Trabalho Prático 4

Bruno Jardim (A91680) e José Ferreira (A91636)

> Todos os problemas deste devem ser resolvidos usando pySMT e SMT's que suportem BitVec

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

1. Pretende-se verificar a correção total deste programa usando a metodologia dos invariantes e a metodologia do "single assignment unfolding". Para isso,
   A. Codifique usando a LPA (linguagem de programas anotadas) a forma recursiva deste programa.
   B. Proponha o invariante mais fraco que assegure a correção, codifique-o em SMT e prove a correção.
   C. Construa a definição iterativa do "single assignment unfolding" usando um parâmetro limite N e aumentando a pré-condição com a condição

   $$(n < N) \land (m < N)$$

   O número de iterações vai ser controlado por este parâmetro N

1. Provar que o programa termina

Para provar que o programa termina iremos utilizar, tal como no Trabalho Prático 3, a função `kinduction_always` desenvolvida nas aulas.

No entanto para podermos usar a função `kinduction_always` teremos que definir as funções `declare` , `init` e `trans`

As variáveis que iremos usar serão as seguintes:

- **pc** $\longrightarrow$ Representará o Program Counter
- **x** $\longrightarrow$ Representará a variável **x** do programa na forma de *BitVec*
- **y** $\longrightarrow$ Representará a variável **y** do programa na forma de *BitVec*
- **r** $\longrightarrow$ Representará a variável **r** do programa na forma de *BitVec*

In [1]:

```python
from z3 import *
def declare(i):
    state = {}
    state['pc'] = Int('pc'+str(i))
    state['x'] = BitVec('x'+str(i),16)
    state['y'] = BitVec('y'+str(i),16)
    state['r'] = BitVec('r'+str(i),16)
    return state

def init(state):
    return And(state['pc'] == 0,state['x'] >= 0, state['y'] >= 0, state['r'] == 0)

def trans(atual,prox):
    trans01 = And(
                atual['pc'] == 0,
                prox['pc'] == 1,
                atual['y'] > 0,
                #Operacao "And" Bitwise -> &
                atual['y'] & BitVecVal(1,16) == BitVecVal(1,16),
                prox['y'] == atual['y'],
                prox['r'] == atual['r']
    )
    trans02 = And(
                atual['pc'] == 0,
                prox['pc'] == 2,
                atual['y'] > 0,
                #Operacao "And" Bitwise -> &
                atual['y'] & BitVecVal(1,16) != BitVecVal(1,16),
                prox['y'] == atual['y'],
                prox['r'] == atual['r']
    )
    trans12 = And(
                atual['pc'] == 1,
                prox['pc']  == 2,
                atual['x']  == prox['x'],
                atual['y']  == prox['y'] + 1,
                prox['r']   == atual['r'] + atual['x']
    )
    trans20 = And(
                atual['pc']   == 2,
                prox['pc']    == 0,
                atual['x']<<1 == prox['x'],
                atual['y']>>1 == prox['y'],
                prox['r']     == atual['r']
    )
    trans03 = And(
                atual['y']  <= 0,
                atual['pc'] == 0,
                prox['pc']  == 3,
                atual['x']  == prox['x'],
                atual['y']  == prox['y'],
                prox['r']   == atual['r']
    )
    return Or(trans01,trans02,trans12,trans20,trans03)
```

De seguida codificamos um variante que nos permitirá verificar se nos encontramos num estado final

In [17]:

```python
def variante(state):
    return If(state['pc'] == 3,0,state['y'])
```

Agora teremos de garantir que o variante atingirá o valor de 0, ou então que é estritamente decrescente

In [18]:

```python
def converge(state):


    next1 = declare(-1)
    next2 = declare(-2)
    next3 = declare(-3)

    converge = variante(next3) < variante(state)
    terminou = variante(next3) == 0


    implicacao = Implies(
        And(
            trans(state,next1),
            trans(next1,next2),
            trans(next2,next3)
        ),
        Or(
            terminou,
            converge
        )
    )


    return ForAll(
                    list(next1.values())+
                    list(next2.values())+
                    list(next3.values()),
                    implicacao
                 )
```

In [19]:

```python
def kinduction_always(declare,init,trans,inv,k):
    s = Solver()
    state = {i: 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:
        m = s.model()
        print("A propriedade falha no estado inicial")
        for i in range(k):
            print("i =", i)
            for v in state[i]:
                print(v, "=", m[state[i][v]])
            print()
        print()
        return False

    s = Solver()
    state = {i: declare(i) for i in range(k+1)}

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

    s.add(Not(inv(state[k])))

    if s.check() == sat:
        m = s.model()
        print("A propriedade falha nos seguintes estados:")
        for i in range(k):
            print("i =", i)
            for v in state[i]:
                print(v, "=", m[state[i][v]])
            print()
        print()
        return False

    print("A propriedade é válida!")
    return True
```

In [20]:

```python
kinduction_always(declare,init,trans,converge,10)
```

```
A propriedade é válida!
```

Out[20]:

```
True
```

2. Correção Total do Programa e Metodologia "SAU"

a)

Para provar a correção do nosso programa utilizando as metodologia **SAU** (*single assignment unfolding*) e dos **invariantes**, teremos que codificar o nosso programa em **LPA** (*linguagem de programas anotadas*)

```
assume m >= 0 and n >= 0  and r == 0 and x == m and y == n
    assert invariante

    havoc x
    havoc y

    (
        #Condição do While True e invariante
        (assume y > 0) and invariante
            # Condição do IF verdadeira + intrucoes
            ((assume (y & 1 == 1)
                  y = y-1
                  r = r + x
             )
             #Condição do IF falsa
             ||
             (assume(not(y & 1 == 1))
              skip
             )

            x = x<<1
            y = y>>1
            assert invariante
            assume False
       ||
       # Condição do While False e invariante
       (
           assume (y <= 0) and invariante
       )
   )
   assert r == m * n
```

Este programa poderá ser resumido às seguintes equações:

$$P \equiv \{\text{assume } (y > 0); Q; P\} \parallel \{\text{assume } (y \leq 0)\}$$

$$Q \equiv \{\text{assume } (y \,\&\, 1 == 1); R; S\} \parallel \{\text{assume } (y \,\&\, 1 \neq 1); S\}$$

$$R \equiv \{y \leftarrow y - 1;\ r \leftarrow r + x\}$$

$$S \equiv \{y \leftarrow y \gg 1;\ x \leftarrow x \ll 1\}$$

b)

Para assegurar a correção através da **WPC** (*weakest pre-condition*) com recurso ao comando **Havoc**

Começemos então por definir as pré e pós condições e também o invariante do ciclo.

invariante = y\ \ge \ 0\ \land\ x\ *\ y\ +\ r\ ==\ m\ * \ n \newline

pre = m\ \ge\ 0\ \land\ n\ \ge\ 0\ \land\ r\ ==\ 0\ \land\ x\ ==\ m\ \land\ y \ ==\ n \newline pos\ =\ r\ ==\ m\ *\ n\newline

Desenvolveremos agora a *weakest pre-condition*:

\quad pre\ \rightarrow\ (\ invariante\land\ [Ciclo]\ )\quad (havoc)\newline

\quad pre\ \rightarrow\ (\ invariante\land\ \forall{x}.\forall{y}.[Ciclo]\ )\newline

\quad pre\ \rightarrow\ invariante\land\ \forall{x}.\forall{y}.[Ciclo]\newline

$$\begin{gathered} pre \ \rightarrow \ invariante \land \ \forall x . \forall y \\ .\ (y \ > \ 0 \ \land invariante \\ \rightarrow ((y \,\&\, 1 \ = \ 1 \ \rightarrow \ invariante[(\ y \ \gg \ 1\,/\,y][x \ \ll \ 1\,/\,x][(r \ + \ x)\,/\,r][y \ - \ 1\,/\,y]) \\ \land \ (\neg\,(\ y \,\&\, 1 \ = \ 1 \ \rightarrow \ invriante[(\ y \ \gg \ 1\,/\,y][x \ \ll \ 1\,/\,x] \ \land \ (\ \neg\,(y \ > \ 0\ ) \ \land \ invariante \ \rightarrow \ pos)\ )\ ) \end{gathered}$$

In [3]:

```python
m,n,x,y,r = BitVecs('m n x y r',8)
pre = And(
            m >= 0,
            #x >= 0,
            n >= 0,
            r == 0,
            y == n,
            x ==m
            )
pos = And(
            r == m * n
            )
invariante = And(
                    y >= 0,
                    x * y + r == m * n
                )


ciclo = And(Implies(y&1 == 1, substitute(invariante,[(y,y>>1),(x,x<<1),(r,r+x),(y,y-1)])),
            Implies(Not(y&1==1),substitute(invariante,[(y,y>>1),(x,x<<1)]))
           )
utilidade = And(
                Implies(And(Not(y>0),invariante),pos),
                Implies(And(y>0,invariante),ciclo)
            )

WPC = Implies(pre,And(invariante, ForAll([x,y,r],utilidade)))
```

In [22]:

```python
def prove(f):
    s = Solver()
    s.add(Not(f))
    r = s.check()
    if r == unsat:
        print('A WPC assegura a correção do programa')
    else:
        print('A WPC não assegura a correção do programa em :')
        m = s.model()
        for v in m:
            print(v,'=', m[v])
prove(WPC)
```

```
A WPC assegura a correção do programa
```

c)

In [9]:

```
!pip install pysmt
```

```
Requirement already satisfied: pysmt in c:\users\bruno\anaconda3\lib\site-packages (0.9.0)
Requirement already satisfied: six in c:\users\bruno\anaconda3\lib\site-packages (from pysmt) (1.15.0)
```

ErrorPropertyUnrolling

De seguida, encontra-se a implementação da classe que faz unfolding de ciclos, denomindada de **EPU** (*error property unrolling*) disponibilizada pelo Professor Valença.

Quando a propriedade $\{\text{assume } pre\ \}\ P\ \{\text{assert } pos\}$ seja verdadeira, quando tal for verdade o algoritmo retornará a iteração em que a propriedade se verifica.

In [4]:

```python
from pysmt.shortcuts import *
from pysmt.typing import *


# Auxiliares

def prime(v):
    return Symbol("next(%s)" % v.symbol_name(), v.symbol_type())

def fresh(v):
    return FreshSymbol(typename=v.symbol_type(),template=v.symbol_name()+"_%d")


class EPU(object):
    """deteção de erro"""

    def __init__(self, variables, init , trans, error, sname="z3"):

        self.variables = variables       # FOTS variables
        self.init  = init                # FOTS init as unary predicate in "variables"
        self.error = error               # FOTS error condition as unary predicate in "variables"
        self.trans = trans               # FOTS transition relation as a binary transition relation
                                         # in "variables" and "prime variables"

        self.prime_variables = [prime(v) for v in self.variables]
        self.frames = [self.error]       # inializa com uma só frame: a situação de error

        self.solver = Solver(name=sname)
        self.solver.add_assertion(self.init)     # adiciona o estado inicial como uma asserção sempre presente

    def new_frame(self):
        freshs = [fresh(v) for v in self.variables]
        T = self.trans.substitute(dict(zip(self.prime_variables,freshs)))
        F = self.frames[-1].substitute(dict(zip(self.variables,freshs)))
        self.frames.append(Exists(freshs, And(T, F)))

    def unroll(self,bound=0):
        n = 0
        while True:
            if n > bound:
                print("falha: tentativas ultrapassam o limite %d "%bound)
                break
            elif self.solver.solve(self.frames):
                self.new_frame()
                n += 1
            else:
                print("sucesso: tentativa %d "%n)
                break


class Cycle(EPU):
    def __init__(self,variables,pre,pos,control,body,sname="z3"):
        init   = pre
        trans = And(control,body)
        error  = Or(control,Not(pos))
        super().__init__(variables, init, trans, error, sname)
```

In [5]:

```python
#Definição de variaveis
x = Symbol("x",BVType(16))
y = Symbol("y",BVType(16))
m = Symbol("m",BVType(16))
n = Symbol("n",BVType(16))
r = Symbol("r",BVType(16))
variaveis = [x,y,m,n,r]
Zero = BVZero(16)
One = BVOne(16)
```

```
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
```

In [7]:

```python
"""
    invariante=y ≥ 0 ∧ x ∗ y + r == m ∗ n
    pre=m ≥ 0 ∧ n ≥ 0 ∧ r == 0 ∧ x == m ∧ y == n
    pos = r == m ∗ n
"""
N = BV(2^16-1,16)
#Pré-condição
pre = And([
            BVUGE(m,Zero),
            BVUGE(n,Zero),
            Equals(r,Zero),
            Equals(x,m),
            Equals(y,n),
            BVULE(n,N),
            BVULE(m,N)
])
#Pós-condição
pos = And(Equals(r,BVMul(m,n)))

#Condição do while - linha 0
l0 = BVUGT(y,Zero)

#Condição do if - linha 1
l1 = Equals(BVAnd(y,One),One)

#Atribuições com if TRUE
true = And([
            Equals(prime(y),BVLShr(BVSub(y,One),One)),
            Equals(prime(r),BVAdd(r,x)),
            Equals(prime(x),BVLShl(x,One))
])
#Identico ao true apenas nao tem a subtração do y nem a adição do r
false = And([
            Equals(prime(y),BVLShr(y,One)),
            #Equals(prime(r),BVAdd(r,x)),
            Equals(prime(x),BVLShl(x,One))
])

transicao = Ite(l1,true,false)
```

In [8]:

```python
W = Cycle(variaveis,pre,pos,l0,transicao)
W.unroll(50)
```

```
sucesso: tentativa 5
```