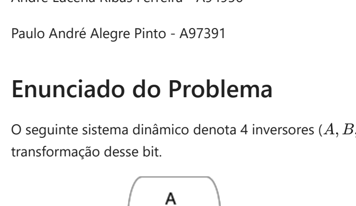
TP3 - Grupo 14

André Lucena Ribas Ferreira - A94956

Paulo André Alegre Pinto - A97391

Enunciado do Problema

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.



1. Cada inversor tem um bit s de estado, inicializado com um valor aleatório.
2. Cada inversor é regido pelas seguintes transformações:

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

3. A escolha neste comando é sempre determinística; isto é, em cada inversor a escolha do comando a executar é sempre a mesma. Porém qual é essa escolha é determinada aleatoriamente na inicialização do sistema.
4. O estado do sistema é um tuplo definido pelos 4 bits s, e é inicializado com um vetor aleatório em $\{0,1\}^4$.
5. O sistema termina em ERRO quando o estado do sistema for $(0,0,0,0)$.

Prentede-se o seguinte:

1. Construa um SFOTS que descreva este sistema e implemente este sistema, numa abordagem BMC ("bouded model checker") num traço com n estados.
2. Verifique se o sistema é seguro usando BMC, k-indução ou model checking com interpolantes.

Análise

O problema exposto tem certas considerações a ter em conta.

Em primeiro lugar, considera-se o in de cada um dos inversores igual ao out do estado anterior, ou seja, o estado anterior com uma rotação cíclica das variáveis, exceto no caso do estado inicial. A ordem da rotação é a descrita nos canais, $a \to b \to d \to c$

No caso do estado inicial, decidiu-se gerar o in aleatoriamente tal como se gera o s para cada inversor, isto é, o estado inicial.

Em segundo lugar, como cada inversor funciona em paralelo com cada outro inversor, as variáveis de cada estado vão ser apenas cada um dos elementos de cada tuplo de $\{0,1\}^4$, não existindo variável que seja considerada o contador do programa.

Em terceiro lugar, deve-se definir aleatoriamente cada um dos comportamentos dos inversores, que será mantido durante a execução inteira.

Definição do SFOTS

Tal como pretendido, o sistema dinâmico será modelado através um SFOTS, nomeadamente um tuplo:

$$\Sigma \equiv < \mathcal{T}, X, next, I, T, E >$$

Onde se verifica o seguinte, para representar o sistema em específico:

1. $\mathcal{T}$ representa um SMT apropriada, que pertence à FOL, que vamos representar no nosso Solver;
2. X é o conjunto das variáveis base do problema;
3. $next$ é um operador que gera "clones" das variáves em X;
4. I é um predicado unário que determina quais os estados iniciais;
5. T é um predicado binário que determina as transições entre dois estados;
6. E é um predicado unário que determina os estados de erro.

X neste caso é constituído por uma variável binária para cada um dos estados dos inversores, $\{s_a, s_b, s_c, s_d\}$. Dependendo da transição ciclica, um dos estados de um dos inversores será considerado o valor de entrada de outro. Ou seja, $in_a = s_c$, $in_b = s_a$, $in_c = s_d$ e $in_d = s_b$.

Considerando o estado inicial, não existem condições que limitam as variáveis, já que é possível o primeiro estado ser um estado de erro.

O predicado de transição depende da definição aleatória no início da execução. No seu geral, é definido por uma conjunção entre todos os inversores, cada um que pode ser um de dois predicados. Por exemplo, para o inversor A:

$$T_a \;\equiv\; s'_a = \neg\, s_c \quad \text{ou} \quad T_a \;\equiv\; s'_a = s_a \oplus s_c$$

O estado de erro, tal como definido no enunciado, ocorre quando cada um dos s é igual a 0.

$$E \quad\equiv\quad s_a = 0 \wedge s_b = 0 \wedge s_c = 0 \wedge s_d = 0$$

Implementação

Para a resolução do problema em questão, decidiu-se usar o módulo `pysmt.shortcuts`, com as funcionalidades possíveis para a utilização de um SMT Solver. Importam-se também os tipos deste Solver, a partir do módulo `pysmt.typing`. Para modelar este problema, usar-se-ão variáveis binárias, então importamos o tipo `BOOL`. Também importamos a função `binomial` do módulo `numpy.random` para poder gerar lançamentos de moedas aleatórios.

In [1]:
```python
from pysmt.shortcuts import *
from pysmt.typing import BOOL

from numpy.random import binomial
import itertools
```

Para gerar os estados, a função `genState(vars,s,i)` recebe as variáveis do problema e, para cada uma, cria-se uma variável binária. Também se marca a variável com uma letra `s` que representa o conjunto de variáveis, nomeadamente X ou Y, tal como um número `i` que representa a i-ésima cópia da variável.

In [2]:
```python
def genState(vars,s,i):
    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+s+str(i),BOOL)
    return state
```

Para se gerar os estados iniciais e as transições, é preciso definir aleatoriamente tanto as variáveis do estado inicial tal como qual das transições será utilizada por cada inversor.

In [3]:
```python
def genInitialRandom():
    s = {}
    s['a']= binomial(1,0.5) == 1
    s['b'] = binomial(1,0.5) == 1
    s['c'] = binomial(1,0.5) == 1
    s['d'] = binomial(1,0.5) == 1
    print("Estados inciais:",s['a'],s['b'],s['c'],s['d'])
    return s

def genTransRandom():
    t = {}
    t['a'] = binomial(1,0.5) == 1
    t['b'] = binomial(1,0.5) == 1
    t['c'] = binomial(1,0.5) == 1
    t['d'] = binomial(1,0.5) == 1
    print("Transição Determinada:", "Neg" if t['a'] else "XOR", "Neg" if t['b'] else "XOR",
          "Neg" if t['c'] else "XOR", "Neg" if t['d'] else "XOR")
    return t
```

As variáveis do problema são os estados internos de cada inversor, que coincidem com o seu out. Dessa forma, não é necessário definir o in dos inversores.

In [4]:
```python
vars = ['s_a','s_b','s_c','s_d'] #in_a = s_c, in_b = s_a, in_c = s_d, in_d = s_b
```

A função inicial depende dos valores aleatórios gerados, apenas limitando cada uma das variáveis a esse valor, através de uma equivalência. `init1` gera os valores aleatórios dentro da função enquanto que `init_notrandom` necessita de receber os valores gerados como parâmetro. Desta forma, cada função que as use será diferente na escrita de acordo com a função pretendida.

In [5]:
```python
def init1(state):
    s = genInitialRandom()
    return And(Iff(state['s_a'], Bool(s['a'])), Iff(state['s_b'], Bool(s['b'])),
               Iff(state['s_c'], Bool(s['c'])), Iff(state['s_d'], Bool(s['d'])))

def init_notrandom(state,a,b,c,d):
    return And(Iff(state['s_a'], Bool(a)), Iff(state['s_b'], Bool(b)),
               Iff(state['s_c'], Bool(c)), Iff(state['s_d'], Bool(d)))
```

A função `error1` é semelhante às iniciais, mas limita exatamente o estado $(0,0,0,0)$ como o estado de erro.

In [6]:
```python
def error1(state):
    return And(Iff(state['s_a'], Bool(False)), Iff(state['s_b'], Bool(False)),
               Iff(state['s_c'], Bool(False)), Iff(state['s_d'], Bool(False)))
```

A função de transição `trans1` depende também dos valores gerados, e que recebe como parâmetro. Para cada um desses valores, escolhe-se uma das transições que será sempre a mesma (já que os valores são gerados apenas uma vez). Neste caso, `True` representa a transição $s \leftarrow \neg x$ e `False` a transição $s \leftarrow s \oplus x$.

In [7]:
```python
def trans1(curr, prox, a, b, c, d):
    t_a = Iff(Not(curr['s_c']), prox['s_a']) if a else Iff(Xor(curr['s_c'], curr['s_a']), prox['s_a'])
    t_b = Iff(Not(curr['s_a']), prox['s_b']) if b else Iff(Xor(curr['s_a'], curr['s_b']), prox['s_b'])
    t_c = Iff(Not(curr['s_d']), prox['s_c']) if c else Iff(Xor(curr['s_d'], curr['s_c']), prox['s_c'])
    t_d = Iff(Not(curr['s_b']), prox['s_d']) if d else Iff(Xor(curr['s_b'], curr['s_d']), prox['s_d'])
    return And(t_a, t_b, t_c, t_d)
```

A função `genTrace` recebe todas estas funções, exceto a do predicado do erro, mais o número `n`, tamanho máximo do traço, e devolve um traço execução sem qualquer consideração pelo estado de erro.

In [8]:
```python
def genTrace(vars,init,trans,n,s,t):
    with Solver(name="z3") as solver:
        X = [genState(vars,'X',i) for i in range(n+1)]    # cria n+1 estados (com etiqueta X)
        I = init(X[0],s['a'],s['b'],s['c'],s['d'])
        Tks = [ trans(X[i],X[i+1],t['a'],t['b'],t['c'],t['d']) for i in range(n) ]

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

In [9]:
```python
t = genTransRandom()
s = genInitialRandom()
genTrace(vars, init_notrandom, trans1, 5,s,t)
```

```
Transição Determinada: Neg Neg Neg XOR
Estados inciais: False False False True
Estado: 0
           s_a = False
           s_b = False
           s_c = False
           s_d = True
Estado: 1
           s_a = True
           s_b = True
           s_c = False
           s_d = True
Estado: 2
           s_a = True
           s_b = False
           s_c = False
           s_d = False
Estado: 3
           s_a = True
           s_b = False
           s_c = True
           s_d = False
Estado: 4
           s_a = False
           s_b = False
           s_c = True
           s_d = False
```

Verificação de Segurança

Para as Verificações de Segurança, utilizaram-se os três métodos descritos. De reparar que o sistema é seguro apenas quando entra em ciclo, já que o único caso que pode ser considerado de paragem é o caso de erro $(0,0,0,0)$. Para além disso, apenas há um número finito de estados diferentes, nomeadamente 16, isto é, 2^4, então qualquer traço é limitado.

Todas as funções aqui definidas são como as funções já estudades, apenas com duas diferenças. Por um lado, recebem dois dicionários, s e t, definidos tal como nas funções aleatórias, para determinar o estado inicial e as transições de uma maneira mais sistemática. Por outro, geram estados a partir da função `genState`.

O invariante que pretendemos provar é o seguinte.

In [10]:
```python
def inv1(state):
    return Not(error1(state))
```

BMC

A função `bmc_always` implementa o algoritmo de *Bounded Model-Checking*, com traços de tamanho máximo K. Como os traços são limitados, com tamanho máximo 15 (15 estados distintos), é possível estar confiante dos resultados dados com traços com esse tamanho máximo.

In [11]:
```python
def bmc_always(declare,init,trans,inv,K,t,s):
    for k in range(1,K+1):
        with Solver(name="z3") as solver:
            trace = [genState(vars,'X',i) for i in range(k)]
            solver.add_assertion(init(trace[0], s['a'], s['b'], s['c'], s['d']))
            for i in range(k-1):
                solver.add_assertion(trans(trace[i],trace[i+1],t['a'], t['b'], t['c'], t['d']))
            solver.add_assertion(Not(inv(trace[k-1])))
            if solver.solve():
                print(f"A propriedade não é valida para o seguinte traço de tamanho <= %d" % k)
                for traco in trace:
                    print("Estado:",i)
                    i+=1
                    for v in traco:
                        print("          ",v,'=',solver.get_value(traco[v]))
                return
    print(f"Propriedade válida para traços de tamanho <= %d." % k)
```

Exemplos

In [12]:
```python
t = {'a': True, 'b': True, 'c': True, 'd': True}
s = {'a': False, 'b': True, 'c': True, 'd': True}
bmc_always(genState,init_notrandom,trans1,inv1,5,t,s)
```

```
Propriedade válida para traços de tamanho <= 5.
```

O seguinte caso é Inseguro, mas não se o consegue verificar no limite (traços de tamanho 14).

In [13]:
```python
t = {'a': True, 'b': True, 'c': True, 'd': False}
s = {'a': True, 'b': True, 'c': True, 'd': False}
bmc_always(genState,init_notrandom,trans1,inv1,14,t,s)
```

```
Propriedade válida para traços de tamanho <= 14.
```

In [14]:
```python
bmc_always(genState,init_notrandom,trans1,inv1,16,t,s)
```

```
A propriedade não é valida para o seguinte traço de tamanho <= 15
Estado: 13
           s_a = True
           s_b = True
           s_c = True
           s_d = False
Estado: 14
           s_a = False
           s_b = False
           s_c = True
           s_d = True
Estado: 15
           s_a = False
           s_b = True
           s_c = False
           s_d = True
Estado: 16
           s_a = True
           s_b = True
           s_c = False
           s_d = False
Estado: 17
           s_a = True
           s_b = False
           s_c = True
           s_d = True
Estado: 18
           s_a = False
           s_b = False
           s_c = False
           s_d = True
Estado: 19
           s_a = True
           s_b = True
           s_c = False
           s_d = True
Estado: 20
           s_a = True
           s_b = False
           s_c = False
           s_d = False
Estado: 21
           s_a = True
           s_b = False
           s_c = True
           s_d = False
Estado: 22
           s_a = False
           s_b = False
           s_c = True
           s_d = False
Estado: 23
           s_a = False
           s_b = True
           s_c = True
           s_d = False
Estado: 24
           s_a = False
           s_b = True
           s_c = True
           s_d = True
Estado: 25
           s_a = False
           s_b = True
           s_c = False
           s_d = False
Estado: 26
           s_a = True
           s_b = True
           s_c = True
           s_d = True
Estado: 27
           s_a = False
           s_b = False
           s_c = False
           s_d = False
```

In [15]:
```python
t = genTransRandom()
s = genInitialRandom()
bmc_always(genState,init_notrandom,trans1,inv1,14,t,s)
```

```
Transição Determinada: Neg Neg XOR Neg
Estados inciais: True True False False
A propriedade não é valida para o seguinte traço de tamanho <= 7
Estado: 5
           s_a = True
           s_b = True
           s_c = False
           s_d = False
Estado: 6
           s_a = True
           s_b = False
           s_c = False
           s_d = False
Estado: 7
           s_a = True
           s_b = False
           s_c = False
           s_d = True
Estado: 8
           s_a = True
           s_b = False
           s_c = True
           s_d = True
Estado: 9
           s_a = False
           s_b = False
           s_c = False
           s_d = True
Estado: 10
           s_a = True
           s_b = True
           s_c = True
           s_d = True
Estado: 11
           s_a = False
           s_b = False
           s_c = False
           s_d = False
```

Indução

O Invariante `inv1` não é forte o suficiente para conseguir provar a condição por indução, já que não impossibilita o estado $(0,0,0,0)$ de ser acessível, por isso foi utilizado *k-indução*.

In [16]:
```python
def induction_always(vars,gen_state,init,trans,inv,s,t):
    with Solver(name="z3") as solver:
        state0 = gen_state(vars, 'X', 0)
        state1 = gen_state(vars, 'X', 1)
        solver.push()
        solver.add_assertion(And(init(state0,s['a'],s['b'],s['c'],s['d']), Not(inv(state0))))
        if solver.solve():
            print("A propriedade não é válida no estado inicial.")
            for v in state0:
                print(v,"=",solverget_value(state0[v]))
            return
        solver.pop()

        solver.add_assertion(And(inv(state0), trans(state0,state1,
                                                     t['a'],t['a'],t['a'],t['a']), Not(inv(state1))))
        if solver.solve():
            print("O passo indutivo não preserva a propriedade.")
            for v in state0:
                print(v,"=",solver.get_value(state0[v]))
            return
        print("A propriedade é valida.")
```

O seguinte exemplo é claramente Seguro.

In [17]:
```python
t = {'a': True, 'b': True, 'c': True, 'd': True}
s = {'a': False, 'b': True, 'c': True, 'd': True}
induction_always(vars, genState, init_notrandom, trans1, inv1,s,t)
```

```
O passo indutivo não preserva a propriedade.
s_a = True
s_b = True
s_c = True
s_d = True
```

K-Indução

Como descrito nas aulas teóricas e práticas, o seguinte algoritmo expande o conceito da indução para k passos de transição, testando também k estados iniciais.

In [18]:
```python
def kinduction_always(vars, gen_state,init,trans,inv,k,s,t):
    with Solver(name="z3") as solver:
        traco = [gen_state(vars, 'X', i) for i in range(k+1) ]
        solver.push()
        solver.add_assertion( init(traco[0], s['a'], s['b'], s['c'], s['d'])  )
        for e in range(k-1):
            solver.add_assertion( trans (traco[e], traco[e+1], t['a'], t['b'], t['c'], t['d'] ) )
        solver.add_assertion( Or([Not(inv(traco[i])) for i in range(k) ]) )

        if solver.solve():
            print("A propriedade não é valida para os k estados iniciais")
            i = 0
            for trace in traco:
                print("Estado:",i)
                i+=1
                for v in trace:
                    print("          ",v,'=',solver.get_value(trace[v]))
            return
        solver.pop()

        solver.push()
        for e in range(k):
            solver.add_assertion( trans (traco[e], traco[e+1], t['a'], t['b'], t['c'], t['d'] ) )
            solver.add_assertion( inv(traco[e]) )

        solver.add_assertion( Not(inv(traco[k])) )

        if solver.solve():
            print(f"O passo indutivo %d não preserva a propriedade" % k)
            i = 0
            for trace in traco:
                print("Estado:",i)
                i += 1
                for v in trace:
                    print("          ",v,'=',solver.get_value(trace[v]))
            return
        solver.pop()

        print("A propriedade é valida para todos os estados acessiveis ")
```

Exemplos

Como os traços são limitados, basta testar *k-indução* para $k = 15$, já que ora o 16º estado é a repetição do primeiro (loop) ou entretanto já encontrou o estado de erro. Este teste é o pior caso do k, sendo possível encontrar k mais pequenos que provem o mesmo.

Este tipo de teste tem falsos negativos, pela falta de acessibilidade de casos que considera ser traços inseguros. No entanto, não regista falsos positivos.

In [19]:
```python
t = {'a': True, 'b': True, 'c': True, 'd': True}
s = {'a': False, 'b': True, 'c': True, 'd': True}
kinduction_always(vars, genState,init_notrandom,trans1,inv1,2,s,t)
```

```
A propriedade é valida para todos os estados acessiveis
```

In [20]:
```python
t = {'a': True, 'b': False, 'c': True, 'd': False}
s = {'a': False, 'b': False, 'c': True, 'd': False}
kinduction_always(vars, genState,init_notrandom,trans1,inv1,5,s,t)
```

```
O passo indutivo 5 não preserva a propriedade
Estado: 0
           s_a = True
           s_b = False
           s_c = True
           s_d = False
Estado: 1
           s_a = False
           s_b = True
           s_c = True
           s_d = False
Estado: 2
           s_a = False
           s_b = True
           s_c = True
           s_d = True
Estado: 3
           s_a = False
           s_b = True
           s_c = False
           s_d = False
Estado: 4
           s_a = True
           s_b = True
           s_c = True
           s_d = True
Estado: 5
           s_a = False
           s_b = False
           s_c = False
           s_d = False
```

In [21]:
```python
kinduction_always(vars, genState,init_notrandom,trans1,inv1,16,s,t)
```

```
A propriedade é valida para todos os estados acessiveis
```

In [22]:
```python
kinduction_always(vars, genState,init_notrandom,trans1,inv1,6,s,t)
```

```
A propriedade é valida para todos os estados acessiveis
```

In [23]:
```python
t = genTransRandom()
s = genInitialRandom()
kinduction_always(vars, genState,init_notrandom,trans1,inv1,10,s,t)
```

```
Transição Determinada: XOR Neg XOR Neg
Estados inciais: False False False False
A propriedade não é valida para os k estados iniciais
Estado: 0
           s_a = False
           s_b = False
           s_c = False
           s_d = False
Estado: 1
           s_a = False
           s_b = True
           s_c = False
           s_d = True
Estado: 2
           s_a = False
           s_b = True
           s_c = True
           s_d = False
Estado: 3
           s_a = True
           s_b = True
           s_c = True
           s_d = False
Estado: 4
           s_a = False
           s_b = False
           s_c = True
           s_d = False
Estado: 5
           s_a = True
           s_b = True
           s_c = True
           s_d = True
Estado: 6
           s_a = False
           s_b = False
           s_c = False
           s_d = False
Estado: 7
           s_a = False
           s_b = True
           s_c = False
           s_d = True
Estado: 8
           s_a = False
           s_b = True
           s_c = True
           s_d = False
Estado: 9
           s_a = True
           s_b = True
           s_c = True
           s_d = False
Estado: 10
           s_a = False
           s_b = False
           s_c = False
           s_d = False
```

Algoritmo de "model-checking" usando interpolantes e invariantes

O algorimto que utiliza interpolantes aqui descrito é o mesmo implementado nas aulas práticas.

In [24]:
```python
def invert(trans,a,b,c,d):
    return (lambda curr,prox: trans(prox, curr, a, b, c, d))
```

In [25]:
```python
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([Iff(state1[x],state2[x]) for x in state1])
```

In [26]:
```python
def model_checking(vars,init,trans,error,N,M,s,t):
    with Solver(name="msat") as solver:

        # 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])

        #falta testar para n = 0 e m = 0

        for (n,m) in order:
            I = init(X[0],s['a'],s['b'],s['c'],s['d'])
            E = error(Y[0])
            Tn = And([trans(X[i], X[i+1],t['a'],t['b'],t['c'],t['d']) for i in range(n)])
            Bm = And([invert(trans,t['a'],t['b'],t['c'],t['d'])(Y[j], Y[j+1]) for j in range(m)])
            Rn = And(I, Tn)
            #print(Rn)
            Um = And(E, Bm)
            Vnm = And(Rn, Um, same(X[n], Y[m]))

            #1º Passo
            if solver.solve([Vnm]):
                print("Unsafe!")
                return

            #2º Passo
            C = binary_interpolant(And(Rn, same(X[n], Y[m])), Um)
            if n == 1 and m == 1:
                print("C:", C)
            if C is None:
                print("Interpolante None!")
                continue
            #3ª Passo
            C0 = rename(C, X[0])
            C1 = rename(C, X[1])
            T = trans(X[0], X[1],t['a'],t['b'],t['c'],t['d'])
            if not solver.solve([C0, T, Not(C1)]):
                print("Safe!")
                return

            #4º Passo - gerar o S
            S = rename(C, X[n])
            while True:
                A = And(S, trans(X[n], Y[m],t['a'],t['b'],t['c'],t['d']))
                if solver.solve([A, Um]):
                    print("Não se encontrou o Majorante.")
                    break
                Cnew = binary_interpolant(A, Um)
                if Cnew is None:
                    print("Interpolante None!")
                    break
                Cn = rename(Cnew, X[n])
                if solver.solve([Cn, Not(S)]):
                    S = Or(S, Cn)
                else:
                    print("Safe!")
                    return

        print("unknown")
```

Exemplos

De facto, com os casos gerados, esta função parece não ser suficiente para conseguir provar que é seguro.

Os seguintes dados iniciais geram um sistema seguro, mas a geração dos interpolantes apenas gera condições que não são limitantes o suficiente, incapacitando a geração de um majorante que não intersete com U_m. Tal verifica-se nos casos que supostamente são seguros, como quando as transições são todas do tipo `neg` e apenas uma das variáveis do estado inicial é 0

O interpolante sendo só $\neg X1_c$ faz com que seja possível considerar X_n como o estado $(0,0,0,0)$, que é claramente inseguro. Para $X1 = (0,0,1,0)$ e $Y1 = (1,1,1,1)$, com todas as transições definidas para `neg`, um interpolante não será forte o suficiente para excluir a possibilidade de Xn ser $(0,0,0,0)$.

In [27]:
```python
t = {'a':True,'b':True,'c':True,'d':True}
s = {'a':True,'b':True,'c':True,'d':False}
genTrace(vars,init_notrandom,trans1,2,s,t)
kinduction_always(vars, genState,init_notrandom,trans1,inv1,2,s,t)
model_checking(vars, init_notrandom, trans1, error1, 5, 5,s,t)
```

```
Estado: 0
           s_a = True
           s_b = True
           s_c = True
           s_d = False
Estado: 1
           s_a = False
           s_b = False
           s_c = True
           s_d = False
A propriedade é valida para todos os estados acessiveis
C: (! s_a!Y1)
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
unknown
```

É possível de se provar a impossibilidade de calcular um interpolante que seja um invariante pretendido para o caso que todas as operações sejam negações, estudando atentamente a definição do algoritmo da geração do interpolante e as características dos estados de erro (nomeadamente, que vão alternar entre $(0,0,0,0)$ e $(1,1,1,1)$).

No caso de ser inseguro, não registamos nenhuma ocorrência de não conseguir chegar a essa conclusão.

In [28]:
```python
t = {'a':True,'b':True,'c':True,'d':False}
s = {'a':True,'b':False,'c':True,'d':False}
model_checking(vars, init_notrandom, trans1, error1, 5, 5,s,t)
```

```
C: (! s_a!Y1)
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Unsafe!
```

In [32]:
```python
t = genTransRandom()
s = genInitialRandom()
model_checking(vars, init_notrandom, trans1, error1, 10, 10,s,t)
```

```
Transição Determinada: Neg Neg XOR XOR
Estados inciais: False False False True
Unsafe!
```

Tendo em conta a consideração que o número de elementos distintos máximo é 15, parece-nos plausível considerar os *unknown* como *safe* para o caso em que se verificaram todos os $n < 16$ ou $m < 16$. Isto dá-se porque considera-se que já se testaram todos os estados distintos de algum dos traços e nenhum levou à acessibilidade de um estado inseguro.

In [31]:
```python
t = {'a':True,'b':True,'c':True,'d':True}
s = {'a':True,'b':True,'c':True,'d':False}
model_checking(vars, init_notrandom, trans1, error1, 16, 1,s,t)
```

```
C: (! s_a!Y1)
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
Não se encontrou o Majorante.
unknown
```

In [ ]: