

TP2-Exercicio1

November 15, 2022

1 TP2

1.1 Grupo 15

Carlos Eduardo Da Silva Machado A96936

Gonçalo Manuel Maia de Sousa A97485

1.2 Problema 1

1.2.1 Descrição do Problema

É nos dado um Control Flow Automaton (CFA) que descreve um programa imperativo cujo objetivo é implementar a multiplicação de dois inteiros a, b , fornecidos como “input” e um n , também fornecido como “input”, de precisão limitada. Para além disso, temos de ter em conta os seguintes aspetos:

- Existe a possibilidade de alguma das operações do programa produzir um erro de “overflow”;
- Os nós do grafo representam ações que actuam sobre os “inputs” do nó e produzem um “output” com as operações indicadas;
- Os ramos do grafo representam ligações que transferem o “output” de um nodo para o “input” do nodo seguinte. Esta transferência é condicionada pela satisfação da condição associada ao ramo.

1.2.2 Abordagem do Problema

Para resolver este problema, vamos construir um First Order Transition System (FOTS) usando *BitVector*'s de tamanho n de forma a descrever o comportamento do autómato acima mencionado.

São parâmetros do problema a , b , n , e k tais que:

1. a é o valor inicial de x
2. b é o valor inicial de y
3. k é o número máximo de estados num traço do problema, toma o valor de $n + 1$ visto que, esse é o número de transições necessárias para computar o pior caso possível.
4. n é o número de *bit*'s máximo das variáveis

O autómato consiste na seguinte estrutura:

1. Um estado final ($pc = 1$).
2. Um estado de erro ($pc = 2$) que marca o estado de *overflow*
3. Um estado de operações ($pc = 0$) no qual todas as operações sobre as variáveis serão realizadas

De modo a tratar de casos de *overflow* as variáveis x , y e z são declaradas como *BitVector*'s de tamanho $n + 1$. Assim se o primeiro bit de uma delas for 1 podemos transitar para o estado de *overflow*

Além disso, por motivos de otimização no caso da variável b ser maior do que a , são trocadas para que o número de transições seja minimizado.

Para além do FOTS, também vamos verificar se $P(xy + z = ab)$ é um invariante do comportamento que estamos a estudar.

1.3 Código Python

Algoritmo básico variaveis $\rightarrow x, y, z, pc$

```
0: while(y!=0):
    if even(y) then x,y,z = 2*x,y/2,z
    if odd(y)  then x,y,z = x,y-1,z+x
1: stop
```

Vamos Utilizar a biblioteca do *Pysmt* e a biblioteca *random* para resolver este exercício.

```
[1]: from pysmt.shortcuts import *
      from pysmt.typing import INT
      import random as rn
```

Construção do FOTS:

Função de declaração:

```
[2]: def declare(i,n):
      state = {}
      state['pc'] = Symbol('pc'+str(i),INT)
      state['x'] = Symbol('x'+str(i),types.BVType(n+1))
      state['y'] = Symbol('y'+str(i),types.BVType(n+1))
      state['z'] = Symbol('z'+str(i),types.BVType(n+1))
      return state
```

Função de inicialização:

```
[3]: def init(state,a,b,n):
      if b > a:
          a,b = b,a

      tPc = Equals(state['pc'],Int(0)) # Program counter a zero
      tZ = Equals(state['z'],BVZero(n+1)) # Z a zero
      tX = Equals(state['x'], BV(a,n+1)) # x inicilizado com valor de a
      tY = Equals(state['y'], BV(b,n+1)) # y inicilizado com valor de b
      return And(tPc,tX,tY,tZ)
```

Função de Transição:

$$\text{trans}(x, y, z, pc, x', y', z', pc') \equiv$$

$$\left\{ \begin{array}{l}
(pc = 0) \wedge \text{even}(y) \wedge (y > 0) \wedge (x' = 2x) \wedge (y' = \frac{y}{2}) \wedge (z' = c) \wedge (pc' = 0) \\
(pc = 0) \wedge \text{odd}(y) \wedge (x' = x) \wedge (y' = y - 1) \wedge (z' = x + z) \wedge (pc' = 0) \\
(pc = 0) \wedge (y = 0) \wedge \text{overflow}(z) \wedge (x' = x) \wedge (y' = y) \wedge (z' = c) \wedge (pc' = 1) \\
(pc = 1) \wedge (x' = x) \wedge (y' = y) \wedge (z' = z) \wedge (pc' = 1) \\
(pc = 0) \wedge \text{overflow}(y) \wedge \text{overflow}(x) \wedge \text{overflow}(z) \wedge (x' = x) \wedge (y' = y) \wedge (z' = c) \wedge (pc' = 2) \\
(pc = 2) \wedge (x' = x) \wedge (y' = y) \wedge (z' = z) \wedge (pc' = 2)
\end{array} \right. \quad \begin{array}{l} \vee \\ \vee \\ \vee \\ \vee \\ \vee \\ \vee \end{array}$$

```

[4]: def BVFirst(n):
    return BV(2**(n-1),n)

def tEven(curr,prox,n):
    tPcZero = Equals(curr['pc'],Int(0))
    tYLast = Equals(BVAnd(curr['y'],BVOne(n+1)),BVZero(n+1)) #ultimo bit = 0
    tYGt = BVUGT(curr['y'],BVZero(n+1)) #y > 0
    tX = Equals(prox['x'], BVLShl(curr['x'],BVOne(n+1))) #2*x
    tY = Equals(prox['y'], BVLShr(curr['y'],BVOne(n+1))) #y/2
    tZ = Equals(prox['z'],curr['z']) #z
    tPc = Equals(prox['pc'],Int(0))
    return And(tPcZero,tYLast,tYGt,tX,tY,tZ,tPc)

def tOdd(curr,prox,n):
    tPcZero = Equals(curr['pc'],Int(0))
    tYLast = Equals(BVAnd(curr['y'],BVOne(n+1)),BVOne(n+1))
    tX = Equals(prox['x'], curr['x'])
    tY = Equals(prox['y'],BVSub(curr['y'],BVOne(n+1)))
    tZ = Equals(prox['z'],BVAdd(curr['x'],curr['z']))
    tPc = Equals(prox['pc'],Int(0))
    return And(tPcZero,tYLast,tX,tY,tZ,tPc)

def tStop(curr,prox,n):
    tPcZero = Equals(curr['pc'],Int(0))
    tYZero = Equals(curr['y'],BVZero(n+1)) #y=0
    tZFirst = Equals(BVAnd(curr['z'],BVFirst(n+1)),BVZero(n+1)) #primero bit de z
    ↪ z = 0
    tX = Equals(prox['x'],curr['x'])
    tY = Equals(prox['y'],curr['y'])
    tZ = Equals(prox['z'],curr['z'])
    tPc = Equals(prox['pc'],Int(1))
    return And(tYZero,tZFirst,tPcZero,tX,tY,tZ,tPc)

def tEnd(curr,prox):
    tPcOne = Equals(curr['pc'],Int(1))
    tX = Equals(prox['x'],curr['x'])
    tY = Equals(prox['y'],curr['y'])
    tZ = Equals(prox['z'],curr['z'])
    tPc = Equals(prox['pc'],Int(1))
    return And(tPcOne,tX,tY,tZ,tPc)

```

```

def tError(curr,prox,n):
    tPcZero = Equals(curr['pc'],Int(0))
    tYFirst = Equals(BVAnd(curr['y'],BVFirst(n+1)),BVFirst(n+1))
    tXFirst = Equals(BVAnd(curr['x'],BVFirst(n+1)),BVFirst(n+1))
    tZFirst = Equals(BVAnd(curr['z'],BVFirst(n+1)),BVFirst(n+1))
    tX = Equals(prox['x'], curr['x'])
    tY = Equals(prox['y'],curr['y'])
    tZ = Equals(prox['z'],curr['z'])
    tPc = Equals(prox['pc'],Int(2))
    return And(tPcZero,Or(tYFirst,tXFirst,tZFirst),tX,tY,tZ,tPc)

def tEndError(curr,prox):
    tPcTwo = Equals(curr['pc'],Int(2))
    tX = Equals(prox['x'], curr['x'])
    tY = Equals(prox['y'],curr['y'])
    tZ = Equals(prox['z'],curr['z'])
    tPc = Equals(prox['pc'],Int(2))
    return And(tPcTwo,tX,tY,tZ,tPc)

def trans(curr,prox,n):
    tToStop = tStop(curr,prox,n)
    tToEven = tEven(curr,prox,n)
    tToError = tError(curr,prox,n)
    tToEndError = tEndError(curr,prox)
    tToOdd = tOdd(curr,prox,n)
    tToEnd = tEnd(curr,prox)
    return Or(tToStop,tToEven,tToError,tToEndError,tToOdd,tToEnd)

```

Função que usa *SMT solver* para gerar um possível traço de execução do programa, imprimindo, para cada estado, as variáveis x,y,z e o program counter e função que auxiliar na conversão das variáveis para inteiro.

```

[5]: def toInt(s):
    return sum([int(x)*2**(len(s)-i-1) for i,x in enumerate(s)])

```

```

[6]: def resolve(a,b,n,k):
    with Solver(name="msat") as s:
        # cria k copias do estado
        trace = [declare(i,n) for i in range(k)]
        #print(trace)
        # criar o traço
        s.add_assertion(init(trace[0],a,b,n))
        #print(init(trace[0]))
        for i in range(k-1):
            s.add_assertion(trans(trace[i], trace[i+1],n))

    if s.solve():

```

```

        for i in range(k):
            print(i)
            print("pc=", pc := s.get_value(trace[i]['pc'])).
→constant_value()
            print("x=", toInt(s.get_value(trace[i]['x']).bv_str()))
            print("y=", toInt(s.get_value(trace[i]['y']).bv_str()))
            print("z=", toInt(s.get_value(trace[i]['z']).bv_str()))
            print()
            if pc in (1,2):
                break
        else:
            print('Não foi possível resolver')

```

O invariante $P(xy + z = ab)$ como função `invariant(state,a,b)` e a função de ordem superior `bmc_always(declare,init,trans,inv,K,a,b,n)` que testa se o invariante é verificado para traços de tamanho máximo k .

```

[7]: def invariant(state,a,b):
        return Equals(BVAdd(BVMul(state['x'], state['y']), state['z']), BVMul(BV(a,
→n+1), BV(b, n+1)))

def bmc_always(declare,init,trans,inv,K,n,a,b):
    for k in range(1,K+1):
        with Solver(name="z3") as s:

            trace = [declare(i,n) for i in range(k)]

            s.add_assertion(init(trace[0],a,b,n))
            for i in range(k-1):
                s.add_assertion(trans(trace[i], trace[i+1],n))

            s.add_assertion(Not(inv(trace[k-1],a,b)))
            if s.solve():
                for i in range(k):
                    for v in trace[0]:
                        print(v, '=', s.get_value(trace[0][v]))
                return

        print("A propriedade é válida para traços de tamanho até " + str(k))

```

1.3.1 Exemplos e Testes de Aplicação

Exemplo 1

```

[8]: n = 7
      a = 4
      b = 3
      k = n+1

```

```
print('x = ', BV(a,n).bv_str())
print('y = ', BV(b,n).bv_str())
```

```
x = 0000100
y = 0000011
```

Resolução do Exemplo 1 Este exemplo é apenas uma mostra de uma multiplicação básica.

[9]: `resolve(a,b,n,k)`

```
0
pc= 0
x= 4
y= 3
z= 0
```

```
1
pc= 0
x= 4
y= 2
z= 4
```

```
2
pc= 0
x= 8
y= 1
z= 4
```

```
3
pc= 0
x= 8
y= 0
z= 12
```

```
4
pc= 1
x= 8
y= 0
z= 12
```

[10]: `bmc_always(declare,init,trans,invariant,k,n,a,b)`

A propriedade é válida para traços de tamanho até 8

Exemplo 2 Neste exemplo procuramos apresentar um dos piores casos em termos de transições de estado.

```
[8]: n = 6
      a = 7
      b = 7
      k = n+1
      k_inv = 1

      print('x = ', BV(a,n).bv_str())
      print('y = ', BV(b,n).bv_str())
```

```
x = 000111
y = 000111
```

Resolução do Exemplo 2

```
[9]: resolve(a,b,n,k)
```

```
0
pc= 0
x= 7
y= 7
z= 0
```

```
1
pc= 0
x= 7
y= 6
z= 7
```

```
2
pc= 0
x= 14
y= 3
z= 7
```

```
3
pc= 0
x= 14
y= 2
z= 21
```

```
4
pc= 0
x= 28
y= 1
z= 21
```

```
5
pc= 0
x= 28
```

```
y= 0
z= 49
```

```
6
pc= 1
x= 28
y= 0
z= 49
```

```
[10]: bmc_always(declare,init,trans,invariant,k,n,a,b)
```

A propriedade é válida para traços de tamanho até 7

Exemplo 3 Neste exemplo procuramos mostrar a optimização feita de modo a que sejam efetuadas o menor número de transições possíveis.

```
[8]: n = 15
a = 1
b = 32767
k = n+1

print('x = ', BV(a,n).bv_str())
print('y = ', BV(b,n).bv_str())
```

```
x = 0000000000000001
y = 1111111111111111
```

Resolução do exemplo 3

```
[9]: resolve(a,b,n,k)
```

```
0
pc= 0
x= 32767
y= 1
z= 0
```

```
1
pc= 0
x= 32767
y= 0
z= 32767
```

```
2
pc= 1
x= 32767
y= 0
z= 32767
```



```
[10]: bmc_always(declare,init,trans,invariant,k,n,a,b)
```

A propriedade é válida para traços de tamanho até 16

Exemplo 4 Neste exemplo procuramos mostrar um caso de *overflow*.

```
[7]: n = 32
a = 65535
b = 131069
k = n+1

print('x = ', BV(a,n).bv_str())
print('y = ', BV(b,n).bv_str())
```

```
x = 00000000000000000111111111111111
y = 000000000000000001111111111111101
```

Resolução do exemplo 4

```
[8]: resolve(a,b,n,k)
```

```
0
pc= 0
x= 131069
y= 65535
z= 0
```

```
1
pc= 0
x= 131069
y= 65534
z= 131069
```

```
2
pc= 0
x= 262138
y= 32767
z= 131069
```

```
3
pc= 0
x= 262138
y= 32766
z= 393207
```

```
4
pc= 0
x= 524276
y= 16383
```

z= 393207

5

pc= 0

x= 524276

y= 16382

z= 917483

6

pc= 0

x= 1048552

y= 8191

z= 917483

7

pc= 0

x= 1048552

y= 8190

z= 1966035

8

pc= 0

x= 2097104

y= 4095

z= 1966035

9

pc= 0

x= 2097104

y= 4094

z= 4063139

10

pc= 0

x= 4194208

y= 2047

z= 4063139

11

pc= 0

x= 4194208

y= 2046

z= 8257347

12

pc= 0

x= 8388416

y= 1023

z= 8257347

13

pc= 0

x= 8388416

y= 1022

z= 16645763

14

pc= 0

x= 16776832

y= 511

z= 16645763

15

pc= 0

x= 16776832

y= 510

z= 33422595

16

pc= 0

x= 33553664

y= 255

z= 33422595

17

pc= 0

x= 33553664

y= 254

z= 66976259

18

pc= 0

x= 67107328

y= 127

z= 66976259

19

pc= 0

x= 67107328

y= 126

z= 134083587

20

pc= 0

x= 134214656

y= 63

z= 134083587

21

pc= 0

x= 134214656

y= 62

z= 268298243

22

pc= 0

x= 268429312

y= 31

z= 268298243

23

pc= 0

x= 268429312

y= 30

z= 536727555

24

pc= 0

x= 536858624

y= 15

z= 536727555

25

pc= 0

x= 536858624

y= 14

z= 1073586179

26

pc= 0

x= 1073717248

y= 7

z= 1073586179

27

pc= 0

x= 1073717248

y= 6

z= 2147303427

28

pc= 0

x= 2147434496

y= 3

```
z= 2147303427
```

```
29
```

```
pc= 0
```

```
x= 2147434496
```

```
y= 2
```

```
z= 4294737923
```

```
30
```

```
pc= 0
```

```
x= 4294868992
```

```
y= 1
```

```
z= 4294737923
```

```
31
```

```
pc= 0
```

```
x= 4294868992
```

```
y= 0
```

```
z= 8589606915
```

```
32
```

```
pc= 2
```

```
x= 4294868992
```

```
y= 0
```

```
z= 8589606915
```

```
[11]: bmc_always(declare,init,trans,invariant,k,n,a,b)
```

A propriedade é válida para traços de tamanho até 33

Exemplo 5 Este exemplo serve para ser possível efetuar testes repetidos com variáveis aleatórias.

```
[19]: n = 32
a = rn.randrange(1, 2**(n))
b = rn.randrange(1, 2**(n))
k = n+1

print('x = ', BV(a,n).bv_str())
print('y = ', BV(b,n).bv_str())
```

```
x = 10011100101110110100100011100000
```

```
y = 01101011111101100010110011100010
```

Resolução do exemplo 5

```
[20]: resolve(a,b,n,k)
```

```
0
```

```
pc= 0
```

x= 2629519584
y= 1811295458
z= 0

1
pc= 0
x= 5259039168
y= 905647729
z= 0

2
pc= 0
x= 5259039168
y= 905647728
z= 5259039168

3
pc= 0
x= 1928143744
y= 452823864
z= 5259039168

4
pc= 0
x= 3856287488
y= 226411932
z= 5259039168

5
pc= 0
x= 7712574976
y= 113205966
z= 5259039168

6
pc= 0
x= 6835215360
y= 56602983
z= 5259039168

7
pc= 2
x= 6835215360
y= 56602983
z= 5259039168

[21]: `bmc_always(declare,init,trans,invariant,k,n,a,b)`

A propriedade é válida para traços de tamanho até 33