Modelling a Queue

Problem 1. A model of a queue. In this assignment, you will write νSMV code to model a queue and verify some properties of your model. We will model a queue that is implemented as a circular buffer using a head pointer and a tail pointer.

A queue is a "first in, first out" (FIFO) data structure supporting `push` and `pop` operations. We can think of the queue as having a head and a tail. We `push` values onto the tail of the queue and `pop` values from the head of the queue. When either the head or the tail pointer reach the end of the buffer, they "wrap around" again to the beginning of the queue.

First, download `queue-template.smv` from the course Piazza page (also included on the last page of this assignment). Implement a νSMV model of the queue as described here and in class. The provided template should run in νSMV and indicate that the example LTL property is false. This is because the functionality of the queue is not yet implemented by you. After implementing the queue model, this property, and some additional properties that you will define, should be true.

Your model should be implemented inside `MODULE queue` only.

The provided template defines the `MODULE main` which you should not change. The user has an `action` variable which can be either `push` or `pop` and variables `push_val` and `pop_val` to represent the data that is posted or popped. Notice that in this model, we enforce in the `main MODULE` that the user never tries to `pop` when the queue is empty or `push` when the queue is full. Otherwise, the user non-deterministically chooses between `pop` or `push`.

You will see that there are two preproccessor `define` directives—one for `SIZE` and one for `DATA`. This means you will need to run `NuSMV` with the `-pre cpp` flags. `DATA` is used to represent the possible data values that can be sored in the queue, specified as a range: `min..max`. For the time being, we will consider only two values, `1` and `2`. Note that in this assignment, we do not have a special `NULL` value as we did in the stack assignment.

There are four variables in the `VAR` block: `buff`, an array that will be used as the circular buffer to store data, `head`, which is an integer index to the head of the queue, initially 0, `tail`, which is an integer index to the tail of the queue, initially 0, and `empty`, which is a boolean value indicating if the queue is empty, initially true. Additionally, there is a variable `full` in the `DEFINE` block which will be used to keep track of whether the queue is full. In the template, `full` is set to `FALSE` so that the code will run without syntax errors. The

`ASSIGN` block has nothing implemented.

You job for this part of the assignment is to fill in the definition of `full` and to encode the initial states and transition relation for `head`, `tail`, `empty`, and `buff` within the `ASSIGN` block.

You should use the following guidelines when implementing your model:

i. The Boolean value `full` can be defined as a logical equivalence to a formula involving `empty`, `head`, and `tail`.

ii. The value of `empty` should be initially true, should be false after any push action, and should become true in the next state any time that the action is pop and the tail pointer is ahead of the head pointer by one position (modulo the size of the buffer).

iii. The `head` pointer, initially 0, should only move when the user chooses a pop action and the queue is not empty, in which case the head pointer is incremented by 1 (modulo the size of the buffer).

iv. The `tail` pointer, initially 0, should only move when the action is push and the queue is not full, in which case the tail pointer should increment by 1 (modulo the size of the buffer).

v. The value of `pop_val` should update to the current value stored in the buffer at the location of the `head` pointer.

vi. The buffer, `buff` should update whenever the action is push and the queue is not full, in which case the buffer value at the location pointed to by the tail pointer should update to the current value of `push_val`

Problem 2. Verifying properties of your queue model. In this part of the assignment, you will write specifications for several queue properties and verify that the model you implemented in the previous part adheres to the specifications. Consequently, you may need to make changes to your model if it does not satisfy all of the desired properties.

For each of the following properties, write a corresponding specification in your file and confirm that νSMV verifies the property. Some of them are similar to the stack properties you wrote previously, but take note that a stack is a LIFO data structure and a queue is a FIFO data structure.

i. LTL property: It is always the case that if the queue is empty then the tail pointer is equal to the head pointer. (This one is done for you.)

ii. LTL property: The queue is never simultaneously full and empty.

iii. CTL property: In all possible states, it is always the case that if the queue happens to be full, then it is possible that the queue eventually becomes empty.

iv. LTL property: If the user never pops then the queue eventually fills up.

v. CTL property: It is always the case that if 1 is pushed on the queue and the queue is not full, then in all possible paths future paths there exists an execution path in which a 1 is popped.

vi. LTL property: It is always the case that that if a 1 is pushed onto the queue when the queue is not full, then starting from the next state, the queue cannot be empty until the user has popped a 1.

vii. LTL property: If the user pushes a 1 followed directly by a 2, and the queue is not full during either of those push actions, then if the user immediately pops, the resulting value will be 1 and if the user immediately pops again, the resulting value will be 2. That is, items come out of the queue in FIFO order.

Problem 3. Scalability of verification.. In this part of the assignment, you will explore the state-space explosion problem.

i. After you have successfully verified the above properties, use `NuSMV` in interactive mode to determine the number of reachable states of the transition system.

ii. Use the Linux command `time` to measure the amount of `real` time needed to run `NuSMV` on your file.

```
> time NuSMV -pre cpp queue.smv
```

Note: before running the `time` command, you may want to run

```
>  TIMEFMT=$'\nreal\t%E'
```

iii. Now, observe that in the template, `DATA` is defined to be a set of only two values. Try changing the number of allowable data values to a larger number, say 10. Again, use `NuSMV` to compute the number of reachable states and also measure the amount of time is takes `NuSMV` to run.

iv. For each possible number of data values from 1 to 20, create either a table, or a plot of the number of possible data values vs the number of reachable states. (If you make a plot, you might want to make it logarithmically scaled on the y-axis.)

v. For each possible number of data values from 1 to 20, create either a table, or a plot of the number of possible data values (on the x-axis) vs the running time of `NuSMV`. (If you make a plot, you might want to make logarithmically scaled on the y-axis.)

```
#define SIZE 5
#define DATA 1..2

MODULE queue(action, push_val, pop_val)
  VAR
    buff  : array 0 .. (SIZE - 1) of DATA;
    head  : 0 .. SIZE - 1;
    tail  : 0 .. SIZE - 1;
    empty : boolean;
  DEFINE
    -- specify a logical equivalence that defines full
    -- in terms of empty, head, and tail.
    -- That is, replace FALSE with an appropriate expression
    full := FALSE;
  ASSIGN
    -- initially head and tail should be 0
    -- initially the queue should be empty
    -- initially the buffer could be anything
    -- specify the next state for head
    -- specify the next state for tail
    -- specify the next state for empty
    -- specify the next state for pop_val
    -- specify the next state for buff[i] for i from 0 to SIZE - 1

MODULE main
  -- do not change code in the main module
  VAR
    action    : {push, pop};
    pop_val   : DATA;
    push_val  : DATA;
    Q         : queue(action, push_val, pop_val);
  ASSIGN
    init(action) := push;
    next(action) :=
      case
        Q.empty : push;
        Q.full  : pop;
        TRUE    : {push, pop};
      esac;

-- It is always the case that if the queue is empty then
-- the tail pointer is equal to the head pointer
LTLSPEC G (Q.empty -> (Q.tail = Q.head))
```