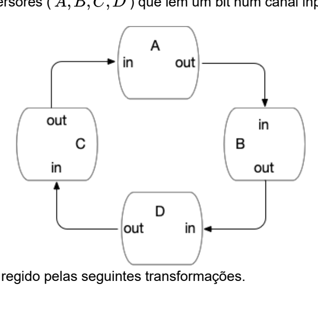
Trabalho 3

Grupo 5
Filipe Barbosa a77252
Hugo Ferreira a78555

In [1]:
```python
from z3 import *
```

Problema 1

O seguinte sistema dinâmico denota 4 inversores (A, B, C, D) que lêem um bit num canal input e escrevem num canal output uma transformação desse bit.



Cada inversor tem um bit s de estado, e é regido pelas seguintes transformações.

$$\begin{gathered}\textbf{invert}\,(in, out)\\ x \leftarrow \mathsf{read}(\mathtt{in})\\ s \leftarrow \neg x \;\;\|\;\; s \leftarrow s\\ \mathsf{write}(\mathtt{out}, s)\end{gathered}$$

O sistema termina quando todos os inversores tiverem o estado $s = 0$.

Construir um FOTS que descreve o sistema a cima numa abordagem BMC ("bounded model checker").

Definimos cada inversor com uma pré condição que restringe o valor de s, $s >= 0$. Começamos com um ciclo while com $pc = 0$ que testa se $s! = 0$. Caso seja então entramos no ciclo passando o $pc = 1$ e aqui lê o out do inversor antecessor na iteração anterior, efetua a operação $s \leftarrow \neg x \| s \leftarrow s$ e de seguida escreve no out o valor de s. Caso $s = 0$ o $pc = 2$ e passa para o estado final $stop$.

Criamos agora a função que cria a i-ésima cópia das variáveis de estado, agrupadas num dicionário que nos permite aceder às mesmas pelo nome.

Define-se as variáveis pc que indica o comando que, em cada momento, está disponivel para executar, s o bit, in como $Input$ do inversor e out como $output$ do inversor.

In [2]:
```python
def declare(i,x):
    state = {}
    state['pc'] = Int('pc'+str(i)+str(x))
    state['s'] = Int('s'+str(i)+str(x))
    state['in'] = Int('in'+str(i)+str(x))
    state['out'] = Int('out'+str(i)+str(x))
    return state
```

Define-se então a função $init$ que declara o estado inicial do programa. O pc tem de começar no comando 0 e o valor do bit s, in e out encontra-se no intervalor entre 0 e 1.

$$pc = 0 \wedge (s = 1 \vee s = 0) \wedge out = s \wedge 0 \leq in \leq 1$$

In [3]:
```python
def init(state):
    return And(state['pc'] == 0, Or(state['s'] == 1, state['s'] == 0), state['out'] == state['s'], stat
e['in'] >= 0,state['in'] <= 1)
```

As transições possiveis no FOTS são caracterizadas pelo seguinte predicado, onde são usadas as variáveis já declaradas anteriormente, mais a variável $outAnt$ que é o out na iteração anterior do inversor que se liga ao atual e $out1$, $out2$ que são os output dos restantes inversores. Precisamos de saber o output de todos os inversores para saber se todos os valores de $s = 0$ e então terminar o programa:

$$\begin{gathered}(pc = 0 \wedge (((s = 0 \wedge out1 = 0 \wedge out2 = 0 \wedge outAnt = 0) \implies pc' = 2) \vee ((s! = 0 \vee out1! = 0 \vee out2! = 0\\ \vee outAnt! = 0) \implies pc' = 1)) \wedge s' = s \wedge out' = s' \wedge in' = outAnt)\\ \vee\\ (pc = 1 \wedge pc' = 0 \wedge ((s' = s) \vee (outAnt = 1 \implies s' = 0 \wedge outAnt = 0 \implies s' = 1)) \wedge out' = s' \wedge in'\\ = outAnt)\\ \vee\\ (pc = 2 \wedge pc' = 2 \wedge s' = s \wedge out' = s' \wedge in' = in)\end{gathered}$$

In [4]:
```python
def trans(curr,prox,outAnt, out1, out2):
    t1 = And(curr['pc'] == 0, If(And(curr['s'] == 0, out1 == 0, out2 == 0, outAnt == 0), prox['pc'] ==
2, prox['pc'] == 1),                  prox['s'] == curr['s'],
      prox['out']==prox['s'], prox['in']==outAnt)
    t2 = And(curr['pc'] == 1,                  prox['pc'] == 0,
                                  Or(prox['s'] == curr['s'],If(outAnt == 1,prox['s'] == 0,prox['s'] ==
 1)), prox['out']==prox['s'], prox['in']==outAnt)
    t3 = And(curr['pc'] == 2,                  prox['pc'] == 2,
                                  prox['s'] == curr['s'],
      prox['out']==prox['s'], prox['in']==curr['in'])
    return Or(t1,t2,t3)
```

Criamos agora a função de ordem superior $gera\_traco$ que, para quatro estados (um para cada inversor), gera uma cópia das variáveis do estado, um predicado que define o estado inicial, outro que adiciona pares de transições entre pares de estados e um número n que define o tamanho do traço.

In [13]:
```python
def gera_traco(declare,init,trans,n):
    s = Solver()
    state0 = [declare(i,0) for i in range(n)]
    state1 = [declare(i,1) for i in range(n)]
    state2 = [declare(i,2) for i in range(n)]
    state3 = [declare(i,3) for i in range(n)]
    s.add(init(state0[0]))
    s.add(init(state1[0]))
    s.add(init(state2[0]))
    s.add(init(state3[0]))
    for i in range(n-1):
        s.add(trans(state0[i],state0[i+1],state3[i]['s'],state1[i]['s'],state2[i]['s']))
        s.add(trans(state1[i],state1[i+1],state0[i]['s'],state2[i]['s'],state3[i]['s']))
        s.add(trans(state2[i],state2[i+1],state1[i]['s'],state0[i]['s'],state3[i]['s']))
        s.add(trans(state3[i],state3[i+1],state2[i]['s'],state0[i]['s'],state1[i]['s']))
    if s.check() == sat:
        m = s.model()
        for i in range(n):
            print("Iteração",i+1)
            print("InversorA")
            for x in state0[i]:
                print(x,'=',m[state0[i][x]])
            print("InversorB")
            for x in state1[i]:
                print(x,'=',m[state1[i][x]])
            print("InversorD")
            for x in state2[i]:
                print(x,'=',m[state2[i][x]])
            print("InversorC")
            for x in state3[i]:
                print(x,'=',m[state3[i][x]])
            print('\n')

gera_traco(declare,init,trans,10)
```

```
Iteração 1
InversorA
pc = 0
s = 1
in = 0
out = 1
InversorB
pc = 0
s = 1
in = 0
out = 1
InversorD
pc = 0
s = 1
in = 0
out = 1
InversorC
pc = 0
s = 1
in = 0
out = 1


Iteração 2
InversorA
pc = 1
s = 1
in = 1
out = 1
InversorB
pc = 1
s = 1
in = 1
out = 1
InversorD
pc = 1
s = 1
in = 1
out = 1
InversorC
pc = 1
s = 1
in = 1
out = 1


Iteração 3
InversorA
pc = 0
s = 1
in = 1
out = 1
InversorB
pc = 0
s = 1
in = 1
out = 1
InversorD
pc = 0
s = 1
in = 1
out = 1
InversorC
pc = 0
s = 1
in = 1
out = 1


Iteração 4
InversorA
pc = 1
s = 1
in = 1
out = 1
InversorB
pc = 1
s = 1
in = 1
out = 1
InversorD
pc = 1
s = 1
in = 1
out = 1
InversorC
pc = 1
s = 1
in = 1
out = 1


Iteração 5
InversorA
pc = 0
s = 0
in = 1
out = 0
InversorB
pc = 0
s = 0
in = 1
out = 0
InversorD
pc = 0
s = 0
in = 1
out = 0
InversorC
pc = 0
s = 1
in = 1
out = 1


Iteração 6
InversorA
pc = 1
s = 0
in = 1
out = 0
InversorB
pc = 1
s = 0
in = 0
out = 0
InversorD
pc = 1
s = 0
in = 0
out = 0
InversorC
pc = 1
s = 1
in = 0
out = 1


Iteração 7
InversorA
pc = 0
s = 0
in = 1
out = 0
InversorB
pc = 0
s = 0
in = 0
out = 0
InversorD
pc = 0
s = 0
in = 0
out = 0
InversorC
pc = 0
s = 1
in = 0
out = 1


Iteração 8
InversorA
pc = 1
s = 0
in = 1
out = 0
InversorB
pc = 1
s = 0
in = 0
out = 0
InversorD
pc = 1
s = 0
in = 0
out = 0
InversorC
pc = 1
s = 1
in = 0
out = 1


Iteração 9
InversorA
pc = 0
s = 0
in = 1
out = 0
InversorB
pc = 0
s = 0
in = 0
out = 0
InversorD
pc = 0
s = 0
in = 0
out = 0
InversorC
pc = 0
s = 1
in = 0
out = 1


Iteração 10
InversorA
pc = 1
s = 0
in = 1
out = 0
InversorB
pc = 1
s = 0
in = 0
out = 0
InversorD
pc = 1
s = 0
in = 0
out = 0
InversorC
pc = 1
s = 1
in = 0
out = 1
```

Verificar por k-induction invariantes.

Implementamos de seguida o método `kinduction_always` para verificar invariantes por k-indução. Começamos por usar o $declare$, $init$ e $trans$ definidos anteriormente. De seguida adicionamos ao solver o contrário do que queremos testar e se então o modelo der sat essa propriedade não é respeitada no programa para os k estados. Se o teste a cima der válido (dar $unsat$) testamos então para $k+1$ estados para provar totalmente o invariante.

In [6]:
```python
def kinduction_always(declare,init,trans,inv,k):
    s = Solver()
    state0 = [declare(i,0) for i in range(k)]
    state1 = [declare(i,1) for i in range(k)]
    state2 = [declare(i,2) for i in range(k)]
    state3 = [declare(i,3) for i in range(k)]
    s.add(init(state0[0]))
    s.add(init(state1[0]))
    s.add(init(state2[0]))
    s.add(init(state3[0]))
    s.add(And([trans(state0[i],state0[i+1],state3[i]['s'],state1[i]['s'],state2[i]['s']) for i in range
(k-1)]))
    s.add(And([trans(state1[i],state1[i+1],state0[i]['s'],state2[i]['s'],state3[i]['s']) for i in range
(k-1)]))
    s.add(And([trans(state2[i],state2[i+1],state1[i]['s'],state0[i]['s'],state3[i]['s']) for i in range
(k-1)]))
    s.add(And([trans(state3[i],state3[i+1],state2[i]['s'],state0[i]['s'],state1[i]['s']) for i in range
(k-1)]))
    s.add(Or([Not(inv(state0[i])) for i in range(k)]))
    s.add(Or([Not(inv(state1[i])) for i in range(k)]))
    s.add(Or([Not(inv(state2[i])) for i in range(k)]))
    s.add(Or([Not(inv(state3[i])) for i in range(k)]))
    if s.check() == sat:
        print('O invariante é falso num dos primeiros',k,'estados.')
        return

    assert (s.check() == unsat)
    s = Solver()
    state0 = [declare(i,0) for i in range(k+1)]
    state1 = [declare(i,1) for i in range(k+1)]
    state2 = [declare(i,2) for i in range(k+1)]
    state3 = [declare(i,3) for i in range(k+1)]
    s.add(init(state0[0]))
    s.add(init(state1[0]))
    s.add(init(state2[0]))
    s.add(init(state3[0]))
    s.add(And([trans(state0[i],state0[i+1],state3[i]['s'],state1[i]['s'],state2[i]['s']) for i in range
(k)]))
    s.add(And([trans(state1[i],state1[i+1],state0[i]['s'],state2[i]['s'],state3[i]['s']) for i in range
(k)]))
    s.add(And([trans(state2[i],state2[i+1],state1[i]['s'],state0[i]['s'],state3[i]['s']) for i in range
(k)]))
    s.add(And([trans(state3[i],state3[i+1],state2[i]['s'],state0[i]['s'],state1[i]['s']) for i in range
(k)]))
    s.add(And([inv(state0[i]) for i in range(k)]))
    s.add(And([inv(state1[i]) for i in range(k)]))
    s.add(And([inv(state2[i]) for i in range(k)]))
    s.add(And([inv(state3[i]) for i in range(k)]))
    s.add(Not(inv(state0[k])))
    s.add(Not(inv(state1[k])))
    s.add(Not(inv(state2[k])))
    s.add(Not(inv(state3[k])))
    if s.check() == sat:
        print('Não é possivel provar o invariante com',k,'indução.')
        return
    assert (s.check() == unsat)
    print("Propriedade válida!")
```

De seguida testa-se se o valor do s está correto de acordo com o pc que o programa se encontra, caso $pc = 1$ o $s = 1$, e se $pc = 0$ o $s = 0$ ou $s = 1$.

In [7]:
```python
def em_execucao(state):
    return Or(If(state['pc'] == 1, state['s'] == 1, state['s'] >= 0), If(state['pc'] == 0, state['s'] <
= 1, state['s'] >= 0))

kinduction_always(declare,init,trans,em_execucao,10)
```

```
Propriedade válida!
```

Testamos de seguida se quando o inversor termina $(pc = 2)$ o s é 0.

In [8]:
```python
def termina_quando_s0(state):
    return (If(state['pc'] == 2, state['s'] == 0, state['s'] >= 0))

kinduction_always(declare,init,trans, termina_quando_s0,10)
```

```
Propriedade válida!
```

Verificar em que condições o sistema termina.

Agora para testar em que condições o programa termina em k iterações criamos o $bmc\_eventually$ que vai de encontro à versão $kinduction\_always$ mas agora adicionamos logo a propriedade que queremos que aconteça e o programa dá o estado inicial e restantes iterações para conseguir cumprir essa propriedade.

In [18]:
```python
def bmc_eventually(declare,init,trans,prop,k):
    s = Solver()
    state0 = [declare(i,0) for i in range(k)]
    state1 = [declare(i,1) for i in range(k)]
    state2 = [declare(i,2) for i in range(k)]
    state3 = [declare(i,3) for i in range(k)]
    s.add(init(state0[0]))
    s.add(init(state1[0]))
    s.add(init(state2[0]))
    s.add(init(state3[0]))
    s.add(And([trans(state0[i],state0[i+1],state3[i]['s'],state1[i]['s'],state2[i]['s']) for i in range
(k-1)]))
    s.add(And([trans(state1[i],state1[i+1],state0[i]['s'],state2[i]['s'],state3[i]['s']) for i in range
(k-1)]))
    s.add(And([trans(state2[i],state2[i+1],state1[i]['s'],state0[i]['s'],state3[i]['s']) for i in range
(k-1)]))
    s.add(And([trans(state3[i],state3[i+1],state2[i]['s'],state0[i]['s'],state1[i]['s']) for i in range
(k-1)]))
    s.add(Or([(prop(state0[i])) for i in range(k)]))
    s.add(Or([(prop(state1[i])) for i in range(k)]))
    s.add(Or([(prop(state2[i])) for i in range(k)]))
    s.add(Or([(prop(state3[i])) for i in range(k)]))
    if s.check() == sat:
        m = s.model()
        for i in range(k):
            print("Iteração",i+1)
            print("InversorA")
            for x in state0[i]:
                print(x,'=',m[state0[i][x]])
            print("InversorB")
            for x in state1[i]:
                print(x,'=',m[state1[i][x]])
            print("InversorD")
            for x in state2[i]:
                print(x,'=',m[state2[i][x]])
            print("InversorC")
            for x in state3[i]:
                print(x,'=',m[state3[i][x]])
            print('\n')
        print('Termina com',k,'iterações!')
        return
    assert (s.check() == unsat)
    print('Não Termina com',k,'iterações!')
```

Criamos a propriedade cuja inevitabilidade queremos que se verifique, que neste caso é o programa terminar e para isso queremos que para os 4 inversores o $pc = 2$.

In [17]:
```python
def termina(state):
    return state['pc']==2

bmc_eventually(declare,init,trans,termina,10)
```

```
Iteração 1
InversorA
pc = 0
s = 1
in = 0
out = 1
InversorB
pc = 0
s = 1
in = 0
out = 1
InversorD
pc = 0
s = 1
in = 0
out = 1
InversorC
pc = 0
s = 1
in = 0
out = 1


Iteração 2
InversorA
pc = 1
s = 1
in = 1
out = 1
InversorB
pc = 1
s = 1
in = 1
out = 1
InversorD
pc = 1
s = 1
in = 1
out = 1
InversorC
pc = 1
s = 1
in = 1
out = 1


Iteração 3
InversorA
pc = 0
s = 1
in = 1
out = 1
InversorB
pc = 0
s = 1
in = 1
out = 1
InversorD
pc = 0
s = 1
in = 1
out = 1
InversorC
pc = 0
s = 1
in = 1
out = 1


Iteração 4
InversorA
pc = 1
s = 1
in = 1
out = 1
InversorB
pc = 1
s = 1
in = 1
out = 1
InversorD
pc = 1
s = 1
in = 1
out = 1
InversorC
pc = 1
s = 1
in = 1
out = 1


Iteração 5
InversorA
pc = 0
s = 1
in = 1
out = 1
InversorB
pc = 0
s = 1
in = 1
out = 1
InversorD
pc = 0
s = 1
in = 1
out = 1
InversorC
pc = 0
s = 1
in = 1
out = 1


Iteração 6
InversorA
pc = 1
s = 1
in = 1
out = 1
InversorB
pc = 1
s = 1
in = 1
out = 1
InversorD
pc = 1
s = 1
in = 1
out = 1
InversorC
pc = 1
s = 1
in = 1
out = 1


Iteração 7
InversorA
pc = 0
s = 1
in = 1
out = 1
InversorB
pc = 0
s = 1
in = 1
out = 1
InversorD
pc = 0
s = 1
in = 1
out = 1
InversorC
pc = 0
s = 1
in = 1
out = 1


Iteração 8
InversorA
pc = 1
s = 1
in = 1
out = 1
InversorB
pc = 1
s = 1
in = 1
out = 1
InversorD
pc = 1
s = 1
in = 1
out = 1
InversorC
pc = 1
s = 1
in = 1
out = 1


Iteração 9
InversorA
pc = 0
s = 0
in = 1
out = 0
InversorB
pc = 0
s = 0
in = 1
out = 0
InversorD
pc = 0
s = 0
in = 1
out = 0
InversorC
pc = 0
s = 0
in = 1
out = 0


Iteração 10
InversorA
pc = 2
s = 0
in = 0
out = 0
InversorB
pc = 2
s = 0
in = 0
out = 0
InversorD
pc = 2
s = 0
in = 0
out = 0
InversorC
pc = 2
s = 0
in = 0
out = 0


Termina com 10 iterações!
```

In [ ]: