TP4-Exercicio1

January 18, 2021

Trabalho Realizado Por:

Carlos Ferreira - a87953 Daniel Ribeiro - a87994

Exercício 1

1. Considere o seguinte programa, em Python anotado, para multiplicação de dois inteiros de precisão limitada a 16 bits. Assume-se que os inteiros são representáveis na teoria `BitVecSort(16)` do Z3.

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
0: while y > 0:
1:    if y & 1 == 1:
            y , r  = y-1 , r+x
2:    x , y = x<<1  ,  y>>1
3: assert r == m * n
```

1. Usando indução verifique a terminação deste programa.

Estado Inicial

$$m >= 0 \wedge n >= 0 \wedge r == 0 \wedge x == m \wedge y == n \wedge pc$$

Transições do FOTS

$$(pc = 0 \wedge pc' = 1 \wedge y <= 0 \wedge m' = m \wedge n' = n \wedge r' = r \wedge x' = x \wedge y' = y)$$
$$\vee$$
$$(pc = 0 \wedge pc' = 0 \wedge y > 0 \wedge m' = m \wedge n' = n \wedge r' = r + x \wedge x' = x << 1 \wedge y' = (y-1) >> 1$$
$$\vee$$
$$(pc = 1 \wedge pc' = 1 \wedge m' = m \wedge n' = n \wedge r' = r \wedge x' = x \wedge y' = y)$$

[25]:
```python
from z3 import *
from random import randint

#declarar
def declare(i):
    state = {}
    state['pc'] = Int('pc'+str(i))
    state['m'] = BitVec('m'+str(i),16)
    state['n'] = BitVec('n'+str(i),16)
    state['r'] = BitVec('r'+str(i),16)
```

```
    state['x'] = BitVec('x'+str(i),16)
    state['y'] = BitVec('y'+str(i),16)
    return state

#iniciar
def init(state):
        return And(state['m'] == randint(0,10),
                   state['n'] == randint(0,10),
                   state['r'] == 0,
                   state['x'] == state['m'],
                   state['y'] == state['n'],
                   state['pc'] == 0 )



#transicoes
def trans(curr,prox):

    aux = And(prox['m'] == curr['m'],
              prox['n'] == curr['n'],
              prox['r'] == curr['r'],
              prox['x'] == curr['x'],
              prox['y'] == curr['y'])


    #y <= 0 acabou ciclo
    t1 = And(curr['pc'] == 0, prox['pc'] == 1, curr['y'] <= 0, aux)

    #continua ciclo
    t2 = And(curr['pc'] == 0, prox['pc'] == 0,
             curr['y'] > 0,
             prox['m'] == curr['m'],
             prox['n'] == curr['n'],
             prox['r'] == curr['r'] + curr['x'],
             prox['x'] == (curr['x'] << 1),
             prox['y'] == ((curr['y']-1) >> 1))

    #fim
    t3 = And(curr['pc'] == 1, prox['pc'] == 1, aux)

    ###
    return Or(t1,t2,t3)




#gerar traco
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,10)
```

```
0
pc = 0
m = 10
n = 3
r = 0
x = 10
y = 3
1
pc = 0
m = 10
n = 3
r = 10
x = 20
y = 1
2
pc = 0
m = 10
n = 3
r = 30
x = 40
y = 0
3
pc = 1
m = 10
n = 3
r = 30
x = 40
y = 0
4
pc = 1
m = 10
n = 3
```

```
r = 30
x = 40
y = 0
5
pc = 1
m = 10
n = 3
r = 30
x = 40
y = 0
6
pc = 1
m = 10
n = 3
r = 30
x = 40
y = 0
7
pc = 1
m = 10
n = 3
r = 30
x = 40
y = 0
8
pc = 1
m = 10
n = 3
r = 30
x = 40
y = 0
9
pc = 1
m = 10
n = 3
r = 30
x = 40
y = 0
```

Usando o bmc_eventually verificamos a terminação do programa

[27]:
```
def termina(state):
    return (state['pc'] == 1)

def bmc_eventually(declare,init,trans,prop,bound):
    for k in range (1,bound+1):
        s = Solver()
        # completar
```

```
        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 ("Property is valid up to traces of length "+str(k))


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

```
Property is valid up to traces of length 1
0
pc = 0
m = 8
n = 0
r = 0
x = 8
y = 0
1
pc = 1
m = 8
n = 0
r = 0
x = 8
y = 0
```

b. Pretende-se verificar a correção parcial deste programa usando duas formas alternativas para lidar com programas iterativos: havoc e unfold. I. Usando o comando havoc e a metodologia WPC (weakest pre-condition) gere a condição de verificação que garanta a correção parcial.

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
0: while y > 0:
1:     if y & 1 == 1:
             y , r  = y-1 , r+x
2:     x , y = x<<1  ,  y>>1
3: assert r == m * n
```

Usando o método de correção havoc obtemos a seguinte estrutura:

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
0: while y > 0:
        invariante
1:      if y & 1 == 1:
```

```
                y , r  = y-1 , r+x
    2:      x , y = x<<1  , y>>1
    3: assert r == m * n
```

Logo com:

```
pre= m >= 0 and n >= 0 and r == 0 and x == m and y == n
pos= r == m * n
inv=
```

Temos:

```
assume pre;
assert inv;
havoc r,havoc x,havoc y;
((assume y>0 and inv;((assume y and 1==1;y==y-1;r==r+x)||
  assume not(y and 1==1););x==x<<1;y==y>>1;assert inv; assume False;assert pos;)||
 (assume not(y>0) and inv;assert pos;))


#== havoc

pre->(inv and ForAll([r,x,y],
     ((assume y>0 and inv;((assume y and 1==1;y==y-1;r==r+x)||
       assume not(y and 1==1);););x==x<<1;y==y>>1;assert inv; assume False;assert pos;)||
       assume not(y>0) and inv;assert pos;)))


#== false->..=TRUE

pre->(inv and ForAll([r,x,y],
     ((assume y>0 and inv;((assume y and 1==1;y==y-1;r==r+x)||
     assume not(y and 1==1);)x==x<<1;y==y>>1;assert inv;)))and assume not(y>0) and inv;assert p

#== transformaçao

pre->(inv and ForAll([r,x,y],
     (y>0 and inv->(((y and 1==1)-> inv;[y>>1/y][x<<1/x][r+x/r][y-1/y]) and
     (not(y and 1==1)->inv;[y>>1/y][x<<i/x])))) and (not(y>0) and inv) -> pos;)
```

II. Usando a metodologia SPC (strongest pos-condition), para um parâmetro inteiro N, gere o fluxo que resulta do unfold do ciclo N vezes e construa a respetiva condição de verificação.

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
0: while y > 0:
1:    if y & 1 == 1:
            y , r  = y-1 , r+x
2:    x , y = x<<1  ,  y>>1
3: assert r == m * n
```

Desenrolando o ciclo em if's ficamos com: (Desenrolamos no máximo 16 vezes pois é o tamanho máximo do BitVec)

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
if (y > 0):
    if y & 1 == 1:
        y , r  = y-1 , r+x
    x , y = x<<1  ,  y>>1
    if (y > 0):
        if y & 1 == 1:
            y , r  = y-1 , r+x
        x , y = x<<1  ,  y>>1
        if (y > 0):
            if y & 1 == 1:
                y , r  = y-1 , r+x
            x , y = x<<1  ,  y>>1
                if (y > 0):
                    if y & 1 == 1:
                        y , r  = y-1 , r+x
                    x , y = x<<1  ,  y>>1

                    (...)

                    if (y > 0):
                        if y & 1 == 1:
                            y , r  = y-1 , r+x
                        x , y = x<<1  ,  y>>1
                        assert not (y > 0)
assert r == m * n
```

Como tem de ser em single assigment (SA)

```
assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
if (y0 > 0):
    if y0 & 1 == 1:
        y1 , r1  = y0-1 , r0+x0
    else:
        r16 = r0
    x1 , y2 = x0<<1  ,  y1>>1
    if (y2 > 0):
        if y2 & 1 == 1:
            y3 , r2  = y2-1 , r1+x1
        else:
            r16 = r1
        x2 , y4 = x1<<1  ,  y3>>1
        if (y4 > 0):
            if y4 & 1 == 1:
                y5 , r3  = y4-1 , r2+x2
            else:
```

```
                r16 = r2
            x3 , y6 = x2<<1   ,   y5>>1

                    (...)

                    if (y30 > 0):
                        if y30 & 1 == 1:
                            y31 , r16  = y30-1 , r15+x15
                        else:
                            r16 = r15
                        x16 , y32 = x15<<1 , y31>>1
                        assert not (y32 > 0)

                    else:
                        r16 = r15
        else:
            r16 = r2
    else:
        r16 = r1
else:
    r16 = r0

assert r16 == m * n
```

-Passo intermédio

```
assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
    assume y0 & 1 == 1;
        y1 , r1  = y0-1 , r0+x0
    ||
    assume not y0 & 1 == 1;
        r16 = r0
    x1 , y2 = x0<<1   ,   y1>>1
    assume (y2 > 0);
        assume y2 & 1 == 1;
            y3 , r2  = y2-1 , r1+x1
        ||
        assume not (y2 & 1 == 1);
            r16 = r1
        x2 , y4 = x1<<1   ,   y3>>1
        assume (y4 > 0);
            assume y4 & 1 == 1;
                y5 , r3  = y4-1 , r2+x2
            ||
            assume not (y4 & 1 == 1);
                r16 = r2
            x3 , y6 = x2<<1   ,   y5>>1
```

```
                    (...)

                    assume (y30 > 0);
                        assume y30 & 1 == 1;
                            y31 , r16  = y30-1 , r15+x15
                        ||
                        assume not (y30 & 1 == 1);
                            r16 = r15
                        x16 , y32 = x15<<1 , y31>>1
                        assert not (y32 > 0);
                        ||
                        assume not (y30 > 0);
                            r16 = r15
        ||
        assume not (y4 > 0);
            r16 = r2
    ||
    assume not (y2 > 0);
        r16 = r1
||
assume not (y0 > 0);
    r16 = r0

assert r16 == m * n
```

-normalizado com 17 fluxos usar a strongest pre condition

```
assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
assume y0 & 1 == 1;
y1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
r16 = r0
x1 , y2 = x0<<1  ,  y1>>1
assume (y2 > 0);
assume y2 & 1 == 1;
y3 , r2  = y2-1 , r1+x1
||
assume not (y2 & 1 == 1);
r16 = r1
x2 , y4 = x1<<1  ,  y3>>1
assume (y4 > 0);
assume y4 & 1 == 1;
y5 , r3  = y4-1 , r2+x2
||
assume not (y4 & 1 == 1);
r16 = r2
x3 , y6 = x2<<1  ,  y5>>1
```

```
(...)

assume (y30 > 0);
assume y30 & 1 == 1;
y31 , r16  = y30-1 , r15+x15
||
assume not (y30 & 1 == 1);
r16 = r15
x16 , y32 = x15<<1 , y31>>1
assert not (y32 > 0) and r16 == m * n


||


(...)


||


assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
assume y0 & 1 == 1;
y1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
r16 = r0
x1 , y2 = x0<<1  ,  y1>>1
assume (y2 > 0);
assume y2 & 1 == 1;
y3 , r2  = y2-1 , r1+x1
||
assume not (y2 & 1 == 1);
r16 = r1
x2 , y4 = x1<<1  ,  y3>>1
assume not (y4 > 0);
r16 = r2
assert r16 == m * n


||


assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
assume y0 & 1 == 1;
```

```
y1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
r16 = r0
x1 , y2 = x0<<1  ,  y1>>1
assume not (y2 > 0);
r16 = r2
assert r16 == m * n



||



assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume not (y0 > 0);
r16 = r0
assert r16 == m * n
```

C. Codifique, em SMT's e em ambos os casos, a verificação da correcção parcial.
i) Usando o Havoc com as deduções préviamente feitas, temos:

[11]:
```
def prove(f):
    s = Solver()
    s.add(Not(f))
    r = s.check()
    if r == unsat:
        print("Proved")
    else:
        print("Failed to prove")
        m = s.model()
        for v in m:
            print(v,'=', m[v])



m= BitVec('m',16)
n= BitVec('n',16)
r= BitVec('r',16)
x= BitVec('x',16)
y= BitVec('y',16)
pre=And( m >=0,n >=0,r == 0,x == m,y == n)
pos= r == m*n
inv=And(y>=0,y<=n,x == m + r)

d1=Implies(And(Not(y==0),1==1),substitute(substitute(substitute(substitute(inv,(y,y>>1)),(x,x<
d2=Implies(Not(And(Not(y==0),1==1)),substitute(substitute(inv,(y,y>>1)),(x,x<<1)))
f1=inv
f2=ForAll([r,x,y],Implies(And(y>0,inv),And(d1,d2)))
```

```
f3=Implies(And(Not(y>0),inv),pos)
prove(Implies(pre,And(f1,f2,f3)))
```

```
Proved
```

ii) Usando o unfold com as deduções previamente feitas temos:

[12]:
```
r  = []
x  = []
y1 = []
y2 = []


for i in range (17):
    r.append(BitVec('r'+str(i),16))
    x.append(BitVec('x'+str(i),16))
    y1.append(BitVec('y1'+str(i),16))
    y2.append(BitVec('y2'+str(i),16))


m  = (BitVec('m'+str(i),16))
n  = (BitVec('n'+str(i),16))

preCond = And(m >=0,n >=0,r[0] == 0,x[0] == m,y1[0] == n)
posCond = r[16] == m*n


def condicao(num):
    if num == 0:
        return Implies(And(preCond, Not(y1[num] > 0),r[16] == r[num]), posCond)

    aux=And([And(r[i+1] == r[i] + x[i],
                y2[i+1] == y1[i] - 1,
                x[i+1] == x[i] << 1,
                y1[i+1] == y2[i+1] >> 1)
                for i in range(num)])

    if num == 16:
        ret = Implies(And(preCond,aux),And(Not(y1[num] > 0), posCond))
    else:
        ret = Implies(And(preCond,aux,Not(y1[num] > 0),r[16] == r[num]),
 ↪posCond)
    return Or(ret, condicao(num - 1))

prove(condicao(16))
```

```
Proved
```

[ ]:

```

```