Trabalho Prático 2

Grupo 22

Alexis Correia - A102495 João Fonseca - A102512

Enunciado

Considere o problema descrito no documento +Lógica Computacional: Multiplicação de Inteiros . Nesse documento usa-se um "Control Flow Automaton" como modelo do programa imperativo que calcula a multiplicação de inteiros positivos representados por vetores de bits.

Pretende-se

1. Construir um SFOTS, usando BitVec's de tamanho n , que descreva o comportamento deste autómato; para isso identifique e codifique em `Z3` ou `pySMT`, as variáveis do modelo, o estado inicial , a relação de transição e o estado de erro.
2. Usando k-indução verifique nesse SFOTS se a propriedade $\left(x*y+z=a*b\right)$ é um invariante do seu comportamento.
3. Usando k-indução no FOTS acima e adicionando ao estado inicial a condição $\left(a<2^{n/2}\right) \wedge \left(b<2^{n/2}\right)$, verifique a segurança do programa; nomeadamente prove que, com tal estado inicial, o estado de erro nunca é acessível.

Resolução

Um SFOTS é definido por $\Sigma \equiv \langle X, next, I, T, E \rangle$. Similiar a um FOTS, os estados são constituídos pelas variáveis do programa mais o **program counter** (pc) e, tanto o estado inicial quanto as relações de trnsição, são caracterizados por predicados. A maior diferença se encontra na existência do estado de erro.

Comecemos, então, por analisar o programa imperativo em questão.

```
{a >= 0 and b >= 0}
0: x , y, z = a , b , 0
1: while not y == 0:
2:  if even(y):
3:      x , y , z = x << 1 , y >> 1 , z
4:  else:
5:      x , y , z = x , y - 1, z + x
6: stop
```

- Note-se que no final deve ser $ z = a\times b $
- Neste pedaço de código, $ x $ e $ y $ são vetores de bits e as operações de `<<`(Shift Left) e `>>`(Shift Right) são equivalentes as operações `*2` e `/2` com inteiros. Podemos, ainda, supor que a função `even` se parece com algo do género:

```
0: def even(n):
1:   return (n%2==0)
```

- As variáveis do programa são: $ a $, $ b $, $ x $, $ y $ e $ z $
- O estado inicial é caracterizado pelo predicado: $pc=0 \wedge a\geq 0 \wedge b\geq 0$
- As transições possíveis são caracterizadas das seguintes formas:

$$(pc=0 \wedge a\geq 0 \wedge b\geq 0 \wedge pc'=1 \wedge a'=a \wedge b'=b \wedge x'=a \wedge y'=b \wedge z'=0)$$
$$\vee$$
$$(pc=1 \wedge y\neq 0 \wedge pc'=2 \wedge a'=a \wedge b'=b \wedge x'=x \wedge y'=y \wedge z'=z)$$
$$\vee$$
$$(pc=1 \wedge y=0 \wedge pc'=6 \wedge a'=a \wedge b'=b \wedge x'=x \wedge y'=y \wedge z'=z)$$
$$\vee$$
$$(pc=2 \wedge (y\%2)=0 \wedge pc'=3 \wedge a'=a \wedge b'=b \wedge x'=x \wedge y'=y \wedge z'=z)$$
$$\vee$$
$$(pc=2 \wedge (y\%2)=1 \wedge pc'=5 \wedge a'=a \wedge b'=b \wedge x'=x \wedge y'=y \wedge z'=z)$$
$$\vee$$
$$(pc=3 \wedge pc'=1 \wedge a'=a \wedge b'=b \wedge x'=x<¿1 \wedge y'=y>¿1 \wedge z'=z)$$
$$\vee$$
$$(pc=5 \wedge pc'=1 \wedge a'=a \wedge b'=b \wedge x'=x \wedge y'=y-1 \wedge z'=z+x)$$
$$\vee$$
$$(pc=6 \wedge pc'=6 \wedge a'=a \wedge b'=b \wedge x'=x \wedge y'=y \wedge z'=z)$$

- O estado de erro acontece caso haja um `overflow` (tanto em $x=x<¿1$ como em $z=z+x$);

```
from pysmt.shortcuts import *
from pysmt.typing import BVType, INT

n = 16 # num de bits

# SFOTS #

def declare(i):
    state = {}
    state['pc'] = Symbol('pc'+str(i), INT)
    state['a'] = Symbol('a'+str(i), BVType(n))
    state['b'] = Symbol('b'+str(i), BVType(n))
    state['x'] = Symbol('x'+str(i), BVType(n))
    state['y'] = Symbol('y'+str(i), BVType(n))
    state['z'] = Symbol('z'+str(i), BVType(n))
    return state

def init(state):
    A = BVSGE(state['a'], BV(0,n))
    B = BVSGE(state['b'], BV(0,n))
    C = Equals(state['pc'], Int(0))
    return And(A,B,C)
```

```
def trans(curr, prox):
    t01 = And(Equals(curr['pc'], Int(0)), BVSGE(curr['a'], BV(0,n)),
BVSGE(curr['b'], BV(0,n)),
              Equals(prox['pc'], Int(1)), Equals(prox['a'],
curr['a']), Equals(prox['b'], curr['b']),
              Equals(prox['x'], curr['a']), Equals(prox['y'],
curr['b']), Equals(prox['z'], BV(0,n)))

    t12 = And(Equals(curr['pc'], Int(1)), Not(Equals(curr['y'],
BV(0,n))),
              Equals(prox['pc'], Int(2)), Equals(prox['a'],
curr['a']), Equals(prox['b'], curr['b']),
              Equals(prox['x'], curr['x']), Equals(prox['y'],
curr['y']), Equals(prox['z'], curr['z']))

    t16 = And(Equals(curr['pc'], Int(1)), Equals(curr['y'], BV(0,n)),
              Equals(prox['pc'], Int(6)), Equals(prox['a'],
curr['a']), Equals(prox['b'], curr['b']),
              Equals(prox['x'], curr['x']), Equals(prox['y'],
curr['y']), Equals(prox['z'], curr['z']))

    t23 = And(Equals(curr['pc'], Int(2)),
Equals(BVExtract(curr['y'],0,0), BV(0,1)),
              Equals(prox['pc'], Int(3)), Equals(prox['a'],
curr['a']), Equals(prox['b'], curr['b']),
              Equals(prox['x'], curr['x']), Equals(prox['y'],
curr['y']), Equals(prox['z'], curr['z']))

    t25 = And(Equals(curr['pc'], Int(2)),
Equals(BVExtract(curr['y'],0,0), BV(1,1)),
              Equals(prox['pc'], Int(3)), Equals(prox['a'],
curr['a']), Equals(prox['b'], curr['b']),
              Equals(prox['x'], curr['x']), Equals(prox['y'],
curr['y']), Equals(prox['z'], curr['z']))

    t31 = And(Equals(curr['pc'], Int(3)), Equals(prox['pc'], Int(1)),
Equals(prox['a'], curr['a']), Equals(prox['b'], curr['b']),
              Equals(prox['x'], BVLShl(curr['x'], 1)),
Equals(prox['y'], BVLShr(curr['y'], 1)), Equals(prox['z'], curr['z']))

    t51 = And(Equals(curr['pc'], Int(5)), Equals(prox['pc'], Int(1)),
Equals(prox['a'], curr['a']), Equals(prox['b'], curr['b']),
              Equals(prox['x'], curr['x']), Equals(prox['y'],
BVSub(curr['y'], BV(1,n))), Equals(prox['z'],
BVAdd(curr['z'],curr['x'])))

    t66 = And(Equals(curr['pc'], Int(6)), Equals(prox['pc'], Int(6)),
Equals(prox['a'], curr['a']), Equals(prox['b'], curr['b']),
```

```
              Equals(prox['x'], curr['x']), Equals(prox['y'],
curr['y']), Equals(prox['z'], curr['z']))

    return Or(t01, t12, t16, t23, t25, t31, t51, t66)

def error(state):
    of_Shl = BVULT(BV(2**n - 1, n), BVLShl(state['x'], 1))
    of_Add = BVULT(BV(2**n - 1, n), BVAdd(state['z'], state['x']))
    return Or(of_Shl, of_Add)
```

Dessa forma, está pronta nosso **SFOTS**. Com as funções acima (`declare`, `init`, `trans` e `error`) devidamente definidas, podemos escrever `genTrace` que escreverá o traço de comprimento K e o valor de cada variável em cada estado i tal que $i \in 0, 1, 2, \ldots, K$.

```
def genTrace(declare, init, trans, error, K):
    with Solver(name="z3") as s:
        X = [declare(i) for i in range(K+1)]
        I = init(X[0])
        Tks = [trans(X[i],X[i+1]) for i in range(K) ]

        if s.solve([I,And(Tks) ]):
            for i in range(K):
                print("Estado:",i)
                print(f"          pc = {s.get_value(X[i]['pc'])}")
                print(f"          a = {s.get_value(X[i]['a'])}, b =
{s.get_value(X[i]['b'])}")
                for v in ['x', 'y', 'z']:
                    p = format(s.get_value(X[i][v]).constant_value(),
f'0{n}b')
                    print(f'          {v} = {p} ({s.get_value(X[i]
[v])})')
        else:
            print("ERROR")

genTrace(declare, init, trans, error, 20)

Estado: 0
          pc = 0
          a = 16384_16, b = 1_16
          x = 0000000000000000 (0_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 1
          pc = 1
          a = 16384_16, b = 1_16
          x = 0100000000000000 (16384_16)
          y = 0000000000000001 (1_16)
          z = 0000000000000000 (0_16)
Estado: 2
```

```
          pc = 2
          a = 16384_16, b = 1_16
          x = 0100000000000000 (16384_16)
          y = 0000000000000001 (1_16)
          z = 0000000000000000 (0_16)
Estado: 3
          pc = 3
          a = 16384_16, b = 1_16
          x = 0100000000000000 (16384_16)
          y = 0000000000000001 (1_16)
          z = 0000000000000000 (0_16)
Estado: 4
          pc = 1
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 5
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 6
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 7
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 8
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 9
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 10
          pc = 6
```

```
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 11
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 12
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 13
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 14
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 15
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 16
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 17
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 18
          pc = 6
          a = 16384_16, b = 1_16
```

```
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
Estado: 19
          pc = 6
          a = 16384_16, b = 1_16
          x = 1000000000000000 (32768_16)
          y = 0000000000000000 (0_16)
          z = 0000000000000000 (0_16)
```

Agora, verificaremos se a propriedade $(x*y+z=a*b)$ é um invariante do comportamento desta SFOTS com k-indução.

Primeiramente, podemos afirmar empiricamente que a variável y sempre chega ao final do código com valor 0 e, por isso, $(x*y + z = a*b)\implies (z=a*b) $ que é da facto o que este código calcula. Logo, sabemos que no fim da execução, esta equação é verdadeira (Desde que não haja `overflow`).

Porém, ainda precisamos verificar com auxílio de k-indução se a afirmação é verdadeira em todos os estados e se esta propriedade é de facto invariante de SFOTS.

```python
def kinduction_always(declare,init,trans,inv,k):
    with Solver(name="z3") as solver:
        s = [declare(i) for i in range(k)]
        solver.add_assertion(init(s[0]))
        for i in range(k-1):
            solver.add_assertion(trans(s[i],s[i+1]))

        for i in range(k):
            solver.push()
            solver.add_assertion(Not(inv(s[i])))
            if solver.solve():
                print(f"> Contradição! O invariante não se verifica 
nos k estados iniciais.")
                return
            solver.pop()

        s2 = [declare(i+k) for i in range(k+1)]

        for i in range(k):
            solver.add_assertion(inv(s2[i]))
            solver.add_assertion(trans(s2[i],s2[i+1]))

        solver.add_assertion(Not(inv(s2[-1])))

        if solver.solve():
            print(f"> Contradição! O passo indutivo não se verifica.")
            return
```

```
        print(f"> A propriedade verifica-se por k-indução (k={k}).")
    return

def inv(state): # Invariante: x*y+z=a*b
    R = BVAdd(BVMul(state['x'],state['y']),state['z'])
    L = BVMul(state['a'], state['b'])
    return Equals(R, L)

kinduction_always(declare,init,trans,inv, 2)

> Contradição! O invariante não se verifica nos k estados iniciais.
```

Com as funções `kinduction_always` e `inv`, pudemos verificar que, de facto, a propriedade não invariante. Ela falha logo no primeiro estado (estado inicial).

Por fim, vamos averiguar a segurança do programa ao adicionar a condição $\left(a<2^{n/2}\right) \wedge \left(b<2^{n/2}\right)$ ao estado inicial. Mais uma vez, nos utilizaremos da k-indução neste exercício.

```
def init1(state):
    I = init(state)
    maxA = BVSLT(state['a'], BV(2**(n//2),n))
    maxB = BVSLT(state['b'], BV(2**(n//2),n))
    return And(I,maxA,maxB)

def safe(state):
    X = BVULE(state['x'], BV((2**n)-1,n))
    Z = BVULE(state['z'], BV((2**n)-1,n))
    return  And(X,Z)

kinduction_always(declare, init1, trans, safe, 20)

> A propriedade verifica-se por k-indução (k=20).
```

Como podemos ver, ao adicionar esta nova condição ao estado inicial, evitamos por completo que aconteça **overflow** do x e do z.