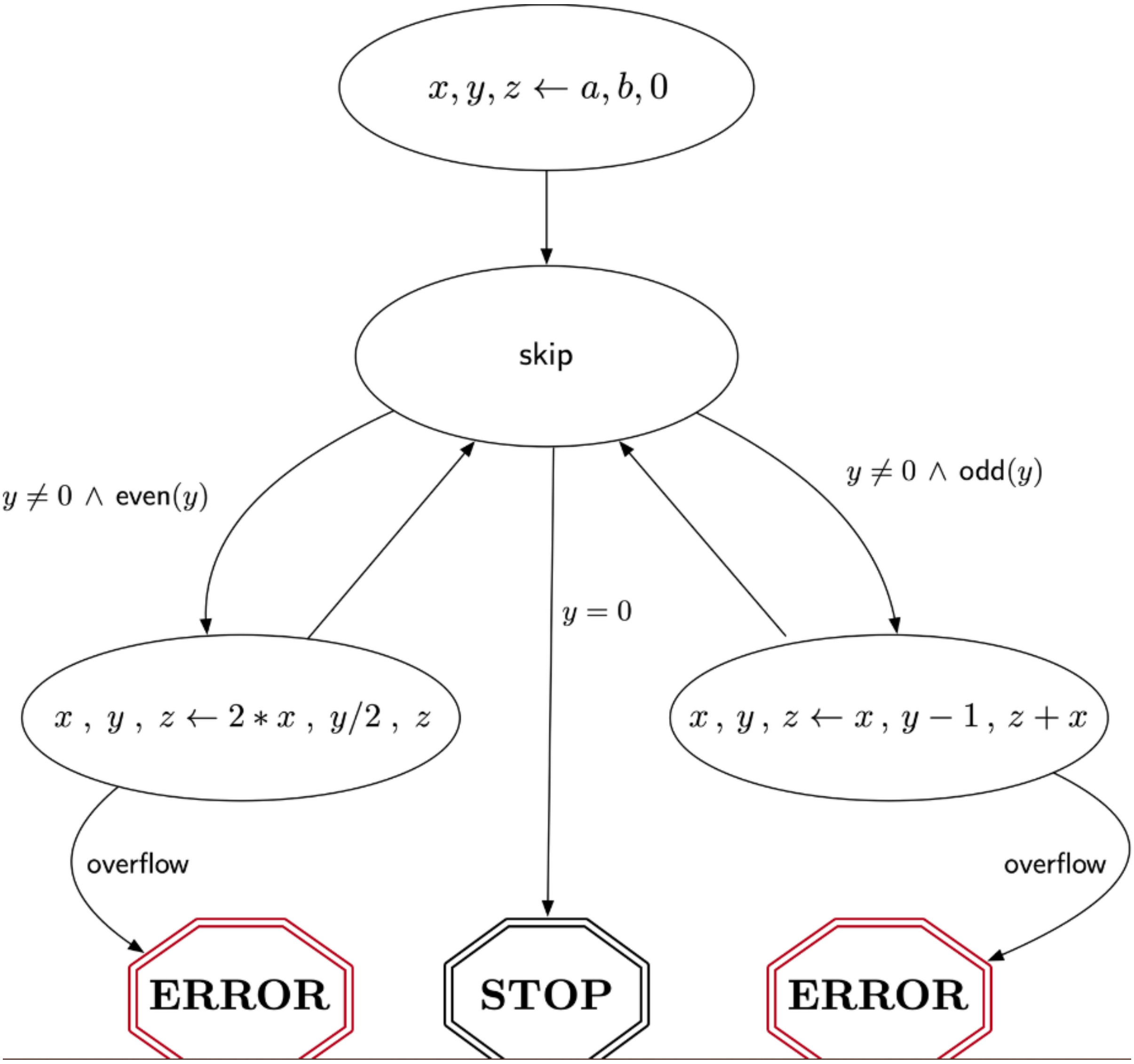
TP2 - Trabalho 1

11 de novembro de 2022

André Oliveira Barbosa - A91684 Francisco Antonio Borges Paulino - A91666

Caso de estudo



Análise do Problema

É possivel verificar que ao longo do programa:

```
. 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 ass
ociada ao ramo .
```

Inicialização

In [4]:

```
!pip install ortools
```

```
Requirement already satisfied: ortools in c:\users\andre\anaconda3\envs\logi
ca\lib\site-packages (9.4.1874)
Requirement already satisfied: absl-py>=0.13 in c:\users\andre\anaconda3\env
s\logica\lib\site-packages (from ortools) (1.2.0)
Requirement already satisfied: numpy>=1.13.3 in c:\users\andre\anaconda3\env
s\logica\lib\site-packages (from ortools) (1.23.3)
Requirement already satisfied: protobuf>=3.19.4 in c:\users\andre\anaconda3
\envs\logica\lib\site-packages (from ortools) (4.21.6)
```

Implementação

Começamos por importar o módulo pysmt.shortcuts que oferece uma API simplificada que disponibiliza as funcionalidades para a utilização usual de um SMT solver. Os tipos estão definidos no módulo pysmt.typing de onde temos que importar o tipo INT e BVType.

In [5]:

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

In [6]:

```
a=2
b=21
n=32
#print(n)
```

```
32
```

Para modelarmos o programa em questão através do FOTS iremos definir:

```
. O estado é constituído pelas variáveis do programa mais uma variável para o resp
ectivo program counter;

. 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
 ;
```

```
```Python
0:{ x=a && y=b && z=0 }
1: while(y!=0):
2:    if (y%2 == 0):
        x = 2*x
        y = y/2
3:    else:
        y = y-1
        z = z+x
4: stop
5:overflow
```
```

Usando estes predicados podemos usar um SMT solver. Para tal precisamos de criar n cópias das variáveis que caracterizam o estado do FOTS e depois impor que a primeira cópia satisfaz o predicado inicial e que cada par de cópias consecutivas satisfazem o predicado de transição.

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

In [7]:

```python
def declare(i):
    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
```

Dado um possível estado do programa (um dicionário de variáveis), a seguinte função devolve um predicado do pySMT que testa se esse estado é um possível estado inicial do programa.

In [14]:

```python
def init(state):
    return And(Equals(state['pc'], Int(0)), Equals(state['x'], BV(a,n)), Equals(state['y'],
```

O estado dos FOTS será um conjunto de inteiros contendo o valor pc (o program counter que será 0,1,2,3,4,5), o segundo o valor da variavel x, o terceiro o valor da variavel y e o quarto do valor da variavel z. O estado inicial é caracterizado pelo seguinte perdicado:

$$pc = 0 \land x = a \land y = b \land z = 0$$

As transições possiveis no FOTS:

$$
\begin{gathered}
(pc = 0 \land pc' = 1 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(pc = 1 \land y! = 0 \land y\%2 = 0 \land pc' = 2 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(pc = 1 \land y! = 0 \land y\%2! = 0 \land pc' = 3 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(pc = 1 \land y = 0 \land pc' = 4 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(pc = 2 \land x \geq 2 * x \land pc' = 5 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(pc = 3 \land z \geq z + x \land pc' = 5 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(pc = 2 \land pc' = 1 \land x' = 2 * x \land y' = y/2 \land z' = z) \\
\lor \\
(pc = 3 \land pc' = 1 \land x' = x \land y' = y - 1 \land z' = z + x) \\
\lor \\
(pc = 4 \land pc' = 4 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(pc = 5 \land pc' = 5 \land x' = x \land y' = y \land z' = z)
\end{gathered}
$$

Dados dois possíveis estados do programa, a seguinte função devolve um predicado do pySMT que testa se é possível transitar do primeiro para o segundo.

In [9]:

```
def trans(curr,prox):
    t0 = And(Equals(curr['pc'], Int(0)),Equals(prox['pc'], Int(1)), Equals(prox['x'], curr[
    t1 = And(Equals(curr['pc'], Int(1)), NotEquals(curr['y'],BV(0,n)),NotEquals(BVAnd(curr[
    t2 = And(Equals(curr['pc'], Int(1)), NotEquals(curr['y'], BV(0,n)),Equals(BVAnd(curr['y
    t3 = And(Equals(curr['pc'], Int(1)), Equals(curr['y'], BV(0,n)),Equals(prox['pc'], Int(
    t4 = And(Equals(curr['pc'], Int(2)),BVUGE(prox['x'], curr['x']*2),Equals(prox['pc'], In
    t5 = And(Equals(curr['pc'], Int(3)),BVUGE(prox['z'], BVAdd(curr['z'],curr['x'])),Equals
    t6 = And(Equals(curr['pc'], Int(2)),Equals(prox['pc'], Int(1)), Equals(prox['x'], curr[
    t7 = And(Equals(curr['pc'], Int(3)),Equals(prox['pc'], Int(1)), Equals(prox['x'], curr[
    t8 = And(Equals(curr['pc'], Int(4)),Equals(prox['pc'], Int(4)), Equals(prox['x'], curr[
    t9 = And(Equals(curr['pc'], Int(5)),Equals(prox['pc'], Int(5)), Equals(prox['x'], curr[

    return Or(t0, t1, t2, t3, t4, t5, t6, t7 ,t8, t9)
```

Função gera_traço

Utilizamos o SMT solver para gerar um possível traço de execução do programa de tamanho k. Para cada estado do traço deverá imprimir o respectivo valor das variáveis.

In [12]:

```
def gera_traco(declare,init,trans,k):

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

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

        # adicionar o estado inicial
        s.add_assertion(init(trace[0]))

        # adicionar a função transição
        for i in range(k-1):
            s.add_assertion(trans(trace[i], trace[i+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('-------')

gera_traco(declare,init,trans,16)
```

```
Passo 0
pc = 0
x = 2_32
y = 21_32
z = 0_32
-------
Passo 1
pc = 1
x = 2_32
y = 21_32
z = 0_32
-------
Passo 2
pc = 3
x = 2_32
y = 21_32
z = 0_32
-------
Passo 3
pc = 1
x = 2_32
y = 20_32
z = 2_32
-------
Passo 4
pc = 2
x = 2_32
y = 20_32
z = 2_32
-------
Passo 5
pc = 1
x = 4_32
y = 10_32
z = 2_32
```

```
-------
Passo 6
pc = 2
x = 4_32
y = 10_32
z = 2_32
-------
Passo 7
pc = 1
x = 8_32
y = 5_32
z = 2_32
-------
Passo 8
pc = 3
x = 8_32
y = 5_32
z = 2_32
-------
Passo 9
pc = 1
x = 8_32
y = 4_32
z = 10_32
-------
Passo 10
pc = 2
x = 8_32
y = 4_32
z = 10_32
-------
Passo 11
pc = 1
x = 16_32
y = 2_32
z = 10_32
-------
Passo 12
pc = 2
x = 16_32
y = 2_32
z = 10_32
-------
Passo 13
pc = 1
x = 32_32
y = 1_32
z = 10_32
-------
Passo 14
pc = 3
x = 32_32
y = 1_32
z = 10_32
-------
Passo 15
pc = 1
x = 32_32
y = 0_32
z = 42_32
-------
```

Verificação do invariante

Segue a verificação do seguinte invariante

$$P \equiv (x * y + z = a * b)$$

In [15]:

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

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

            # adicionar o estado inicial
            s.add_assertion(init(trace[0]))

            # adicionar a função transição
            for i in range(k-1):
                s.add_assertion(trans(trace[i], trace[i+1]))

            # adicionar a negação do invariante
            s.add_assertion(Not(And(inv(trace[i]) 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("A propriedade não é invariante")
                return

    print(f"O invariante mantém-se nos primeiros {K} passos")

def nonnegative(state):
    c = BV(a,n)
    f = BV(b,n)
    return (Equals(BVAdd((BVMul(state['x'],state['y'])),state['z']),BVMul(c,f)))

bmc_always(declare,init,trans,nonnegative,20)
```

```
O invariante mantém-se nos primeiros 20 passos
```