FOTS

1 First Order Transition Systems

Grupo 7:

- Luís Almeida A84180
- João Pedro Antunes A86813

1. O seguinte sistema dinâmico denota 4 inversores (A, B, C, D) que lêm 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

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

3. O sistema termina quando todos os inversores tiverem o estado

$$s = 0$$

a) Construa um FOTS que descreva este sistema e implemente este sistema, numa abordagem BMC ("bouded model checker") num traço com n estados.

b) Verifique usando k-lookahead se o sistema termina ou, em alternativa,

c) Explore as técnicas que estudou para verificar em que condições o sistema termina.

1.1 Construção do FOTS e a sua implementação

Um FOTS é um modelo de sistemas dinâmicos constituído por uma SMT τ, um vetor de variáveis X onde para cada variável $x \in X, x \in \tau$, um predicado unário *init* que atua sobre o vetor X para determinar os estados iniciais do sistema e um predicado binário *trans* que indica como é possível o estado X transitar para o estado X'. Como tal, comecemos por verificar quais as variáveis presentes no sistema:

- Temos 4 inversores (A, B, C, D), cada um com o seu canal de *input* e o seu canal de *output*. Estes canais podem ser vistos como variáveis do sistema, pelo que as vamos incluir no FOTS
- Cada inversor tem também uma variável x, lida do seu canal *input* e uma variável s, que indica o estado do inversor. Acrescentemos então estas variáveis ao FOTS

Temos agora que escolher a SMT que iremos utilizar para modelar as variáveis do sistema apresentado. Dado que todas as variáveis do sistema terão o valor 0 ou 1, faz sentido utilizarmos Lógica Inteira Binária para as modelar. De seguida apresenta-se uma função que declara as variáveis de um certo estado i do sistema:

[1]:
```
from z3 import *
import random as rn

def declare(i):
    state = {}

    state['inA'] = Int('inA'+str(i))
    state['inB'] = Int('inB'+str(i))
    state['inC'] = Int('inC'+str(i))
    state['inD'] = Int('inD'+str(i))

    state['sA'] = Int('sA'+str(i))
    state['sB'] = Int('sB'+str(i))
    state['sC'] = Int('sC'+str(i))
    state['sD'] = Int('sD'+str(i))

    state['xA'] = Int('xA' + str(i))
    state['xB'] = Int('xB' + str(i))
    state['xC'] = Int('xC' + str(i))
    state['xD'] = Int('xD' + str(i))

    state['outA'] = Int('outA'+str(i))
    state['outB'] = Int('outB'+str(i))
    state['outC'] = Int('outC'+str(i))
    state['outD'] = Int('outD'+str(i))

    return state
```

Passemos então à definição do predicado *init* para obtermos um estado inicial do FOTS. Observando as transformações de cada inversor, verifica-se que a variável x de cada inversor toma o valor do seu *input*, pelo que podemos adicionar as seguintes conjunções:

$$x_A == in_A \land x_B == in_B \land x_C == in_C \land x_D == in_D$$

De seguida, temos que o *input* do inversor A é o *output* do inversor C, o *input* do inversor C é o *output* do inversor D, o *input* do inversor D é o *output* do inversor B e o *input* do inversor B é o *output* do inversor A:

$$in_A == out_C \land in_C == out_D \land in_D == out_B \land in_B == out_A$$

Falta-nos então inicializar com um bit aleatório as variáveis s de cada inversor. Como sabemos que cada uma destas variáveis tem o valor de 0 ou 1, adiciona-se o seguinte predicado:

$$0 \leq s_A \leq 1 \land 0 \leq s_B \leq 1 \land 0 \leq s_C \leq 1 \land 0 \leq s_D \leq 1$$

O predicado *init* será então a conjunção destes predicados apresentados:

[2]:
```
def init(state):

    return And(0 <= state['sA'], state['sA'] <= 1, 0 <= state['sB'], state['sB']␣
↪<= 1,
               0 <= state['sC'], state['sC'] <= 1, 0 <= state['sD'], state['sD']␣
↪<= 1,
               state['outA'] == state['sA'], state['outB'] == state['sB'],
               state['outC'] == state['sC'], state['outD'] == state['sD'],
               state['inA'] == state['outC'], state['inB'] == state['outA'],
               state['inC'] == state['outD'], state['inD'] == state['outB'],
               state['xA'] == state['inA'], state['xB'] == state['inB'],
               state['xC'] == state['inC'], state['xD'] == state['inD'])
```

Para definirmos o predicado binário *trans* que permite a transição entre os diferentes estados do FOTS precisamos de distinguir o estado final dos estados restantes, pois a partir do estado final apenas é possível transitar para ele próprio. O FOTS atinge o estado final quando todos os bits de estado dos diferentes inversores são iguais a 0. Portanto, para restringirmos a transição em que as variáveis de estado do FOTS se mantêm inalteradas ao estado final, podemos simplesmente adicionar a "condição":

$$s_A == 0 \land s_B == 0 \land s_C == 0 \land s_D == 0$$

Tendo em conta as transformações especificadas em cada inversor, podemos começar por estabelecer os predicados que determinam o próximo valor que cada variável do FOTS vai tomar. Olhemos primeiro para o que acontece em cada canal de *input* de cada inversor. Verifica-se que o canal de *input* do inversor A vai tomar o valor do próximo *output* do inversor C, o canal de *input* do inversor B toma o valor do *output* do inversor A, o canal de *input* do inversor D toma o valor do *output* do inversor B e que o *input* do inversor C toma o valor do *output* do inversor D. Temos então o seguinte predicado:

$$in'_A == out'_C \quad \wedge \quad in'_B == out'_A \quad \wedge \quad in'_D == out'_B \quad \wedge \quad in'_C == out'_D$$

Observa-se também que a variável x toma o valor do canal $input$ em todos os inversores. Como tal, o próximo valor da variável x será o próximo valor do canal de $input$. Temos portanto as seguintes conjunções:

$$x'_A == in'_A \quad \wedge \quad x'_B == in'_B \quad \wedge \quad x'_C == in'_C \quad \wedge \quad x'_D == in'_D$$

Para obtermos o próximo valor da variável s, que representa o estado do inversor, a transformação apresentada é uma escolha não determinística entre o valor atual de s e a negação da variável x. Ora, como a negação de uma variável x em lógica inteira binária é dada por $1 - x$, podemos utilizar a função Or() do Z3 para implementar a escolha não determinística entre os predicados $s' == s$ e $s' == 1 - x$, e assim implementar o predicado que modela a transformação apresentada:

$$(s'_A == s_A \quad \vee \quad s'_A == 1 - x_A) \quad \wedge \quad (s'_B == s_B \quad \vee \quad s'_B == 1 - x_B) \quad \wedge \quad (s'_C == s_C \quad \vee \quad s'_C == 1 - x_C)$$
$$\wedge \quad (s'_D == s_D \quad \vee \quad s'_D == 1 - x_D)$$

Resta-nos apenas determinar o próximo valor do canal de $output$ de cada inversor. Ora, a última transformação refere-nos que é o valor da variável s escrito no canal de $output$ em cada inversor, pelo que o canal de $output$ de cada inversor toma o valor da variável s. Temos então o seguinte predicado:

$$out'_A == s'_A \quad \wedge \quad out'_B == s'_B \quad \wedge \quad out'_C == s'_C \quad \wedge \quad out'_D == s'_D$$

O predicado que determina a transição entre estados "não-finais" será então uma conjunção destes predicados, sendo o predicado *trans* uma disjunção entre o predicado que determina a transição entre estados "não-finais" e o predicado que determina a transição para o estado final. De seguida apresenta-se a implentação do predicado *trans*:

[3]:
```
def trans(curr,prox):
    T = []

    #Estado final transita para ele próprio
    T.append(And(curr['sA'] == 0, curr['sB'] == 0, curr['sC'] == 0, curr['sD']␣
↪== 0, prox['sA'] == curr['sA'],
                 prox['sB'] == curr['sB'], prox['sC'] == curr['sC'], prox['sD']␣
↪== curr['sD'],
                 prox['xA'] == curr['xA'], prox['xB'] == curr['xB'], prox['xC']␣
↪== curr['xC'],
                 prox['xD'] == curr['xD'], prox['inA'] == curr['inA'],␣
↪prox['inB'] == curr['inB'],
                 prox['inC'] == curr['inC'], prox['inD'] == curr['inD'],␣
↪prox['outA'] == curr['outA'],
```

```
                prox['outB'] == curr['outB'], prox['outC'] == curr['outC'],␣
↪prox['outD'] == curr['outD']))

    #Todos os inversores operam ao mesmo tempo
    T.append(And(Sum(curr['sA'], curr['sB'], curr['sC'], curr['sD']) >= 1,
                 prox['inA'] == prox['outC'], prox['inB'] == prox['outA'],  #␣
↪leitura do input
                 prox['inC'] == prox['outD'], prox['inD'] == prox['outB'],
                 prox['xA'] == prox['inA'], prox['xB'] == prox['inB'],     #␣
↪atribuição ao x
                 prox['xC'] == prox['inC'], prox['xD'] == prox['inD'],
                 Or(prox['sA'] == 1 - curr['xA'], prox['sA'] == curr['sA']), #␣
↪escolha do s
                 Or(prox['sB'] == 1 - curr['xB'], prox['sB'] == curr['sB']),
                 Or(prox['sC'] == 1 - curr['xC'], prox['sC'] == curr['sC']),
                 Or(prox['sD'] == 1 - curr['xD'], prox['sD'] == curr['sD']),
                 prox['outA'] == prox['sA'], prox['outB'] == prox['sB'],    ␣
↪#escrita do s
                 prox['outC'] == prox['sC'], prox['outD'] == prox['sD']
                ))

    return Or(T)
```

Adotando uma abordagem BMC, em que os traços do FOTS são restringidos aos traços limitados e com k estados distintos, podemos definir uma função que gera um traço (com um dado tamanho k) do FOTS construído. Começa-se por declarar k estados, inicia-se o estado α_0 inicial com o predicado *init* e força-se

$$trans(\alpha_i, \alpha_{i+1}), \quad \forall 0 \leq i \leq k-1$$

[4]:
```
def gera_traco(declare,init,trans,k):
    trace = [declare(i) for i in range(k)]
    s = Solver()

    s.add(init(trace[0]))

    for i in range(k-1):
        s.add(trans(trace[i],trace[i+1]))

    r = s.check()
    if r == sat:
        m = s.model()
        for i in range(k):
            print(i)
            for v in trace[i]:
                print(v, '=', m[trace[i][v]])
```

```python
        return

    print('UNSAT')
    return


gera_traco(declare,init,trans,4)
```

```
0
inA = 1
inB = 0
inC = 1
inD = 0
sA = 0
sB = 0
sC = 1
sD = 1
xA = 1
xB = 0
xC = 1
xD = 0
outA = 0
outB = 0
outC = 1
outD = 1
1
inA = 0
inB = 0
inC = 1
inD = 1
sA = 0
sB = 1
sC = 0
sD = 1
xA = 0
xB = 0
xC = 1
xD = 1
outA = 0
outB = 1
outC = 0
outD = 1
2
inA = 0
inB = 1
inC = 0
inD = 1
```

```
sA = 1
sB = 1
sC = 0
sD = 0
xA = 0
xB = 1
xC = 0
xD = 1
outA = 1
outB = 1
outC = 0
outD = 0
3
inA = 1
inB = 1
inC = 0
inD = 1
sA = 1
sB = 1
sC = 1
sD = 0
xA = 1
xB = 1
xC = 0
xD = 1
outA = 1
outB = 1
outC = 1
outD = 0
```

1.2 Verificação da Terminação do Sistema

A propriedade que queremos verificar é a terminação do sistema, informalmente, queremos saber se "é inevitável que o sistema termine". Ora, este enunciado pode ser traduzido para *Bounded Temporal Logic* com recurso ao operador F da seguinte forma:

$$F(s_A == 0 \land s_B == 0 \land s_C == 0 \land s_D == 0)$$

Queremos portanto fazer a verificação de uma propriedade de animação utilizando *Bounded Model Checking*. Como tal, não basta provar que a fórmula $s_A == 0 \land s_B == 0 \land s_C == 0 \land s_D == 0$ nunca é válida nos primeiros k estados do traço, pois nada impediria que esta fórmula fosse válida num estado futuro do traço. Por esta razão, apenas podemos provar a validade (ou invalidade) semântica da fórmula apresentada em traços limitados e com k estados diferentes, ou seja, em traços onde exista uma transição do estado final α_{k-1} para o estado α_i, sendo $0 \leq i < k-1$ (um *loop*), pois esta condição permite-nos determinar todos os estados futuros a partir do *loop*. Ora, caso $\neg(s_A == 0 \land s_B == 0 \land s_C == 0 \land s_D == 0)$ seja válida para um qualquer traço $\alpha \in FOTS$ nas condições referidas anteriormente, então um qualquer traço servirá como um contra exemplo que refutará a validade semântica de $F(s_A == 0 \land s_B == 0 \land s_C == 0 \land s_D == 0)$, e saberemos que o sistema não termina. De seguida, apresenta-se uma função que implementa o procedimento *BMC* para traços limitados de k estados de um dado FOTS:

[5]:
```
def bmc_eventually(declare,init,trans,prop,K):
    s = Solver()
    trace = [declare(i) for i in range(K)]

    s.add(init(trace[0]))

    for i in range(K-1):
        s.add(trans(trace[i],trace[i+1]))

    s.add(Or([trans(trace[K-1],trace[i]) for i in range(K)]))

    s.add(Not(prop(trace[K-1])))

    if s.check() == sat:
        print('Property is invalid')
        m = s.model()

        for k in range(K):
            if m.eval(trans(trace[K-1],trace[k])):
                print('Loop starts here:')
                for v in trace[i]:
                    print(v,'=',m[trace[i][v]])
                break

        for i in range(K):
            print(i)
            for v in trace[i]:
                print(v,'=',m[trace[i][v]])
        return

    print("Property is valid")
    return

def terminates(state):
    return And(state['sA'] == 0, state['sB'] == 0, state['sC'] == 0, state['sD']␣
 ↪== 0)

bmc_eventually(declare,init,trans,terminates,4)
```

```
Property is invalid
Loop starts here:
inA = 1
inB = 0
inC = 0
```

```
inD = 0
sA = 0
sB = 0
sC = 1
sD = 0
xA = 1
xB = 0
xC = 0
xD = 0
outA = 0
outB = 0
outC = 1
outD = 0
0
inA = 0
inB = 0
inC = 0
inD = 1
sA = 0
sB = 1
sC = 0
sD = 0
xA = 0
xB = 0
xC = 0
xD = 1
outA = 0
outB = 1
outC = 0
outD = 0
1
inA = 1
inB = 1
inC = 0
inD = 1
sA = 1
sB = 1
sC = 1
sD = 0
xA = 1
xB = 1
xC = 0
xD = 1
outA = 1
outB = 1
outC = 1
outD = 0
2
```

```
inA = 1
inB = 0
inC = 0
inD = 0
sA = 0
sB = 0
sC = 1
sD = 0
xA = 1
xB = 0
xC = 0
xD = 0
outA = 0
outB = 0
outC = 1
outD = 0
3
inA = 1
inB = 0
inC = 0
inD = 1
sA = 0
sB = 1
sC = 1
sD = 0
xA = 1
xB = 0
xC = 0
xD = 1
outA = 0
outB = 1
outC = 1
outD = 0
```

Antes de passarmos para a prova seguinte, podemos criar uma função que calcule os traços em que o sistema termina. Tal pode ser feito acrescentando ao procedimento adotado na função *gera_traco* a restrição de que o último estado do traço é necessariamente o estado final. Temos então a função *gera_term*:

[6]:
```python
def gera_term(declare,init,trans,k):
    s = Solver()
    trace = [declare(i) for i in range(k)]

    s.add(init(trace[0]))

    for i in range(k-1):
        s.add(trans(trace[i],trace[i+1]))
```

```
    s.add(trace[k-1]['sA'] == 0,trace[k-1]['sB'] == 0,trace[k-1]['sC'] ==␣
↪0,trace[k-1]['sD'] == 0)

    r = s.check()
    if r == sat:
        m = s.model()
        for i in range(k):
            print(i)
            for v in trace[i]:
                print(v, '=', m[trace[i][v]])
        return

    print('UNSAT')
    return

gera_term(declare,init,trans,3)
```

```
0
inA = 1
inB = 1
inC = 1
inD = 1
sA = 1
sB = 1
sC = 1
sD = 1
xA = 1
xB = 1
xC = 1
xD = 1
outA = 1
outB = 1
outC = 1
outD = 1
1
inA = 1
inB = 1
inC = 1
inD = 1
sA = 1
sB = 1
sC = 1
sD = 1
xA = 1
xB = 1
xC = 1
xD = 1
```

```
outA = 1
outB = 1
outC = 1
outD = 1
2
inA = 0
inB = 0
inC = 0
inD = 0
sA = 0
sB = 0
sC = 0
sD = 0
xA = 0
xB = 0
xC = 0
xD = 0
outA = 0
outB = 0
outC = 0
outD = 0
```

Foi então demonstrado usando *BMC* que, em regra geral, o sistema não termina. No entanto, este procedimento apenas nos permite fazer a prova de propriedades de FOTS sobre traços limitados. Podemos utilizar outras técnicas para demonstrar estas propriedades quando não é possível assumir que os traços do FOTS são limitados. Mais uma vez, como queremos demonstrar uma propriedade de *liveness*, podemos utilizar *k-lookahead* para fazer esta demonstração. Ora, temos portanto de encontrar um variante V que satisfaça as seguintes condições:

- O variante é sempre positivo, ou seja, $G\ (V(s) \geq 0)$
- O variante descresce sempre (estritamente) ou atinge o valor 0, ou seja, $G\ (\forall s'.trans(s, s') \rightarrow (V(s') < V(s) \lor V(s') = 0))$
- Quando o variante é 0 verifica-se necessariamente ϕ, ou seja, $G\ (V(s) = 0 \rightarrow \phi(s))$

Observe-se que, no sistema apresentado, a condição de terminação é $s_A == 0 \land s_B == 0 \land s_C == 0 \land s_D == 0$. Temos também que os únicos casos em que o sistema termina é quando todas as variáveis de estado de cada inversor são inicializadas com o valor 0 ou então quando todas elas são inicializadas com o valor 1, e todas as variáveis s "escolhem" então o valor da negação da variável x (que terá o valor 1 em todos os inversores). Sendo assim, podemos utilizar o variante:

$$V(s) = \begin{cases} 0, & \text{se} \quad s_A == 0 \land s_B == 0 \land s_C == 0 \land s_D == 0 \\ 1, & \text{caso contrário} \end{cases}$$

No entanto, este variante não satisfaz todas as condições apresentadas (para os casos em que o sistema termina), pois caso os inversores "mantenham" as suas variáveis s com o valor 1 durante mais do que uma transição, o variante não irá diminuir. Para termos em conta estes traços podemos então "relaxar" a segunda condição e permitir que o variante diminua apenas de k em k transições, o que nos permite ter um *lookahead* de k. Para a implementação vamos considerar um *lookahead* de 2.

- Comecemos por provar que o variante especificado é sempre positivo:

[7]:
```
def kinduction_always(declare,init,trans,inv,k):
    trace = [declare(i) for i in range(k+1)]
    s = Solver()
    s.add(init(trace[0]))
    for i in range(k-1):
        s.add(trans(trace[i],trace[i+1]))

    l = [Not(inv(trace[i])) for i in range(k)]

    s.add(Or(l))

    r = s.check()

    if r == sat:
        print('Falhou no caso base')
        m = s.model()
        for i in range(k):
            for v in trace[i]:
                print(v,'=',m[trace[i][v]])
        return

    s = Solver()
    for i in range(k):
        s.add(trans(trace[i],trace[i+1]))
        s.add(inv(trace[i]))

    s.add(Not(inv(trace[k])))

    r = s.check()

    if r == sat:
        print('Falhou no passo de k-indução')
        m = s.model()
        for i in range(k+1):
            print(i)
            for v in trace[i]:
                print(v,'=',m[trace[i][v]])

        return

    if r == unsat:
        print('Verifica-se')
        return

    return
```

```
def variante(state):
    return If( And(state['sA'] == 0, state['sB'] == 0, state['sC'] == 0,␣
 ↪state['sD'] == 0), 0, 1)

def positivo(state):
    return variante(state)>= 0

kinduction_always(declare,init,trans,positivo,2)
```

```
Verifica-se
```

- Mostremos que caso o variante tenha o valor 0, então o sistema termina necessariamente:

[8]:
```
def term(state):
    return Implies(variante(state)==0, And(state['sA'] == 0, state['sB'] == 0,␣
 ↪state['sC'] == 0, state['sD'] == 0))

kinduction_always(declare,init,trans,term,2)
```

```
Verifica-se
```

- Falta agora mostrar que o variante diminui de 2 em 2 transições. Conforme referido anteriormente, caso o variante não decresça ou não atinja o valor 0, então o sistema não irá terminar:

[16]:
```
def decresce(state):
    state1 = declare(-1)
    state2 = declare(-2)

    return ForAll(list(state1.values()) + list(state2.values()),
                  Implies(And(trans(state,state1),trans(state1,state2))
                                                    ,Or(variante(state2) <␣
 ↪variante(state), variante(state2) == 0)))

kinduction_always(declare,init,trans,decresce,5)
```

```
Falhou no caso base
inA = 1
inB = 1
inC = 1
inD = 1
sA = 1
sB = 1
sC = 1
sD = 1
xA = 1
xB = 1
xC = 1
```

```
xD = 1
outA = 1
outB = 1
outC = 1
outD = 1
inA = 1
inB = 1
inC = 0
inD = 1
sA = 1
sB = 1
sC = 1
sD = 0
xA = 1
xB = 1
xC = 0
xD = 1
outA = 1
outB = 1
outC = 1
outD = 0
inA = 1
inB = 1
inC = 0
inD = 1
sA = 1
sB = 1
sC = 1
sD = 0
xA = 1
xB = 1
xC = 0
xD = 1
outA = 1
outB = 1
outC = 1
outD = 0
inA = 1
inB = 1
inC = 0
inD = 1
sA = 1
sB = 1
sC = 1
sD = 0
xA = 1
xB = 1
xC = 0
```

```
xD = 1
outA = 1
outB = 1
outC = 1
outD = 0
inA = 1
inB = 1
inC = 0
inD = 1
sA = 1
sB = 1
sC = 1
sD = 0
xA = 1
xB = 1
xC = 0
xD = 1
outA = 1
outB = 1
outC = 1
outD = 0
```