Trabalho 4

Grupo 5
Filipe Barbosa a77252
Hugo Ferreira a78555

In [1]:

```
from z3 import *
```

Problema 1

Considera-se programa seguinte, em Python anotado, para multiplicação de dois inteiros, representáveis na teoria BitVecSort(16) do Z3, 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.a. Verificar que o programa termina usando indução.

Para verificar que o programa terminal usando indução criamos um $FOTS$ que representa o programa.

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

In [2]:

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

Define-se função $init$ que declara o estado inicial do programa. O pc tem de começar no comando 0 e o valor do bit s, in e out encontra-se no intervalo entre 0 e 1.

$$m \geq 0 \wedge n \geq 0 \wedge r = 0 \wedge x = m \wedge y = n$$

In [3]:

```
def init(state):
    return And(state['pc'] == 0, state['m'] >= 0, state['n'] >= 0, state['r'] == 0, state['
```

As transições possiveis no FOTS são caracterizadas pelo seguinte predicado, onde são usadas as variáveis já declaradas anteriormente.

$$(pc = 0 \wedge pc' = 1 \wedge m' = m \wedge n' = n \wedge x' = x \wedge y' = y \wedge y > 0 \wedge r' = r)$$
$$\vee$$
$$(pc = 0 \wedge pc' = 3 \wedge m' = m \wedge n' = n \wedge x' = x \wedge y' = y \wedge y \leq 0 \wedge r' = r)$$
$$\vee$$
$$(pc = 1 \wedge pc' = 2 \wedge m' = m \wedge n' = n \wedge y\&1 = 1 \wedge y = y - 1 \wedge r' = r + x)$$
$$\vee$$
$$(pc = 1 \wedge pc' = 2 \wedge m' = m \wedge n' = n \wedge x' = x \wedge y' = y \wedge \neg y\&1 = 1 \wedge r' = r)$$
$$\vee$$
$$(pc = 3 \wedge pc' = 0 \wedge m' = m \wedge n' = n \wedge x' = x << 1 \wedge y' = y >> 1 \wedge r' = r)$$
$$\vee$$
$$(pc = 3 \wedge pc' = 3 \wedge m' = m \wedge n' = n \wedge x' = x \wedge y' = y \wedge r = m * n)$$

In [4]:

```
def trans(curr,prox):
    t1 = And(curr['pc'] == 0, prox['pc'] == 1, prox['m'] == curr['m'], prox['n'] == curr['n
    t2 = And(curr['pc'] == 0, prox['pc'] == 3, prox['m'] == curr['m'], prox['n'] == curr['n
    t3 = And(curr['pc'] == 1, prox['pc'] == 2, prox['m'] == curr['m'], prox['n'] == curr['n
    t4 = And(curr['pc'] == 1, prox['pc'] == 2, prox['m'] == curr['m'], prox['n'] == curr['n
    t5 = And(curr['pc'] == 2, prox['pc'] == 0, prox['m'] == curr['m'], prox['n'] == curr['n
    t6 = And(curr['pc'] == 3, prox['pc'] == 3, prox['m'] == curr['m'], prox['n'] == curr['n
    return Or(t1,t2,t3,t4,t5,t6)
```

Implementamos de seguida o método `kinduction_always` para verificar invariantes por k-indução. Começamos por usar o $declare$, $init$ e $trans$ definidos anteriormente. De seguida adicionamos ao solver o invariante que queremos testar e se então o modelo der sat essa propriedade é respeitada no programa para os k estados.

Se o teste a cima der válido (dar $unsat$) testamos então para $k + 1$ estados para provar totalmente o invariante.

In [5]:

```
def kinduction_always(declare,init,trans,inv,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]))

    s.add(Or([Not(inv(state[i])) for i in range(k)]))

    if s.check() == sat:
        print('O invariante é inválido.')
        return
    print('O invariante pode não ser válido.')
    assert (s.check() == unsat)
    s = Solver()
    state = [declare(i) for i in range(k+1)]
    s.add(And([trans(state[i],state[i+1]) for i in range(k)]))
    s.add(And([inv(state[i]) for i in range(k)]))
    s.add(Not(inv(state[k])))
    if s.check() == sat:
        print('Não é possível provar o invariante com',k,'indução.')
        return
    assert (s.check() == unsat)
    print("Propriedade válida!")
```

Testa-se então se o programa termina.

$$(y = 0 \implies pc = 0 \vee pc = 3) \vee (y! = 0 \implies pc = 0)$$

In [6]:

```
def termina(state):
    return If(state['y']==0, Or(state['pc'] == 0, state['pc'] == 3), Or(state['pc'] == 0, s

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

```
O invariante pode não ser válido.
Propriedade válida!
```

1.b. Verificar correção parcial do programa.

Começamos por descobrir qual é o invariante do programa. Neste caso é $inv = y \geq 0 \wedge r = x * (n - y)$

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

Usando o comando havoc e a metodologia WPC (weakest pre-condition).

Para provar que este programa é correcto pelo método havoc procedemos 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;
assert inv
((assume y & 1 == 1 and inv; y=y-1; r=r+x; assert inv; havoc x; havoc y; assume Fa
lse) || assume not (y>0) and inv);
assert r==m*n

  ((assume i<n and inv; x=x+a; i=i+1; assert inv; assume False) || assume not(i<n)
and inv);
```

Em seguida calcula-se a denotação lógica deste programa de fluxos (a sua VC) pela WPC.

$$\text{inv} = y \geq 0 \wedge r = x * (n - y)$$
$$\text{pre} = m \geq 0 \wedge n \geq 0 \wedge r = 0 \wedge x = m \wedge y = n$$
$$\text{pos} = r = m * n$$

$$\text{pre} \rightarrow \text{inv}$$
$$\equiv$$
$$\text{pre} \rightarrow \text{inv} \ \wedge \ (\forall x. \forall y. \ (y\&1 == 1 \wedge \text{inv} \rightarrow \text{inv}[(y-1)/y][(r+x)/r])[(y >> 1)/y][(x << 1)/x] \ \wedge \ (\neg(y$$

Cria-se de seguida a prova de correção.

In [7]:

```
m, n, r, x, y = BitVecs("m n r x y",16)
pre = And(m >= 0, n >= 0, r == 0, x == m, y == n)
pos = r == m * n
inv = And(y >= 0, r == x * (n - y))

f1 = inv
f2 = ForAll([x,y], Implies(And(y > 0, inv), And(Implies(y & 1 == 1, And(substitute(substitu
f3 = Implies(And(Not(y > 0), inv), pos)

wpc = Implies(pre, And(f1, f2, f3))
```

De seguida usamos a função prove que verifica a validade da fórmula lógica usando o Z3.

In [8]:

```
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])
```

In [9]:

```
prove(wpc)
```

```
Proved
```

Usando a metodologia SPC (strongest pos-condition), para um parâmetro inteiro N.

Fazemos o unfold do ciclo para o N definido.

```
       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
```

```
    assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
    assume y > 0;
        assume (y & 1 == 1);
            y0 , r0  = y - 1 , r + x;
        x0 , y1 = x << 1 , y0 >> 1;
        assume(not(y1 > 0))
        ||
        assume(not(y & 1 == 1);
        x0, y0 = x<<1, y >> 1;
        assume(not(y0 > 0))
    assert r == m * n
```

E prova-se a denotação lógica, primeiro traduzimos para linguagem de fluxos.

In [15]:

```
# unfold -------------------------------------------------
def unfold():
    pre = "assume m >= 0 and n >= 0 and r == 0 and x == m and y == n"
    pos = "r15 == m * n"
    pOr = "||"
    x,y,r = 0,0,-1
    iteracoes = 16
    iteracoes = iteracoes-2
    # pre condicao
    print(pre)
    # iteracao inicial while e if
    unfoldWhileInicio(x,y,r)
    i1 = unfoldIfInicio(x,y,r)
    # iteracao recuriva if
    unfoldRecursivo(i1[0],i1[1],i1[2],1,iteracoes)
    # or
    print(pOr)
    # iteracao inicial not if
    unfoldNotInicio(x,y,r)
    # iteracao recuriva not if
    unfoldRecSeg(x+1,y,r,1,iteracoes)
    # pos condicao
    print(pos)

# recursivo ----------------------------------------------
def unfoldRecursivo(nx,ny,nr,tabs,iteracoes):
    pOr = "\t"*tabs+"||"
    unfoldWhile(nx,ny,nr,tabs)
    nx1, ny1, nr1 = unfoldIf(nx,ny,nr,tabs,iteracoes)
    if(iteracoes > 0):
        unfoldRecursivo(nx1,ny1,nr1,tabs+1,iteracoes-1)
    print(pOr)
    nx2, ny2, nr2 = unfoldNot(nx,ny,nr,tabs,iteracoes)
    if(iteracoes > 0):
        unfoldRecursivo(nx2,ny2,nr2,tabs+1,iteracoes-1)

# assume que o while e verdadeiro
def unfoldWhile(nx,ny,nr, tabs):
    wh1 = "\t"*tabs+f"assume y{ny} > 0;"
    print(wh1)

# parte quando o if é verdadeiro
def unfoldIf(nx, ny, nr, tabs, iteracoes):
    if1 = "\t"*tabs+f"(assume(y{ny} & 1 == 1);\n"
    if nr == -1:
        if2 = "\t"*tabs+f"    y{ny+1} , r{nr+1}  = y{ny} - 1 , r + x{nx};\n"
        if3 = "\t"*tabs+f"x{nx+1} , y{ny+2} = x{nx} << 1 , y{ny+1} >> 1;\n"
        if4 = "\t"*tabs+f"assert(Not(y{ny+2} > 0) and r15 == r{nr+1})"
    else:
        if2 = "\t"*tabs+f"    y{ny+1} , r{nr+1}  = y{ny} - 1 , r{nr} + x{nx};\n"
        if3 = "\t"*tabs+f"x{nx+1} , y{ny+2} = x{nx} << 1 , y{ny+1} >> 1;\n"
        if4 = "\t"*tabs+f"assert(Not(y{ny+2} > 0) and r15 == r{nr+1})"
    if iteracoes==0:
        print(if1,if2,if3,if4)
    else:
        print(if1,if2,if3)
    return nx+1, ny+2, nr+1

# parte quando o if e falso
```

```
def unfoldNot(nx, ny, nr, tabs, iteracoes):
    # not1 = "\t"*tabs+f"assume(not(y{ny} > 0))\n"
    if nr == -1:
        not2 = "\t"*tabs+f"assume(Not (y{ny} & 1 == 1);\n"
        not3 = "\t"*tabs+f"x{nx+1}, y{ny+1} = x{nx} << 1, y{ny} >> 1;\n"
        not4 = "\t"*tabs+f"assert(Not(y{ny+1} > 0) and r15 == r)"
    else:
        not2 = "\t"*tabs+f"assume(Not (y{ny} & 1 == 1);\n"
        not3 = "\t"*tabs+f"x{nx+1}, y{ny+1} = x{nx} << 1, y{ny} >> 1;\n"
        not4 = "\t"*tabs+f"assert(Not(y{ny+1} > 0) and r15 == r{nr})"
    if iteracoes==0:
        print(not2,not3,not4)
    else:
        print(not2,not3)
    return nx+1, ny+1, nr

# iteracao inicial ----------------------------------------
def unfoldWhileInicio(nx,ny,nr):
    wh1 = f"assume y > 0;"
    print(wh1)

# parte quando o if é verdadeiro na iteracao inicial quando as variaveis ainda sao x,y,r
def unfoldIfInicio(nx, ny, nr):
    if1 = f"assume(y & 1 == 1);\n"
    if2 = f"    y{ny} , r{nr+1}  = y - 1 , r + x;\n"
    if3 = f"x{nx}, y{ny+1} = x << 1, y{nr+1} >> 1;"
    print(if1,if2,if3)
    return(nx,ny+1,nr+1)

# parte quando o if é falso na iteracao inicial quando as variaveis ainda sao x,y,r
def unfoldNotInicio(nx, ny, nr):
    not1 = f"assume(Not (y & 1 == 1);\n"
    not2 = f"x{nx}, y{ny} = x << 1, y >> 1;"
    print(not1,not2)

# segunda iteracao not ------------------------------------
def unfoldRecSeg(nx, ny, nr, tabs, iteracoes):
    pOr = "\t" * tabs + "||"
    unfoldWhile(nx, ny, nr, tabs)
    nx1, ny1, nr1 = unfoldIfSeg(nx, ny, nr, tabs)
    if(iteracoes > 0):
        unfoldRecursivo(nx1,ny1,nr1,tabs+1,iteracoes-1)
    print(pOr)
    nx2, ny2, nr2 = unfoldNotSeg(nx, ny, nr, tabs)
    if(iteracoes > 0):
        unfoldRecursivo(nx2,ny2,nr2,tabs+1,iteracoes-1)

# parte quando o if é verdadeiro na segunda iteracao quando na primeira iteracao o if e fal
def unfoldIfSeg(nx, ny, nr, tabs):
    if1 = "\t"*tabs+f"assume(y{ny} & 1 == 1);\n"
    if2 = "\t"*tabs+f"    y{ny+1} , r{nr}  = y{ny} - 1 , r + x{nx-1};\n"
    if3 = "\t"*tabs+f"x{nx} , y{ny+2} = x{nx-1} << 1 , y{nr+1} >> 1;"
    print(if1,if2,if3)
    return nx, ny+2, nr

# parte quando o if é falso na segunda iteracao quando na primeira iteracao o if e falso e
def unfoldNotSeg(nx, ny, nr, tabs):
    not2 = "\t"*tabs+f"assume(Not (y{ny} & 1 == 1);\n"
    not3 = "\t"*tabs+f"x{nx}, y{ny+1} = x{nx-1} << 1, y{ny} >> 1;"
    print(not2,not3)
    return nx, ny+1, nr
```

```
unfold()
```

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assume y > 0;
assume(y & 1 == 1);
     y0 , r0  = y - 1 , r + x;
 x0, y1 = x << 1, y0 >> 1;
        assume y1 > 0;
        (assume(y1 & 1 == 1);
            y2 , r1  = y1 - 1 , r0 + x0;
        x1 , y3 = x0 << 1 , y2 >> 1;

                assume y3 > 0;
                (assume(y3 & 1 == 1);
                    y4 , r2  = y3 - 1 , r1 + x1;
                x2 , y5 = x1 << 1 , y4 >> 1;

                        assume y5 > 0;
                        (assume(y5 & 1 == 1);
                            y6 , r3  = y5 - 1 , r2 + x2;
                        x3 , y7 = x2 << 1 , y6 >> 1;
```

A execução acima foi interrompida devida ao ficheiro com tanto texto ficar demasiado lento.

E de seguida fazendo o unfold do ciclo.

In [12]:

```
# unfold ------------------------------------------------
def unfold():
    pre1 = "m>=0"
    pre2 = "n>=0"
    pre3 = "r==0"
    pre4 = "x==m"
    pre5 = "y==n"
    pos = "r15==m*n"
    pOr = "||"
    x,y,r = 0,0,-1
    iteracoes = 16
    dic = {}
    path = 0
    # iteracao inicial while
    wh1 = unfoldWhileInicio(x,y,r)
    dic[path] = [pre1,pre2,pre3,pre4,pre5,wh1]
    # iteracao inicial do if
    i1 = unfoldIfInicio(x,y,r)
    dic[path] = dic[path]+[i1[3],i1[4],i1[5],i1[6],i1[7]]
    # iteracao recuriva if
    unfoldRecursivo(i1[0],i1[1],i1[2],1,iteracoes-2,dic,path)
    # iteracao inicial not if
    i2 = unfoldNotInicio(x,y,r)
    path = int(pow(2,iteracoes)/2)
    dic[path] = [pre1,pre2,pre3,pre4,pre5,wh1,i2[3],i2[4],i2[5]]
    # iteracao recuriva not if
    unfoldRecSeg(i2[0],i2[1],i2[2],1,iteracoes-2,dic,path)
    l = list(dic.values())
    caminhos = []
    prove1 = "Implies(And("
    prove2 = "),"
    prove3 = ")"
    for i in l:
        caminho = ""
        for a in i:
            caminho+=","+a
        caminho = caminho[1::]
        caminhos.append(prove1+caminho+prove2+pos+prove3)
    return caminhos

# recursivo ---------------------------------------------
def unfoldRecursivo(nx,ny,nr,tabs,iteracoes,dic,path):
    pOr = "\t"*tabs+"||"
    wh1 = unfoldWhile(nx,ny,nr,tabs)
    i1 = unfoldIf(nx,ny,nr,tabs,iteracoes)
    l = dic[path]
    if len(i1) == 8:
        dic[path] = dic[path]+[wh1,i1[3],i1[4],i1[5],i1[6],i1[7]]
    else:
        dic[path] = dic[path]+[wh1,i1[3],i1[4],i1[5],i1[6],i1[7],i1[8],i1[9]]
    if(iteracoes > 0):
        unfoldRecursivo(i1[0],i1[1],i1[2],tabs+1,iteracoes-1,dic,path)
    i2 = unfoldNot(nx,ny,nr,tabs,iteracoes)
    path = int((pow(2,iteracoes))/2+(path/2))
    if len(i2) == 6:
        dic[path] = l+[wh1,i2[3],i2[4],i2[5]]
    else:
        dic[path] = l+[wh1,i2[3],i2[4],i2[5],i2[6],i2[7]]
    if(iteracoes > 0):
```

```
        unfoldRecursivo(i2[0],i2[1],i2[2],tabs+1,iteracoes-1,dic,path)

# assume que o while e verdadeiro
def unfoldWhile(nx,ny,nr, tabs):
    wh1 = f"y{ny}>0"
    return(wh1)

# parte quando o if é verdadeiro
def unfoldIf(nx, ny, nr, tabs, iteracoes):
    if1 = f"y{ny}&1==1"
    if nr == -1:
        if2 = f"y{ny+1}==y{ny}-1"
        if3 = f"r{nr+1}==r+x{nx}"
        if4 = f"x{nx+1}==x{nx}<<1"
        if5 = f"y{ny+2}==y{ny+1}>>1"
    else:
        if2 = f"y{ny+1}==y{ny}-1"
        if3 = f"r{nr+1}==r{nr}+x{nx}"
        if4 = f"x{nx+1}==x{nx}<<1"
        if5 = f"y{ny+2}==y{ny+1}>>1"
    if iteracoes==0:
        if6 = f"Not(y{ny+2}>0)"
        if nr == -1:
            if7 = f"r15==r"
        else:
            if7 = f"r15==r{nr+1}"
        return nx+1,ny+2,nr+1,if1,if2,if3,if4,if5,if6,if7
    return nx+1,ny+2,nr+1,if1,if2,if3,if4,if5

# parte quando o if e falso
def unfoldNot(nx, ny, nr, tabs, iteracoes):
    not1 = f"Not(y{ny}&1==1)"
    if nr == -1:
        not2 = f"x{nx+1}==x{nx}<<1"
        not3 = f"y{ny+1}==y{ny}>>1"
    else:
        not2 = f"x{nx+1}==x{nx}<<1"
        not3 = f"y{ny+1}==y{ny}>>1"
    if iteracoes==0:
        not4 = f"Not(y{ny+1}>0)"
        if nr == -1:
            not5 = "r15==r"
        else:
            not5 = f"r15==r{nr}"
        return nx+1,ny+1,nr,not1,not2,not3,not4,not5
    return nx+1,ny+1,nr,not1,not2,not3

# iteracao inicial ----------------------------------------
def unfoldWhileInicio(nx,ny,nr):
    wh1 = f"y>0"
    return wh1

# parte quando o if é verdadeiro na iteracao inicial quando as variaveis ainda sao x,y,r
def unfoldIfInicio(nx, ny, nr):
    if1 = f"y&1==1"
    if2 = f"y{ny}==y-1"
    if3 = f"r{nr+1}==r+x"
    if4 = f"x{nx}==x<<1"
    if5 = f"y{ny+1}==y{nr+1}>>1"
    return(nx,ny+1,nr+1,if1,if2,if3,if4,if5)
```

```
# parte quando o if é falso na iteracao inicial quando as variaveis ainda sao x,y,r
def unfoldNotInicio(nx, ny, nr):
    not1 = "Not(y&1==1)"
    not2 = f"x{nx}==x<<1"
    not3 = f"y{ny}==y>>1"
    return (nx,ny,nr,not1,not2,not3)

# segunda iteracao not ----------------------------------
def unfoldRecSeg(nx, ny, nr, tabs, iteracoes, dic, path):
    wh1 = unfoldWhile(nx, ny, nr, tabs)
    i1 = unfoldIfSeg(nx, ny, nr, tabs)
    l = dic[path]
    dic[path] = dic[path]+[wh1,i1[3],i1[4],i1[5],i1[6],i1[7]]
    if(iteracoes > 0):
        unfoldRecursivo(i1[0],i1[1],i1[2],tabs+1,iteracoes-1,dic,path)
    i2 = unfoldNotSeg(nx, ny, nr, tabs)
    path = int((pow(2,iteracoes)/2)+(path/2))
    dic[path] = l+[wh1,i2[3],i2[4],i2[5]]
    if(iteracoes > 0):
        unfoldRecursivo(i2[0],i2[1],i2[2],tabs+1,iteracoes-1,dic,path)


# parte quando o if é verdadeiro na segunda iteracao quando na primeira iteracao o if e fal
def unfoldIfSeg(nx, ny, nr, tabs):
    if1 = f"y{ny}&1==1"
    if2 = f"y{ny+1}==y{ny}-1"
    if3 = f"r{nr}==r+x"
    if4 = f"x{nx}==x<<1"
    if5 = f"y{ny+2}==y{nr+1}>>1"
    return nx,ny+2,nr,if1,if2,if3,if4,if5

# parte quando o if é falso na segunda iteracao quando na primeira iteracao o if e falso e
def unfoldNotSeg(nx, ny, nr, tabs):
    not1 = f"Not(y{ny}&1==1)"
    not2 = f"x{nx+1}==x{nx}<<1"
    not3 = f"y{ny+1}==y{ny}>>1"
    return nx+1,ny+1,nr,not1,not2,not3




paths = unfold()
```

De seguida cria-se as variáveis e provamos os diferentes caminho do ciclo recorrendo ao prove a cima.

In [11]:

```
m,n,r,x,y,x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,y0,y1,y2,y3,y4,y5,y6,y7,y8,

for i in paths:
    #print(i)
    prove(eval(i))
```

```
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
Proved
```