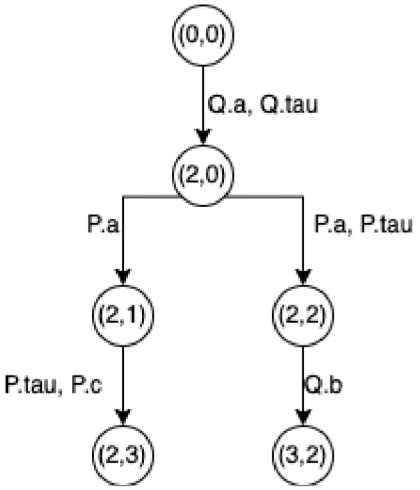
Taller 5

Ejercicio 2

P y Q son debilmente bisimilares ya que el defensor tiene siempre una estrategia de defensa siguiendo la siguiente relacion de bisimulación:

[(0,0), (1,2), (2,3), (3,1), (4,2)]

Ejercicio 3



Ejercicio 5

Usamos un allowlist para solo permitir el alfabeto de PRIMES_SPEC (Agregando esta linea al final de la definicion de PRIMES: `@{filter[0..3].prime[2..15],end}`) y de esta forma probamos que son debilmente bisimilares con MTSA ya que nos devolvio una relacion de bisimulación.

Ejercicio 9

1

```
property SHALL_NOT_PASS_WHEN_CLOSED = CLOSED,

CLOSED = (open -> OPEN),
OPEN = (entry -> OPEN
       | close -> CLOSED).
```

2

```
property SHALL NOT LEAVE WHEN CLOSED = CLOSED,

CLOSED = (open -> OPEN),
```

```
OPEN = (exit -> OPEN
     | close -> CLOSED).
```

3

```
ENTRADA = (entry -> ENTRADA).
SALIDA = (exit -> SALIDA).
DIRECTOR = (open -> close -> empty -> DIRECTOR).
const N=3
range T = 0..N
CONTROL = CONTROL_CERRADO,
CONTROL_CERRADO = (open -> CONTROL_ABIERTO[0]),
CONTROL_ABIERTO[t:T] = (when(t<N) entry -> CONTROL_ABIERTO[t+1]
                                | when(t>0) exit -> CONTROL_ABIERTO[t-1]
                      | close -> CONTROL_CERRANDO[t]),
CONTROL_CERRANDO[t:T] = (when(t>0) exit -> CONTROL_CERRANDO[t-1]
                         | when(t==0) empty -> CONTROL_CERRADO).

||MUSEO = (ENTRADA || SALIDA || DIRECTOR || CONTROL).

||CHECK_MUSEO = (MUSEO || SHALL_NOT_PASS_WHEN_CLOSED || SHALL_NOT_LEAVE_WHEN_CLOSED).
```

Y la traza que devolvió mtsa es

```
Trace to property violation in SHALL_NOT_LEAVE_WHEN_CLOSED:
    open
    entry
    close
    exit
```

Ejercicio 10

Con la siguiente propiedad pudimos verificar que el proceso LCR no tiene errores:

```
property ONLY ONE LEADER = NO LEADER,
NO_LEADER = (proc[uid:1..N].leader -> LEADER[uid]),
LEADER[uid:1..N] =  (proc[uid].leader -> LEADER[uid]).

||CHECK LCR = (LCR || ONLY ONE LEADER).
```

Ejercicio 12

Agregariamos la siguiente validacion de progreso para validar que algun proceso siempre sea electo como lider:

```
progress EXISTS LEADER = {proc[uid:1..N].leader}
```

Para chequear que en toda traza siempre pueda quedar electo el proceso 2 usariamos la siguiente validacion de progreso:

```
progress TWO ALWAYS CAN BE LEADER = {proc[2].leader}
```

Para este caso con el código de ejemplo nos da una violación de progreso ya que en ese caso el proceso 1 siempre es electo lider.

Ejercicio 13

La propiedad no es cierta ya que R restringe el comportamiento de Q. Por ejemplo si Q = COIN entonces:

```
COIN = (toss -> TAILS | toss -> HEADS),
TAILS = (tails -> COIN),
HEADS = (heads -> COIN).

R = (toss -> heads -> R).

||R_COIN = (COIN || R).

progress HEADS = {heads}
progress TAILS = {tail}
```

Podemos ver que COIN cumple ambas validaciones de progreso pero R_COIN no cumple con TAILS.

Ejercicio 14

```
a) [](enBase) (Es una propiedad de safety)
b) [](bateriaBaja -> (modoAhorro U enBase)) (Es una propiedad de liveness)
c) [](paredDelante -> (girandoAIzquierda U !paredDelante)) (Es una propiedad de
liveness)
```

Ejercicio 17

```
a) [](entroABase -> !<>salioDeBase)
b) [](bateriaBaja -> (
        (X modoAhorroOn) &&
        (!modoAhorroOff U entroABase) &&
        ([](entroABase -> X modoAhorroOff) U salioDeBase)
    )
c) [](paredDelanteDetectado -> X gira1GradoIzquierda)
```

Ejercicio 20

Las propiedades anteriores reescritas en LTL quedaron así:

```
assert SHALL_NOT_PASS_WHEN_CLOSED = [](close -> (!entry U open))

assert SHALL_NOT_LEAVE_WHEN_CLOSED = [](close -> (!leave U open))
```

Quedó mucho más compacto y claro en LTL. En el caso del observador hay que interpretar el modelo LTS y en el caso de la expresión LTL se lee casi de manera literal.