TP2 - Control Flow Automaton

Novembro, 2022

Bruno Miguel Ferreira Fernandes - a95972

Hugo Filipe de Sá Rocha - a96463

Funções auxiliares:

- even(x) - Verifica se x é par.
- odd(x) - Verifica se x é par.

In [1]:
```
def even(x):
    return (x%2 == 0)

def odd(x):
    return (x%2 != 0)
```

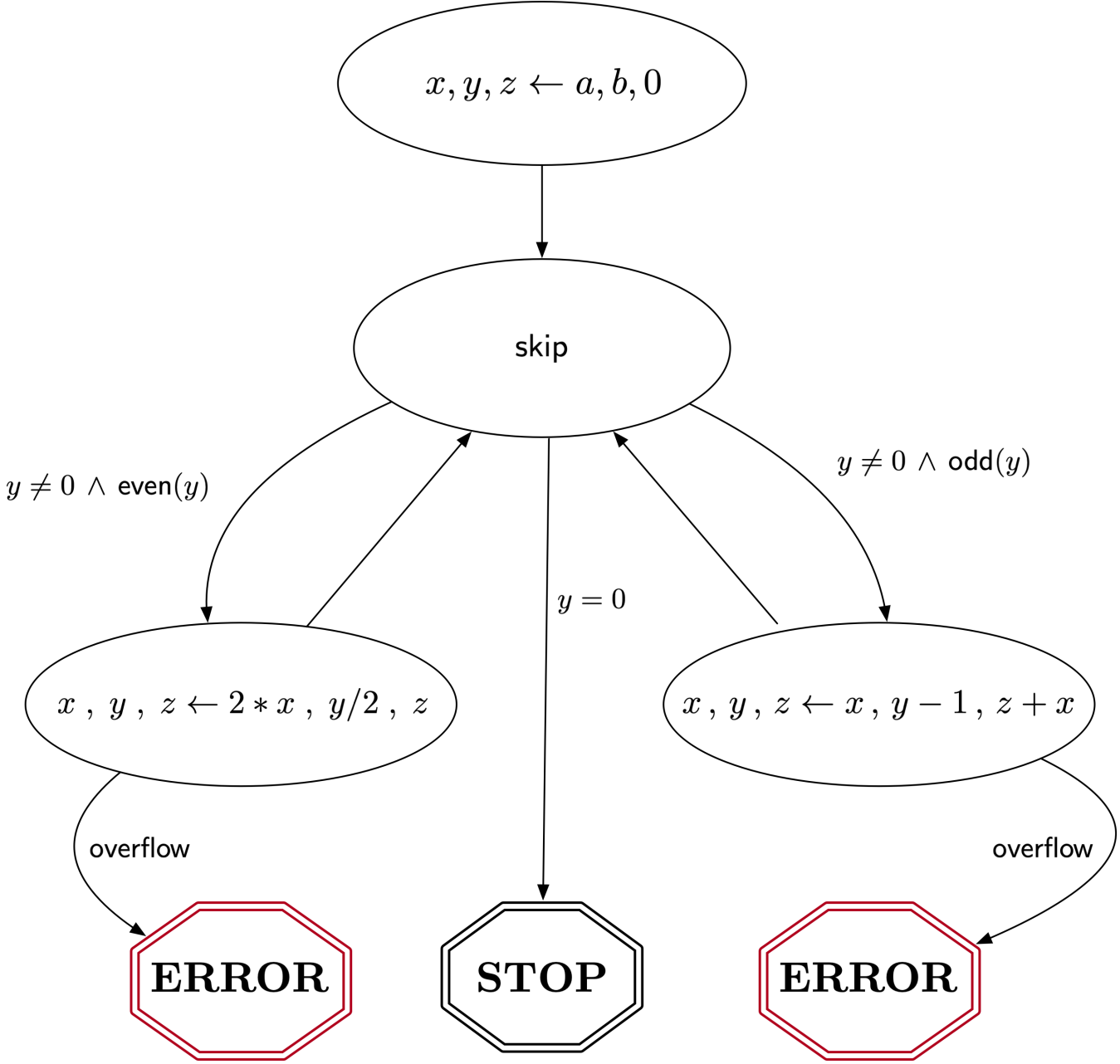
Definir valores de input do problema

In [2]:
```
#imports
from pysmt.shortcuts import *
from pysmt.typing import INT
from z3 import *

#inputs do problema
global a
global b
global n
a = 2
b = 3
n = 3
# 9 passos



#a = 2
#b = 3    caso overflow
#n = 2
# 6 passos
```



O estado inicial é caracterizado pelo seguinte predicado:

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

As transições possíveis no FOTS são caracterizadas pelo seguinte predicado:

$$(pc = 0 \wedge pc' = 1 \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\vee$$
$$(pc = 1 \wedge y \neq 0 \wedge even(y) \wedge pc' = 2 \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\vee$$
$$(pc = 1 \wedge y \neq 0 \wedge odd(y) \wedge pc' = 3 \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\vee$$
$$(pc = 1 \wedge y = 0 \wedge pc' = 4 \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\vee$$
$$(pc = 2 \wedge (x \geq 2x) \wedge pc' = 5 \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\vee$$
$$(pc = 2 \wedge (x < 2x) \wedge pc' = 1 \wedge x' = 2 * x \wedge y' = y/2 \wedge z' = z)$$
$$\vee$$
$$(pc = 3 \wedge pc' = 1 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x)$$
$$\vee$$
$$(pc = 3 \wedge (z \geq (z + x)) \wedge pc' = 6 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x)$$

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

In [3]:
```
def declare(i):
    state = {}
    state['pc'] = Int('pc'+str(i))
    state['x'] = BitVec('x'+str(i),n)
    state['y'] = BitVec('y'+str(i),n)
    state['z'] = BitVec('z'+str(i),n)
    return state
```

A seguinte função `init`, dado um possível estado do programa (um dicionário de variáveis), devolve um predicado do pySMT que testa se esse estado é um possível estado inicial do programa.

In [4]:
```
def init(state):
    return And((state['pc'] == 0), (state['x'] == a),
               (state['y'] == b), (state['z'] == 0))
```

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

In [5]:
```
def trans(curr,prox):
    t0 = And((curr['pc'] == 0),
             (prox['pc'] == 1),
             (prox['x'] == curr['x']),
             (prox['y'] == curr['y']),
             (prox['z'] == curr['z'])
         )



    t1 = And((curr['pc'] == 1),
             (curr['y'] != 0),
            even(curr['y']),
             (prox['pc'] == 2),
             (prox['x'] == curr['x']),
             (prox['y'] == curr['y']),
             (prox['z'] == curr['z'])
         )
```

```
    t2 = And((curr['pc'] == 1),
             (curr['y'] != 0),
             odd(curr['y']),
             (prox['pc'] == 3),
             (prox['x'] == curr['x']),
             (prox['y'] == curr['y']),
             (prox['z'] == curr['z'])

         )

    #atinge estado final 4
    t3 = And((curr['pc'] == 1),
             (curr['y'] == 0),
             (prox['pc'] == 4),
             (prox['x'] == curr['x']),
             (prox['y'] == curr['y']),
             (prox['z'] == curr['z'])
         )


    #OVERFLOW (atinge estado final 5)
    t4 = And((curr['pc'] == 2),
             UGE(curr['x'], curr['x']*2),
             (prox['pc'] == 5),
             (prox['x'] == curr['x']),
             (prox['y'] == curr['y']),
             (prox['z'] == curr['z'])
            )



    t5 = And((curr['pc'] == 2),
             Not(UGE(curr['x'], curr['x']*2)),
             (prox['pc'] == 1),
             (prox['x'] == curr['x']*2),
             (prox['y'] == curr['y']/2),
             (prox['z'] == curr['z'])
            )

    t6 = And((curr['pc'] == 3),
             (prox['pc'] == 1),
             (prox['x'] == curr['x']),
             (prox['y'] == curr['y']-1),
             (prox['z'] == curr['z']+curr['x'])
         )

    #OVERFLOW DIREITA (atinge estado final 6)
    t7 = And((curr['pc'] == 3),
             UGE(curr['z'], curr['z'] + curr['x']),
             (prox['pc'] == 6),
             (prox['x'] == curr['x']),
             (prox['y'] == curr['y']-1),
             (prox['z'] == curr['z'] + curr['x'])
            )

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

A função `gera_traco`, 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 gera um possível traço de execução do programa de tamanho k.

In [6]:
```
def gera_traco(declare,init,trans,k):
```

```
        s = Solver()

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

        s.add(init(trace[0]))

        for i in range(k - 1):
            s.add(trans(trace[i], trace[i+1]))

        if s.check() == sat:
            m = s.model()
            for i in range(k):
                print("Passo", i)
                for v in trace[i]:
                    print(v, "=", m[trace[i][v]])
                print("----------------")
        else:
            print("Não foi possível gerar o traço.\n")


gera_traco(declare,init,trans,9)
```

```
Passo 0
pc = 0
x = 2
y = 3
z = 0
----------------
Passo 1
pc = 1
x = 2
y = 3
z = 0
----------------
Passo 2
pc = 3
x = 2
y = 3
z = 0
----------------
Passo 3
pc = 1
x = 2
y = 2
z = 2
----------------
Passo 4
pc = 2
x = 2
y = 2
z = 2
----------------
Passo 5
pc = 1
x = 4
y = 1
z = 2
----------------
Passo 6
pc = 3
x = 4
y = 1
z = 2
----------------
Passo 7
```

```
pc = 1
x = 4
y = 0
z = 6
----------------
Passo 8
pc = 4
x = 4
y = 0
z = 6
----------------
```

A seguinte função `bmc_always`, 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, um invariante a verificar, e um número positivo de passos K, averigua se o candidato a invariante é ou não válido.

In [7]:
```
def bmc_always(declare,init,trans,inv,K):
    s = Solver()


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


    s.add(init(trace[0]))

    for i in range(K-1):
         s.add(trans(trace[i], trace[i+1]))

    s.add(Not(And([inv(trace[i]) for i in range(K-1)])))

    if s.check() == sat:
                print("A propriedade não é um invariante.\n")
                return

    else:
        print(f'O invariante verificou-se em todos os passos.\n')


def inv(state):
    return ((state['x'] * state['y'] + state['z']) == a*b)

bmc_always(declare,init,trans,inv,9)
```

```
O invariante verificou-se em todos os passos.
```