Trabalho Prático 3

Grupo 04 - Renato Garcia (A101987) & Bernardo Moniz (A102497)

Problema 1

Enunciado

O algoritmo estendido de Euclides (EXA) aceita dois inteiros constantes $a, b > 0$ e devolve inteiros r, s, t tais que $a * s + b * t = r$ e $r = \gcd(a, b)$.

Para além das variáveis r, s, t o código requer 3 variáveis adicionais r', s', t' que representam os valores de r, s, t no "próximo estado".

```
INPUT  a, b
assume  a > 0 and b > 0
r, r', s, s', t, t' = a, b, 1, 0, 0, 1
while r' != 0
  q = r div r'
  r, r', s, s', t, t' = r', r − q × r', s', s − q × s', t', t − q × t'
OUTPUT r, s, t
```

a. Construa um SFOTS usando BitVector's de tamanho n que descreva o comportamento deste programa. Considere estado de erro quando $r = 0$ ou alguma das variáveis atinge o "overflow".

b. Prove, usando a metodologia dos invariantes e interpolantes, que o modelo nunca atinge o estado de erro.

Resolução

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

- A **genState** cria a declaração das variáveis em BVType

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
```

- A função **init** é responsável por inicializar as variáveis do programa

In [3]:
```python
def init(s, n):
    return And(
        BVSGT(s['a'], BV(0, n)),
        BVSGT(s['b'], BV(0, n)),
        Equals(s['pc'], BV(0, n)),
        Equals(s['r'], s['a']),
        Equals(s['r_prox'], s['b']),
        Equals(s['s'], BV(1, n)),
        Equals(s['s_prox'], BV(0, n)),
        Equals(s['t'], BV(0, n)),
        Equals(s['t_prox'], BV(1, n)),
        Equals(s['q'], BV(0, n))
    )
```

- A função **trans**, recebe **curr** que é o estado atual, **prox** que é o próximo estado e **n** o número de bits, desenvolve as transições para os estados possíveis

In [4]:
```python
def trans(curr, prox, n):
    same_values = And(
        Equals(prox['a'], curr['a']),
        Equals(prox['b'], curr['b']),
        Equals(prox['r'], curr['r']),
        Equals(prox['r_prox'], curr['r_prox']),
        Equals(prox['s'], curr['s']),
        Equals(prox['s_prox'], curr['s_prox']),
        Equals(prox['t'], curr['t']),
        Equals(prox['t_prox'], curr['t_prox']),
        Equals(prox['q'], curr['q'])
    )

    Max_value = BV((1 << (n - 1)) - 1, n)
    Min_value = BV(0, n) - BV(1 << (n - 1), n)

    # (init) Q0 -> Q1 (skip)
    t01 = And(
        Equals(curr['pc'], BV(0, n)),
        Equals(prox['pc'], BV(1, n)),
        same_values
    )

    # (skip) Q1 -> Q4 (stop)
    t14 = And(
        Equals(curr['pc'], BV(1, n)),
        Equals(prox['pc'], BV(4, n)),
        Equals(curr['r_prox'], BV(0, n)),
        Equals(curr['r'], BVAdd(BVMul(curr['a'], curr['s']), BVMul(curr['b'], curr['t']))),
        same_values
    )

    # (skip) Q1 -> Q5 (error)
    t15 = And(
        Equals(curr['pc'], BV(1, n)),
        Equals(prox['pc'], BV(5, n)),
        Equals(curr['r'], BV(0, n)),
        same_values
    )

    # (skip) Q1 -> Q2 (loop)
    t12 = And(
        Equals(curr['pc'], BV(1, n)),
        Equals(prox['pc'], BV(2, n)),
        Not(Equals(curr['r_prox'], BV(0, n))),
        same_values
    )

    overflow_div = Or(
        BVSGT(BVSDiv(curr['r'], curr['r_prox']), Max_value),
        BVSLT(BVSDiv(curr['r'], curr['r_prox']), Min_value)
    )

    # Q2 -> Q3  ou Q2 -> Q5 (verificar overflow)
    t23_25 = And(
        Equals(curr['pc'], BV(2, n)),
        Equals(prox['a'], curr['a']),
        Equals(prox['b'], curr['b']),
        Equals(prox['r'], curr['r']),
        Equals(prox['r_prox'], curr['r_prox']),
        Equals(prox['s'], curr['s']),
        Equals(prox['s_prox'], curr['s_prox']),
        Equals(prox['t'], curr['t']),
        Equals(prox['t_prox'], curr['t_prox']),
        Equals(prox['q'], BVSDiv(curr['r'], curr['r_prox'])),

        Or(
            # Q2 -> Q3
            And(
                Equals(prox['pc'], BV(3, n)),
                Not(overflow_div) # se não houver overflow
            ),

            # Q2 -> Q5 com overflow
            And(
                Equals(prox['pc'], BV(5, n)),
                overflow_div # se houver overflow
            )
        )
    )

    overflow_calc = Or(
        Or(
            BVSGT(BVMul(curr['q'], curr['r_prox']), Max_value),
            BVSLT(BVMul(curr['q'], curr['r_prox']), Min_value)
        ),

        Or(
            BVSGT(BVSub(curr['r'], BVMul(curr['q'], curr['r_prox'])), Max_value),
            BVSLT(BVSub(curr['r'], BVMul(curr['q'], curr['r_prox'])), Min_value)
        ),

        Or(
            BVSGT(BVSub(curr['s'], BVMul(curr['q'], curr['s_prox'])), Max_value),
            BVSLT(BVSub(curr['s'], BVMul(curr['q'], curr['s_prox'])), Min_value)
        ),

        Or(
            BVSGT(BVMul(curr['q'], curr['s_prox']), Max_value),
            BVSLT(BVMul(curr['q'], curr['s_prox']), Min_value)
        ),


        Or(
            BVSGT(BVSub(curr['t'], BVMul(curr['q'], curr['t_prox'])), Max_value),
            BVSLT(BVSub(curr['t'], BVMul(curr['q'], curr['t_prox'])), Min_value)
        ),

        Or(
            BVSGT(BVMul(curr['q'], curr['t_prox']), Max_value),
            BVSLT(BVMul(curr['q'], curr['t_prox']), Min_value)
        ),
    )

    # Q3 -> Q1 ou Q3 -> Q5 (verificar overflow)
    t31_35 = And(
        Equals(curr['pc'], BV(3, n)),
        Equals(prox['a'], curr['a']),
        Equals(prox['b'], curr['b']),
        Equals(prox['r'], curr['r_prox']),
        Equals(prox['r_prox'], BVSub(curr['r'], BVMul(curr['q'], curr['r_prox']))),
        Equals(prox['s'], curr['s_prox']),
        Equals(prox['s_prox'], BVSub(curr['s'], BVMul(curr['q'], curr['s_prox']))),
        Equals(prox['t'], curr['t_prox']),
        Equals(prox['t_prox'], BVSub(curr['t'], BVMul(curr['q'], curr['t_prox']))),
        Equals(prox['q'], curr['q']),

        Or(
            # Q3 -> Q1
            And(
                Not(overflow_calc), # se não houver overflow
                Equals(prox['pc'], BV(1, n)),
                Not(Equals(curr['r'], BV(0, n)))
            ),

            # Q3 -> Q5 com overflow
            And(
                Equals(prox['pc'], BV(5, n)),
                Or(
                    Equals(curr['r'], BV(0, n)), # se r = 0
                    overflow_calc # se houver overflow
                )
            )
        )
    )

    # (stop) Q4 -> Q4 (stop)
    t_stop = And(
        Equals(curr['pc'], BV(4, n)),
        Equals(prox['pc'], BV(4, n)),
        same_values
    )

    # (error) Q5 -> Q5 (error)
    t_error = And(
        Equals(curr['pc'], BV(5, n)),
        Equals(prox['pc'], BV(5, n)),
        same_values
    )

    return Or(t01, t14, t15, t12, t23_25, t31_35, t_stop, t_error)
```

- A função **error** desenvolve o estado de erro

In [5]:
```python
def error(s, n):
    return Or(
        Equals(s['pc'], BV(5, n)),
        Equals(s['r'], BV(0, n))
    )
```

- A função **genTrace** gera um possível traço de execução com N transições

In [6]:
```python
def genTrace(vars, init, trans, N, n):
    with Solver(name='z3') as s:

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

        if s.solve([I,And(Tks)]):
            for i in range(N):
                print("Estado:",i)
                for v in X[i]:
                    value = s.get_value(X[i][v])
                    raw_value = value.constant_value()
                    signed_value = raw_value if raw_value < 2**(n-1) else raw_value - 2**n
                    print(f"          {v} = {signed_value}")
                print("----------------")
```

In [7]:
```python
# N estados -> 10 | n bits bitvector -> 10
genTrace(["pc", "a", "b", "r", "r_prox", "s", "s_prox", "t", "t_prox", "q"], init, trans, 15, 10)
```

```
Estado: 0
          pc = 0
          a = 140
          b = 70
          r = 140
          r_prox = 70
          s = 1
          s_prox = 0
          t = 0
          t_prox = 1
          q = 0
----------------
Estado: 1
          pc = 1
          a = 140
          b = 70
          r = 140
          r_prox = 70
          s = 1
          s_prox = 0
          t = 0
          t_prox = 1
          q = 0
----------------
Estado: 2
          pc = 2
          a = 140
          b = 70
          r = 140
          r_prox = 70
          s = 1
          s_prox = 0
          t = 0
          t_prox = 1
          q = 0
----------------
Estado: 3
          pc = 3
          a = 140
          b = 70
          r = 140
          r_prox = 70
          s = 1
          s_prox = 0
          t = 0
          t_prox = 1
          q = 2
----------------
Estado: 4
          pc = 1
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 5
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 6
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 7
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 8
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 9
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 10
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 11
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 12
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 13
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
Estado: 14
          pc = 4
          a = 140
          b = 70
          r = 70
          r_prox = 0
          s = 0
          s_prox = 1
          t = 1
          t_prox = -2
          q = 2
----------------
```

- A função **invert** recebe a função Python que codifica a transição e devolve a relação de transição inversa
- 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 [8]:
```python
def invert(trans, n_bits):
    return (lambda curr, prox: trans(prox, curr, n_bits))

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])
```

- A função **model_checking** implementa o algoritmo Model Checking orientado aos invariantes e interpolantes

In [9]:
```python
def model_checking(vars, init, trans, error, N, M, n_bits):
    with Solver(name="z3") as solver:

        # 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)]

        # Estabelecer a ordem pela qual os pares (n,m) vão surgir. Por exemplo:
        order = sorted([(a,b) for a in range(1,N+1) for b in range(1,M+1)],key=lambda tup:tup[0]+tup[1])

        # Step 1 implícito na ordem de 'order' e nas definições de Rn, Um.
        for (n,m) in order:
            # Step 2.
            I = init(X[0], n_bits)
            Tn = And([trans(X[i], X[i+1], n_bits) for i in range(n)])
            Rn = And(I, Tn)

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

            Vnm = And(Rn, same(X[n], Y[m]), Um)
            if solver.solve([Vnm]):
                print("> O sistema é inseguro.")
                return
            else:
                # Step 3.
                A = And(Rn, same(X[n], Y[m]))
                B = Um
                C = binary_interpolant(A, B)

                # Salvaguardar cálculo bem-sucedido do interpolante.
                if C is None:
                    print("> O interpolante é None.")
                    break

                # Step 4.
                C0 = rename(C, X[0])
                T = trans(X[0], X[1], n_bits)
                C1 = rename(C, X[1])

                if not solver.solve([C0, T, Not(C1)]):
                    # C é invariante de T.
                    print("> O sistema é seguro.")
                    return
                else:
                    # Step 5.1.
                    S = rename(C, X[n])
                    while True:
                        # Step 5.2.
                        T = trans(X[n], Y[m], n_bits)
                        A = And(S, T)
                        if solver.solve([A, Um]):
                            print("> Não foi encontrado majorante.")
                            break
                        else:
                            # Step 5.3.
                            C = binary_interpolant(A, Um)
                            Cn = rename(C, X[n])
                            if not solver.solve([Cn, Not(S)]):
                                # Step 5.4.
                                # C(Xn) -> S é tautologia.
                                print("> O sistema é seguro.")
                                return
                            else:
                                # Step 5.5.
                                # C(Xn) -> S não é tautologia.
                                S = Or(S, Cn)

    print("> Não foi provada a segurança ou insegurança do sistema.")
```

In [10]:
```python
# N -> 50 | M -> 50 | n_bits bitvector -> 10
model_checking(["pc", "a", "b", "r", "r_prox", "s", "s_prox", "t", "t_prox", "q"], init, trans, error, 50, 50, 10)
```

```
> Não foi encontrado majorante.
> O sistema é seguro.
```