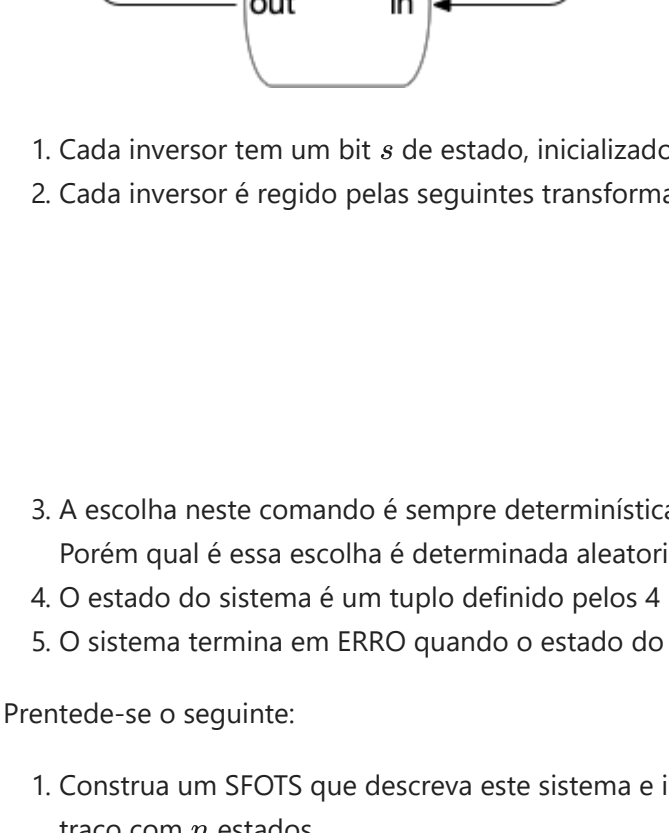


# TP3 - Grupo 14

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 têm um bit num canal input e escrevem num canal output uma transformação desse bit.



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

$$\text{invert}(in, out)$$
$$x \leftarrow \text{read}(in)$$
$$s \leftarrow x \oplus s$$
$$\text{write}(out, s)$$
- 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.

- O estado do sistema é um tuplo defendido das 4 bits  $s$ , e é inicializado com um valor aleatório em  $\{0, 1\}^4$ .
- O sistema termina em ERRO quando o estado do sistema for  $(0, 0, 0, 0)$ .

Prentede-se o seguinte:

- Construa um SFOTS que descreva este sistema e implemente este sistema, numa abordagem BMC ("bounded model checker") num traço com  $n$  estados.
- 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,  $a \rightarrow b \rightarrow d \rightarrow c$ .

No caso do estado inicial, decidiu-se gerar o *in* aleatoriamente ao mesmo que 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:

$$\Sigma \equiv \langle \mathcal{T}, X, \text{start}, I, T, E \rangle$$

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

- $\mathcal{T}$  representa um SMT apropriada, que pertence à FOL, que vamos representar no nosso Solver;
- $X$  é o conjunto das variáveis base do problema;
- $\text{start}$  é um operador que gera "clones" das variáveis em  $X$ ;
- $I$  é um predicado unário que determina quais os estados iniciais;
- $T$  é um predicado unário que determina as transições entre dois estados;
- $E$  é um predicado binário 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 cíclica, um dos estados de um dos inversores será considerado o valor de entrada de outro. Ou seja,  $in_a = s_a$ ,  $in_b = s_a$ ,  $in_c = s_d$  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  $A$ :

$$T_a \equiv s'_a = -s_c \quad \text{ou} \quad T_a \equiv s'_a = s_b \oplus s_c$$

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

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

## 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. Importamos-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]: 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]: def genState(vars,s,i):
    state = {}
    for v in vars:
        state[v] = Symbol('s'+str(i)+'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]: 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 iniciais:",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
    neg = "Neg"
    xor = "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]: vars = ['s_a','s_b','s_c','s_d']
in_a = s_a, in_b = s_a, in_c = s_d, in_d = s_b

A função inicial define dos valores aleatórios gerados, apenas limitando cada uma das variáveis a esse valor, através de uma equivalência. init 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]: def init(state):
    s = genInitialRandom()
    return And(If(state['s_a'], Bool(s['a'])), If(state['s_b'], Bool(s['b'])),
               If(state['s_c'], Bool(s['c'])), If(state['s_d'], Bool(s['d'])))

def init_notrandom(state,a,b,c,d):
    return And(If(state['s_a'], Bool(a)), If(state['s_b'], Bool(b)),
               If(state['s_c'], Bool(c)), If(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]: def error1(state):
    return And(If(state['s_a'], Bool(False)), If(state['s_b'], Bool(False)),
               If(state['s_c'], Bool(False)), If(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  $s \leftarrow \neg s$  e False a transição  $s \leftarrow s \oplus s$ .
```

```
In [41]: def trans1(curr, prox,s_a,b,c,d):
    t_a = If(Not(curr['s_a']), prox['s_a']) if a else If(Xor(curr['s_a'], curr['s_a']), prox['s_a'])
    t_b = If(Not(curr['s_b']), prox['s_b']) if b else If(Xor(curr['s_b'], curr['s_b']), prox['s_b'])
    t_c = If(Not(curr['s_c']), prox['s_c']) if c else If(Xor(curr['s_c'], curr['s_c']), prox['s_c'])
    t_d = If(Not(curr['s_d']), prox['s_d']) if d else If(Xor(curr['s_d'], 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]: def genTrace(vars,gen_state,init,trans,inv,k,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)
        t = init(X[0])
        Tks = [trans(X[i],X[i+1],t['a'],t['b'],t['c'],t['d']) for i in range(n)]

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

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

Transição Determinada: Neg Neg XOR XOR
Estados iniciais: False False True True
Propriedade válida para traços de tamanho <= 15
```

Estado: 0

s\_a = False  
s\_b = False  
s\_c = True  
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 = True  
s\_c = False  
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 aleatórias, apenas com duas diferenças. Por um lado, recebem dois dicionários,  $s$  e  $t$ , definidos tal como nas funções `genState`, para determinar o estado inicial e as transições de uma maneira mais sistemática. Por outro, geram estados a partir da função `state1`.

O invariante que pretendemos provar é o seguinte.

```
In [10]: def inv(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]: def bmc_always(declare,init,trans,inv,k,t,s):
    for i 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']))
            solver.add_assertion(inv(trace[k-1]))
            if solver.solve():
                print(f"A propriedade não é válida para o seguinte traço de tamanho <= {k}")
                for trace in trace:
                    print("Estado:",i)
                    prox = X[i]
                    "v,v'=",solver.get_value(trace[v])
            print(f"Propriedade válida para traços de tamanho <= {k}")
```

## Exemplos

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

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

```
In [14]: bmc_always(genState,init_notrandom,trans1,inv1,6,t,s)

A propriedade não é válida 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 = False

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 = True  
s\_d = True

Estado: 26

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = False

Estado: 27

s\_a = False  
s\_b = False  
s\_c = False  
s\_d = False

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

Transição Determinada: XOR Neg XOR XOR
Estados iniciais: True False False True
Propriedade válida para traços de tamanho <= 14.
```

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

O seguinte exemplo é claramente Seguro.

```
In [17]: 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
print("None")
```

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

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

        print("A propriedade é válida para todos os estados acessíveis")
```

## Exemplos

Como os traços são limitados, basta testar  $k$ -indução para  $k = 15$ , já que o ora 0<sup>o</sup> estado é a repetição do primeiro (loop) ou entretanto já encontrou o estado de erro. Este teste é o pior caso de  $k$ , 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 [36]: t = {'a': True, 'b': True, 'c': True, 'd': True}
s = {'a': False, 'b': True, 'c': True, 'd': False}
kinduction_always(vars, genState,init_notrandom,trans1,inv1,2,s,t)

A propriedade é válida para todos os estados acessíveis
```

```
In [37]: 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 = True  
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 [38]: kinduction\_always(vars, genState,init\_notrandom,trans1,inv1,6,s,t)  
A propriedade é válida para todos os estados acessíveis

In [39]: kinduction\_always(vars, genState,init\_notrandom,trans1,inv1,6,s,t)  
A propriedade é válida para todos os estados acessíveis

In [22]: t = genTransRandom()
s = genInitialRandom()
kinduction\_always(vars, genState,init\_notrandom,trans1,inv1,10,s,t)

Transição Determinada: Neg Neg XOR Neg
Estados iniciais: False True False False
A propriedade não é válida para os k estados iniciais
Estado: 0

s\_a = False  
s\_b = True  
s\_c = False  
s\_d = False

Estado: 1

s\_a = True  
s\_b = False  
s\_c = True  
s\_d = True

Estado: 2

s\_a = True  
s\_b = True  
s\_c = False  
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

Estado: 6

s\_a = True  
s\_b = True  
s\_c = False  
s\_d = True

Estado: 7

s\_a = True  
s\_b = False  
s\_c = False  
s\_d = False

Estado: 8

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 9

s\_a = False  
s\_b = False  
s\_c = False  
s\_d = False

Estado: 10

s\_a = False  
s\_b = True  
s\_c = False  
s\_d = False

Estado: 11

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 12

s\_a = False  
s\_b = False  
s\_c = False  
s\_d = False

Estado: 13

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 14

s\_a = False  
s\_b = True  
s\_c = False  
s\_d = False

Estado: 15

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 16

s\_a = False  
s\_b = False  
s\_c = False  
s\_d = False

Estado: 17

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 18

s\_a = False  
s\_b = True  
s\_c = False  
s\_d = False

Estado: 19

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 20

s\_a = False  
s\_b = False  
s\_c = False  
s\_d = False

Estado: 21

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 22

s\_a = False  
s\_b = True  
s\_c = False  
s\_d = False

Estado: 23

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 24

s\_a = False  
s\_b = False  
s\_c = False  
s\_d = False

Estado: 25

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 26

s\_a = False  
s\_b = True  
s\_c = False  
s\_d = False

Estado: 27

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 28

s\_a = False  
s\_b = False  
s\_c = False  
s\_d = False

Estado: 29

s\_a = True  
s\_b = True  
s\_c = True  
s\_d = True

Estado: 30

s\_a = False  
s\_b = True  
s\_c = False  
s\_d = False