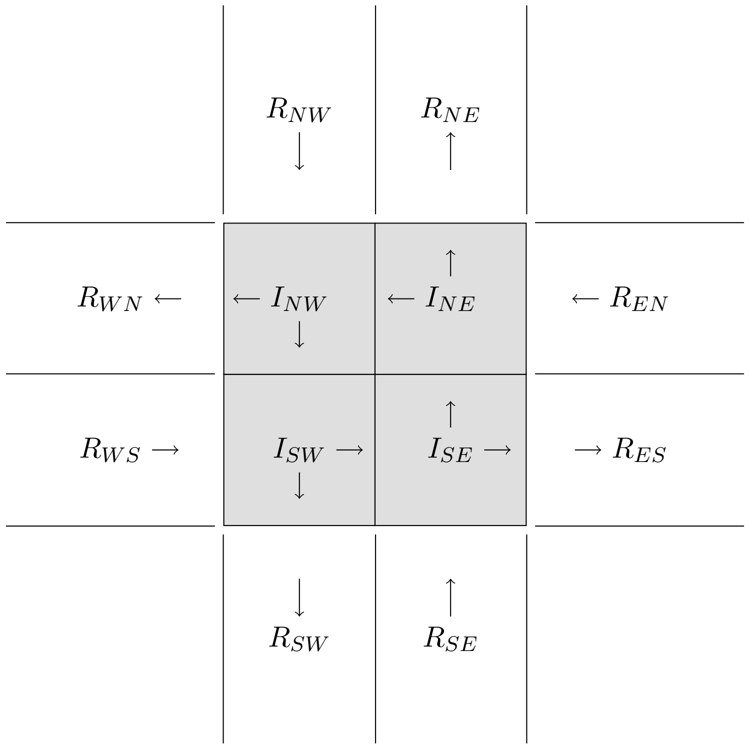

Figure 1: A Four-way Intersection

(b) Define the `can_move` procedure in `Intersection_BMC.ucl`. This function should determine which car(s) are allowed to move during the current transition. Explain your implementation of `can_move` in your write-up.

Use bounded model checking (the `unroll` command in UCLID5) to verify that the `no_collision` invariant is satisfied up to a bound of 16 steps.

(c) Does your model ensure that every car exits the intersection in a bounded number of steps? In your write-up, state a invariant encoding this requirement. State this as the `bounded_exit` invariant in `Intersection_BMC.ucl`. Verify `bounded_exit` using bounded model checking up to a bound of 16 steps.

Hint: Use the `wait_cnt1`, `wait_cnt2` and `wait_cnt3` variables to define this invariant.

(d) Copy over your definitions of the procedure `can_move` and invariant `no_collision` to `Intersection_Induction.ucl`. The proof script in this file attempts an unbounded proof of `no_collision` using induction. However, the invariant by itself may not satisfy an inductive argument. Add "strengthing" invariants that describe reachable states of the system so that the inductive proof succeeds. Describe the invariants you added in your write-up.

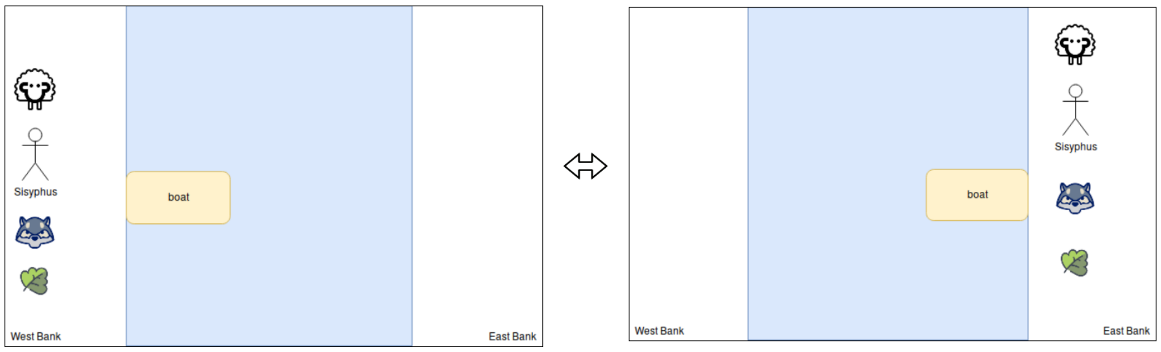

Figure 2: Illustration of Sisyphus' river crossing problem

Hint: Study the counterexamples to induction carefully to determine which invariants to add.

3. **Linear Temporal Logic (LTL)** (Asynchronous Systems) *(40 points)*
In this question, you are asked to model and prove properties about the following river puzzle (`https://en.wikipedia.org/wiki/River_crossing_puzzle`) using the Spin model checker (`http://spinroot.com/spin/whatispin.html`).

   Consider a man (Sisyphus), a goat, a wolf, and a cabbage. Sisyphus is tasked with transporting himself, the goat, wolf, and the cabbage from the east bank of a river to the west bank via a boat. Once all four have crossed, Sisyphus must then transport all four from the west bank to the east bank of the river. This process of moving back and forth between the east and west banks continues indefinitely, see Fig 2.

   The boat only has room for two entities, one of which must be Sisyphus. Further, if the goat is left alone with cabbage, then the goat will eat the cabbage, making it impossible to cross the river with all four. Similarly, if the wolf is left with the goat, the wolf will eat the goat. One can play with an interactive version of this puzzle at the following url: `http://www.mathcats.com/explore/river/crossing.html`

   (a) Provide a labeled transition system that encodes the dynamics of the puzzle.
   (b) Give LTL formula which encode the objective of Sisyphus' task as well as the constraint that the goat/cabbage and wolf/goat cannot be left alone.
   (c) Give a strategy that solves Sisyphus' river crossing puzzle.
   (d) Encode your labeled transition system and strategy from parts (a) and (c) as a Spin model.
   (e) Use Spin to prove that your strategy along with the world model satisfies LTL formula from part (b).

4. OPTIONAL/EXTRA CREDIT: **Binary Search Trees** (Software Model Checking) *(30 points)*

In this question, you will use the Dafny program verification system from Microsoft Research to prove correctness of algorithms that operate on binary search trees.

You can download binary releases of Dafny from here: `https://github.com/Microsoft/dafny/releases`. There is also a web-interface to Dafny at rise4fun: `https://rise4fun.com/dafny/tutorial/guide`; this is also a good introduction and tutorial to Dafny. A number of other excellent resources for learning Dafny are linked from the project's Github homepage: `https://github.com/Microsoft/dafny`.

```
datatype Tree = Leaf | Node(Tree, int, Tree)

function Contains(t : Tree, v : int) : bool
{
    match t
    case Leaf => false
    case Node(left, x, right) =>
        x == v || Contains(left, v) || Contains(right, v)

}
// File continues.
```

Figure 3: Snippet of code from `Tree.dfy`

Download `Tree.dfy` from the Homeworks/HW3 folder on bCourses. The first few lines of this file are shown in Figure 3. The first line shown defines a binary tree data structure: each tree is either a leaf node, which contains no other data or a left and right child along with a integer value. The figure also shows the implementation of the function `Contains`, which returns true if a value `v` is present in the binary tree `t`.

(a) In your written response, formally define the condition for a binary tree to be a binary search tree. Informally stated, for every node in a binary search tree, all values in its left sub-tree must be strictly less than the node's value, and the node value must be less than or equal to all values in the node's right sub-tree.

    Implement the function `Ordered` in `Tree.dfy` which returns true if and only if the binary tree is a binary search tree.

(b) In your written response, define recurrences for (i) a binary tree's size (number of nodes in the tree), (ii) a count of the number of occurrences of a value `v` in the tree `t`. Implement the functions `Size` and `Count` in `Tree.dfy` as per your definitions.

(c) Consider a procedure that insert nodes to a binary search tree. In your written response, state the post-conditions that should be satisfied by a correct implementation of this procedure. Now implement the method `Insert(t, v)` in `Tree.dfy`, state your post-conditions in Dafny and prove them.

(d) Implement the procedure `Min(t)` which returns `Some(v)` if `v` is the minimum value in the binary tree `t`, and `None` if `t` is empty.

(e) Implement the procedure `Remove(t, v)` which removes a node with a value `v` from the binary search tree `t`. If the value `v` does not exist in the tree, the tree is returned unchanged.

In your response, list the post-conditions that a correct implementation of this procedure should satisfy. State these post-conditions in Dafny and prove them.

Hints: You may need to use `Min` in your implementation of `Remove`. Make sure that your post-conditions relate (i) the sizes of the tree and (ii) counts of values that occur in the tree, before and after deletion. You may need to prove and use the auxiliary lemma `Lemma_CountEquivContains` that relates the `Count` and `Contains` functions.