Paul Beglin

Formal Verification

VER Coursework: Modulo 8 and 16 counter

**First Part:** Modulo 8 counter (with easy expansion to 16) with a free variable dictating the state transitions.  
We want to go from 000 🡪 001 🡪 010 🡪 … 🡪 111 🡪 000. Let’s name the bits ‘x’, ‘y’ and ‘z’; in this order, such that ‘z’ is the smallest bit and ‘x’ is the largest. For example, for “011” we have x=0, y=1, z=1; or another way: “!x & y & z”.  
At first we can write the states individually (like in the second method shown). Giving:  
!x & !y & !z 🡪 !x & !y & z 🡪 !x & y & !z 🡪 !x & y & z 🡪 … 🡪 x & y & z 🡪 !x & !y & !z  
But if we use our knowledge from Logic Design & FC2/VER, we can simplify these states, instead of having to write them all out (and making it complex to add a 4th bit).

We can see that whenever we change states, z changes as well (!z 🡪 z 🡪 !z 🡪 z …) and the other variables stay the same. Therefore, one of the rules is: !z: z:=true.

Similarly, we can find other rules to optimize our code, here is a table showing (using colors) which optimized rules deal with which ‘intuitive’ cases (the rules of every state transition).

Table of my findings on the optimized rules:

|  |  |
| --- | --- |
| Rules (every state transition) | Optimized Rules |
| !x & !y & !z : !x & !y & z  !x & !y & z : !x & y & !z  !x & y & !z : !x & y & z  !x & y & z : x & !y & !z  x & !y & !z : x & !y & z  x & !y & z : x & y & !z  x & y & !z : x & y & z  x & y & z : !x & !y & !z | !z : z:= true  !y & z : z:=false; y:=true  !x & y & z : x & !y & !z  x & y & z : !x & !y & !z |

And here is the code written with the free variable in mind.  
  
NAME Modulo8FreeVariable

VAR

x : boolean;

y : boolean;

z : boolean;

V : boolean;

INIT

!x & !y & !z;

RULES

!z & V :

z := true

!z & !V :

z := false

z & !y & V:

z := false; y := true

z & !y & !V:

z := true; y := false

z & y & !x & V:

z := false; y := false; x := true

z & y & !x & !V:

z := true; y := true; x := false

z & y & x & V:

z := false; y := false; x := false

z & y & x & !V:

z := true; y := true; x := true

We essentially duplicate the “Optimized Rules” so that we have a state transition if the free variable ‘V’ is true (1); otherwise, if ‘V’ is false (0) we keep the same state.

Model Structure Diagram (with !x&!y&!z syntax)

Graphical user interface, timeline

Description automatically generated

Kripke structure:  
Let’s define the Kripke structure: M = (S, S0, R, L)

S : set of all states  
S0 : set of initial states  
R : transition relation between states

L : labelling function that associates each state with the set of propositions true in that state

Graphical user interface, application

Description automatically generated

S = {state0, state1, state2, state3, state4, state5, state6, state7}  
S0 = {state0}

R = {(state0, state1), (state1, state2), (state2, state3), (state3, state4), (state4, state5), (state5, state6), (state6, state7), (state7, state0)}  
L = {(state0, {}), (state1, {z}), (state2, {y}), (state3, {y, z}), (state4, {x}), (state5, {x, z}), (state6, {x, y}), (state7, {x, y, z})}

**Second Part:** Write the CTL properties to check correctness of your model.  
I started by writing out the 8 state transitions of a normal modulo 8 counter (without a free variable), hoping it would work despite the free variable; for example:   
AG ((!x & !y & !z) -> AX(!x & !y & z)). The problem is that this means “For every reachable state (!x & !y &!z), for all paths starting at that state, the next state must be (!x & !y & z)” but that’s not true in our case; because one path with V (the free variable being True) will indeed go to the next state, but another path with !V will remain on the same state…

We could write the same things as the rules we used, for example: AG ((!z & V) -> AX(z & V)), and then with !V etc. But it wouldn’t test that our model works, it would just test that we wrote something correctly (without bugs) and that xCheck works… It doesn’t make sure that we indeed built a good counter (if for example the optimized rules aren’t correct, then this test wouldn’t tell us, since we are testing the rules against themselves…).

We could also write out all 16 possibilities of transition (the same 8 with V and another 8 with !V), but I’m sure there is a smarter way to go about this (if I can’t find it I will write the 16 CTL properties).

I’ve been reticent to use the quantifiers “U” for until, and “AF” for eventually, because as I can see it currently, it could work because the state transition does happen eventually, but it might be when V is False for example, and that would be problematic…   
So we need to make sure that the state transition only happens in the correct conditions.

In the end I think I’ll write all 16 transition cases because it took me too much time already and I don’t want to write the same 8 properties as the rules because I believe testing against your own rules (even though we know they are true as shown in the table) doesn’t make a lot of sense. So here are all 16 properties (every single transition possible):

With V (changes to next state)  
1. AG ((!x & !y & !z & V) -> AX(!x & !y & z))

2. AG ((!x & !y & z & V) -> AX(!x & y & !z))

3. AG ((!x & y & !z & V) -> AX(!x & y & z))

4. AG ((!x & y & z & V) -> AX(x & !y & !z))

5. AG ((x & !y & !z & V) -> AX(x & !y & z))

6. AG ((x & !y & z & V) -> AX(x & y & !z))

7. AG ((x & y & !z & V) -> AX(x & y & z))

8. AG ((x & y & z & V) -> AX(!x & !y & !z))

With !V (state stays the same)

9. AG ((!x & !y & !z & !V) -> AX(!x & !y & !z))

10. AG ((!x & !y & z & !V) -> AX(!x & !y & z))

11. AG ((!x & y & !z & !V) -> AX(!x & y & !z))

12. AG ((!x & y & z & !V) -> AX(!x & y & z))

13. AG ((x & !y & !z & !V) -> AX(x & !y & !z))

14. AG ((x & !y & z & !V) -> AX(x & !y & z))

15. AG ((x & y & !z & !V) -> AX(x & y & !z))

16. AG ((x & y & z & !V) -> AX(x & y & z))

My model satisfies all those properties. Therefore, it is a modulo 8 counter that counts the number of times the free variable had the value 1.

These properties represent the intuitive specification of the counter because they are literally every single possibility (as shown on the model structure diagram) every state transition if ‘V’ is true and every loop into the same state when it is ‘!V’.

We can analyze one specifically to prove that. For example ‘000’ 🡪 ‘001’:

1. AG ((!x & !y & !z & V) -> AX(!x & !y & z))  
As we wrote previously, this means: “For every reachable state (!x & !y &!z & V, 000 in binary), for all paths starting at that state, the next state must be (!x & !y & z, 001 in binary)”. Therefore, we know that ‘000’ 🡪 ‘001’.   
Similarly with the other properties, we have ‘001’ 🡪 ‘010’; and ‘010’ 🡪 ‘011’.   
In addition, when ‘V’ is false, we have: 9. AG ((!x & !y & !z & !V) -> AX(!x & !y & !z)) which shows that the state remains the same (‘000’ in this case).  
Finally, we also have the final property that resets the state when reaching a value of 7, which is what makes it a modulo: 8. AG ((x & y & z & V) -> AX(!x & !y & !z))

Therefore, we have proven that we built a counter that goes from 0 to 7, counting the number of times the free variable ‘V’ obtained the value 1.

All of the properties here are safety properties, because they guarantee that “something wrong will never happen”. They were created using the definition of what a counter should do. There is no “eventuality” so there are no liveness properties. As shown in the lecture notes, a counter-example for a safety property is a finite path showing violation of the property.

**Third Part:** Show that one of your properties passes vacuously in the system.  
A vacuous truth is a statement that is only true because the antecedent cannot be satisfied; since we know that False 🡪 anything is True…   
None of my properties pass vacuously in the system, but we can create one that does.

For example, if we were to unfortunately forget to put a pair of parentheses in the first property, we would get: AG (!x & !y & !z & V) -> AX(!x & !y & z). This translates to “If every reachable state is (!x & !y & !z & V), then for all paths starting at the initial state, the immediate successor is (!x & !y & z)”. We know that (!x & !y & !z & V) is not the only state in our model, so the antecedent is False; therefore, this property is vacuously true.   
It’s especially bad if we don’t realize it because then we would think our system is correct when it isn’t; what we meant to test was that given a certain state, the only possible transition is another specific state; but here it wouldn’t have helped us.

**4.** Introduce a bug in the model that isn’t caught by any of the properties.  
  
NAME Modulo8FreeVariableBUG

VAR

x : boolean;

y : boolean;

z : boolean;

V : boolean;

INIT

x & y & z;

RULES

!z & V :

z := true

!z & !V :

z := false

z & !y & V:

z := false; y := true

z & !y & !V:

z := true; y := false

z & y & !x & V:

z := false; y := false; x := true

z & y & !x & !V:

z := true; y := true; x := false

z & y & x & V:

z := false; y := false; x := false

z & y & x & !V:

z := true; y := true; x := true

Here is the Kripke structure for this model (with a BUG):

S = {state0, state1, state2, state3, state4, state5, state6, state7}  
S0 = {state7} (this is the difference with the previous model)

R = {(state0, state1), (state1, state2), (state2, state3), (state3, state4), (state4, state5), (state5, state6), (state6, state7), (state7, state0)}  
L = {(state0, {}), (state1, {z}), (state2, {y}), (state3, {y, z}), (state4, {x}), (state5, {x, z}), (state6, {x, y}), (state7, {x, y, z})}

Here is the Kripke structure for this model (with a BUG):

Graphical user interface, application

Description automatically generated

**5.** Explain why the bug happened, write an additional property that exposes this bug.  
Without changing the rules, we can make the model fail to count correctly the number of times the free variable had a value of 1; simply by initializing the values of xyz in such a way that the first state we start with is not the initial state0 as shown in the Model Structure Diagram, but another state, like the state7 for example. This means that when V has the value 1, the counter will reset and go from 7 to 0, thinking that V never had the value 1…

Maybe we can write a property that checks that the initial state is state0 with values (!x & !y & !z). This works! So let’s add a 17th property:  
17. (!x & !y & !z). This checks that the initial state is the correct one (state0).  
A counter-example is the BUG case for example, or any other initialization that isn’t the 000 case (!x & !y & !z). The BUG model wouldn’t satisfy the 17th property because its initial state is (x & y & z); in so doing, we would know that the counter doesn’t work correctly.

(Because of the indications in the Second Part, I’ll specify for this property as well that it is a Safety property, since it guarantees that “something wrong will never happen”.)

**6.** Does your initial model satisfy this new property, if not explain why and fix the model.  
My initial model does indeed start at the correct state (state0 in the Model Structure Diagram); with the values xyz being initialized as (!x & !y & !z). So it does satisfy the 17th property.

**7.** Expand model to count modulo 16.

I’m not drawing the 16 states of the Kripke structure (8 states makes the text very tiny already); but we can imagine that it’s basically the same thing as the first Kripke structure shown for the modulo8 counter, where instead of looping back at state7 it continues counting and loops back at state15 to go to state0:  
  
S = {state0, state1, state2, state3, state4, state5, state6, state7, state8, state9, state10, state11, state12, state13, state14, state15}  
S0 = {state0}

R = {(state0, state1), (state1, state2), (state2, state3), (state3, state4), (state4, state5), (state5, state6), (state6, state7), (state7, state8), (state8, state9), (state9, state10), (state10, state11), (state11, state12), (state12, state13), (state13, state14), (state14, state15), (state15, state0)}  
L = {(state0, {}), (state1, {z}), (state2, {y}), (state3, {y, z}), (state4, {x}), (state5, {x, z}), (state6, {x, y}), (state7, {x, y, z}), (state8, {w}), (state9, {w, z}), etc. }

Here is a table of the 16 basic rules necessary for modulo 16, and the optimized rules:

|  |  |
| --- | --- |
| Rules (every state transition, modulo 16) | Optimized Rules (modulo 16) |
| !w & !x & !y & !z : !w & !x & !y & z  !w & !x & !y & z : !w & !x & y & !z  !w & !x & y & !z : !w & !x & y & z  !w & !x & y & z : !w & x & !y & !z  !w & x & !y & !z : !w & x & !y & z  !w & x & !y & z : !w & x & y & !z  !w & x & y & !z : !w & x & y & z  !w & x & y & z : w & !x & !y & !z  w & !x & !y & !z : w & !x & !y & z  w & !x & !y & z : w & !x & y & !z  w & !x & y & !z : w & !x & y & z  w & !x & y & z : w & x & !y & !z  w & x & !y & !z : w & x & !y & z  w & x & !y & z : w & x & y & !z  w & x & y & !z : w & x & y & z  w & x & y & z : !w & !x & !y & !z | !z : z:= true  !y & z : z:=false; y:=true  !x & y & z : x & !y & !z  !w & x & y & z : w & !x & !y & !z  w & x & y & z : !w & !x & !y & !z |

Changes from modulo 8 to modulo 16:  
Changed {x & y & z : !x & !y & !z} to {!w & x & y & z : w & !x & !y & !z} to take into account !w 🡪 w.   
Added { w & x & y & z : !w & !x & !y & !z } for the reset (what makes it a modulo…)

NAME Modulo16FreeVariable

VAR

w : boolean;

x : boolean;

y : boolean;

z : boolean;

V : boolean;

INIT

!w & !x & !y & !z;

RULES

!z & V :

z := true

!z & !V :

z := false

z & !y & V:

z := false; y := true

z & !y & !V:

z := true; y := false

z & y & !x & V:

z := false; y := false; x := true

z & y & !x & !V:

z := true; y := true; x := false

z & y & x & !w & V:

z := false; y := false; x := false; w := true

z & y & x & !w & !V:

z := true; y := true; x := true; w:= false

z & y & x & w & V:

z := false; y := false; x := false; w := false

z & y & x & w & !V:

z := true; y := true; x := true; w:= true

**8.** Which properties pass/fail on modulo 16 counter. If none fail, introduce a property that behaves differently based on modulo 8 or 16.

All of the properties (1 to 17) are true (because ‘w’ stays the same).   
So every property is true because ‘w’ stays the same in both cases of each property (once when ‘!w’ and once when ‘w’).   
To create a property that distinguishes between modulo8 and modulo16, we could use the fact that there is 1 more bit (w) in modulo16. We could change the code of modulo8 and initialize a ‘w’ that doesn’t serve any purpose (just give it a constant value) and then use a property that works on modulo16 but doesn’t in modulo8.   
I have an idea, let’s try to use this:

!w & x & y & z : w & !x & !y & !z

w & x & y & z : !w & !x & !y & !z  
Here we can see that ‘w’ changes value. The properties that we used previously do not test this, so technically the model could not work correctly because it could reset at 7 for example, doing this: {!w & x & y & z : !w & !x & !y & !z} instead of what was written above: {!w & x & y & z : w & !x & !y & !z}.  
Therefore we can create an additional property that tests the change of value for ‘w’:

18. AG ((!w & x & y & z & V) -> AX(w & !x & !y & !z))  
This property works on the Modulo16 counter; but doesn’t on Modulo8 counter (if we initialize the value of ‘w’ to be false, like written in the file “Modulo8TestingQ8.gc”).  
And to make sure the modulo16 counter is fully correct, we can also add this:  
19. AG ((w & x & y & z & V) -> AX(!w & !x & !y & !z))  
Again, this works on Modulo16 but not Modulo8 (if we initialize the value of ‘w’ to be true in Modulo8).