TP 1 EX 1

Grupo 5:

Breno Fernando Guerra Marrão A97768

Tales André Rovaris Machado A96314

Inicialização

Usamos as bibliotecas pysmt para resovler o problema proposto

In [1]:
```
from pysmt.shortcuts import *
from pysmt.typing import INT
```

Transformando CFA em um ciclo while

```
0:{ z == 0 and y >= 0 }
1: while (while y != 0:):
2:    if y % 2:
        x = 2*x
        y = y/2
        if overflow:
5:           OVERFLOW
3:     else:
        y = y-1
        z = z+x

4: stop
```

Transições

O estado inicial é caracterizado pelo seguinte predicado:

$$z = 0 \wedge pc = 0 \wedge y \geq 0$$

As transições possíveis no FOTS são caracterizadas pelo seguinte predicado:

$$(pc = 0 \wedge pc' = 1 \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\vee$$
$$(pc = 1 \wedge pc' = 2 \wedge y \mod 2 = 0 \wedge x' = x \wedge y' = y \wedge z' = z \wedge y \neq 0)$$
$$\vee$$
$$(pc = 2 \wedge pc' = 1 \wedge x' \geq x \wedge x' = x * 2 \wedge y' = y \div 2 \wedge z' = z)$$
$$\vee$$
$$(pc = 2 \wedge pc' = 5 \wedge x > x * 2 \wedge y' = y \div 2 \wedge z' = z)$$
$$\vee$$
$$(pc = 5 \wedge pc' = 5 \wedge x' = x \wedge y' = y \div 2 \wedge z' = z)$$
$$\vee$$
$$(pc = 1 \wedge pc' = 3 \wedge y \mod 2 = 1 \wedge x' = x \wedge y' = y \div 2 \wedge z' = z, \wedge y \neq 0)$$
$$\vee$$
$$(pc = 3 \wedge pc' = 1 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x)$$
$$\vee$$
$$(pc = 1 \wedge pc' = 4 \wedge y = 0 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x)$$
$$\vee$$
$$(pc = 4 \wedge pc' = 4 \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\vee$$
$$(pc = 3 \wedge pc' = 5 \wedge z > z + x \wedge y' = y - 1 \wedge x' = x)$$

Implementação

A seguinte função cria a i-ésima cópia das variáveis de estado, agrupadas num dicionário que nos permite aceder às mesmas pelo nome.

In [2]:
```
def declare(i,n):
    state = {}
    state['pc'] = Symbol('pc'+str(i),INT)
    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
```

Função que dado um possível estado do programa , devolve um predicado do pySMT que testa se esse estado é um possível estado inicial do programa

In [3]:
```
def init(state,a,b,n):

    return And(Equals(state['x'],BV(a,n)),
               Equals(state['y'],BV(b,n)),
               Equals(state['z'],BV(0,n)),
               Equals(state['pc'],Int(0)))
```

Função `trans` que, dados dois possíveis estados do programa, devolve um predicado do pySMT que testa se é possível transitar do primeiro para o segundo

In [4]:
```
def trans(curr,prox,bv2,n):
    bv1 = BVOne(n)
    t0 = And(Equals(curr['pc'],Int(0)),
             Equals(prox['pc'],Int(1)),
             Equals(curr['x'],prox['x']),
             Equals(curr['y'],prox['y']),
             Equals(curr['z'],prox['z']))

    t1 = And(Equals(curr['pc'],Int(1)),
             Equals(BVAdd(curr['y'],bv1),BVXor(curr['y'],bv1)),
             Not(Equals(prox['y'],BVZero(n))),
             Equals(prox['pc'],Int(2)),
             Equals(curr['x'],prox['x']),
             Equals(curr['y'],prox['y']),
             Equals(curr['z'],prox['z']))

    t2 = And(Equals(curr['pc'],Int(2)),
             Equals(prox['pc'],Int(1)),
             BVUGE(prox['x'],curr['x']),
             Equals(BVMul(curr['x'],bv2),prox['x']),
             Equals(BVUDiv(curr['y'],bv2),prox['y']),
             Equals(curr['z'],prox['z']))

    te = And(Equals(curr['pc'],Int(2)),
             Equals(prox['pc'],Int(5)),
             BVUGT(curr['x'],BVMul(curr['x'],bv2)),
             Equals(curr['y'],prox['y']),
             Equals(curr['z'],prox['z']))

    te2 = And(Equals(curr['pc'],Int(5)),
             Equals(prox['pc'],Int(5)),
             Equals(curr['x'],prox['x']),
             Equals(curr['y'],prox['y']),
             Equals(curr['z'],prox['z']))

    t3 = And(Equals(curr['pc'],Int(1)),
             Equals(BVSub(curr['y'],bv1),BVXor(curr['y'],bv1)),
             Not(Equals(prox['y'],BVZero(n))),
             Equals(prox['pc'],Int(3)),
             Equals(curr['x'],prox['x']),
             Equals(curr['y'],prox['y']),
             Equals(curr['z'],prox['z']))

    t4 = And(Equals(curr['pc'],Int(3)),
             Equals(prox['pc'],Int(1)),
             Equals(curr['x'],prox['x']),
             Equals(BVSub(curr['y'],BVOne(n)),prox['y']),
             Equals(curr['z']+curr['x'],prox['z']),
             BVUGE(prox['z'],curr['z']))

    te3 = And(Equals(curr['pc'],Int(3)),
             Equals(prox['pc'],Int(5)),
             Equals(curr['x'],prox['x']),
             Equals(BVSub(curr['y'],BVOne(n)),prox['y']),
             Equals(curr['z']+curr['x'],prox['z']),
             BVUGT(curr['z'],curr['x']+curr['z']))

    t5 = And(Equals(curr['pc'],Int(1)),
             Equals(prox['pc'],Int(4)),
             Equals(curr['x'],prox['x']),
             Equals(curr['y'],prox['y']),
             Equals(curr['z'],prox['z']),
             Equals(curr['y'],BVZero(n)))

    t6 = And(Equals(curr['pc'],Int(4)),
             Equals(prox['pc'],Int(4)),
             Equals(curr['x'],prox['x']),
             Equals(curr['y'],prox['y']),
             Equals(curr['z'],prox['z']))

    return Or(t1,t2,t3,t0,t4,t5,t6,te,te2,te3)
```

Função de ordem superior `gera_traco` que, dada uma função que gera uma cópia das variáveis do estado, um predicado que testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, e um número positivo `k`, usa o SMT solver para gerar um possível traço de execução do programa de tamanho `k`.

In [5]:
```
def gera_traco(declare,init,trans,a,b,n,k):

    with Solver(name="z3") as s:

        bv2 = BV(2,n)

        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],bv2,n))

        if s.solve():
            for i in range(k):
                print("Passo", i)
                for v in trace[i]:
                    print(v,"=",s.get_value(trace[i][v]))
                if i>1:
                    if s.get_value(trace[i]["pc"]) == s.get_value(trace[i-1]["pc"]):
                        if s.get_value(trace[i]["pc"]) == Int(4):
                            print("Multiplicação efetuada com sucesso e valor :", s.get_value(BVToNatural(trace[i][
                            return
                        else:
                            print("OVERFLOW")
                            return
```

Testes e análise dos resultados

Caso normal que o programa consegue calcular facilmente:

In [6]:
```
gera_traco(declare,init,trans,5,5,5,30)
```

```
Passo 0
pc = 0
x = 5_5
y = 5_5
z = 0_5
Passo 1
pc = 1
x = 5_5
y = 5_5
z = 0_5
Passo 2
pc = 3
x = 5_5
y = 5_5
z = 0_5
Passo 3
pc = 1
x = 5_5
y = 4_5
z = 5_5
Passo 4
pc = 2
x = 5_5
y = 4_5
z = 5_5
Passo 5
pc = 1
x = 10_5
y = 2_5
z = 5_5
Passo 6
pc = 2
x = 10_5
y = 2_5
z = 5_5
Passo 7
pc = 1
x = 20_5
y = 1_5
z = 5_5
Passo 8
pc = 3
x = 20_5
y = 1_5
z = 5_5
Passo 9
pc = 1
x = 20_5
y = 0_5
z = 25_5
Passo 10
pc = 4
x = 20_5
y = 0_5
z = 25_5
Passo 11
pc = 4
x = 20_5
y = 0_5
z = 25_5
Multiplicação efetuada com sucesso e valor : 25
```

Caso em que da overflow

In [7]:
```
gera_traco(declare,init,trans,2,31,5,30)
```

```
Passo 0
pc = 0
x = 2_5
y = 31_5
z = 0_5
Passo 1
pc = 1
x = 2_5
y = 31_5
z = 0_5
Passo 2
pc = 3
x = 2_5
y = 31_5
z = 0_5
Passo 3
pc = 1
x = 2_5
y = 30_5
z = 2_5
Passo 4
pc = 2
x = 2_5
y = 30_5
z = 2_5
Passo 5
pc = 1
x = 4_5
y = 15_5
z = 2_5
Passo 6
pc = 3
x = 4_5
y = 15_5
z = 2_5
Passo 7
pc = 1
x = 4_5
y = 14_5
z = 6_5
Passo 8
pc = 2
x = 4_5
y = 14_5
z = 6_5
Passo 9
pc = 1
x = 8_5
y = 7_5
z = 6_5
Passo 10
pc = 3
x = 8_5
y = 7_5
z = 6_5
Passo 11
pc = 1
x = 8_5
y = 6_5
z = 14_5
Passo 12
pc = 2
x = 8_5
y = 6_5
z = 14_5
Passo 13
pc = 1
x = 16_5
y = 3_5
z = 14_5
Passo 14
pc = 3
x = 16_5
y = 3_5
z = 14_5
Passo 15
pc = 1
x = 16_5
y = 2_5
z = 30_5
Passo 16
pc = 2
x = 16_5
y = 2_5
z = 30_5
Passo 17
pc = 5
x = 0_5
```

```
y = 2_5
z = 30_5
Passo 18
pc = 5
x = 0_5
y = 2_5
z = 30_5
OVERFLOW
```

Limite de resolução do programa

In [6]:
```
gera_traco(declare,init,trans,67108864,67108864,52,1000)
```

```
Passo 0
pc = 0
x = 67108864_52
y = 67108864_52
z = 0_52
Passo 1
pc = 1
x = 67108864_52
y = 67108864_52
z = 0_52
Passo 2
pc = 2
x = 67108864_52
y = 67108864_52
z = 0_52
Passo 3
pc = 1
x = 134217728_52
y = 33554432_52
```

Invariante

Para verificação do invariante temos que garantir que é valido no init, também que a transição é valida e que a negação do invariante para todas as transições é verdadeira, ou seja:

$$P(init) \wedge trans(x, x') \wedge \neg P(x') \implies \bot$$

é sempre unsolvable, portanto o invariante é verdadeiro

In [6]:
```
def bmc_always(declare,init,trans,inv,a,b,n,K):

    bv2 = BV(2,n)
    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],bv2,n))

            s.add_assertion(Not(And(inv(trace[i],a,b) for i in range(k-1))))
            if s.solve():
                for i in range(k):
                    print("Passo", i)
                    for v in trace[i]:
                        print(v,"=",s.get_value(trace[i][v]))
                    print("----------------")
                print("NAO E INV")
                return
    print("É invariante")

def inv(trace,a,b):
    return Equals(BVToNatural(trace['x'])*BVToNatural(trace['y'])+BVToNatural(trace['z']), Int(a*b))
```

In [7]:
```
bmc_always(declare,init,trans,inv,5,5,5,15)
```

```
É invariante
```