Verification Lab Report

1. Vending Machine Design

I implemented the design using the given specifications with the following additions:

- on reset, the vending machine credit is set to 0
- the user can only either insert a coin, or request a beverage (if user tries both, coin insertion is registered and beverage request is ignored)

I chose to represent the signals coin_in, button_in, change_out and beverage_out as integer types. I used an internal integer variable to keep track of the user's credit. Change is given out after output of a drink only if credit is smaller than 30 (no further beverage can be bought). This is implemented by a status variable that indicates if a beverage was retrieved, and therefore the user may receive his change. The delay (for change or beverage retrieval) is implemented by an integer counter variable that is set to the desired delay value (M or N) when the action is triggered, and causes "stall" cycles while the counter is decremented until it reaches 0.

I implemented the vending machine design in the file rtl/vending.sv, with the interface of it being defined in rtl/vendingInterface.sv. Following the example we saw for the counter and FSM design, I implemented the verification using the generator and driver architecture (see tb/). For the test pattern generation I constrained my coin_in input to be one of {0, 10, 20, 50, 100, 200} and my button_in input to be one of {0, 1, 2}. Whenever the coin_in is not 0, the machine updates the credit and is not able to do anything else. Having coin_in distributed uniformly, the generated test patterns will cover mostly credit incrementing, as there is a probability of ⅚ that my coin_in is not 0. To avoid this, I chose to constrain my coin_in with a weighted distribution to improve the coverage of the other code paths.

For the formalization of the specifications, please refer to the assertion definition part.

2. Assertion Definition

Assertions are defined in assertions/ass.sv.

My property p0_insert_coin_increment_credit checks if the credit got increased after a coin was inserted and the machine was not blocked.
My property p1_retrieve_beverage_decrease_credit checks if the credit got decreased after a beverage was retrieved.
My property p2_retrieve_beverage_water_set_output checks if the output is set correctly when a beverage is retrieved.
My property p3_ignore_request_water_credit_insufficient checks if the beverage request is correctly ignored in case the credit is not sufficiently high.
My property p4_retrieve_beverage_block_machine checks if the machine is blocked after a beverage was retrieved.

3. Assertion Mining

I used HARM to mine assertions for my design. The image shows the results. Sadly I was only able to use my primary inputs/outputs as antecedent/consequent candidates, referencing my internal variables resulted in errors. (I tried using the correct path and checked my vcd file but it did not work. I even explicitly had to remove any path prefix before my PIs/POs, otherwise I would have gotten an error message as well.)

I got the following results:

N	Assertion (Context : default)	final	causality	frequency
0	G((button_in) ##1 (button_in) |-> (beverage_out))	1.00	1.00	1.00
1	G((!coin_in && button_in) ##1 (!coin_in) |-> (beverage_out))	0.44	0.58	0.41
2	G((coin_in) ##1 (!coin_in && button_in) |-> (beverage_out))	0.05	0.53	0.27
3	G((button_in && coin_in) ##1 (coin_in) |-> (beverage_out))	0.01	0.47	0.18

The second and third assertion demonstrates the desired behavior of the machine (how it should react on a button press while no coin is being inserted). However, if the assertion miner could have used information about my internal state variables, I think it would have been able to produce a lot more valuable assertions.