tp3_g20_Ex1

December 13, 2020

1 Lógica Computacional

Grupo 20

- Francisco Domingos Martins Oliveira, A82066
- José Luís Cerqueira Pires, A84552

1.1 Trabalho Prático 1 - Exercício 1

Neste exercício, nós decidimos representar o estado do FOTS respectivo com quatro inteiros, contendo o valor do *s* de cada *inversor* (0 ou 1) e uma constante contendo o valor do *modo* em que o estado se encontra $(INIT, WHILE, STOP)$. O estado inicial é caracterizado pelo seguinte predicado:

$$(S_A = 0 \lor S_A = 1) \land (S_B = 0 \lor S_B = 1) \land (S_D = 0 \lor S_D = 1) \land (S_C = 0 \lor S_C = 1) \land modo = INIT$$

As transições possíveis no FOTS são caracterizadas pelo seguinte predicado:

$$\begin{array}{c}
(modo = INIT \land (S_A + S_B + S_D + S_C > 0) \land modo' = WHILE \land ({S_A}' = S_A \lor {S_A}' = 1 - S_C) \land ({S_B}' = S_B \lor {S_B}' = \\
\lor \\
(modo = INIT \land (S_A + S_B + S_D + S_C = 0) \land modo' = STOP \land {S_A}' = S_A \land S \\
\lor \\
(modo = WHILE \land (S_A + S_B + S_D + S_C > 0) \land modo' = WHILE \land ({S_A}' = S_A \lor {S_A}' = 1 - S_C) \land ({S_B}' = S_B \lor {S_B}' = \\
\lor \\
(modo = WHILE \land (S_A + S_B + S_D + S_C = 0) \land modo' = STOP \land {S_A}' = S_A \land \\
\lor \\
(modo = STOP \land modo' = STOP \land {S_A}' = S_A \land {S_B}' = S_B \land {S_D}'
\end{array}$$

Note-se 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.

Usando estes predicados podemos usar um SMT solver (nomeadamente o Z3) para, por exemplo, gerar uma possível execução de $k-1$ passos do programa (em que $k > 0$). Para tal precisamos de criar k cópias das variáveis que caracterizam o estado do FOTS e depois impor que a primeira cópia satisfaz o predicado inicial e que cada par de cópias consecutivas satisfazem o predicado de transição.

A seguinte função cria a *i*-ésima cópia das variáveis de estado, agrupadas num dicionário que nos permite aceder às mesmas pelo nome.

[16]:
```
from z3 import *

Mode, (INIT,WHILE,STOP) = EnumSort('Mode', ('INIT','WHILE','STOP'))

def declare(i):
    state = {}
    state['modo'] = Const('modo'+str(i),Mode)
    state['A'] = Int('A_'+str(i))
    state['B'] = Int('B_'+str(i))
    state['D'] = Int('D_'+str(i))
    state['C'] = Int('C_'+str(i))
    return state

#declare(2)
```

A função `init`, dado um possível estado do programa (um dicionário de variáveis), devolva um predicado Z3 que testa se esse estado é um possível estado inicial do programa.

[17]:
```
def init(state):
    r = []
    r.append(state['modo']==INIT)
    r.append(Or(state['A']==1,state['A']==0))
    r.append(Or(state['B']==1,state['B']==0))
    r.append(Or(state['D']==1,state['D']==0))
    r.append(Or(state['C']==1,state['C']==0))
    return(And(r))

#init(declare(0))
```

Aa função `trans`, dados dois possíveis estados do programa, devolva um predicado Z3 que testa se é possível transitar do primeiro para o segundo.

[22]:
```
def trans(curr,prox):
    r=[]
    initwhile =␣
 ↪And(curr['modo']==INIT,prox['modo']==WHILE,(curr['A']+curr['B']+curr['D']+curr['C'])>0,Or(pro
                         ␣
 ↪Or(prox['B']==curr['B'],prox['B']==1-curr['A']),Or(prox['D']==curr['D'],prox['D']==1-curr['B'

    initstop =␣
 ↪And(curr['modo']==INIT,prox['modo']==STOP,(curr['A']+curr['B']+curr['D']+curr['C'])==0,prox['
```

```
    whilewhile =␣
↪And(curr['modo']==WHILE,prox['modo']==WHILE,(curr['A']+curr['B']+curr['D']+curr['C']>0),Or(pr
                          Or(prox['B']==curr['B'],prox['B']==1-curr['A']),␣
↪Or(prox['D']==curr['D'],prox['D']==1-curr['B']),␣
↪Or(prox['C']==curr['C'],prox['C']==1-curr['D']))

    whilestop =␣
↪And(curr['modo']==WHILE,prox['modo']==STOP,(curr['A']+curr['B']+curr['D']+curr['C'])==0,prox[

    stopstop =␣
↪And(curr['modo']==STOP,prox['modo']==STOP,prox['A']==curr['A'],prox['B']==curr['B'],prox['D']

    r.append(initwhile)
    r.append(initstop)
    r.append(whilewhile)
    r.append(whilestop)
    r.append(stopstop)

    return Or(r)

#trans(declare(0),declare(1))
```

A função de ordem superior gera_traco, dada uma função que gera uma cópia das variáveis do estado, um predicado que testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, e um número positivo k, usa o Z3 para gerar um possível traço de execução do programa de tamanho k. Para cada estado do traço imprime o respectivo valor das variáveis.

[30]:
```
def gera_traco(declare,init,trans,k):
    s = Solver()
    # completar
    trace = [declare(i) for i in range(k)]
    #print(trace)
    s.add(init(trace[0]))

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

    if s.check()==sat:
        m=s.model()
        #print(m)
        for i in range(k):
            print(i)
            print('modo '+'=',m[trace[i]['modo']])
            print('s_A '+'=',m[trace[i]['A']])
            print('s_B '+'=',m[trace[i]['B']])
```

```python
            print('s_D '+'=',m[trace[i]['D']])
            print('s_C '+'=',m[trace[i]['C']])

gera_traco(declare,init,trans,4)
print('---------')
gera_traco(declare,init,trans,5)
print('---------')
gera_traco(declare,init,trans,10)
```

```
0
modo = INIT
s_A = 1
s_B = 1
s_D = 1
s_C = 1
1
modo = WHILE
s_A = 1
s_B = 1
s_D = 1
s_C = 1
2
modo = WHILE
s_A = 0
s_B = 0
s_D = 0
s_C = 0
3
modo = STOP
s_A = 0
s_B = 0
s_D = 0
s_C = 0
---------
0
modo = INIT
s_A = 1
s_B = 1
s_D = 0
s_C = 0
1
modo = WHILE
s_A = 1
s_B = 1
s_D = 0
s_C = 0
2
```

```
modo = WHILE
s_A = 1
s_B = 0
s_D = 0
s_C = 1
3
modo = WHILE
s_A = 0
s_B = 0
s_D = 0
s_C = 1
4
modo = WHILE
s_A = 0
s_B = 0
s_D = 1
s_C = 1
---------
0
modo = INIT
s_A = 1
s_B = 0
s_D = 0
s_C = 0
1
modo = WHILE
s_A = 1
s_B = 0
s_D = 0
s_C = 0
2
modo = WHILE
s_A = 1
s_B = 0
s_D = 0
s_C = 0
3
modo = WHILE
s_A = 1
s_B = 0
s_D = 0
s_C = 0
4
modo = WHILE
s_A = 1
s_B = 0
s_D = 0
s_C = 1
```

```
5
modo = WHILE
s_A = 0
s_B = 0
s_D = 1
s_C = 1
6
modo = WHILE
s_A = 0
s_B = 1
s_D = 1
s_C = 0
7
modo = WHILE
s_A = 0
s_B = 1
s_D = 1
s_C = 0
8
modo = WHILE
s_A = 0
s_B = 1
s_D = 0
s_C = 0
9
modo = WHILE
s_A = 1
s_B = 1
s_D = 0
s_C = 1
```

Sobre este FOTS podemos querer verificar várias propriedades temporais, como por exemplo: 1. $S_A \wedge S_B \wedge S_D \wedge S_C$ são sempre superior ou igual a zero 2. $S_A + S_B + S_D + S_C$ é sempre superior a 0 3. $S_A + S_B + S_D + S_C$ é sempre inferior ou igual a 4 4. $S_A \wedge S_B \wedge S_D \wedge S_C$ chegam inevitavelmente a 0 5. O programa termina

A lógica LTL introduz *operadores temporais* que nos permitem escrever estas propriedades formalmente. Os operadores mais conhecidos são o G, que informalmente significa "*é sempre verdade que*", e o F, que informalmente significa "*é inevitável que*". Com estes operadores, as propriedades acima podem ser especificadas formalmente do seguinte modo 1. $G\ (S_A \geq 0 \wedge S_B \geq 0 \wedge S_D \geq 0 \wedge S_C \geq 0)$ 2. $G\ (S_A + S_B + S_D + S_C > 0)$ 2. $G\ (S_A + S_B + S_D + S_C \leq 4)$ 3. $F\ (S_A = 0 \wedge S_B = 0 \wedge S_D = 0 \wedge S_C = 0)$ 4. $F\ (modo = STOP)$

A função de ordem superior `bmc_always`, dada uma função que gera uma cópia das variáveis do estado, um predicado que testa se um estado é inicial, um predicado, que testa se um par de estados é uma transição válida, um invariante a verificar, e um número positivo `K`, usa o Z3 para verificar se esse invariante é sempre válido nos primeiros `K-1` passos de execução do programa, ou devolva um contra-exemplo mínimo caso não seja.

[53]:

```
def bmc_always(declare,init,trans,inv,K):
    for k in range(1,K+1):
        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(Not(inv(trace[k-1])))

        if s.check()==sat:
            m = s.model()
            for i in range(k):
                print(i)
                print('modo '+'=',m[trace[i]['modo']])
                print('s_A '+'=',m[trace[i]['A']])
                print('s_B '+'=',m[trace[i]['B']])
                print('s_D '+'=',m[trace[i]['D']])
                print('s_C '+'=',m[trace[i]['C']])
            return
        # completar

    print ("Property is valid up to traces of length "+str(K))

def positive(s):
    return And(s['A']>=0,s['B']>=0,s['D']>=0,s['C']>=0)

def great0(s):
    return Sum([s[x] for x in ['A','B','D','C']])>0

def less4(s):
    return Sum([s[x] for x in ['A','B','D','C']])<=4


print('Verificação da propriedade 1:')
bmc_always(declare,init,trans,positive,10)
print('--------')

print('Verificação da propriedade 2:')
bmc_always(declare,init,trans,great0,10)
print('--------')

print('Verificação da propriedade 3:')
bmc_always(declare,init,trans,less4,10)
```

```
Verificação da propriedade 1:
Property is valid up to traces of length 10
--------
Verificação da propriedade 2:
0
modo = INIT
s_A = 0
s_B = 0
s_D = 0
s_C = 0
--------
Verificação da propriedade 3:
0
modo = INIT
s_A = 1
s_B = 1
s_D = 1
s_C = 1
```

Para fazer BMC de propriedades de animação da forma $F\ \phi$ usaremos a função de ordem superior `bmc_eventually` que, dada uma função que gera uma cópia das variáveis do estado, um predicado que testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, uma propriedade cuja inevitabilidade se pretende verificar, e um número positivo `K`, usa o Z3 para encontrar um contra-exemplo para essa propriedade considerando apenas os primeiros `K` estados de execução do programa. Note que neste caso um contra-exemplo tem que ser necessariamente um *loop* onde a propriedade desejada nunca seja válida.

[55]:
```python
def bmc_eventually(declare,init,trans,prop,bound):
    for k in range(1,bound+1):
        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(Not(prop(trace[k-1])))

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

        if s.check()==sat:
            m = s.model()
            for i in range(k):
                if(m.eval(trans(trace[k-1],trace[i]))):
                    print("Loop starts here")
                print(i)
```

```
                print('modo '+'=',m[trace[i]['modo']])
                print('s_A '+'=',m[trace[i]['A']])
                print('s_B '+'=',m[trace[i]['B']])
                print('s_D '+'=',m[trace[i]['D']])
                print('s_C '+'=',m[trace[i]['C']])
            return

    print ("Property is valid up to traces of length "+str(bound))

def zero(state):
    return (state['A']+state['B']+state['D']+state['C'] == 0)

def terminates(state):
    return (state['modo'] == STOP)

print('Verificação da propriedade 4:')
bmc_eventually(declare,init,trans,zero,10)
print('----------------------------')

print('Verificação da propriedade 5:')
bmc_eventually(declare,init,trans,terminates,10)
```

```
Verificação da propriedade 4:
0
modo = INIT
s_A = 0
s_B = 0
s_D = 1
s_C = 0
Loop starts here
1
modo = WHILE
s_A = 1
s_B = 0
s_D = 1
s_C = 0
----------------------------
Verificação da propriedade 5:
0
modo = INIT
s_A = 1
s_B = 1
s_D = 0
s_C = 0
Loop starts here
1
modo = WHILE
```

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

Como vimos acima, há casos em que o programa não termina. Por isso vamos calcular em que situações isso acontece.

[31]:
```python
def possible_states(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]['modo']==STOP)

    while s.check() == sat:
        m = s.model()

        outputs= []
        for i in trace[0]:
            outputs.append(m[trace[0][i]])

        print(f"Estado de Init em que o programa tem solução: {outputs}")
        outputs.pop(0)

        s.add(And( (trace[0]['A'] != outputs[0]),(trace[0]['B'] !=␣
↪outputs[1]),(trace[0]['D'] != outputs[2]),(trace[0]['C'] != outputs[3])))


    return


possible_states(declare, init, trans, 42)
```

```
Estado de Init em que o programa tem solução: [INIT, 0, 0, 0, 0]
Estado de Init em que o programa tem solução: [INIT, 1, 1, 1, 1]
```

O programa termina se o valor de *s* de cada inversor começar a 0 ou a 1

[ ]:
```

```