Branching time – CTL:

- **EF**(Fail) = **E**(tt **U** Fail)

- **AG**(Req **=>** **AF**(Ack))
  =**!**(**EF**(**!**(**!**Req **or** **AF**(Ack)))
  =**!**(**EF**(Req **and** **EG**(**!**Ack))
  =**!**(**E**(tt **U** (Req **and** **EG**(**!** Ack))

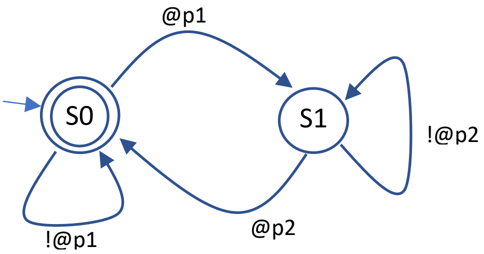
- **AG**(**AF**(DeviceEnabled))
  =**!EF!**(**AF**(DeviceEnabled))
  =**!EF**(**EG!**(DeviceEnabled))
  =**!E**(tt **U** (**EG**(**!** DeviceEnabled)))

- **AG** (**EF** (Restart))
  = **!EF!**(**EF**(Restart))
  = **!E**(tt **U** (**!EF**(Restart)))
  = **!E**(tt **U** (**!E**(tt **U** Restart))

Mutual exclusion

Part A

- Mutual exclusion phi_mx= G(!@p2 or !@q2)
- Starvation freedom phi_eat= G(@p1 => F @p2)
- Let Sigma = Set of subsets of the union of atomic propositions and their negations. For instance, the element {!@p2,!@p3,w0=1} in Sigma captures all configurations where the state of process p is neither p2 nor p3 and where the value of variable w0 is 1.
  A Büchi automaton for phi_eat:
  Two states: s0 (initial and accepting) and s1 (not accepting):
  - s0 to s0 on any element in Sigma except those including {!@p1} (intuitively, capture any configuration except those satisfying "!@p1").
  - s0 to s1 on any element in Sigma including {@p1}.
  - s1 to s1 on any element in Sigma including {!@p2}.
  - s1 to s0 on any element in Sigma including @p2.



Infinite words accepted by the automaton have to visit the accepting state s0 infinitely often. They do that by either never witnessing a configuration where @p1 holds (process p is interested in accessing its critical section), or by always witnessing a configuration where @p2 holds (process p at its critical section) sometime after they witness a configuration where @p1 holds.

Part B

The condition on the scheduler corresponds to “weak fairness”: the scheduler should not ignore a continuously enabled transition.

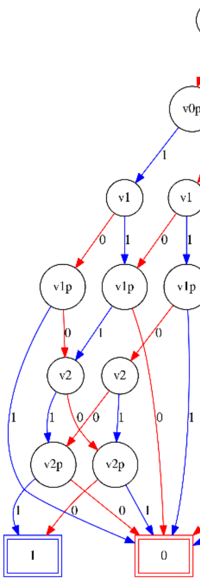
Suppose @p1 is true (we are at a configuration where p wants to access its critical section). We show @p2 will be eventually true.

P is the only process writing to w0. All incoming transitions to p1 assign 1 to w0. So w0 is 1.

If variable w1 is continuously 0, by fairness, t12 should be taken.

Suppose w1 is/becomes 1 while p is scheduled at p1 (and hence w0 is 1). Either t is 1 or 0. If t is 0 then q will eventually block at q6 after it assigned w1 to 0 (hence continuously enabling t12 and allowing p to access p2). If t is 1, p will eventually block at p6 after it assigned 0 to w0 (resulting q accessing q2 and assigning t to 0).

3. Symbolic representation



4. Partial and total correctness

Use {Inv: 0 <= x <= 100 and y = 2*x} as an invariant:

Partial correctness:

1. Q => Inv: (x=0 and y=0) => (0 <= x <= 100 and y = 2*x)
2. {Inv and 0 < 100} x:= x+1; y:= y+2 {Inv}:
   wp(‘x:= x+1; y:= y+2’, 0 <= x <= 100 and y = 2*x)
   = wp(‘x:= x+1’, 0 <= x <= 100 and y+2=2*x)
   = 0 <= x+1 <= 100 and y+2 = 2*(x + 1)
   = -1 <= x < 100 and y=2*x

   Indeed: (0 <= x <= 100 and y = 2*x and x < 100) => (-1 <= x < 100 and y=2*x)

3. (Inv and not (x < 100)) => Q:
   Inv and not (x < 100)
   = 0 <= x <= 100 and y = 2*x and x >= 100
   = (x = 100 and y = 2*x)
   => y = 200 => y < 201 = Q
   So, the program is partially correct.

   Termination: variant: v= 100 – x
4. (Inv and x < 100) => (v > 0):
   (0 <= x <= 100 and y = 2*x and x < 100) => (x < 100) => (100 – x > 0) => (v > 0)
5. {Inv and x < 100 and 100 – x = v0} x:= x+1; y:=y+1 {v < v0}
   We have :
   wp(‘x:= x+1; y:= y+2’, v < v0)
   = wp(‘x:= x+1; y:= y+2’, 100 - x < v0)
   = wp(‘x:= x+1’, 100 – x < v0)
   = 100 – x < v0 + 1

   In addition:
   Inv and x < 100 and 100 – x = v0 => 100 – x = v0

   Since:
   (100 – x = v0) => (100 – x < v0 +1) we get that:
   {Inv and x < 100 and 100 – x = v0} x:= x+1; y:=y+1 {v < v0}

   So, the program terminates.

Abstract interpretation

1. Fixpoint :

```
//x: [-oo,+oo]
L1. x:= 0
//x: [] widening [0,0]
L2. x:= x + 1
//x: [] widening [1,1]
L3. if x < 100 goto L2
//x: []
L4. nop
//x: []
L5. End

--------

//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [] widening [1,1]
L3. if x < 100 goto L2
//x: []
L4. nop
//x: []
L5. End

--------

//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [1,1] widening [2,2]
L3. if x < 100 goto L2
```

```
//x: []
L4. nop
//x: []
L5. end

--------

//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [1,+oo]
L3. if x < 100 goto L2
//x: [] widening [100,+oo]
L4. nop
//x: []
L5. end

--------

//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [1,+oo]
L3. if x < 100 goto L2
//x: [100,+oo]
L4. nop
//x: [] widening [100,+oo]
L5. end

--------

//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [1,+oo]
L3. if x < 100 goto L2
//x: [100,+oo]
L4. nop
//x: [100,+oo]
L5. end
```

2. Some precision can be recovered using narrowing.

```
//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [1,+oo] narrowing [1,100]
L3. if x < 100 goto L2
//x: [100,+oo]
L4. nop
//x: [100,+oo]
L5. End

-----

//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [1,100]
L3. if x < 100 goto L2
//x: [100,+oo] narrowing [100,100]
L4. nop
//x: [100,+oo]
L5. End

-----

//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [1,100]
```

```
L3. if x < 100 goto L2
//x: [100,100]
L4. nop
//x: [100,+oo] narrowing [100,100]
L5. End

-------

//x: [-oo,+oo]
L1. x:= 0
//x: [0,0]
L2. x:= x + 1
//x: [1,100]
L3. if x < 100 goto L2
//x: [100,100]
L4. nop
//x: [100,100]
L5. end
```