LCTrabalho4 Exercicio1

January 17, 2021

1 Trabalho 4

1.1 Lógica Computacional 2020-2021

Trabalho realizado por:


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


1.1.1 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. Tenha em atenção a atribuição simultânea do Python : `x , y = f(x,y) , g(x,y)` implica que simultaneamente a `x` é atribuído o valor `f(x,y)` e a `y` é atribuído `g(x,y)` .
2. Recorde que anotações não modificam o estado, nomeadamente o "program conter"

a. Usando indução verifique a terminação deste programa.

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
>ii. Usando a metodologia SPC (strongest pos-condition), para um parâmetro inteiro $N$,  g
\
```

c. Codifique, em SMT's e em ambos os casos, a verificação da correcção parcial.

1.2 Resolução:

a)

Primeiramente, apresentamos o estado inicial:

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

Passamos agora, a mostrar as transições possíveis no FOTS, estas são caracterizadas pelo seguinte predicado:

$$\begin{gathered}
(pc = 0 \land pc' = 1 \land y <= 0 \land m' = m \land n' = n \land r' = r \land x' = x \land y' = y) \\
\lor \\
(pc = 0 \land pc' = 0 \land y > 0 \land m' = m \land n' = n \land r' = r + x \land x' = x << 1 \land y' = (y - 1) >> 1 \land \neg(y = 0)) \\
\lor \\
(pc = 0 \land pc' = 0 \land y > 0 \land m' = m \land n' = n \land r' = r \land x' = x << 1 \land y' = y >> 1 \land (y = 0)) \\
\lor \\
(pc = 1 \land pc' = 1 \land m' = m \land n' = n \land r' = r \land x' = x \land y' = y)
\end{gathered}$$

Apartir de agora, vamos passar a apresentar as funções criadas para a resolução deste exercício. Todas elas com comentário apresentado, para ilucidar o intuito de cada uma delas.

[1]:
```python
from z3 import *
from random import randint
def declare(i): # declara as variaveis de cada estado #
    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

def init(state): # inicia o primeiro estado #
    return And(state['m']==randint(0,20),state['n']==randint(0,20),
               state['r']==0,state['x']==state['m'],
               state['y']==state['n'],state['pc']==0)

def trans(curr,prox): # define as trancições possiveis #
    # define igualdade dos valores do estado anterior para o seguinte #
    ␣
 ↪ti=And(prox['m']==curr['m'],prox['n']==curr['n'],prox['r']==curr['r'],prox['x']==curr['x'],

    # as possiveis trancições #
    t1=And(curr['pc']==0,prox['pc']==1,curr['y']<=0,ti)
```

```
↪t2=And(curr['pc']==0,prox['pc']==0,curr['y']>0,prox['m']==curr['m'],prox['n']==curr['n'],pr
↪prox['x']==(curr['x']<<1),prox['y']==((curr['y']-1)>>1),Not(curr['y']==0))
↪t3=And(curr['pc']==0,prox['pc']==0,curr['y']>0,prox['m']==curr['m'],prox['n']==curr['n'],pr
           prox['x']==(curr['x']<<1),prox['y']==(curr['y']>>1),curr['y']==0)
    t4=And(curr['pc']==1,prox['pc']==1,ti)
    return Or(t1,t2,t3,t4)
```

[9]:
```
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
m = 11
n = 7
r = 0
x = 11
y = 7
1
pc = 0
m = 11
n = 7
r = 11
x = 22
y = 3
2
pc = 0
m = 11
n = 7
r = 33
x = 44
y = 1
3
pc = 0
```

```
m = 11
n = 7
r = 77
x = 88
y = 0
4
pc = 1
m = 11
n = 7
r = 77
x = 88
y = 0
```

usamos o bmc_eventually para verificar se a propriedade termina, que verifica se o *pc* chega a 1.

[14]:
```
def termina(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(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,16)
```

```
0
pc = 0
m = 16
n = 1
r = 0
x = 16
y = 1
1
pc = 0
m = 16
n = 1
```

```
r = 16
x = 32
y = 0
2
pc = 1
m = 16
n = 1
r = 16
x = 32
y = 0
3
pc = 1
m = 16
n = 1
r = 16
x = 32
y = 0
```

b)

```
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
```

Queremos também colocar aqui uma introdução sobre dois aspetos que iremos utilizar, tanto na resolução da alínea 1 como da alínea 2, que se denominam por havoc e unfold, respetivamente. >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

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

Frequentemente o comando `havoc x` aparece combinado com um invariante `P` num comando `havoc x : P` que designamos "*havoc such that*". Informalmente este comando designa uma atribuição arbitrária a `x` mas dentro dos valores que verificam `P`. Ou seja, é equivalente a `havoc x ; assume P`.
Esta noção pode ser generalizada para um conjunto de variáveis $\vec{x}$.

Unfold:
Uma outra metodologia (chamada *bounded model checking of software*) passa por simular a execução do ciclo, $\mathsf{while}\ b\ \mathsf{do}\ C$, um determinado número de vezes.
Consiste basicamente em desenrolar os ciclos um certo número de vezes (k):

$\mathsf{if}\ b\ \mathsf{then}\ C;$ $\quad\mathsf{if}\ b\ \mathsf{then}\ C;$ $\qquad\ldots$ $\qquad\quad\mathsf{if}\ b\ \mathsf{then}\ \{C\ ; \mathsf{assert}\ \neg b\}$	$\mathsf{if}\ b\ \mathsf{then}\ C;$ $\quad\mathsf{if}\ b\ \mathsf{then}\ C;$ $\qquad\ldots$ $\qquad\quad\mathsf{if}\ b\ \mathsf{then}\ \{C\ ; \mathsf{assume}\ \neg b\}$

i)

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

tomando como inv= y>=0 and y<=n and x == m + r
assim,

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
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;)||assume not(y>0) and inv;)
assert r == m * n;
```

```
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;))

#==
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;)))

#== 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 pos;)
```

```
#== 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;)
```

[15]:
```
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])
```

[16]:
```
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
```

b-ii)

```
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:
        ya1 , r1  = y0-1 , r0+x0
    else:
        r16 = r0
    x1 , y1 = x0<<1  ,  ya1>>1
    if (y2 > 0):
        if y1 & 1 == 1:
            ya2 , r2  = y1-1 , r1+x1
        else:
            r16 = r1
        x2 , y2 = x1<<1  ,  ya2>>1
        if (y2 > 0):
            if y2 & 1 == 1:
                ya3 , r3  = y2-1 , r2+x2
            else:
                r16 = r2
            x3 , y3 = x2<<1  ,  ya3>>1
```

```
                    (...)

                    if (y15 > 0):
                        if y15 & 1 == 1:
                            ya16 , r16  = y16-1 , r15+x15
                        else:
                            r16 = r15
                        x16 , y16 = x15<<1 , ya16>>1
                        assert not (y16 > 0)

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

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;
        ya1 , r1  = y0-1 , r0+x0
    ||
    assume not y0 & 1 == 1;
        ya1=y0)
    x1 , y1 = x0<<1  ,  ya1>>1
    assume (y1 > 0);
        (assume y1 & 1 == 1;
            ya2 , r2  = y1-1 , r1+x1
        ||
        assume not (y1 & 1 == 1);
            ya2=y1)
        x2 , y2 = x1<<1  ,  ya2>>1
        assume (y2 > 0);
            (assume y2 & 1 == 1;
                ya3 , r3  = y2-1 , r2+x2
            ||
            assume not (y2 & 1 == 1);
                ya3=y2)
            x3 , y3 = x2<<1  ,  ya3>>1

                    (...)

                    assume (y15 > 0);
                        (assume y15 & 1 == 1;
```

```
                            ya16 , r16  = y15-1 , r15+x15
                        ||
                        assume not (y15 & 1 == 1);
                            ya16=y15)
                        x16 , y16 = x15<<1 , ya16>>1
                        assert not (y16 > 0);
                        ||
                        assume not (y16 > 0);
                            r16 = r15
        ||
        assume not (y2 > 0);
            r16 = r2
    ||
    assume not (y1 > 0);
        r16 = r1
||
assume not (y0 > 0);
    r16 = r0

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;
ya1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
ya1=y0)
x1 , y1 = x0<<1  ,  ya1>>1
assume (y1 > 0);
(assume y1 & 1 == 1;
ya2 , r2  = y1-1 , r1+x1
||
assume not (y1 & 1 == 1);
ya2=y1)
x2 , y2 = x1<<1  ,  ya2>>1
assume (y2 > 0);
(assume y2 & 1 == 1;
ya3 , r3  = y2-1 , r2+x2
||
assume not (y2 & 1 == 1);
ya3=y3)
x3 , y3 = x2<<1  ,  ya3>>1

(...)

assume (y15 > 0);
(assume y15 & 1 == 1;
```

```
ya16 , r16  = y15-1 , r15+x15
||
assume not (y15 & 1 == 1);
ya16=y15)
x16 , y16 = x15<<1 , ya16>>1
assert not (y16 > 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;
ya1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
ya1=y0)
x1 , y1 = x0<<1  ,  ya1>>1
assume (y1 > 0);
(assume y1 & 1 == 1;
ya2 , r2  = y1-1 , r1+x1
||
assume not (y1 & 1 == 1);
ya2=y1)
x2 , y2 = x1<<1  ,  ya2>>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 (y0 > 0);
(assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
ya1=y0)
x1 , y1 = x0<<1  ,  ya1>>1
```

```
assume not (y1 > 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
```

como $y > 0 \implies y \neq 0$ e $1 = 1 \implies True$ temos que $(y > 0 \implies y \neq 0) = True$ por causa de antes vir uma condição que verifica se $y > 0$ e caso não seja essa parte do codigo não é executada.

```
assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
x1 , y1 = x0<<1  ,  ya1>>1
assume (y1 > 0);
assume y1 & 1 == 1;
ya2 , r2  = y1-1 , r1+x1
x2 , y2 = x1<<1  ,  ya2>>1
assume (y2 > 0);
assume y2 & 1 == 1;
ya3 , r3  = y2-1 , r2+x2

x3 , y3 = x2<<1  ,  ya3>>1

(...)

assume (y15 > 0);
assume y15 & 1 == 1;
ya16 , r16  = y15-1 , r15+x15
x16 , y16 = x15<<1 , ya16>>1
assert not (y16 > 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;
ya1 , r1  = y0-1 , r0+x0
x1 , y1 = x0<<1  ,  ya1>>1
assume (y1 > 0);
assume y1 & 1 == 1;
ya2 , r2  = y1-1 , r1+x1
x2 , y2 = x1<<1  ,  ya2>>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 (y0 > 0);
assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
x1 , y1 = x0<<1  ,  ya1>>1
assume not (y1 > 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
```

[17]:

```
r=[BitVec('r'+str(i),16) for i in range(17)]
x=[BitVec('x'+str(i),16) for i in range(17)]
ya=[BitVec('ya'+str(i),16) for i in range(17)]
y=[BitVec('y'+str(i),16) for i in range(17)]
m= BitVec('m',16)
n= BitVec('n',16)
pre=And( m >=0,n >=0,r[0] == 0,x[0] == m,y[0] == n)
pos= r[16] == m*n
```

```
def cond(u):
    if u==0:
        return Implies(And(pre,Not(y[u]>0),r[16]==r[u]),pos)

    ␣
↪a=And([And(r[i+1]==r[i]+x[i],ya[i+1]==y[i]-1,x[i+1]==x[i]<<1,y[i+1]==ya[i+1]>>1)␣
↪for i in range(u)])

    if u==16:
        con=Implies(And(pre,a),And(Not(y[u]>0),pos))
    else:
        con=Implies(And(pre,a,Not(y[u]>0),r[16]==r[u]),pos)
    return Or(con,cond(u-1))

prove(cond(16))
```

```
Proved
```

1.3 Conclusão:

Como conclusão, achamos que deveríamos tocar em alguns aspetos.
Primeiramente, referir que foi um dos problemas/exercícios, mais difíceis que esta cadeira nos proporcionou. Isto porque tinha vários pontos que necessitavam de ser refletidos, isto é, tanto os tópicos *Unfold* e *Havoc* foram difíceis de abordar e de pôr em prática, mas com algum estudo foi possível realizar o pedido.

Este exercício foi ainda, bastante "chato", mais na parte de termos de realizar parte em single assigment (SA), que deu um trabalho a ser feito.

Apesar de tudo isto, achamos que conseguimos concluir o exercício de uma forma muito positiva e cumprindo os objetivos pretendidos.
Esperemos que este Trabalho 4, tenha sido realizado ao gosto do professor e que sobretudo tenhamos cumprido os requisitos necessário, algo que achamos que foi bem sucedido!

```
[ ]:
```