Implementação

Começamos por implementar as bibliotecas que necessitamos para realizar o problema. O **pysmt.shortcuts** oferece uma API simplificada que disponibiliza as funcionalidades para a utilização usual de um SMT solver. Chamamos o tipo Int de **pysmt.typing** e também as funções randint e choice da biblioteca **random**

[1]:
```
from pysmt.shortcuts import *
from pysmt.typing import INT
from random import randint, choice

import itertools
```

Parâmetros do programa

O Sistema dinâmico denota 4 inversores (A,B,C,D) e cada um deles tem um bit de estado inicializado de forma aleatoria, com valor de 0 ou de 1.

[2]:
```
a = randint(0,1)
b = randint(0,1)
c = randint(0,1)
d = randint(0,1)
```

Função genState

Definimos a função **genState** que recebe a lista com o nome das variáveis do estado, uma etiqueta e um inteiro, e cria a i-ésima cópia das variáveis do estado para essa etiqueta. As variáveis lógicas começam sempre com o nome de base das variáveis dos estado, seguido do separador `!`.

[3]:
```
def genState(vars,s,i):
    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+s+str(i),INT)
    return state
```

Função init

Dado um estado do programa, devolve um predicado do pySMT que testa se esse estado é um possível estado inicial do programa.

Função error

Dado um estado do programa, devolve um predicado do pySMT que testa se esse estado é um possível estado de erro do programa. Neste caso é quando um estado do sistema é (0,0,0,0).

Funções not_ e xor_

Seja p e q predicados logicos sabemos que $p \implies q \Leftrightarrow \neg p \vee q$, as funções com prefixo **not_** definem a operação logica "negação", variando consoante o inversor em que estamos, isto é a, b, c ou d. As funções com prefixo **xor_** definem-se de forma parecida, sendo que neste caso se realiza a função logica do "ou exclusivo", variando consoante o inversor assim com a outra função.

Função trans

Como todos os inversores (A,B,C,D) foram inicializados com um valor aleatorio e com apenas uma das duas funcionalidades **not** ou **xor**, também realizada de forma aleatoria.

[4]:
```
def init1(state):
    return And(
            Equals(state['a'], Int(a)),
            Equals(state['b'], Int(b)),
            Equals(state['c'], Int(c)),
            Equals(state['d'], Int(d))
    )

def error1(state):
    return And(
            Equals(state['a'], Int(0)),
            Equals(state['b'], Int(0)),
            Equals(state['c'], Int(0)),
            Equals(state['d'], Int(0))
    )

def not_a(curr, prox):
    t1 = Or(
        NotEquals(prox['c'], Int(0)),
        Equals(prox['a'], Int(1))
     )

    t2 = Or(
            NotEquals(prox['c'], Int(1)),
            Equals(prox['a'], Int(0))
         )

    return And(t1, t2)

def not_b(curr, prox):
    t1 = Or(
            NotEquals(curr['a'], Int(0)),
            Equals(prox['b'], Int(1))
```

```
        )

    t2 = Or(
            NotEquals(curr['a'], Int(1)),
            Equals(prox['b'], Int(0))
        )

    return And(t1, t2)

def not_c(curr, prox):
    t1 = Or(
        NotEquals(prox['d'], Int(0)),
        Equals(prox['c'], Int(1))
     )

    t2 = Or(
            NotEquals(prox['d'], Int(1)),
            Equals(prox['c'], Int(0))
        )

    return And(t1, t2)

def not_d(curr, prox):
    t1 = Or(
            NotEquals(prox['b'], Int(0)),
            Equals(prox['d'], Int(1))
        )

    t2 = Or(
            NotEquals(prox['b'], Int(1)),
            Equals(prox['d'], Int(0))
        )

    return And(t1, t2)

def xor_a(curr, prox):
    t1 = Or(
            NotEquals(curr['a'], prox['c']),
            Equals(prox['a'], Int(0))
        )

    t2 = Or(
            Equals(curr['a'], prox['c']),
            Equals(prox['a'], Int(1))
        )

    return And(t1, t2)
```

```
def xor_b(curr, prox):
    t1 = Or(
            NotEquals(curr['a'], curr['b']),
            Equals(prox['b'], Int(0))
         )

    t2 = Or(
            Equals(curr['a'], curr['b']),
            Equals(prox['b'], Int(1))
         )

    return And(t1, t2)

def xor_c(curr, prox):
    t1 = Or(
            NotEquals(curr['c'], prox['d']),
            Equals(prox['c'], Int(0))
         )

    t2 = Or(
            Equals(curr['c'], prox['d']),
            Equals(prox['c'], Int(1))
         )

    return And(t1, t2)

def xor_d(curr, prox):
    t1 = Or(
            NotEquals(curr['d'], prox['b']),
            Equals(prox['d'], Int(0))
         )

    t2 = Or(
            Equals(curr['d'], prox['b']),
            Equals(prox['d'], Int(1))
         )

    return And(t1, t2)

def trans1(curr,prox):

    t1 = f1(curr, prox)
    t2 = f2(curr, prox)
    t3 = f3(curr, prox)
    t4 = f4(curr, prox)
```

```
    return And(t1, t2, t3, t4)
```

[5]:
```
f1 = choice([not_a, xor_a])
f2 = choice([not_b, xor_b])
f3 = choice([not_c, xor_c])
f4 = choice([not_d, xor_d])

# print(f1, f2, f3, f4)
```

Função genTrace

A fórmula $\mathsf{I} \wedge \mathsf{T}^n$ denota um traço finito com n transições em Σ, $\mathsf{X}_0, \cdots, \mathsf{X}_n$, que descrevem estados acessíveis com n ou menos transições. Inspirada nesta notação, a seguinte função `genTrace` gera um possível traço de execução com n transições.

[6]:
```
def genTrace(vars,init,trans,error,n):
    with Solver(name="z3") as s:

        X = [genState(vars,'X',i) for i in range(n+1)]   # cria n+1 estados (com␣
↪etiqueta X)
        I = init(X[0])
        Tks = [ trans(X[i],X[i+1]) for i in range(n) ]

        if s.solve([I,And(Tks)]):      # testa se I /\ T^n  é satisfazível
            for i in range(n):
                print("Estado:",i)
                for v in X[i]:
                    print("          ",v,'=',s.get_value(X[i][v]))

genTrace(['a', 'b', 'c', 'd'], init1, trans1, error1, 10)
```

Relembremos o algoritmo de model checking(automatico) usado no exercicio 1, que utilizamos aqui, para verificar a segurança do SFOTS.

[7]:
```
def invert(trans):
    return( lambda u, v : trans(v,u) )

def baseName(s):
    return ''.join(list(itertools.takewhile(lambda x: x!='!', s)))

def rename(form,state):
    vs = get_free_variables(form)
    pairs = [ (x,state[baseName(x.symbol_name())]) for x in vs ]
    return form.substitute(dict(pairs))

def same(state1,state2):
    return And([Equals(state1[x],state2[x]) for x in state1])
```

```
def model_checking(vars,init,trans,error,N,M):
    with Solver(name="z3") as s:

        # Criar todos os estados que poderão vir a ser necessários.
        X = [genState(vars,'X',i) for i in range(N+1)]
        Y = [genState(vars,'Y',i) for i in range(M+1)]

        # Estabelecer a ordem pela qual os pares (n,m) vão surgir. Por exemplo:
        order = sorted([(a,b) for a in range(1,N+1) for b in␣
 ↪range(1,M+1)],key=lambda tup:tup[0]+tup[1])

        for (n,m) in order:

            I  = init(X[0])
            Tn = And([trans(X[i],X[i+1]) for i in range(n) ])
            Rn = And (I,Tn)

            E  = error(Y[0])
            Bm = And( [invert(trans)(Y[i],Y[i+1]) for i in range(m) ])
            Um = And (E,Bm)

            Vnm = And( Rn, same(X[n],Y[m]), Um )

            if s.solve([Vnm]):
                print("inseguro")
                return

            C = binary_interpolant(And (Rn, same(X[n],Y[m])), Um)

            if C is None:
                print(" interpolante None ")
                continue

            C0 = rename(C,X[0])
            C1 = rename(C,X[1])
            T  = trans(X[0],X[1])

            if not s.solve([C0,T, Not(C1) ]):
                print("safe")
                return

            else:
                # gerar o S
                S = rename(C,X[n])

                while(True):
                    A = And(S, trans(X[n],Y[n]) )
```

```
                    if s.solve( [A,Um] ):
                        # print("Não encontramos o majorante")
                        break
                    else:
                        Cnew = binary_interpolant( A, Um )
                        Cn   = rename(Cnew, X[n])

                        if s.solve( [Cn,Not(S)] ):
                            S = Or(S,Cn)

                        else:
                            print("safe")
                            return
        print("unknown")

#####

model_checking(['a', 'b', 'c', 'd'], init1, trans1, error1, 50, 50)
```