TP4 - Grupo 06

Grupo 06

- Tomás Vaz de Carvalho Campinho A91668
- Miguel Ângelo Alves de Freitas A91635

```
!pip install z3-solver

from z3 import *
from random import randint
```

Considere o seguinte programa, em Python anotado, para multiplicação de dois inteiros de precisão limitada a 16 bits.

```
           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. Prove por indução a terminação deste programa

```
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
```

Estado Inicial

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

```
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)
```

Transições do FOTS

$$(pc = 0 \quad \wedge \quad pc' = 1 \quad \wedge \quad y <= 0 \quad \wedge \quad m' = m \quad \wedge \quad n' = n \quad \wedge \quad r' = r \quad \wedge$$

$$\vee$$

$$(pc = 0 \quad \wedge \quad pc' = 0 \quad \wedge \quad y > 0 \quad \wedge \quad y \& 1 = 1 \quad \wedge \quad m' = m \quad \wedge \quad n' = n \quad \wedge$$
$$\wedge \quad y' = (y - 1) >> 1)$$

$$\vee$$

$$(pc = 0 \quad \wedge \quad pc' = 0 \quad \wedge \quad y > 0 \quad \wedge \quad y \& 1 \neq 1 \quad \wedge \quad m' = m \quad \wedge \quad n' = n \quad \wedge$$
$$y' = y >> 1)$$

$$\vee$$

$$(pc = 1 \quad \wedge \quad pc' = 1 \quad \wedge \quad m' = m \quad \wedge \quad n' = n \quad \wedge \quad r' = r \quad \wedge \quad x' = x \quad \wedge$$

```
def trans(curr,prox):
    t0 = And(prox['m'] == curr['m'], prox['n'] == curr['n'], prox['r'] == curr['r'], prox['x'
              prox['y'] == curr['y'])


    t1 = And(curr['pc'] == 0, prox['pc'] == 1, curr['y'] <= 0, t0) #não entra no ciclo

    t2 = And(curr['pc'] == 0, prox['pc'] == 0, curr['y'] > 0, curr['y']&1 == 1, #entra no if
             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))

    t3 = And(curr['pc'] == 0, prox['pc'] == 0, curr['y'] > 0, curr['y']&1 != 1, #não entra no
             prox['m'] == curr['m'],
             prox['n'] == curr['n'],
             prox['r'] == curr['r'],
             prox['x'] == (curr['x'] << 1),
             prox['y'] == (curr['y'] >> 1))

    t4 = And(curr['pc'] == 1, prox['pc'] == 1, t0)

    return Or(t1,t2,t3,t4)
```

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("\n",i)
            for x in state[i]:
                print(x,"=", m[state[i][x]])

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

Verificação da terminação do programa

$pc == 1$

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

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("\n",i)
                for x in state[i]:
                    print(x,"=", m[state[i][x]])
            return
    print("Não foi possivel verificar a propriedade com "+ str(bound) + ' tracos')

bmc_eventually(declare, init, trans, cond_term, 16)
```

```
 0
pc = 0
m = 7
n = 6
r = 0
x = 7
y = 6

 1
pc = 0
m = 7
n = 6
r = 0
x = 14
y = 3

 2
pc = 0
m = 7
n = 6
r = 14
x = 28
y = 1

 3
pc = 0
m = 7
n = 6
r = 42
x = 56
y = 0

 4
pc = 1
m = 7
n = 6
r = 42
x = 56
y = 0
```

2. Pretende-se verificar a correção total deste programa usando a metodologia dos invariantes e a metodologia do “single assignment unfolding”.

a. Codifique usando a LPA (linguagem de programas anotadas) a forma recursiva deste programa.

$$W \equiv \{\textsf{assume}\ (y > 0);\ S;\ W\} \parallel \{\textsf{assume}\ (y \leq 0)\}$$
$$S \equiv \{\textsf{assume}\ (y\ \&\ 1 = 1);\ C;\ Z\} \parallel \{\textsf{assume}\ (y\ \&\ 1 \neq 1);\ Z\}$$
$$C \equiv \{y \leftarrow y - 1;\ r \leftarrow r + x\}$$
$$Z \equiv \{y \leftarrow y \gg 1;\ x \leftarrow x \ll 1\}$$

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n;
assert inv;
# Inicio
havoc x; havoc y; havoc r;

# Ciclo
((assume (y > 0) and inv;
    ((assume (y & 1 == 1);
        y = y-1;
        r = r+x;
    ) || (
        assume (not (y & 1 == 1));
        skip;
    ))
    x = x<<1;
    y = y>>1;
    assert inv;
    assume False;
) || (
    assume (not (y > 0)) and inv;
));

# Fim
assert r == m * n;
```

b. Proponha o invariante mais fraco que assegure a correção, codifique-o em SMT e prove a correção.

Havoc

O comando `havoc x` pode ser descrito informalmente como uma atribuição a `x` de um valor arbitrário. Em termos de denotação lógica usando a denotação WPC teremos

$$[\textsf{havoc } x\ ; C] = \forall x.\ [C]$$

Na metodologia *havoc*, o ciclo ($\textsf{while } b \textsf{ do}\{\theta\}\ C$), com anotação de invariante θ é transformado num fluxo não iterativo da seguinte forma

$$\textsf{assert } \theta\ ;\textsf{havoc } \vec{x}\ ;(\ (\textsf{assume } b \wedge \theta\ ;\ C\ ;\textsf{assert } \theta\ ;\textsf{assume } False) \ ||\ \textsf{assume } \neg b \wedge \theta\ )$$

onde $\vec{x}$ representa as *variáveis atribuídas em* C.

Observe como a denotação do triplo de Hoare $\{\phi\}\textsf{while } b \textsf{ do}\{\theta\}\ C\ \{\psi\}$, traduzido desta forma, permite garantir as propriedades de "inicialização", "preservação" e "utilidade" do invariante θ

$$[\,\textsf{assume } \phi\ ;\textsf{assert } \theta\ ;\textsf{havoc } \vec{x}\ ;(\ (\textsf{assume } b \wedge \theta\ ;\ C\ ;\textsf{assert } \theta\ ;\textsf{assume } False) \ ||\ \textsf{assume } \neg b$$

$$=$$

$$\phi \rightarrow \theta \wedge \forall \vec{x}.\ (\ (b \wedge \theta \rightarrow [C\ ;\textsf{assert } \theta]) \wedge (\neg b \wedge \theta \rightarrow \psi)\ )$$

Note que $[\,\textsf{assume } False\ ;\textsf{assert } \psi\,] = False \rightarrow \psi = True.$

Para provar que este programa é correcto pelo método havoc, teremos que, em primeiro lugar, proceder à sua tradução para a linguagem de fluxos com havoc

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

Pré-condição

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

Pós-condição

$$r = m * n$$

Invariante

$$y >= 0 \quad \wedge \quad y >= n \quad \wedge \quad x = m + r$$

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

```
inv = y>=0 and y<=n and x == m + r

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;))



# Inicialização
pre->(inv and (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;)))

# Preservação

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;)))

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 pos;)

# Utilidade

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;)
```

Prova da correção

```
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])
```

```
N = 16
s = BitVecSort(N)
m,n,r,x,y = Consts("m n r x y",s)


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
d2 = Implies(Not(And(Not(y==0),1==1)),substitute(substitute(inv,(y,y>>1)),(x,x<<1)))

inicio = inv
ciclo = ForAll([r,x,y],Implies(And(y>0,inv),And(d1,d2)))
fim = Implies(And(Not(y>0),inv),pos)

prove(Implies(pre,And(inicio,ciclo,fim)))
```

```
Proved
```

c. Construa a definição iterativa do “single assignment unfolding” usando um parâmetro limite

$$N$$

e aumentando a pré-condição com a condição $(n < N) \wedge (m < N)$

O número de iterações vai ser controlado por este parâmetro N

```
  assume m>=0 and n>=0 and r==0 and x==m and y==n and n<N and m<N);

assume y<=0;
assert y*x+r == n*m;

||

assume y>0;
assume y&1==1;
y0 = y - 1;
```

```
r0 = r + x;
x0 = x;
y1 = y0 >> 1;
x1 = x0 << 1;
r1 = r0;
||
assume y&1!=1;
y1 = y >> 1;
x1 = x << 1;
r1 = r;

assume y1<=0;
assert y1*x1+r1 = n*m;

...
```

Após o *unfold* do ciclo, temos um programa sem ciclos cuja denotação lógica (a sua VC) pode ser obtida por qualquer das técnicas apresentadas na aula anterior (transformação do programa na linguagem intermédia de fluxo, seguida da geração da VC com WPC).

```
def unfold(N):

    pre = (And(m>=0,n>=0,r==0,x==m,y==n,n<N,m<N))
    l1 = []
    v = {}
    for i in range(2*N+1):
        v['x'+str(i)] = Const('x'+str(i),s)
        v['y'+str(i)] = Const('y'+str(i),s)
        v['r'+str(i)] = Const('r'+str(i),s)

    l1.append(And(v['y0'] == y,v['x0'] == x,v['r0'] == r))
    for i in range(1,2*N+1,2):

        y0 = v['y'+str(i-1)]
        y1 = v['y'+str(i)]
        y2 = v['y'+str(i+1)]
        x0 = v['x'+str(i-1)]
        x1 = v['x'+str(i)]
        x2 = v['x'+str(i+1)]
        r0 = v['r'+str(i-1)]
        r1 = v['r'+str(i)]
        r2 = v['r'+str(i+1)]

        k = And(y0>0,Or(And(y0&1==1,y1==y0-1,r1==r0+x0,x1==x0,y2==y1>>1,x2==x1<<1,r2==r1),And
        l1.append(k)
```

```
        l1.append(k)

    r0 = v['r'+str(2*N)]

    return Implies(And(pre,And(l1)),r0==m*n)

prove(unfold(N))
```

```
Proved
```