ECE/CS 584: Embedded and cyberphysical system verification	Fall 2019
Dawei Sun, daweis2	*Homework 1: Modeling computation and invariance*
	Due September 17th 11:59pm

Typeset your solutions using LaTeX zip your writeup (.pdf) and code (.py) in a single file called `nedid-584-F19.zip` *and upload this file through Compass.*

Problem 1. Bidirectional array (20 points). Model the following token-based mutual exclusion algorithm that works on a *bidirectional array*. There are N processes $\{0, \ldots, N-1\}$ in an array. Each process i, has a single variable $s[i]$ that takes values in the set $\{0, 1, 2, 3\}$ independent of the size of the array. The two processes 0 and $N-1$ behave differently from the rest, they can take two values each $s[0]$ can take values $\{1, 3\}$ and $s[N-1]$ can take values $\{0, 2\}$. Let $Nbrs(i)$ be the set of neighboring processes for process i.

1. Program for processes, i, $i = 0$ or $i = N-1$:
   if $\exists\, j \in Nbrs(i)$:$s[j] = s[i] + 1 \mod 4$ then $s[i] = s[i] + 2 \mod 4$
2. Program for processes, i, $0 < i, N-1$:
   if $\exists\, j \in Nbrs(i)$:$s[j] = s[i] + 1 \mod 4$ then $s[i] = s[i] + 1 \mod 4$

In this protocol, process i *has token* iff

$$\exists j \in Nbrs(i) : s[j] = s[i] + 1 \mod 4.$$

(a) Write the model of the bidirectional array system with N processes using the language we saw in class.

(b) Write an execution of the algorithm that starts from a state with a single token. Mark the process with the token.

(c) Write an execution (of length at least 6) that starts from a state with multiple tokens.

(d) Prove the invariant "system has a single token" using the inductive invariance theorem.

Solution (a)

---

automaton BidirectionalArray(N:Nat)
  type ID:**enumeration** [0,· · ·,N-1]
  type Val:**enumeration** [0,1,2,3]
  actions
    update(i:ID)
  Variables
    s:[ID→Value]

transitions
update(i:ID)
pre $i = 0 \wedge s[i+1] = (s[i]+1)\%4$
eff s[i] := (s[i]+2) % 4
update(i:ID)
pre $i = N-1 \wedge s[i-1] = (s[i]+1)\%4$
eff s[i] := (s[i]+2) % 4
update(i:ID)
pre $i > 0 \wedge i < N-1 \wedge (s[i-1] = (s[i]+1)\%4 \vee s[i+1] = (s[i]+1)\%4)$
eff s[i] := (s[i]+1) % 4

(b) Let $N = 5$. For every step, the state are represented by a number like $1333\hat{2}$, which means $s[0] = 1, s[1] = s[2] = s[3] = 3, s[4] = 2$ and process 4 has the token. An execution that starts from a state with a single token: $1333\hat{2} \rightarrow 133\hat{3}0 \rightarrow 13\hat{3}00 \rightarrow 1\hat{3}000 \rightarrow 1\hat{0}000 \rightarrow 11\hat{0}00 \rightarrow 111\hat{0}0 \rightarrow 1111\hat{0} \rightarrow 111\hat{1}2 \rightarrow \ldots$.

(c) Let $N = 5$. An execution that starts from a state with multiple tokens: $1\hat{0}1\hat{0}0 \rightarrow 111\hat{0}0 \rightarrow 1111\hat{0} \rightarrow 111\hat{1}2 \rightarrow 11\hat{1}22 \rightarrow 1\hat{1}222 \rightarrow \hat{1}2222 \rightarrow \ldots$.

(d) It is easy to verify that if the variables of two neighboring processes has different parity, then at least one of them has the token. Because process 0 must take an odd variable and process $N-1$ must take an even one, we have the following consequence: the state has only one token $\iff$ there exists a parity-separating process $p(0 \leq p < N-1)$ so that $\forall j \leq p, s[j] \in 1, 3$ and $\forall j > p, s[j] \in 0, 2$. Next we prove that if v_k has only one token, then v_{k+1} also has only one token. For a state with only one token and the parity-separating ID p, process p or $p+1$ has the token. There are 4 possible situations. 1) $p = 0$. If process 0 has the token, then in the next state there exists an parity-separating ID $p' = 0$. If process 1 has the token, then $p' = 1$. 2) $p = N-2$. If process $N-2$ has the token, then $p' = N-3$. If process $N-1$ has the token, then $p' = N-2$. 3) $1 < p < N-2$. If process p has the token, then $p' = p-1$. If process $p+1$ has the token, then $p' = p+1$. Therefore, in all situations the next state has a parity-separating ID which means this state has only one token. With the inductive invariance theorem, we proved that "system has only one token" is an invariant.

Problem 2. LCR Leader election (20 points). In this problem, you will create a model of a leader election algorithm in a unidirectional ring [2]. Here is the informal description of the protocol:

Each process sends its identifier to its successor around the ring. When a process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming identifier; if it is equal to its own the process declares itself as the leader.

(a) Write the model of the system with n processes in the ring using the language we saw in class. To get you started, the set of variables is:

- $send$: The identifier to send or *null*,
- $status$: Takes values in $\{unknown, leader\}$ to indicate that the leader has been elected or not.

(b) Write an execution of the system in which status of at least one process is eventually set to $leader$.

(c) Write two candidate invariants.

Solution (a) In order to avoid the message lost problem, the algorithm should be synchronous. Also, as described in [2], every process has an unique ID (UID).

---

automaton LCRLeaderElection(N:Nat)
  type indices:**enumeration** [0,...,N-1] // *index for every process*
  type UID:**enumeration** [N different IDs] // *every process has an unique ID*
  type MSG:**enumeration** UID ∪ [null] // *possible values in* ***send[i]***
  type STA:**enumeration** [unknown, leader] // *possible status values*
  actions
    step
  Variables
    uid:[indices→UID]
    send:[indices→MSG] **initially** send[i] = uid[i], $\forall i \in indices$
    send_old:[indices→MSG]
    status:[indices→STA] **initially** status[i] = unknown, $\forall i \in indices$
  transitions
    step
      pre True
      eff send_old := send
        for i=0 **to** N-1:
          if (send_old[(i-1) mod N] > uid[i]) or (send_old[(i-1) mod N] == null):
          then send[i] := send_old[(i-1) mod N]
          if send_old[(i-1) mod N] < uid[i]:
          then send[i] := null
          if send_old[(i-1) mod N] == uid[i]:
          then status[i] := leader; send[i] = null

(b) Let $N = 5, uid = [1, 3, 2, 4, 5]$, "u" is short for "unkonwn" and "n" is short for "null". $send = [1, 3, 2, 4, 5], status = [u, u, u, u, u] \rightarrow send = [5, n, 3, n, n], status = [u, u, u, u, u] \rightarrow send = [n, 5, n, n, n], status = [u, u, u, u, u] \rightarrow send = [n, n, 5, n, n], status = [u, u, u, u, u] \rightarrow send = [n, n, n, 5, n], status = [u, u, u, u, u] \rightarrow send = [n, n, n, n, n], status = [u, u, u, u,$ **leader**]

(c) $I_1 = whole\ state\ space.\ I_2 = [\![(\exists i,\ send[i] == the\ largest\ UID\ uid_{max}) \vee (\exists i,\ status[i] == leader)]\!]$.

Problem 3. Impossibility of election (20 points). A *symmetric function* $f : S^k \rightarrow S$ has the property that for all $s_1, s_2 \in S^k$, if s_1 is a permutation of s_2 then $f(s_1) = f(s_2)$. That is, for $k = 3$, $f(\langle 1, 2, 3\rangle) = f(\langle 2, 3, 1\rangle) = f(\langle 3, 1, 2\rangle)$, etc.

Consider a synchronous algorithm $SymGk$ running on a graph G with *indegree* k. That is, each process P_j reads the states of exactly k other processes. In this algorithm. $SymGk$, every node updates its state x_i according to some symmetric function $f : S^k \to S$, that is,

$$x[i] := f(\langle x[j] \mid (j,i) \in G)$$

Show that it is impossible for $SymGk$ to elect a leader if initially every process has the same value of $x[i]$. *[Hint: State an invariant and prove it by induction on rounds.]*

Solution Let Θ be the set of all states where every process has the same value. Next, we prove that Θ is an invariant for initial states set Θ. First, every initial state is in the invariant. Then, for any execution $\alpha = v_0\, step\, v_1\, step\, v_2 step\, v_3 \ldots$. If v_k is in the invariant, then $v_k\lceil x[i] = C, \forall i$, where C a constant. Therefore, we have $v_{k+1}\lceil x[i] = f(\langle v_k\lceil x[j] \mid (j,i) \in G) = f(C, C, \ldots, C) = C', \forall i$, which means $v_{k+1} \in \Theta$ still holds. Therefore Θ is an invariant. If initially every process has the same value, they will always have the same value which means a leader cannot be elected.

Problem 4. Dijkstra invariance with Z3 (40 points). Use the Z3 SMT solver to encode and check that for the Dijkstra's token ring mutual exclusion algorithm ($DijkstraTR$ of Chapter 2) the predicate ϕ_{legal} is an inductive invariant.

First install z3 in your computer. This is a very quick process. On the MacOS `pip install z3-solver` does it in less than a minute. For Windows and Linux you can get it from: https://github.com/Z3Prover/z3. There is a lot of help available online for installation related issues.

Next, download the file `DijkstraTRind.py` given for this homework and read it carefully. There are several function given and you have to complete several other functions. The documentation given with the program, the lecture slides, and the discussions in the book chapter 2 and 7 should be adequate for you to complete this problem. Here are some additional notes.

- `has_token(x_list, j)`: Generates a Z3 Boolean expression (predicate) that represents whether process P_j holds the token. Here `x_list` is a list of Z3 variables, for example, `[x[0], x[1], x[2], x[3]]`, and `j` is the index of process P_j.
- `legal_config(x_list)`: Generates a Z3 Boolean expression that represents whether the system is in a legal configuration. This is the implementation of ϕ_{legal}. This function is given to and **you do not have to change it**.
- `transition_relation(old_x_list, new_x_list)`: Generates a Z3 Boolean expression representing transition from `old_x_list` to `new_x_list`.
- `prove(conjecture)`: Checks whether the Boolean predicate `conjecture` is valid using the Z3 solver to check the *unsatisfiability* of the negation of `conjecture`. **You do not have to change this.**

Run your program with `python DijkstraTRind.py`. If the functions are written correctly, then validity of the base case and the induction case imply (by the induction invariance Theorem) that `legal_config` is indeed an invariant.

Class contribution and project ideas

- Verify the self-stabilization property of Dijkstra's algorithm and other self-stabilizing algorithms from Chapter 17 of [1]. Note self-stabilization is not an invariant property and requires a different proof technique with ranking functions that we will discuss later in this course.
- Verify inductive invariants of other distributed algorithms, such as the bidirectional array in Problem 1, following the pattern of Problem 4. Several interesting examples appear in Chapter 17 of [1].

References

[1] Sukumar Ghosh. *Distributed systems: an algorithmic approach*. Chapman and Hall/CRC, 2014.

[2] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.