Trabalho 4 - Verificação Formal de Software

Janeiro, 2022

Inês Pires Presa - A90355
Tiago dos Santos Silva Peixoto Carriço - A91695

Descrição do Problema

Pretende-se verificar a correção total do seguinte programa usando a metodologia dos invariantes e a metodologia do "single assignement unfolding":

```
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
```

Provar por indução a terminação do programa

Pretendemos provar uma propriedade de animação por indução, para isso começaremos por modelar o programa com FOTS e de seguida teremos de descobrir um variante que satisfaça as condições:

- O variante nunca é negativo, 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, $G\ (V(s) = 0 \rightarrow \phi(s))$

Modelação do programa com FOTS

O estado inicial é caracterizado pelo seguinte predicado:

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

As transições possíveis no FOTS são caracterizadas pelo seguinte predicado:

$$\begin{array}{c}
(pc = 0 \wedge y > 0 \wedge pc' = 1 \wedge y' = y \wedge m' = m \wedge n' = n \wedge x' = x \wedge r' = r) \\
\vee \\
(pc = 0 \wedge y \leq 0 \wedge pc' = 3 \wedge y' = y \wedge m' = m \wedge n' = n \wedge x' = x \wedge r' = r) \\
\vee \\
(pc = 1 \wedge y \,\&\, 1 = 1 \wedge pc' = 2 \wedge y' = y - 1 \wedge m' = m \wedge n' = n \wedge x' = x \wedge r' = r + x) \\
\vee \\
(pc = 1 \wedge y \,\&\, 1 \neq 1 \wedge pc' = 2 \wedge y' = y \wedge m' = m \wedge n' = n \wedge x' = x \wedge r' = r) \\
\vee \\
(pc = 2 \wedge pc' = 0 \wedge y' = y >> 1 \wedge m' = m \wedge n' = n \wedge x' = x << 1 \wedge r' = r) \\
\vee \\
(pc = 3 \wedge pc' = 3 \wedge y' = y \wedge m' = m \wedge n' = n \wedge x' = x \wedge r' = r \wedge r = m \cdot n)
\end{array}$$

In [9]:
```python
from z3 import *
```

Funções:

- `declare(i)` - gera uma cópia das variáveis do estado
- `init(state)` - testa se um estado é inicial
- `trans(curr, prox)` - testa se um par de estados é uma transição válida.

In [10]:
```python
def declare(i):
    state = {}
    state['pc'] = BitVec('pc_'+str(i), 16)
    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)
    return state

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

def trans(curr, prox):
    t_0_1 = And(curr['pc'] == 0, prox['pc'] == 1, curr['y'] > 0 , prox['y'] == curr['y'],
                prox['n'] == curr['n'], prox['m'] == curr['m'], prox['r'] == curr['r'], prox['x'] == curr['x'])
    t_0_3 = And(curr['pc'] == 0, prox['pc'] == 3, curr['y'] <= 0, prox['y'] == curr['y'],
                prox['n'] == curr['n'], prox['m'] == curr['m'], prox['r'] == curr['r'], prox['x'] == curr['x'])

    t_1_2 = Or(
            And(curr['pc'] == 1, prox['pc'] == 2, curr['y'] & 1 == 1, prox['n'] == curr['n'],
                prox['m'] == curr['m'], prox['x'] == curr['x'], prox['y'] == curr['y'] - 1,
                prox['r'] == curr['r'] + curr['x']),
            And(curr['pc'] == 1, prox['pc'] == 2, curr['y'] & 1 == 1, prox['y'] == curr['y'],
                prox['n'] == curr['n'], prox['m'] == curr['m'], prox['r'] == curr['r'],
                prox['x'] == curr['x'])
    )

    t_2_0 = And(curr['pc'] == 2, prox['pc'] == 0, prox['n'] == curr['n'], prox['m'] == curr['m'],
                prox['r'] == curr['r'], prox['x'] == curr['x'] << 1, prox['y'] == curr['y'] >> 1)

    t_3_3 = And(curr['pc'] == 3, prox['pc'] == 3, prox['y'] == curr['y'], prox['n'] == curr['n'],
                prox['m'] == curr['m'], prox['x'] == curr['x'], curr['r'] == prox['r'],
                curr['r'] == curr['m'] * curr['n'])

    return Or(t_0_1, t_0_3, t_1_2, t_2_0, t_3_3)
```

Descobir um variante

De forma a facilitar a procura de um variante que permite provar por indução que o programa acima termina, iremos relaxar a segunda condição acima e exigir que o variante apenas tenha que decrescer estritamente a cada 3 transições, ou seja, vamos usar um *lookahead* de 3.

In [11]:
```python
x =  10
y =  15
r = 0

from tabulate import tabulate

# variante: y - pc +3

headers = ['pc', 'x', 'y', 'r', 'var']
tabela = []
while y > 0:
    l = [0, x, y, r, (y-0+3)]
    tabela.append(l)
    if y & 1 == 1:
        y , r  = y-1 , r+x
        l = [1, x, y, r, (y-1+3)]
        tabela.append(l)
    x , y = x<<1  ,  y>>1
    l = [2, x, y, r, (y-2+3)]
    tabela.append(l)
l = [3, x, y, r, (y-3+3)]
tabela.append(l)

print(tabulate(tabela, headers))
```

```
  pc    x    y    r    var
----  ---  ---  ---  -----
   0   10   15    0     18
   1   10   14   10     16
   2   20    7   10      8
   0   20    7   10     10
   1   20    6   30      8
   2   40    3   30      4
   0   40    3   30      6
   1   40    2   70      4
   2   80    1   70      2
   0   80    1   70      4
   1   80    0  150      2
   2  160    0  150      1
   3  160    0  150      0
```

Pela análise da tabela anterior concluimos que o variante $(y - pc + 3)$ satisfaz as condições pretendidas.

De seguida, iremos confirmar, através de k-indução, que tal se verifica para quaisquer valores de m e n.

Provar as condições acima por k-indução

In [12]:
```python
def variante(state):
  return (BV2Int(state['y']) - BV2Int(state['pc']) + 3)

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

def variante_decrescente(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)))

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


def kinduction_always(declare,init,trans,inv,k):
    # completar
    trace = [declare(i) for i in range(k+1)]

    # testar inv para os estados iniciais
    s = Solver()
    s.add(init(trace[0]))
    for i in range(k-1):
      s.add(trans(trace[i],trace[i+1]))
    s.add(Or([Not(inv(trace[i])) for i in range(k)]))
    if s.check()==sat:
      m = s.model()
      print("A propriedade falha em pelo menos um dos ", k, " primeiros estados")
      for v in trace[0]:
        print(v,"=",m[trace[0][v]])
      return
    if s.check() == unknown:
      print("Não sabemos.")
      return

    # testar o passo indutivo
    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])))
    if s.check() == sat:
      m = s.model()
      print('O passo indutivo falha no estado.')
      for v in trace[0]:
        print(v,'=', m[trace[0][v]])
      return
    if s.check() == unknown:
      print("Não sabemos.")
      return

    print("A propriedade é válida")
```

- O variante nunca é negativo, ou seja, $G\ (V(s) \geq 0)$

In [13]:
```python
kinduction_always(declare,init,trans,variante_positivo,3)
```

```
A propriedade é válida
```

- O variante descresce sempre (estritamente) ou atinge o valor 0, ou seja, $G\ (\forall s'.\ trans(s,s') \rightarrow (V(s') < V(s) \vee V(s') = 0))$

In [14]:
```python
kinduction_always(declare,init,trans,variante_decrescente,4)
```

```
A propriedade é válida
```

- Quando o variante é 0 verifica-se necessariamente ϕ, ou seja, $G\ (V(s) = 0 \rightarrow \phi(s))$

In [15]:
```python
kinduction_always(declare,init,trans,termina,3)
```

```
A propriedade é válida
```

Codificar usando a LPA a forma recursiva deste programa

$$W \quad \equiv \quad \{\,\text{assume}\, b\,;\, S\,;\, W\,\} \,\|\, \{\,\text{assume}\, \neg b\,\}$$

```
S = (assume (y & 1 == 1); y = y - 1; r = r + x || assume (y & 1 != 1); skip); x = x<<1; y = y>>1;
W = {assume y > 0; (assume (y & 1 == 1); y = y - 1; r = r + x || assume (y & 1 != 1); skip); x = x<<1; y =
y>>1; W} ||
    {assume not (y > 0)}
```

Provar a correção usando a metodologia dos invariantes

Propor o invariante que assegure a correção

In [16]:
```python
from tabulate import tabulate

# assume m >= 0 and n >= 0 and r == 0 and x == m and y == n

x = x0 = 10
y = y0 = 15
r = 0

#inv: x0 * y0 = x * y + r and y >= 0

headers = ['x', 'y', 'r', 'val de inv']
tabela = []

while y > 0:
    l = [x, y, r, (x0 * y0 == x*y+r and y >= 0)]
    tabela.append(l)
    if y & 1 == 1:
        y , r  = y-1 , r+x
        l = [x, y, r, (x0 * y0 == x*y+r and y >= 0)]
        tabela.append(l)
    x , y = x<<1  ,  y>>1
l = [x, y, r, (x0 * y0 == x*y+r and y >= 0)]
tabela.append(l)

print(tabulate(tabela, headers))
```

```
  x    y    r  val de inv
---  ---  ---  ------------
 10   15    0  True
 10   14   10  True
 20    7   10  True
 20    6   30  True
 40    3   30  True
 40    2   70  True
 80    1   70  True
 80    0  150  True
160    0  150  True
```

Pela tabela acima verificamos que o invariante $(m * n = x * y + r \wedge y \geq 0)$ é verdade em qualquer iteração do ciclo.

Codificar em SMT e provar a correção

Iremos provar a correção do ciclo usando *havoc*.

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

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

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

Iremos então traduzir o triplo de Hoare $\{\phi\}$while b do$\{\theta\}$ C $\{\psi\}$, da seguinte forma, de modo a garantir as propriedades de "inicialização", "preservação" e "utilidade" do invariante θ

$$\begin{array}{l}
[\,\text{assume } \phi\,;\text{assert } \theta\,;\text{havoc } \vec{x}\,;(\,(\text{assume } b \wedge \theta\,;\ C\,;\text{assert } \theta\,;\text{assume } False) \,||\, \text{assume } \neg b \wedge \theta\,)\,;\text{assert } \psi\,] \\
= \\
\phi \rightarrow \theta \wedge \forall \vec{x}.\ (\,(b \wedge \theta \rightarrow [C\,;\text{assert } \theta]) \wedge (\neg b \wedge \theta \rightarrow \psi)\,)
\end{array}$$

Em primeiro lugar, iremos traduzir o programa para a linguagem de fluxos com *havoc*:

```
assume pre;
assert invariante;
havoc x; havoc y; havoc r;
((assume b and invariante; C; assert invariante; assume False) || assume (not b) and invariante);
assert pos
```

Em seguida iremos calcular a denotação lógica deste programa de fluxos pela WPC, calculada pelas seguintes regras:

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

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

```
[assume pre; assert invariante; havoc x,y,r ; ((assume (b and invariante); C; assert invariante; assume False)
||
assume ((not b) and invariante)); assert pos]
=
pre -> inv and forall (x, y, r) . ((b and inv -> [C; assert inv]) and (not b and inv -> pos))
=
pre -> inv and forall (x, y, r) . ((y > 0 and inv -> [
  (assume (y & 1 == 1); y = y - 1; r = r + x || assume (y & 1 != 1); skip);
  x = x<<1;
  y = y>>1;
  assert inv
  ])
  and (not y > 0 and inv -> pos))
=
pre -> inv and forall (x, y, r) . ((y > 0 and inv -> ([
  assume (y & 1 == 1); y = y - 1; r = r + x;
  x = x<<1;
  y = y>>1;
  assert inv
  ] || [
  assume (y & 1 != 1); skip;
  x = x<<1;
  y = y>>1;
  assert inv
  ]))
  and (not y > 0 and inv -> pos))
=
pre -> inv and forall (x, y, r) . ((y > 0 and inv -> (
  (y & 1 == 1) -> [
  y = y - 1;
  r = r + x;
  x = x<<1;
  y = y>>1;
  assert inv
  ] and
  (y & 1 != 1) -> [
  x = x<<1;
  y = y>>1;
  assert inv
  ]))
  and (not y > 0 and inv -> pos))
=
pre -> inv and forall (x, y, r) . ((y > 0 and inv -> (
  (y & 1 == 1) -> [
  x = x<<1;
  y = y>>1;
  assert inv
  ] [y-1 / y] [r+x / r] and
  (y & 1 != 1) -> [
  x = x<<1;
  y = y>>1;
  assert inv
  ] ))
  and (not y > 0 and inv -> pos))
=
pre -> inv and forall (x, y, r) . ((y > 0 and inv -> (
  (y & 1 == 1) -> [
  assert inv
  ] [y-1 / y] [r+x / r] [x<<1 / x] [y>>1 / y] and
  (y & 1 != 1) -> [
  assert inv
  ] [x<<1 / x] [y>>1 / y] ))
  and (not y > 0 and inv -> pos))
=
pre -> inv and forall (x, y, r) . ((y > 0 and inv -> (
  (y & 1 == 1) -> inv [y-1 / y] [r+x / r] [x<<1 / x] [y>>1 / y] and
  (y & 1 != 1) -> inv [x<<1 / x] [y>>1 / y] ))
  and (not y > 0 and inv -> pos))
```

Finalmente, passaremos à prova da correção do programa, usando o Z3.

In [17]:
```python
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 [18]:
```python
#pre -> inv and forall (x, y, r) . ((y > 0 and inv -> ((y & 1 == 1) -> inv [y-1 / y] [r+x / r] [x<<1 / x] [y>>
#                                     and (y & 1 != 1) -> inv [x<<1 / x] [y>>1 / y] ))
#                                    and (not y > 0 and inv -> pos))

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

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

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

# (y & 1 == 1) -> inv [y-1 / y] [r+x / x] [x<<1 / r] [y>>1 / y]
if_true = Implies(y & 1 == 1,
                  substitute(substitute(substitute(substitute(inv, (y, y>>1)), (x, x<<1)), (r,r+x)), (y, y-1)))

# (y & 1 != 1) -> inv [x<<1 / x] [y>>1 / y]
if_false = Implies(y & 1 != 1, substitute(substitute(inv, (y, y>>1)),(x, x<<1)))

#pre -> inv and forall (x,y,r) . ((y > 0 and inv -> if_true and if_false) and (not y > 0 and inv -> pos))
vc = Implies(pre, And(inv, ForAll([x,y,r], And(Implies(And(y > 0, inv), And(if_true, if_false)),
                                              Implies(And(Not(y > 0), inv), pos)))) )

prove(vc)
```

```
Proved
```

Provar a correção usando a metodologia do "single assignment unfolding"

O algoritmo SAU consiste em calcular sucessivamente os vários predicados

$$W_n(v)$$

e testar

$$\phi(v) \rightarrow \bigvee_n W_n(v)$$

é uma tautologia.

Para a prova da correção através da metodologia SAU utilizaremos a seguinte implementação, que é uma versão deste algoritmo.

In [19]:
```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)
```

Modelação do ciclo com a pré-condição $(m >= 0 \wedge n >= 0 \wedge r == 0 \wedge x == m \wedge y == n \wedge n < N \wedge m < N)$ e a pós-condição $(r == m * n)$

In [20]:
```python
bits = 16
L = 2^16 -1

# constantes auxiliares
N    = BV(L,width=bits)
zero = BV(0,width=bits)
um   = BV(1,width=bits)

# O ciclo
x  = Symbol("x",BVType(bits))
m  = Symbol("m",BVType(bits))
n  = Symbol("n",BVType(bits))
y  = Symbol("y",BVType(bits))
r  = Symbol("r",BVType(bits))
variables = [x,m,n,r,y]

pre  =  And([BVUGE(m,zero),  # m > 0
             BVUGE(n,zero),  # n > 0
             Equals(r,zero), # r = 0
             Equals(x,m),    # x = m
             Equals(y,n),    # y = n
             #BVULT(m,N),    # m < N
             #BVULT(n,N)     # n < N
            ])

pos  =  Equals(r,BVMul(m,n)) # r = m * n

cond =  BVUGT(y , zero)      # y > 0

left     = And(Equals(prime(y), BVLShr(BVSub(y, um), um)), # y = (y-1) >> 1
               Equals(prime(r), BVAdd(r, x)),              # r = r + x
               Equals(prime(x), BVLShl(x, um)))            # x = x << 1

right    = And(Equals(prime(y), BVLShr(y, um)),            # y = y >> 1
               Equals(prime(x), BVLShl(x, um)))            # x = x << 1

trans = Ite(Equals(BVAnd(y, um), um), left, right)       # if (y & 1 == 1) then left else right

W = Cycle(variables,pre,pos,cond,trans)

W.unroll(bits + 1)
```

```
sucesso: tentativa 17
```