TP3 - 1

1. Pretende-se construir uma implementação simplificada do algoritmo “model checking” orientado aos interpolantes seguindo a estrutura apresentada nos apontamentos onde no passo (n, m) na impossibilidade de encontrar um interpolante invariante se dá ao utilizador a possibilidade de incrementar um dos índices n e m à sua escolha. Pretende-se aplicar este algoritmo ao problema da da multiplicação de inteiros positivos em `BitVec` (apresentado no TP2).

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

autómato

Função genState(vars, s, i, n):

```
vars - Variáveis a serem declaradas
s - Nome da variável
i - Valor do traço atual
n - Número de bits utilizados
```

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

In [2]:
```python
def genState(vars, s, i, n):
    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+s+str(i), BVType(n))
    return state
```

Função init(state, a, b, n):

```
state - Dicionário de variáveis de estado
a - Valor associado ao 1º número a multiplicar
b - Valor associado ao 2º número a multiplicar
n - Número de bits a serem usados
```

A função `init` tem como objetivo devolver um predicado do Solver que testa se é um possível estado inicial do programa, através do `state`, um dicionário de variáveis.

In [3]:
```python
def init(state, a, b, n):
    return And(
        Equals(state['pc'], BV(0, n)),
        Equals(state['x'], BV(a, n)),
        Equals(state['y'], BV(b, n)),
        Equals(state['z'], BV(0, n))
    )
```

Função trans(curr, prox, n):

```
curr - Estado das variáveis no momento atual
prox - Estado das variáveis no momento da próxima iteração
n - Número de bits
```

A função `trans` tem como objetivo devolver um predicado do Solver, através dos três estados disponíveis, que teste se é possível transitar entre os estados possíveis.

Função error(state, n):

```
state - Variáveis do programa num certo estado do programa
n - Número de bits
```

A função `error` tem como objetivo devolver um predicado do Solver que verifica se o programa se encontra num estado de erro.

In [4]:
```python
def error(state, n):
    return Or(
        Equals(state['pc'], BV(4, n)),
        Equals(state['pc'], BV(6, n))
    )


def trans(curr, prox, n):
    same_values = And(
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z'])
    )

    t0 = And(
        Equals(curr['pc'], BV(0, n)),
        Equals(prox['pc'], BV(1, n)),
        same_values
    )

    # y = 0
    t1 = And(
        Equals(curr['y'], BV(0, n)),
        Equals(curr['pc'], BV(1, n)),
        Equals(prox['pc'], BV(5, n)),
        same_values
    )

    # y != 0 ^ odd(y)
    t2 = And(
        NotEquals(curr['y'], BV(0, n)),
        Equals(BVURem(curr['y'], BV(2, n)), BV(1, n)),
        Equals(curr['pc'], BV(1, n)),
        Equals(prox['pc'], BV(2, n)),
        same_values
    )


    # y != 0 ^ even(y)
    t3 = And(
        NotEquals(curr['y'], BV(0, n)),
        Equals(BVURem(curr['y'], BV(2, n)), BV(0, n)),
        Equals(curr['pc'], BV(1, n)),
        Equals(prox['pc'], BV(3, n)),
        same_values
    )



    # transição em que o solver decide se vai para o estado de overflow ou se continua
    magia_left = And(
        Equals(prox['x'], BVLShl(curr['x'], BV(1, n))),
        Equals(prox['y'], BVLShr(curr['y'], BV(1, n))),
        Equals(prox['z'], curr['z']),

        Equals(curr['pc'], BV(3, n)),

        Or(
            And(BVUGE(prox['x'], curr['x']), Equals(prox['pc'], BV(1, n))),  # curr['x'] <= prox['x']  - não há overflow
            And(BVUGT(curr['x'], prox['x']), Equals(prox['pc'], BV(4, n))),  # curr['x'] > prox['x'] - há overflow
        )
    )

    # transição em que o solver decide se vai para o estado de overflow ou se continua
    magia_right = And(
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], BVSub(curr['y'], BV(1, n))),
        Equals(prox['z'], BVAdd(curr['z'], curr['x'])),

        Equals(curr['pc'], BV(2,n)),

        Or(
            And(BVUGE(prox['x'], curr['x']), Equals(prox['pc'], BV(1, n))),  # curr['x'] <= prox['x']  - não há overflow
            And(BVUGT(curr['x'], prox['x']), Equals(prox['pc'], BV(6, n))),  # curr['x'] > prox['x'] - há overflow
        )
    )


    # caso de paragem no overflow e no estado final
    stop_case = And(
        Equals(prox['pc'], curr['pc']),
        same_values,

        Or(
            And(Equals(curr['pc'], BV(4, n)), Equals(prox['pc'], BV(4, n))),
            And(Equals(curr['pc'], BV(5, n)), Equals(prox['pc'], BV(5, n))),
            And(Equals(curr['pc'], BV(6, n)), Equals(prox['pc'], BV(6, n)))
        )
    )


    return Or(t0, t1, t2, t3, stop_case, magia_left, magia_right)
```

Função gera_traco(vars, init, trans, error, k, n, a, b)

```
vars - Variáveis a declarar
init - Função que devolve um predicado que representa o estado inicial do programa
trans - Função transição
k - Tamanho do traço
n - Número de bits a utilizar
a - Valor para a multiplicação
b - Valor para a multiplicação
```

A função `gera_traco` tem como objetivo imprimir o valor das variáveis à medida que vão percorrendo os estados, através das variáveis do estado, de um predicado que testa se um estado é inicial, um número positivo para gerar um possível traço de execução do programa de tamanho `k` , com `n` bits, multiplicando `a` por `b` .

In [5]:
```python
def gera_traco(vars,init,trans, error, k, n, a, b):
    with Solver(name='z3') as s:

        X = [genState(vars, 'X', i, n) for i in range(k+1)]
        I = init(X[0], a, b, n)
        Tks = [trans(X[i], X[i+1], n) for i in range(k)]

        if s.solve([I,And(Tks)]):
            for i in range(k):
                print("Estado:",i)
                for v in X[i]:
                    print("          ",v,'=',str(s.get_value(X[i][v]))[0:-2])
                print("----------------")
        else:
            print(check)

gera_traco(['pc', 'x', 'y', 'z'],init,trans, error, 20, 8, 150, 2)
```

```
Estado: 0
           pc = 0
           x = 150
           y = 2
           z = 0
----------------
Estado: 1
           pc = 1
           x = 150
           y = 2
           z = 0
----------------
Estado: 2
           pc = 3
           x = 150
           y = 2
           z = 0
----------------
Estado: 3
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 4
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 5
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 6
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 7
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 8
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 9
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 10
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 11
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 12
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 13
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 14
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 15
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 16
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 17
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 18
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
Estado: 19
           pc = 4
           x = 44
           y = 1
           z = 0
----------------
```

Função invert(trans, n_bits)

```
trans - Função que codifica as relações de transição entre estados
n_bits - Número de bits utilizados
```

Função invert que recebe a função python que codifica a relação de transição e devolve a relação de transição inversa.

In [6]:
```python
def invert(trans, n_bits):
    return (lambda c, p: trans(p,c, n_bits))
```

O algoritmo de "model-checking"

O algoritmo de “model-checking” manipula as fórmulas $R_n \equiv I \wedge T^n$ e $U_m \equiv E \wedge B^m$ fazendo crescer os índices n, m . Neste exemplo, os índices n, m crescem de acordo com o `input` do utilizador.

Para auxiliar na implementação deste algoritmo, começamos por definir duas funções. A função `rename` renomeia uma fórmula (sobre um estado) de acordo com um dado estado. A função `same` testa se dois estados são iguais.

In [7]:
```python
def baseName(s):
    return ''.join(list(itertools.takewhile(lambda x: x!='!', s)))

def rename(form,state):
    vs = get_free_variables(form)
    pairs = [ (x,state[baseName(x.symbol_name())]) for x in vs ]
    return form.substitute(dict(pairs))

def same(state1,state2):
    return And([Equals(state1[x], state2[x]) for x in state1])
```

Função model_checking(vars,init,trans,error,N, M, n_bits, a, b)

```
vars - Variáveis a declarar
init - Função que devolve um predicado que representa o estado inicial do programa
trans - Função transição
error - Função que devolve um predicado que representa o estado de erro do programa
N - Tamanho máximo do N
M - Tamanho máximo do M
n_bits - Número de bits a utilizar
a - Valor para a multiplicação
b - Valor para a multiplicação
```

Esta função implementa o algoritmo de `Model Checking orientado aos Interpolantes` onde o utilizador tem a livre vontade de aumentar `N` ou `M` caso não seja possível encontrar um majorante.

In [8]:
```python
def model_checking(vars,init,trans,error,N, M, n_bits, a, b):
    with Solver(name="z3") as s:

        # Criar todos os estados que poderão vir a ser necessários.
        X = [genState(vars, 'X', i, n_bits) for i in range(N+1)]
        Y = [genState(vars, 'Y', i, n_bits) for i in range(M+1)]


        (n,m) = (1,1)
        command = 0
        while command != 3 and n != N and m != M:
            Tn = And([trans(X[i], X[i+1], n_bits) for i in range(n)])
            I = init(X[0], a, b, n_bits)
            Rn = And(I, Tn)

            Bm = And([invert(trans, n_bits)(Y[i], Y[i+1]) for i in range(m)])
            E = error(Y[0], n_bits)
            Um = And(E, Bm)

            Vnm = And(Rn, same(X[n], Y[m]), Um)

            if s.solve([Vnm]):
                print("Unsafe")
                return

            else:                        # Vnm é instatisfazível
                C = binary_interpolant(And(Rn, same(X[n], Y[m])), Um)
                #C =1
                if C is None:
                    print("Interpolant None")
                    break

                C0 = rename(C, X[0])
                C1 = rename(C, X[1])
                T = trans(X[0], X[1], n_bits)
                if not s.solve([C0, T, Not(C1)]):   # C é invariante de T
                    print("Safe")
                    return
                else:
                    ### tenta gerar o majorante S
                    S = rename(C, X[n])
                    while True:
                        A = And(S, trans(X[n], Y[m], n_bits))
                        if s.solve([A,Um]):
                            print("Não é possível encontrar um majorante")
                            break
                        else:
                            Cnew = binary_interpolant(A, Um)
                            Cn = rename(Cnew, X[n])
                            if s.solve([Cn, Not(S)]):   # Se Cn -> S não é tautologia
                                S = Or(S, Cn)
                            else:             # S foi encontrado
                                print("Safe")
                                return

            command = int(input("1- Aumentar n\n2- Aumentar m\n3- Sair\nOpção: "))
            if command == 1:
                (n,m) = (n+1, m)
            elif command == 2:
                (n,m) = (n, m+1)

            print(f"N = {n}\nM = {m}\n")
        print("unknown")



model_checking(['pc', 'x', 'y', 'z'], init, trans, error, 20, 20, 8, 150, 2)
```

```
Não é possível encontrar um majorante
1- Aumentar n
2- Aumentar m
3- Sair
Opção: 2
N = 1
M = 2

Unsafe
```

In [ ]: