LCTrabalho3 Exercicio1

December 13, 2020

1 Trabalho 3

1.1 Lógica Computacional 2020-2021

O objetivo deste trabalho é a utilização do sistema Z3 na análise de propriedades temporais de sistemas dinâmicos modelados por FOTS ("First Order Transition Systems")
.

Trabalho realizado por:

`> 1. Paulo Costa - A87986`
`> 2. André Araújo - A87987`

1.1.1 Exercício 1

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. > i. Cada inversor tem um bit s de estado, inicializado com um valor aleatório.
   > ii. Cada inversor é regido pelas seguintes transformações

$$\textbf{invert}(in, out)$$

$$x \leftarrow \mathsf{read}(\texttt{in})$$

$$s \leftarrow \neg x \;\; \| \;\; s \leftarrow s$$

$$\mathsf{write}(\texttt{out}, s)$$

   > iii. 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.

1.2 Resolução:

a- Neste caso o estado do FOTS respectivo serão 5 de inteiros, o primeiro contendo o valor do pc (o *program counter* que neste caso pode ser binario 0 ou 1) e os restantes os valores das variáveis s. O estado inicial é caracterizado pelo seguinte predicado:

$$pc = 0 \land (s_A = 1 \lor s_A = 0) \land (s_B = 1 \lor s_B = 0) \land (s_D = 1 \lor s_D = 0) \land (s_C = 1 \lor s_C = 0)$$

As transições possíveis no FOTS são caracterizadas pelo seguinte predicado:

$$\begin{gathered}
(pc = 0 \land s_A = 0 \land s_B = 0 \land s_D = 0 \land s_C = 0 \land \\
pc' = 1 \land s'_A = s_A \land s'_B = s_B \land s'_D = s_D \land s'_C = s_C)) \\
\lor \\
(pc = 0 \land pc' = 0 \land (s_A = 1 \lor s_A = 0) \land \\
(s_B = 1 \lor s_B = 0) \land (s_D = 1 \lor s_D = 0) \land (s_C = 1 \lor s_C = 0) \land \\
(s'_A = s_A \lor s'_A = \neg s_C) \land (s'_B = s_B \lor s'_B = \neg s'_A) \land (s'_D = s_D \lor s'_D = \neg s'_B) \land (s'_C = s_C \lor s'_C = \neg s'_D) \\
\lor \\
(pc = 1 \land pc' = 1 \land s'_A = s_A \land s'_B = s_B \land s'_D = s_D \land s'_C = s_C)
\end{gathered}$$

Note que este predicado é uma disjunção de todas as possíveis transições que podem ocorrer no programa. Cada transição é caracterizada por um predicado onde uma variável do programa denota o seu valor no pré-estado e a mesma variável com apóstrofe denota o seu valor no pós-estado.

Este procedimento designa-se model checking e, quando uma propriedade não é válida, produz um contra-exemplo (um traço do FOTS correspondente a uma execução do programa onde a propriedade falha). Bounded model checking é uma técnica particular de model checking, onde o objectivo é determinar se uma propriedade temporal é válida nos primeiros   estados da execução do FOTS.

Definimos então a função *declare*, seguida da função *init* e da *trans* e finalmente a função *gera_traco*.

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

def declare(i): #declara as variaveis de cada estado
    state = {}
    state['pc'] = Int('pc'+str(i))
    state['s_A'] = Int('s_A'+str(i))
    state['s_B'] = Int('s_B'+str(i))
    state['s_D'] = Int('s_D'+str(i))
    state['s_C'] = Int('s_C'+str(i))
    return state

def init(state): #inicia o primeiro estado
```

```
    return␣
↪And(Or(state['s_A']==0,state['s_A']==1),Or(state['s_B']==0,state['s_B']==1),
             ␣
↪Or(state['s_D']==0,state['s_D']==1),Or(state['s_C']==0,state['s_C']==1)
              ,state['pc']==0)
```

[2]:
```
def trans(curr,prox): #define as tranciçoes possiveis
    # delimita os valores do proximo estado para as variaveis
    tl=And(Or(prox['s_A']==0,prox['s_A']==1),Or(prox['s_B']==0,prox['s_B']==1),
             ␣
↪Or(prox['s_D']==0,prox['s_D']==1),Or(prox['s_C']==0,prox['s_C']==1))
    # define igualdade dos valores do estado anterior para o seguinte
    ti=And(prox['s_A']==curr['s_A'],prox['s_B']==curr['s_B'],
           prox['s_D']==curr['s_D'],prox['s_C']==curr['s_C'])
    # operaçoes para quando nao alcança o objetivo
    A=Or(prox['s_A']==curr['s_A'],prox['s_A']!=curr['s_C'])
    B=Or(prox['s_B']==curr['s_B'],prox['s_B']!=prox['s_A'])
    D=Or(prox['s_D']==curr['s_D'],prox['s_D']!=prox['s_B'])
    C=Or(prox['s_C']==curr['s_C'],prox['s_C']!=prox['s_D'])

    # as possiveis tranciçoes
  ␣
↪t1=And(curr['pc']==0,prox['pc']==1,curr['s_A']==0,curr['s_B']==0,curr['s_D']==0,curr['s_C']
  ␣
↪t2=And(curr['pc']==0,prox['pc']==0,A,B,D,C,tl,Or(curr['s_A']==1,curr['s_B']==1,curr['s_D']=
↪=curr['s_A'],prox['s_B']!=curr['s_B'],prox['s_D']!=curr['s_D'],prox['s_C']!
↪=curr['s_C']))
    t3=And(curr['pc']==1,prox['pc']==1,ti)
    return Or(t1,t2,t3)
```

[3]:
```
def gera_traco(declare,init,trans,k):
    s = Solver()
    state =[declare(i) for i in range(k)]
    s.add(init(state[0]))
    for i in range(k-1):
        s.add(trans(state[i],state[i+1]))
    if s.check()==sat:
        m=s.model()
        for i in range(k):
            print(i)
            for x in state[i]:
                print(x,"=",m[state[i][x]])


gera_traco(declare,init,trans,5)
```

0

```
pc = 0
s_A = 1
s_B = 1
s_D = 0
s_C = 0
1
pc = 0
s_A = 1
s_B = 1
s_D = 0
s_C = 1
2
pc = 0
s_A = 1
s_B = 0
s_D = 1
s_C = 1
3
pc = 0
s_A = 0
s_B = 1
s_D = 1
s_C = 0
4
pc = 0
s_A = 1
s_B = 0
s_D = 1
s_C = 0
```

1.2.1 Conclusão (a):

Bem, achamos que a primeira alínea deste exercício foi realizada com bastante sucesso. Acabamos por conseguir pôr em prática mais uma vez, os conhecimentos teóricos adquiridos na disciplina e acabar por entendê-los de uma forma bastante interessante.

Utilizando a função *declare* declara as variáveis de estado, agrupadas num dicionário que nos permite aceder às mesmas pelo nome.
A função *init* inicia o primeiro estado a utilizar. Em seguida, a função *trans* permite-nos definir as transições possíveis, ou seja, testa se é possível transitar de um estado para o seguinte, através das condições colocadas.
E por fim, a função *gera_traco* que gera uma cópia das variáveis do estado, testando condicões impostas, tais como: se um estado é inicial e se um par de estados é uma transição válida.

Assim foi-nos possível realizar esta alínea com sucesso e realizando os objetivos pretendidos.

1.3 Resolução:

c)

Aqui tivemos várias "étapas", que vamos passar a explicar, antes de apresentarmos cada uma das funções respetivas.
Primeiramente, para provar que a transiçao acaba temos de verificar se em algum estado o pc (program counter) chega a ter valor 1, e portanto definimos a função *termina*:

[4]:
```
def termina(state):
    return state['pc'] ==1
```

Em seguida, definimos a função *bmc_eventually*, esta que tem como objetivo verificar se de facto, o programa termina.

[5]:
```
def bmc_eventually(declare,init,trans,prop,bound):
    for k in range(1,bound+1):
        s = Solver()
        state =[declare(i) for i in range(k)]
        s.add(init(state[0]))
        for i in range(k-1):
            s.add(trans(state[i],state[i+1]))
        s.add(prop(state[k-1]))
        if s.check()==sat:
            m=s.model()
            for i in range(k):
                print(i)
                for x in state[i]:
                    print(x,"=",m[state[i][x]])
            return
    print ("Não foi possivel verificar a proposicao com "+str(bound)+' tracos')


bmc_eventually(declare,init,trans,termina,20)
```

```
0
pc = 0
s_A = 0
s_B = 0
s_D = 0
s_C = 0
1
pc = 1
s_A = 0
s_B = 0
s_D = 0
s_C = 0
```

Agora, tendo verificado que o programa termina quando todos começam a 0, tivemos de verificar se ele termina quando pelo menos um deles começa a 1, ou seja:

$$S_A = 1 \lor S_B = 1 \lor S_D = 1 \lor S_C = 1$$

E portanto, definimos a função *bmceventuallynot*0, que nos torna possível esta verificação necessária para a conclusão do exercício.

[6]:
```
def bmc_eventually_not0(declare,init,trans,prop,bound):
    for k in range(1,bound+1):
        s = Solver()
        state =[declare(i) for i in range(k)]
        s.add(init(state[0]))
        s.
 ↪add(Or(state[0]['s_A']==1,state[0]['s_B']==1,state[0]['s_D']==1,state[0]['s_C']==1))
        for i in range(k-1):
            s.add(trans(state[i],state[i+1]))
        s.add(prop(state[k-1]))
        if s.check()==sat:
            m=s.model()
            for i in range(k):
                print(i)
                for x in state[i]:
                    print(x,"=",m[state[i][x]])
            return
    print ("Não foi possivel verificar a proposicao com "+str(bound)+' tracos')

bmc_eventually_not0(declare,init,trans,termina,50)
```

```
Não foi possivel verificar a proposicao com 50 tracos
```

1.3.1 Conclusão (c):

Decidimos resolver pela alternativa, da alínea (c), e achamos que acabamos por conseguir apresentar o objetivo pretendido.
Conseguimos mais uma vez, colocar em prática os conhecimentos obtidos nas aulas e desta vez, achamos interessante o facto de termos de escolher perante duas opções, isto porque nos deixou a pensar em ambas as formas e optar por apresentar uma delas.
Achamos que esta alínea acabou por ser completa da melhor forma.

Em jeito de conclusão, esperemos que tenhamos realizado o exercício da forma que o professor pretendia e que de facto, tenhamos cumprido todos os requisitos impostos, algo que achamos que foi cumprido!

[ ]:
```
```