VFS

January 17, 2021

1 Verificação Formal de um Programa

Grupo 7:

- Luís Almeida A84180
- João Pedro Antunes A86813

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.

```
python             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.
2. Pretende-se verificar a correção parcial deste programa usando duas formas alternativas para lidar com programas iterativos: havoc e unfold.
   1. Usando o comando havoc e a metodologia WPC (weakest pre-condition) gere a condição de verificação que garanta a correção parcial.
   2. 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.
3. Codifique, em SMT's e em ambos os casos, a verificação da correcção parcial.

1.1 Construção do FOTS que modela o programa

Para usarmos indução e provarmos que o programa apresentado termina vamos começar por construir um FOTS que o modela. Temos então as variáveis x, y, r, m, n que irão fazer parte do FOTS. Podemos ainda considerar uma variável pc que nos indicará em que instrução nos encontramos. Define-se então a função que declara as variáveis do FOTS:

[1]:
```
from z3 import *


def declare(i):
    state = {}

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

```
    state['m'] = BitVec('m'+str(i),16)
    state['n'] = BitVec('n'+str(i),16)
    state['pc'] = Int('pc'+str(i))

    return state
```

Para definirmos o predicado *init* que determina o estado inicial do FOTS basta olharmos para a pré-condição do programa. Assim, o predicado *init* será:

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

[2]:
```
def init(state):
    l = []

    l.append(state['m'] >= 0)
    l.append(state['n'] >= 0)
    l.append(state['r'] == 0)
    l.append(state['x'] == state['m'])
    l.append(state['y'] == state['n'])
    l.append(state['pc'] == 0)

    return And(l)
```

Defina-se agora a função de transição. Sabemos que, quando a variável pc tiver o valor 0, estamos na condição do ciclo, portanto 2 situações podem ocorrer: transitar para dentro do ciclo caso a condição seja verdadeira ou transitar para o fim do programa caso a condição seja falsa. Estas situações correspondem aos predicados:

$$pc == 0 \wedge y > 0 \wedge y' == y \wedge m' == m \wedge r' == r \wedge x' == x \wedge n' == n \wedge pc' == 1$$

$$pc == 0 \wedge y \leq 0 \wedge y' == y \wedge m' == m \wedge r' == r \wedge x' == x \wedge n' == n \wedge pc' == 3$$

Ora, quando o valor de pc for 1, sabemos que entramos dentro do ciclo. A próxima instrução volta a ser uma condição e portanto, temos novamente duas situações: a condição é verdadeira e as variáveis y e r são alteradas ou então a condição é falsa e as variáveis y e r não são alteradas. Assim, temos os seguintes predicados:

$$pc == 1 \wedge \quad y\&1 == 1 \quad \wedge \quad y' == y - 1 \quad \wedge \quad m' == m \quad \wedge \quad r' == r + x \quad \wedge$$
$$x' == x \quad \wedge \quad n' == n \quad \wedge \quad pc' == 2$$

$$pc == 1 \quad \wedge \quad \neg(y\&1 == 1) \quad \wedge \quad y' == y \quad \wedge \quad m' == m \quad \wedge \quad r' == r \quad \wedge$$
$$x' == x \quad \wedge \quad n' == n \quad \wedge \quad pc' == 2$$

Quando o valor de pc é 2, temos apenas de executar as últimas instruções do ciclo e voltar a colocar o valor de pc a 0, para a condição de ciclo ser testada novamente.

$$pc == 2 \wedge y' == y >> 1 \wedge m' == m \wedge r' == r \wedge x' == x << 1 \wedge n' == n \wedge pc' == 0$$

Falta apenas adicionar o lacete final, em que o estado final transita para ele próprio:

$$pc == 3 \wedge y' == y \wedge m' == m \wedge r' == r \wedge x' == x \wedge n' == n \wedge pc' == 3$$

[3]:
```
def trans(curr,prox):
    l = []

    l.append(And(curr['pc'] == 0, curr['y'] > 0, prox['y'] == curr['y'],␣
↪prox['m'] == curr['m'],
                 prox['r'] == curr['r'], prox['x'] == curr['x'], prox['n'] ==␣
↪curr['n'], prox['pc'] == 1))

    l.append(And(curr['pc'] == 0, curr['y'] <= 0, prox['y'] == curr['y'],␣
↪prox['m'] == curr['m'],
                 prox['r'] == curr['r'], prox['x'] == curr['x'], prox['n'] ==␣
↪curr['n'], prox['pc'] == 3))

    l.append(And(curr['pc'] == 1, curr['y'] & 1 == 1, prox['y'] == curr['y'] -␣
↪1,
                 prox['r'] == curr['r'] + curr['x'], prox['x'] == curr['x'],␣
↪prox['m'] == curr['m'],
                prox['n'] == curr['n'], prox['pc'] == 2))

    l.append(And(curr['pc'] == 1, Not(curr['y'] & 1 == 1), prox['y'] ==␣
↪curr['y'],
                 prox['r'] == curr['r'], prox['x'] == curr['x'], prox['m'] ==␣
↪curr['m'],
                prox['n'] == curr['n'], prox['pc'] == 2))

    l.append(And(curr['pc'] == 2, prox['x'] == curr['x'] << 1, prox['y'] ==␣
↪curr['y'] >> 1,
                prox['m'] == curr['m'], prox['n'] == curr['n'], prox['r'] ==␣
↪curr['r'], prox['pc'] == 0))

    l.append(And(curr['pc'] == 3, prox['y'] == curr['y'], prox['m'] ==␣
↪curr['m'],
                 prox['r'] == curr['r'], prox['x'] == curr['x'], prox['n'] ==␣
↪curr['n'], prox['pc'] == 3))

    return Or(l)
```

De seguida apresenta-se uma função que gera um traço de execução do FOTS construído

[4]:
```
def gera_traco(declare,init,trans,k):
    trace = [declare(i) for i in range(k)]
    s = Solver()

    s.add(init(trace[0]))

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

    r = s.check()
    if r == sat:
        m = s.model()
        for i in range(k):
            print(i)
            for v in trace[i]:
                print(v, '=', m[trace[i][v]])
        return

    print('UNSAT')
    return

gera_traco(declare,init,trans,20)
```

```
0
x = 512
y = 39
r = 0
m = 512
n = 39
pc = 0
1
x = 512
y = 39
r = 0
m = 512
n = 39
pc = 1
2
x = 512
y = 38
r = 512
m = 512
n = 39
pc = 2
3
x = 1024
```

```
y = 19
r = 512
m = 512
n = 39
pc = 0
4
x = 1024
y = 19
r = 512
m = 512
n = 39
pc = 1
5
x = 1024
y = 18
r = 1536
m = 512
n = 39
pc = 2
6
x = 2048
y = 9
r = 1536
m = 512
n = 39
pc = 0
7
x = 2048
y = 9
r = 1536
m = 512
n = 39
pc = 1
8
x = 2048
y = 8
r = 3584
m = 512
n = 39
pc = 2
9
x = 4096
y = 4
r = 3584
m = 512
n = 39
pc = 0
10
```

```
x = 4096
y = 4
r = 3584
m = 512
n = 39
pc = 1
11
x = 4096
y = 4
r = 3584
m = 512
n = 39
pc = 2
12
x = 8192
y = 2
r = 3584
m = 512
n = 39
pc = 0
13
x = 8192
y = 2
r = 3584
m = 512
n = 39
pc = 1
14
x = 8192
y = 2
r = 3584
m = 512
n = 39
pc = 2
15
x = 16384
y = 1
r = 3584
m = 512
n = 39
pc = 0
16
x = 16384
y = 1
r = 3584
m = 512
n = 39
pc = 1
```

```
17
x = 16384
y = 0
r = 19968
m = 512
n = 39
pc = 2
18
x = 32768
y = 0
r = 19968
m = 512
n = 39
pc = 0
19
x = 32768
y = 0
r = 19968
m = 512
n = 39
pc = 3
```

1.2 Prova da terminação do programa

Vamos então utilizar indução para demonstrar que o programa termina. Como queremos provar uma propriedade de *liveness*, podemos utilizar *k-lookahead* para fazer esta demonstração. Ora, temos portanto de encontrar um variante V que satisfaça as seguintes condições:

- O variante é sempre positivo, ou seja, $G\ (V(s) \geq 0)$
- O variante descresce sempre (estritamente) ou atinge o valor 0, ou seja, $G\ (\forall s'.trans(s, s') \rightarrow (V(s') < V(s) \vee V(s') = 0))$
- Quando o variante é 0 verifica-se necessariamente ϕ, ou seja, $G\ (V(s) = 0 \rightarrow \phi(s))$

No entanto, como vamos utilizar *k-lookahead*, acabamos por relaxar a 2 condição e permitir que o variante decresça apenas de 3 em 3 transições. Considera-se um lookahead de 3 pois é o valor que nos permite "saltar" o corpo do ciclo na última iteração. Sendo assim, podemos utilizar o variante:

$$V(s) \equiv \quad y_s - pc_s + 3$$

Comecemos por provar que V é sempre positivo:

[5]:
```
def variante(state):
    return BV2Int(state['y']) - state['pc'] + 3

def kinduction_always(declare,init,trans,inv,k):
    trace = [declare(i) for i in range(k+1)]
    s = Solver()
    s.add(init(trace[0]))
    for i in range(k-1):
```

```
        s.add(trans(trace[i],trace[i+1]))

    l = [Not(inv(trace[i])) for i in range(k)]

    s.add(Or(l))

    r = s.check()

    if r == sat:
        print('Falhou no caso base')
        m = s.model()
        for i in range(k):
            print(i)
            for v in trace[i]:
                print(v,m[trace[i][v]])
        return

    s = Solver()
    for i in range(k):
        s.add(trans(trace[i],trace[i+1]))
        s.add(inv(trace[i]))
    s.add(Not(inv(trace[k])))

    r = s.check()

    if r == sat:
        print('Falhou no passo de k-indução')
        return

    if r == unsat:
        print('Verifica-se')
        return
    return

def positivo(state):
    return variante(state) >= 0

kinduction_always(declare,init,trans,positivo,3)
```

```
Verifica-se
```

De seguida, temos de provar a 2 condição, ou seja, V decresce de 3 em 3 transições:

[6]:
```
def decresce(state):
    state1 = declare(-1)
    state2 = declare(-2)
    state3 = declare(-3)
```

```
    return ForAll(list(state1.values()) + list(state2.values()) + list(state3.
↪values()),
                  ␣
↪Implies(And(trans(state,state1),trans(state1,state2),trans(state2,state3))
                                                    ,Or(variante(state3) <␣
↪variante(state), variante(state3) == 0)))

kinduction_always(declare,init,trans,decresce,3)
```

```
Verifica-se
```

Finalmente, prova-se que quando V tem o valor 0, o programa termina:

[7]:
```
def term(state):
    return Implies(variante(state) == 0, state['pc'] == 3)

kinduction_always(declare,init,trans,term,4)
```

```
Verifica-se
```

1.3 Verificação da Correção Parcial do programa

Comecemos por provar a correção parcial do programa utilizando o comando *havoc* e a metodologia *WPC*. Na metodologia *havoc*, o ciclo (while b do$\{\theta\}$ C), com um invariante θ é transformado num fluxo não iterativo da seguinte forma:

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

Assim, temos agora de definir um invariante θ. Seja $\theta \equiv m * n == x * y + r \wedge y \geq 0$. Desta forma podemos traduzir o programa na seguinte linguagem de fluxos:

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n;
assert m * n == x * y + r and y >= 0;
havoc x; havoc y;

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

Ora, vamos agora utilizar as regras da metodologia *WPC* para gerar a condição de verificação:

$$[\text{skip}] = \mathit{True}$$

$$[\text{assume } \phi] = \mathit{True}$$

$$[\text{assert } \phi] = \phi$$
$$[x = e] = True$$
$$[(C_1||C_2)] = [C_1] \wedge [C_2]$$
$$[\text{skip}\,; C] = [C]$$
$$[\text{assume } \phi\,; C] = \phi \rightarrow [C]$$
$$[\text{assert } \phi\,; C] = \phi \wedge [C]$$
$$[x = e\,; C] = [C][e/x]$$
$$[(C_1||C_2)\,; C] = [(C_1; C)||(C_2; C)]$$

```
m >= 0 and n >= 0 and r == 0 and x == m and y == n ->
m * n == x * y + r and
forall x forall y

(y > 0 and m * n == x * y + r ->
(y & 1 == 1 -> m * n == x * y + r and False -> r == m * n)[y-1/y][r+x/r][x<<1/x][y>>1/y] and
not(y & 1 == 1) -> (m * n == x * y + r and False -> r == m * n)[x<<1/x][y>>1/y]
and
not(y > 0) and m * n == x * y + r -> r == m * n)
```

De seguida implementamos esta fórmula no *z3* e tentamos provar a sua veracidade:

[15]:
```
def prove(f):
    s = Solver()
    s.add(Not(f))
    if s.check() == unsat:
        return "Valid"

    if s.check() == sat:
        mod = s.model()

        print('x = ' + str(mod[x].as_long()))
        print('y = ' + str(mod[y].as_long()))
        print('m = ' + str(mod[m].as_long()))
        print('n = ' + str(mod[n].as_long()))
        print('r = ' + str(mod[r].as_long()))

        return "False"

    return "Unknown"

x,y,m,n,r = BitVecs("x y m n r",8)

pre = And(m >= 0, n >= 0, r == 0, x == m, y == n)

pos = (r == m * n)
```

```
inv = And(m * n == x * y + r, y>=0)

f4 = Implies(y & 1 ==␣
 ↪1,substitute(substitute(substitute(substitute(inv,(y,y>>1)),(x,x<<1)),(r,r+x)),(y,y-1)))

f5 = Implies(Not(y & 1 == 1),substitute(substitute(inv,(y,y>>1)),(x,x<<1)))

f2 = Implies(And(y > 0, inv),And(f4,f5))

f3 = Implies(And(Not(y > 0), inv), pos)

resto = ForAll([x,y],f2)

F = Implies(pre,And(inv,resto,f3))


prove(F)
```

[15]: 'Valid'

Vamos agora utilizar a metodologia *SPC* e a técnica de *unfold*. Esta técnica consiste em desenrolar o ciclo k-vezes, adicionando uma *unwinding assertion* para verificar que não existem execuções que exigem mais do que k-iterações ou uma *unwinding assumption* para excluir as execuções que exigem mais do que k-iterações para serem verificadas.

if b then C;
  if b then C;
  ...
  if b then $\{C$; assert $\neg b\}$
	if b then C;
	  if b then C;
	  ...
	  if b then $\{C$; assume $\neg b\}$

Como sabemos que no fim de cada iteração do ciclo é feito um *shift-right* do BitVec y e que este tem no máximo $16bits$, então o ciclo irá ter no máximo 16 iterações. Sendo assim, apresenta-se um exemplo de um *unfold* do ciclo de 16 convertido para SA:

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0;

if(y0 > 0):                          #1
   if y0 & 1 == 1:
         y1 , r1  = y0-1 , r0+x0;
   else:
       y1 = y0;
       r1 = r0;

   x1 , y2 = x0<<1  ,  y1>>1;

    if(y2 > 0):                      #2
       if y1 & 1 == 1:
            y3 , r2  = y2-1 , r1+x1;
```

```
else:
 y3 = y2;
 r2 = r1;

x2 , y4 = x1<<1  ,  y3>>1;

 if(y4 > 0):                    #3
    if y4 & 1 == 1:
         y5 , r3  = y4-1 , r2+x2;

    else:
     y5 = y4;
     r3 = r2;

    x3 , y6 = x2<<1  ,  y5>>1;

     if(y6 > 0):                    #4
        if y6 & 1 == 1:
          y7 , r4  = y6-1 , r3+x3;

        else:
         y7 = y6;
         r4 = r3;

        x4 , y8 = x3<<1  ,  y7>>1;

         if(y8 > 0):                    #5
            if y8 & 1 == 1:
              y9 , r5  = y8-1 , r4+x4;

            else:
             y9 = y8;
             r5 = r4;

            x5 , y10 = x4<<1  ,  y9>>1;

             if(y10 > 0):                    #6
                if y10 & 1 == 1:
                  y11 , r6  = y10-1 , r5+x5;

                else:
                 y11 = y10;
                 r6 = r5;

                x6 , y12 = x5<<1  ,  y11>>1;

                 if(y12 > 0):                    #7
                    if y12 & 1 == 1:
```

```
  y13 , r7  = y12-1 , r6+x6;

else:
 y13 = y12;
 r7 = r6;

x7 , y14 = x6<<1  ,  y13>>1;

 if(y14 > 0):                    #8
    if y14 & 1 == 1:
      y15 , r8  = y14-1 , r7+x7;

    else:
     y15 = y14;
     r8 = r7;

    x8 , y16 = x7<<1  ,  y15>>1;

     if(y16 > 0):                    #9
        if y16 & 1 == 1:
          y17 , r9  = y16-1 , r8+x8;

        else:
         y17 = y16;
         r9 = r8;

        x9 , y18 = x8<<1  ,  y17>>1;

         if(y18 > 0):                    #10
            if y18 & 1 == 1:
              y19 , r10  = y18-1 , r9+x9;

            else:
             y19 = y18;
             r10 = r9;

            x10 , y20 = x9<<1  ,  y19>>1;

             if(y20 > 0):                    #11
                if y20 & 1 == 1:
                  y21 , r11  = y20-1 , r10+x10;

                else:
                 y21 = y20;
                 r11 = r10;

                x11 , y22 = x10<<1  ,  y21>>1;
```

```
if(y22 > 0):                    #12
   if y22 & 1 == 1:
     y23 , r12  = y22-1 , r11+x11;

   else:
    y23 = y22;
    r12 = r11;

   x12 , y24 = x12<<1  ,  y23>>1;

    if(y24 > 0):                    #13
       if y24 & 1 == 1:
         y25 , r13  = y24-1 , r12+x12;

       else:
        y25 = y24;
        r13 = r12;

       x13 , y26 = x12<<1  ,  y25>>1;

        if(y26 > 0):                    #14
           if y26 & 1 == 1:
             y27 , r14  = y26-1 , r13+x13;

           else:
            y27 = y26;
            r14 = r13;

           x14 , y28 = x13<<1  ,  y27>>1;

            if(y28 > 0):                    #15
               if y28 & 1 == 1:
                 y29 , r15  = y28-1 , r14+x14;

               else:
                y29 = y28;
                r15 = r14;

               x15 , y30 = x14<<1  ,  y29>>1;

                if(y30 > 0):
                   if y30 & 1 == 1:
                     y31 , r16  = y30-1 , r15+x

                   else:
                    y31 = y30;
                    r16 = r15;
```

```
                                                    x16 , y32 = x15<<1   ,  y31>>
                                                    assert not(y32>0);

                                                else:
                                                    r16 = r15;

                                            else:
                                                r16 = r14;

                                        else:
                                            r16 = r13;

                                    else:
                                        r16 = r12;

                                else:
                                    r16 = r11;

                            else:
                                r16 = r10;

                        else:
                            r16 = r9;

                    else:
                        r16 = r8;

                else:
                    r16 = r7;

            else:
                r16 = r6;

        else:
            r16 = r5;

    else:
        r16 = r4;

else:
    r16 = r3;

else:
    r16 = r2;

else:
    r16 = r1;
```

```
    else:
        r16 = r0;


    assert r16 == m0 * n0;
```

No entanto, a anotação gerada acima engloba os casos em que o ciclo não é executado 16 vezes, pelo que não irá representar a anotação em linguagem de fluxos que modela a execução do ciclo 16 vezes. Neste caso específico não existirão alternativas aos "*if*" que testam a continuidade do ciclo, ou seja, força-se que o ciclo corra 16 vezes. Temos então a seguinte anotação em linguagem de fluxos:

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0;
assume y0 > 0;
(  assume y0 & 1 == 1;
  y1 = y0 - 1;
  r1 = r0 + x0;

||
  assume not(y0 & 1 == 1);
  y1 = y0;
  r1 = r0;
);
y2 = y1 >> 1;
x1 = x0 << 1;
assume y2 > 0;
(  assume y2 & 1 == 1;
  y3 = y2 - 1;
  r2 = r1 + x1;

||
  assume not(y2 & 1 == 1);
  y3 = y2;
  r2 = r1;
);
y4 = y3 >> 1;
x2 = x1 << 1;
assume y4 > 0;
(  assume y4 & 1 == 1;
  y5 = y4 - 1;
  r3 = r2 + x2;

||
  assume not(y4 & 1 == 1);
  y5 = y4;
  r3 = r2;
);
y6 = y5 >> 1;
x3 = x2 << 1;
```

```
assume y6 > 0;
(  assume y6 & 1 == 1;
  y7 = y6 - 1;
  r4 = r3 + x3;

||
  assume not(y6 & 1 == 1);
  y7 = y6;
  r4 = r3;
);
y8 = y7 >> 1;
x4 = x3 << 1;
assume y8 > 0;
(  assume y8 & 1 == 1;
  y9 = y8 - 1;
  r5 = r4 + x4;

||
  assume not(y8 & 1 == 1);
  y9 = y8;
  r5 = r4;
);
y10 = y9 >> 1;
x5 = x4 << 1;
assume y10 > 0;
(  assume y10 & 1 == 1;
  y11 = y10 - 1;
  r6 = r5 + x5;

||
  assume not(y10 & 1 == 1);
  y11 = y10;
  r6 = r5;
);
y12 = y11 >> 1;
x6 = x5 << 1;
assume y12 > 0;
(  assume y12 & 1 == 1;
  y13 = y12 - 1;
  r7 = r6 + x6;

||
  assume not(y12 & 1 == 1);
  y13 = y12;
  r7 = r6;
);
y14 = y13 >> 1;
x7 = x6 << 1;
```

```
assume y14 > 0;
(  assume y14 & 1 == 1;
  y15 = y14 - 1;
  r8 = r7 + x7;

||
  assume not(y14 & 1 == 1);
  y15 = y14;
  r8 = r7;
);
y16 = y15 >> 1;
x8 = x7 << 1;
assume y16 > 0;
(  assume y16 & 1 == 1;
  y17 = y16 - 1;
  r9 = r8 + x8;

||
  assume not(y16 & 1 == 1);
  y17 = y16;
  r9 = r8;
);
y18 = y17 >> 1;
x9 = x8 << 1;
assume y18 > 0;
(  assume y18 & 1 == 1;
  y19 = y18 - 1;
  r10 = r9 + x9;

||
  assume not(y18 & 1 == 1);
  y19 = y18;
  r10 = r9;
);
y20 = y19 >> 1;
x10 = x9 << 1;
assume y20 > 0;
(  assume y20 & 1 == 1;
  y21 = y20 - 1;
  r11 = r10 + x10;

||
  assume not(y20 & 1 == 1);
  y21 = y20;
  r11 = r10;
);
y22 = y21 >> 1;
x11 = x10 << 1;
```

```
assume y22 > 0;
(  assume y22 & 1 == 1;
  y23 = y22 - 1;
  r12 = r11 + x11;

||
  assume not(y22 & 1 == 1);
  y23 = y22;
  r12 = r11;
);
y24 = y23 >> 1;
x12 = x11 << 1;
assume y24 > 0;
(  assume y24 & 1 == 1;
  y25 = y24 - 1;
  r13 = r12 + x12;

||
  assume not(y24 & 1 == 1);
  y25 = y24;
  r13 = r12;
);
y26 = y25 >> 1;
x13 = x12 << 1;
assume y26 > 0;
(  assume y26 & 1 == 1;
  y27 = y26 - 1;
  r14 = r13 + x13;

||
  assume not(y26 & 1 == 1);
  y27 = y26;
  r14 = r13;
);
y28 = y27 >> 1;
x14 = x13 << 1;
assume y28 > 0;
(  assume y28 & 1 == 1;
  y29 = y28 - 1;
  r15 = r14 + x14;

||
  assume not(y28 & 1 == 1);
  y29 = y28;
  r15 = r14;
);
y30 = y29 >> 1;
x15 = x14 << 1;
```

```
assume y30 > 0;
(  assume y30 & 1 == 1;
  y31 = y30 - 1;
  r16 = r15 + x15;

||
  assume not(y30 & 1 == 1);
  y31 = y30;
  r16 = r15;
);
y32 = y31 >> 1;
x16 = x15 << 1;
assert r16 == m0 * n0 and not(y32 > 0);
```

Ora, resta-nos agora utilizar a metodologia *SPC* para gerar a condição de verificação do programa. Temos:

$$[\text{skip}] = True$$
$$[\text{assume } \phi] = \phi$$
$$[\text{assert } \phi] = \phi$$
$$[x = e] = (x = e)$$
$$[(C_1||C_2)] = [C_1] \vee [C_2]$$
$$[C\,;\text{skip }] = [C]$$
$$[C\,;\text{assume } \phi] = [C] \wedge \phi$$
$$[C\,;\text{assert } \phi] = [C] \rightarrow \phi$$
$$[C\,;x = e] = [C] \wedge (x = e)$$
$$[C\,;(C_1||C_2)] = [(C;C_1)||(C;C_2)]$$

No entanto, devido à dimensão da fórmula lógica a ser gerada, resolvemos automatizar este processo com algumas funções:

[9]:
```
def SPC(inst):
        return SPCaux(inst)[0]


def SPCaux(inst):
        if not inst:
                return False,False

        h = inst[-1]
        typ,pars = h

        if typ == "alternative":
                lef = SPCaux(inst[:-1] + pars[0])
```

```
                rig = SPCaux(inst[:-1] + pars[1])
                rzt = Or(lef[0],rig[0])

                return rzt,True



        zt,h = SPCaux(inst[:-1])
        if h:
                if typ == "assume":
                        rzt = And(zt,pars[0])

                elif typ == "assert":
                        rzt = Implies(zt,pars[0])

                elif typ == "skip":
                        rzt = zt

                elif typ == "atrib":
                        rzt = And(zt,pars[0] == pars[2])

        else:
                if typ == "assume":
                        rzt = pars[0]

                elif typ == "assert":
                        rzt = pars[0]

                elif typ == "skip":
                        rzt = True

                elif typ == "atrib":
                        rzt = pars[0] == pars[2]

        return rzt,True
```

Podemos olhar para o programa como uma lista de comandos à qual temos de aplicar certas regras. Então podemos iterar esta lista seguindo as regras da *SPC* para gerar a fórmula em *z3* que modela a condição de verificação. É apenas este procedimento que a função SPC() implementa. Acabamos por fazer o mesmo para a metodologia *WPC*:

[10]:
```
def WPC(inst):
        return WPCaux(inst)[0]


def WPCaux(inst):
        if not inst:
```

```
                return False,False

        typ,pars = inst[0]

        if typ == "alternative":
                e = pars[0].copy()
                d = pars[1].copy()
                e.extend(inst[1:])
                d.extend(inst[1:])
                lef = WPCaux(e)
                rig = WPCaux(d)

                rzt = And(lef[0],rig[0])
                return rzt,True


        zt,h = WPCaux(inst[1:])
        if h:
                if typ == "assume":
                        rzt = Implies(pars[0],zt)

                elif typ == "assert":
                        rzt = And(pars[0],zt)

                elif typ == "havoc":
                        rzt = ForAll(pars[0],zt)



                elif typ == "skip":
                        rzt = zt

                elif typ == "atrib":
                        rzt = substitute(zt,(pars[0],pars[2]))

        else:
                if typ == "assume":
                        rzt = BoolVal(True)

                elif typ == "assert":
                        rzt = pars[0]

                elif typ == "skip":
                        rzt = BoolVal(True)

                elif typ == "atrib":
                        rzt = BoolVal(True)
```

```
        return rzt,True
```

Podemos também escrever uma função que gere a tradução em linguagem de fluxos do nosso programa:

[11]:
```
def FLOW(inst):
        return FLOWaux(inst,0)



def FLOWaux(inst,ident):
        if not inst:
                return ""

        h = inst[0]
        typ,pars = h

        if typ == "alternative":
                lef = FLOWaux(pars[0].copy(), ident+1)
                rig = FLOWaux(pars[1].copy(), ident+1)

                rst = ("  " * ident) + f"({lef} \n{'  ' * ident}||\n{rig}{'  '␣
↪* ident});\n{FLOWaux(inst[1:],ident)}"

                return rst

        a = FLOWaux(inst[1:],ident)

        if typ == "assume":
                rst =  "  " * ident + f"assume {pars[1]};\n{a}"

        elif typ == "assert":
                rst = "  " * ident + f"assert {pars[1]};\n{a}"

        elif typ == "skip":
                rst = "  " * ident + f"skip;\n{a}"

        elif typ == "atrib":
                rst = "  " * ident + f"{pars[1]} = {pars[3]};\n{a}"

        elif typ == "havoc":
                rst = "  " * ident + f"havoc {pars[1]};\n{a}"

        return rst
```

De seguida apresentam-se duas funções que geram a condição de verificação (em *string*) de uma anotação em linguagem de fluxos segundo a metodologia *SPC* e *WPC*, respetivamente:

[12]:
```
def SPCstr(inst):
        return SPCstraux(inst)[0]

def SPCstraux(inst):
        if not inst:
                return "",False

        h = inst[-1]
        typ,pars = h

        if typ == "alternative":
                lef = SPCstraux(inst[:-1] + pars[0])
                rig = SPCstraux(inst[:-1] + pars[1])

                rst = f"({lef[0]}) V ({rig[0]})\n"

                return rst,True



        st,h = SPCstraux(inst[:-1])
        if h:
                if typ == "assume":
                        rst = f" ({st})   ({pars[1]})"

                elif typ == "assert":
                        rst = f"({st}) → ({pars[1]})"


                elif typ == "skip":
                        rst = st


                elif typ == "atrib":
                        rst = f"({pars[1]} == {pars[3]})   ({st})"

        else:
                if typ == "assume":
                        rst = pars[1]

                elif typ == "assert":
                        rst = pars[1]


                elif typ == "skip":
                        rst = "True"
```

```
                elif typ == "atrib":
                        rst = f"({pars[1]} == {pars[3]})"


        return rst,True

def WPCstr(inst):
        return WPCstraux(inst)[0]


def WPCstraux(inst):
        if not inst:
                return False,""

        typ,pars = inst[0]

        if typ == "alternative":
                e = pars[0].copy()
                d = pars[1].copy()
                e.extend(inst[1:])
                d.extend(inst[1:])
                lef = WPCstraux(e)
                rig = WPCstraux(d)

                rst = f"({lef[0]})   ({rig[0]})"
                return rst,True


        st,h = WPCstraux(inst[1:])
        if h:
                if typ == "assume":
                        rst = f"({pars[1]}) → ({st})"

                elif typ == "assert":
                        rst = f"({pars[1]})   ({st})"

                elif typ == "havoc":
                        rst = f" {pars[1]} ({st})"



                elif typ == "skip":
                        rst = st


                elif typ == "atrib":
```

```
                        rst = f"({st}) [{pars[3]}/{pars[1]}]"

        else:
                if typ == "assume":
                        rst = "True"


                elif typ == "assert":
                        rst = pars[1]

                elif typ == "skip":
                        rst = "True"


                elif typ == "atrib":
                        rst = "True"

        return rst,True
```

Para podermos parametrizar o *unfold* e gerarmos todas as condições de verificação respetivas a cada *unfold* de 0 a 16 para assim garantirmos a correção parcial do programa, criamos a função unfold(N) que recebe um número N e cria a lista de instruções do programa para um *unfold* de N-vezes. Assim, para verificarmos a correção parcial do programa simplesmente executamos a função unfold() com todos os números de 0 a 16, aplicamos a função SPC() ao seu resultado, que irá gerar a condição de verificação em *z3* e tentamos prova-la.

[14]:
```
import gc

maxFold = 8
bits = 8
varbs = {
         "y":[BitVec(f"y{i}",bits) for i in range((maxFold+1)*2)],
         "x":[BitVec(f"x{i}",bits) for i in range(maxFold+1)],
         "r":[BitVec(f"r{i}",bits) for i in range(maxFold+1)],
         "m":[BitVec(f"m0",bits)],
         "n":[BitVec(f"n0",bits)]
        }




def internPattern(varbs,i):
    internThen = [
                    ("assume",[varbs["y"][(2*i)] & 1 == 1, f"y{(2*i)} & 1 ==␣
 ↪1"]),
                    ("atrib",[varbs["y"][(2*i)+1],␣
 ↪f"y{(2*i)+1}",varbs["y"][(2*i)] - 1,f"y{(2*i)} - 1"]),
```

```
                    ("atrib",[varbs["r"][i+1], f"r{i+1}",varbs["r"][i] +␣
↪varbs["x"][i],f"r{i} + x{i}"])
                ]

    internElse = [
                    ("assume",[Not(varbs["y"][(2*i)] & 1 == 1), f"not(y{(2*i)}␣
↪& 1 == 1)"]),
                    ("atrib",[varbs["y"][(2*i)+1], f"y{(2*i)+1}",␣
↪varbs["y"][(2*i)],  f"y{(2*i)}"]),
                    ("atrib",[varbs["r"][i+1], f"r{i+1}", varbs["r"][i], ␣
↪f"r{i}"])
                ]


    res = [
            ("alternative",[internThen,internElse]),
            ("atrib",[varbs["y"][(2*i)+2],f"y{(2*i)+2}",varbs["y"][(2*i)+1] >>␣
↪1, f"y{(2*i)+1} >> 1"]),
            ("atrib",[varbs["x"][i+1],f"x{i+1}",varbs["x"][i] << 1, f"x{i} <<␣
↪1"])
        ]

    return res



def unfold(N):

    pre = And(varbs["m"][0]>= 0, varbs["n"][0]>= 0, varbs["r"][0] == 0,␣
↪varbs["x"][0] == varbs["m"][0], varbs["y"][0] == varbs["n"][0])
    prestr = "m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0"
    pos = And(varbs["r"][maxFold] == varbs["m"][0] * varbs["n"][0])
    postr = f"r{maxFold} == m0 * n0"
    acc = [("assume", [pre, prestr])]



    for i in range(N):
        body = internPattern(varbs,i)

        acc += [
                ("assume",[varbs["y"][(2*i)] > 0, f"y{(2*i)} > 0"]),
                *body
                ]
```

```
    if N != maxFold:
        acc += [
                ("assume", [Not(varbs["y"][(2*N)] > 0), f"not(y{(2*N)} > 0)"]),
                ("atrib", [varbs["y"][2*maxFold], f"y{maxFold}",␣
↪varbs["y"][2*N],  f"y{N}"]),
                ("atrib", [varbs["x"][maxFold], f"x{maxFold}", varbs["x"][N], ␣
↪f"x{N}"]),
                ("atrib", [varbs["r"][maxFold], f"r{maxFold}", varbs["r"][N], ␣
↪f"r{N}"]),
                ("assert", [pos, postr])
               ]

    else:
        acc += [
        ("assert",[And(pos,Not(varbs["y"][2*maxFold] > 0)), postr + f" and␣
↪not(y{2*maxFold} > 0)"])
        ]



    return acc



def proveUnfold():

    for i in range(maxFold+1):
        gc.collect()
        print(f"Unfolding {i}...")
        res = unfold(i)
        zExp = SPC(res)

        print(f"Solving for unfold {i}...")

        s = Solver()
        s.add(Not(zExp))
        res = s.check()

        if res == unsat:
            print(f"Proved for unfold {i}!")

        elif res == sat:
            m = s.model()
            for i in m:
                print(f"{i} = {m[i]}")
            print(f"Could not Prove for unfold {i}")
```

```
        else:
            print(f"Unknown result for unfold {i}")

    return

#unfold8 = unfold(16)

#print(FLOW(unfold5))
#print(SPCstr(unfold5))


#z3_form = SPC(unfold8)
#prove(z3_form)
proveUnfold()
```

```
Unfolding 0…
Solving for unfold 0…
Proved for unfold 0!
Unfolding 1…
Solving for unfold 1…
Proved for unfold 1!
Unfolding 2…
Solving for unfold 2…
Proved for unfold 2!
Unfolding 3…
Solving for unfold 3…
Proved for unfold 3!
Unfolding 4…
Solving for unfold 4…
Proved for unfold 4!
Unfolding 5…
Solving for unfold 5…
Proved for unfold 5!
Unfolding 6…
Solving for unfold 6…
Proved for unfold 6!
Unfolding 7…
Solving for unfold 7…
Proved for unfold 7!
Unfolding 8…
Solving for unfold 8…
Proved for unfold 8!
```

Podemos adotar o mesmo procedimento para o comando *havoc* e utilizar a metodologia *WPC*:

[ ]:
```
bits = 16
x,y,m,n,r = BitVecs("x y m n r",bits)
```

```
inv = And(m*n == x*y + r,y >= 0)
invStr = "m * n == x * y + r and y >= 0"

pre = And( m>=0, n>=0, r == 0, x == m, y == n)
preStr = "m >= 0 and n >= 0 and r == 0 and x == m and y == n"

pos = r == m*n
posStr = "r == m*n"



havocInternIF = [
                                        ("assume", [y&1 == 1, "y & 1 == 1"]),
                                        ("atrib", [y, "y", y-1, "y-1"]),
                                        ("atrib", [r, "r", r+x, "r+x"])
                                ]

havocInternElse = [
                                        ("assume", [Not(y&1 == 1), "not(y & 1␣
 ↪== 1)"]),
                                        ("skip",[])
                                  ]


havocThen = [
                                ("assume", [And(y > 0, inv) , f"y > 0 and␣
 ↪{invStr}"]),
                                ("alternative", [havocInternIF,␣
 ↪havocInternElse]),
                                ("atrib", [x, "x", x<<1, "x<<1"]),
                                ("atrib", [y, "y", y>>1, "y>>1"]),
                                ("assert", [inv,invStr]),
                                ("assume",[False,"False"])
                        ]



havocElse = [
                                ("assume", [And(Not(y>0),inv), f"not(y > 0) and␣
 ↪{invStr}"])
                        ]


havoc  = [
                        ("assume", [pre,preStr]),
                         ("assert", [inv,invStr]),
                         ("havoc", [x,"x"]),
```

```
                        ("havoc", [y,"y"]),
                        ("alternative", [havocThen,havocElse]),
                        ("assert", [pos,posStr])
                ]

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

# tudo com 16 bits
#print(FLOW(havoc))
#print(WPCstr(havoc))

#logic = WPC(havoc)

#prove(logic)
```

[ ]: