TP3

Daniel Francisco Texeira Andrade - A100057

Pedro André Ferreira Malainho - A100050

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 "proximo 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.

Implementação

Imports

In [4]:
```
import itertools
from pysmt.shortcuts import *
```

Alinea a)

GenState Function

Generates a state dictionary for the given variables and state label.

`Parameters:`

- `vars:` List of variables names to include in the state.
- `s:` The label for the state.
- `i:` The index of the state.

`Returns:` A dictionary mapping variable names to their corresponding BitVec symbols.

Init1 function

Defines the initial conditions for the state.

`Parameters:`

- `state:` The current state containing variables.

`Returns:` A boolean expression representing the initial conditions.

Error1 function

Defines the error conditions based on the state.

`Parameters:`

- `state:` The current state containing variables.

`Returns:` A boolean expression representing the error conditions.

Trans1 function

Defines the transition relations between the current and next state.

`Parameters:`

- `curr:` The current state.
- `prox:` The next state.

`Returns:` A boolean expression representing the valid transitions.

GenTrace function

Generates a trace of the system based on initial conditions and transitions.

`Parameters:`

- `vars:` List of variable names.
- `init:` The initial condition function.
- `trans:` The transition function.
- `n:` The number of transitions to simulate.

`Returns:` None: Prints the state of the system at each step if satisfiable.

In [8]:
```python
global a, b

a = 60
b = 27

n = 8


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


def init1(s):
    assert not (a == 0 and b == 0)

    return (
        And(
            Equals(s['pc'], BVZero(n)),  # pc Inicial = 0
            Equals(s['r'], BV(a, n)),  # r Inicial = a
            Equals(s['r_linha'], BV(b, n)),  # r_linha Inicial = b
            Equals(s['s'], BV(1, n)),  # s Inicial = 1
            Equals(s['s_linha'], BVZero(n)),  # s_linha Inicial = 0
            Equals(s['t'], BVZero(n)),  # t Inicial = 0
            Equals(s['t_linha'], BV(1, n))  # t_linha Inicial = 1
        )
    )


def error1(s):
    # Considere o estado de erro quando
    # r = 0 ou alguma das variáveis atinge o "overflow"
    max_val = (1 << n) - 1
    return Or(
        Equals(s['r'], BVZero(n)),  # Se r for zero, erro
        Or(*[s[v] > BV(max_val, n) for v in vars]),  # Verifica overflow
    )



def trans1(curr, prox):

    # de fora para dentro do ciclo
    # pc == 0 fora do ciclo
    t0 = And(
        Equals(curr['pc'], BVZero(n)),
        Equals(prox['pc'], BV(1, n)),

        # r = r'
```

```
        Equals(prox['r'], curr['r_linha']),
        # r' = r - q * r'
        Equals(
            prox['r_linha'],
            BVSub(
                curr['r'],
                BVMul(
                    BVUDiv(
                        curr['r'],
                        curr['r_linha']
                    ),
                    curr['r_linha'],
                )
            )
        ),

        # s = s'
        Equals(prox['s'], curr['s_linha']),
        # s' = s - q * s'
        Equals(
            prox['s_linha'],
            BVSub(
                curr['s'],
                BVMul(
                    BVUDiv(
                        curr['r'],
                        curr['r_linha']
                    ),
                    curr['s_linha'],
                )
            )
        ),

        # t = t'
        Equals(prox['t'], curr['t_linha']),
        # t' = t - q * t'
        Equals(
            prox['t_linha'],
            BVSub(
                curr['t'],
                BVMul(
                    BVUDiv(
                            curr['r'],
                            curr['r_linha']
                        ),
                    curr['t_linha'],
                )
            ),
        )
    )

    # de dentro para dentro do ciclo
    t1 = And(
        Not(Equals(curr['r_linha'], BVZero(n))),

        Equals(curr['pc'], BV(1, n)),
        Equals(prox['pc'], BV(1, n)),

        Equals(prox['r'], curr['r_linha']),
        Equals(
```

```
            prox['r_linha'],
            BVSub(
                curr['r'],
                BVMul(
                    BVUDiv(
                            curr['r'],
                            curr['r_linha']
                        ),
                    curr['r_linha'],
                )
            )
        ),

        Equals(prox['s'], curr['s_linha']),
        Equals(
            prox['s_linha'],
            BVSub(
                curr['s'],
                BVMul(
                    BVUDiv(
                            curr['r'],
                            curr['r_linha']
                    ),
                    curr['s_linha'],
                )
            )
        ),

        Equals(prox['t'], curr['t_linha']),
        Equals(
            prox['t_linha'],
            BVSub(
                curr['t'],
                BVMul(
                    BVUDiv(
                            curr['r'],
                            curr['r_linha']
                    ),
                    curr['t_linha'],
                )
            )
        )
    )

    # de dentro para fora do ciclo
    t2 = And(
        Equals(curr['r_linha'], BVZero(n)),

        Equals(curr['pc'], BV(1, n)),
        Equals(prox['pc'], BV(2, n)),

        Equals(prox['r'], curr['r']),
        Equals(prox['r_linha'], curr['r_linha']),

        Equals(prox['s'], curr['s']),
        Equals(prox['s_linha'], curr['s_linha']),

        Equals(prox['t'], curr['t']),
        Equals(prox['t_linha'], curr['t_linha'])
    )
```

```
    # mantem tudo igual para ser possível gerar o traço em vários passos
    t3 = And(
        Equals(curr['r_linha'], BVZero(n)),

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

        Equals(prox['r'], curr['r']),
        Equals(prox['r_linha'], curr['r_linha']),

        Equals(prox['s'], curr['s']),
        Equals(prox['s_linha'], curr['s_linha']),

        Equals(prox['t'], curr['t']),
        Equals(prox['t_linha'], curr['t_linha'])
    )

    return Or(t0, t1, t2, t3)


def genTrace(vars, init, trans, error, n):
    with Solver(name="z3") as s:
        X = [genState(vars, 'X', i) for i in range(n + 1)]  # cria n+1 estados (
        I = init(X[0])
        Tks = [trans(X[i], X[i + 1]) for i in range(n)]

        if s.solve([I, And(Tks)]):  # testa se I /\ T^n  é satisfazível
            for i in range(n):
                print("Estado:", i)
                for v in X[i]:
                    print("          ", v, '=', s.get_value(X[i][v]))


vars = ['pc', 'r', 'r_linha', 's', 's_linha', 't', 't_linha']
genTrace(vars, init1, trans1, error1, 5)
```

```
Estado: 0
          pc = 0_8
          r = 60_8
          r_linha = 27_8
          s = 1_8
          s_linha = 0_8
          t = 0_8
          t_linha = 1_8
Estado: 1
          pc = 1_8
          r = 27_8
          r_linha = 6_8
          s = 0_8
          s_linha = 1_8
          t = 1_8
          t_linha = 254_8
Estado: 2
          pc = 1_8
          r = 6_8
          r_linha = 3_8
          s = 1_8
          s_linha = 252_8
          t = 254_8
          t_linha = 9_8
Estado: 3
          pc = 1_8
          r = 3_8
          r_linha = 0_8
          s = 252_8
          s_linha = 9_8
          t = 9_8
          t_linha = 236_8
Estado: 4
          pc = 2_8
          r = 3_8
          r_linha = 0_8
          s = 252_8
          s_linha = 9_8
          t = 9_8
          t_linha = 236_8
```

Alinea b)

BaseName function

Extracts the base name of a string by stopping at the first occurrence of the `!` character.

`Parameters:`

- `s:` A string containing the symbol name.

`Returns:` A string up to (but not including) the first occurrence of `!`.

Rename function

Renames free variables in a formula based on a given state mapping.

`Parameters:`

- `form:` The formula containing the free variables.
- `state:` A mapping of variable base names to their new values.

`Returns:` A new formula with the variables substituted according to the state.

Same function

Checks if two states are equivalent by comparing their values for all variables.

`Parameters:`

- `state1:` The first state as a dictionary of variable-value pairs.
- `state2:` The second state as a dictionary of variable-value pairs.

`Returns:` A boolean expression indicating equivalence of the two states.

Invert function

Inverts the direction of a transition relation function.

`Parameters:`

- `trans:` A transition relation function from one state to the next.

`Returns:` A new transition relation function with the direction reversed.

Model_Checking function

Implements a model-checking algorithm to verify system safety using interpolation.

`Parameters:`

- `vars:` The variables used in the states.
- `init:` A function defining the initial state condition.
- `trans:` A function defining the transition relation between states.
- `error:` A function defining the error condition.
- `N:` Maximum depth for forward reachability exploration.
- `M:` Maximum depth for backward reachability exploration.

`Returns:` Prints a message indicating the system's safety status: safe, unsafe, or inconclusive.

In [9]:
```
def baseName(s):
    return ''.join(list(itertools.takewhile(lambda x: x!='!', s)))
```

```python
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]
    )


def invert(trans):
    return lambda curr, prox: trans(prox, curr)


def model_checking(vars, init, trans, error, N, M):
    with Solver(name="z3") as solver:

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

        # 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)

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

            E = error(Y[0])
            Bm = And([transt(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])
                C1 = rename(C, X[1])
```

```python
                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])
                        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.")


model_checking(vars, init1, trans1, error1, 50, 50)
```

```
> O sistema é seguro.
```