TP3 - Exercício 1

Grupo 1

- Diogo Coelho da Silva A100092
- Pedro Miguel Ramôa Oliveira A97686

Problema proposto: Exercício 1 O algoritmo extendido 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 × Nr', s', s − q × s',
t', t − q × t'
OUTPUT r, s, t
```

1. 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".
2. Prove, usando a metodologia dos invariantes e interpolantes, que o modelo nunca atinge o estado de erro.

Proposta de resolução problema 1: O problema apresentado tem como objetivo a criação de SFOTS para descrever o comportamento de um programa, neste caso o algoritmo de Euclides. Na solução apresentada o sistema finito de transições foi definido utilizando BitVectors de tamanho n(32 no caso da nossa solução, para garantir a representação de números inteiros em 32 bits). Para verificar que o programa nunca atinge um estado de erro, utilizamos prova por interpolantes e invariantes,com a k-indução. Foram também considerados as restrições e variáveis dadas no enunciado do problema.

Resolução Exercício 1

1. Importar as bibliotecas importantes

In [95]:
```
from pysmt.shortcuts import *
import pysmt.typing as type
import random
from pysmt.typing import BOOL, REAL, INT, BVType, STRING
```

- `pysmt.shortcuts` : Importa as funções principais da biblioteca PySMT para trabalhar com SMT (Satisfiability Modulo Theories).
- `pysmt.shortcuts` : Permite definir os tipos de variáveis, como BOOL, INT, REAL, etc.
- `random` : Gera aleatoriedade para simular a disponibilidade de colaboradores e equipas dos projetos.
- `pysmt.typing` : Importa tipos específicos usados para criar variáveis simbólicas.

2. Construir um SFOTS usando BitVector's de tamanho n

In [96]:
```
n = 32     #número de bits das variáveis

def declare(i):
    state = {}
    state['pc'] = Symbol('pc'+str(i),BVType(n))
    state['r'] = Symbol('r'+str(i),BVType(n))
    state['s'] = Symbol('s'+str(i),BVType(n))
    state['t'] = Symbol('t'+str(i),BVType(n))
    state['q'] = Symbol('q'+str(i),BVType(n))
    state['r_'] = Symbol('r_'+str(i),BVType(n))
    state['s_'] = Symbol('s_'+str(i),BVType(n))
    state['t_'] = Symbol('t_'+str(i),BVType(n))
    return state
```

Neste pedaço de código estão definidos os estados iniciais do sistema.

In [97]:
```
def init(state, a, b):
    A = BVUGT(state['r'], SBV(0, n))
    B = BVUGT(state['r_'], SBV(0, n))
    C = Equals(state['pc'], SBV(0, n))
    D = Equals(state['r'], SBV(a, n))
    E = Equals(state['s'], SBV(1, n))
    F = Equals(state['t'], SBV(0, n))
    G = Equals(state['r_'], SBV(b, n))
    H = Equals(state['s_'], SBV(0, n))
    I = Equals(state['t_'], SBV(1, n))
    J = Equals(state['q'], SBV(0, n))

    r = And(A, B, C, D, E, F, G, H, I, J)

    return r
```

Tendo em conta a pré-condição do programa:

$$a > 0 \quad \wedge \quad b > 0 \quad \wedge \quad r = a \quad \wedge \quad r\_ = b \quad \wedge \quad s = 1 \quad \wedge \quad s\_ = 0 \quad \wedge \quad t = 0 \quad \wedge$$

Definimos o predicado $init$, que dado um estado e dois inteiros a e b, verifica se o mesmo é um estado inicial.

In [98]:
```
def trans(curr, prox):
    # transição estado 0 -> 1
    A = Equals(curr['pc'], SBV(0, n))
    B = Equals(prox['pc'], SBV(1, n))
    C = Equals(prox['r'], curr['r'])
    D = Equals(prox['s'], curr['s'])
    E = Equals(prox['t'], curr['t'])
    F = Equals(prox['r_'], curr['r_'])
    G = Equals(prox['s_'], curr['s_'])
    H = Equals(prox['t_'], curr['t_'])
    I = Equals(prox['q'], curr['q'])
    t01 = And(A, B, C, D, E, F, G, H, I)

    # transição estado 1 -> 2
    A = Equals(curr['pc'], SBV(1, n))
    B = Equals(prox['pc'], SBV(2, n))
    C = Equals(prox['r'], curr['r'])
    D = Equals(prox['s'], curr['s'])
    E = Equals(prox['t'], curr['t'])
    F = Equals(prox['r_'], curr['r_'])
    G = Equals(prox['s_'], curr['s_'])
    H = Equals(prox['t_'], curr['t_'])
    z = Not(Equals(curr['r_'], SBV(0, n)))
    t12 = And(A, B, C, D, E, F, G, H, z)

    #transição estado  2 -> 1
    A = Equals(curr['pc'], SBV(2, n))
    B = Equals(prox['pc'], SBV(1, n))
    D = Equals(prox['q'], BVSDiv(curr['r'], curr['r_']))
    E = Equals(prox['r_'], BVSub(curr['r'], BVMul(prox['q'], curr['r_']))
    F = Equals(prox['s_'], BVSub(curr['s'], BVMul(prox['q'], curr['s_']))
    G = Equals(prox['t_'], BVSub(curr['t'], BVMul(prox['q'], curr['t_']))
    H = Equals(prox['r'], curr['r_'])
    I = Equals(prox['s'], curr['s_'])
    J = Equals(prox['t'], curr['t_'])
    t21 = And(A, B, D, E, F, G, H, I, J)

    #transição estado 1 -> 3
    A = Equals(curr['pc'], SBV(1, n))
    B = Equals(prox['pc'], SBV(3, n))
    C = Equals(curr['r_'], SBV(0, n))
    D = Equals(prox['r'], curr['r'])
    E = Equals(prox['s'], curr['s'])
    F = Equals(prox['t'], curr['t'])
    G = Equals(prox['r_'], curr['r_'])
    H = Equals(prox['s_'], curr['s_'])
```

```
    I = Equals(prox['t_'], curr['t_'])
    J = Equals(prox['q'], curr['q'])
    t13 = And(A, B, C, D, E, F, G, H, I, J)

    #transição estado 3 -> 3
    A = Equals(curr['pc'], SBV(3, n))
    B = Equals(prox['pc'], SBV(3, n))
    C = Equals(prox['r'], curr['r'])
    D = Equals(prox['s'], curr['s'])
    E = Equals(prox['t'], curr['t'])
    F = Equals(prox['r_'], curr['r_'])
    G = Equals(prox['s_'], curr['s_'])
    H = Equals(prox['t_'], curr['t_'])
    I = Equals(prox['q'], curr['q'])
    t33 = And(A, B, C, D, E, F, G, H, I)

    return Or(t01, t12, t21, t13, t33)
```

Nesta parte do código são definidos os estados de transição e a função que trata da transição dos mesmos. Como argumento da função, a mesma recebe um estado atual e o estado para o qual pretendemos transitar. Após a verificação se a transição é válida, prossegue-se com a mesma.

In [99]:
```
def gera_traco(declare,init,trans,k,a,b):
    if(a>0 and b>0):
        with Solver() as solver:

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

            solver.add_assertion(init(traco[0],a,b))

            for i in range(k-1):
                solver.add_assertion(trans(traco[i], traco[i+1]))

            if solver.solve():
                print("> is sat")
                for i, s in enumerate(traco):
                    print("Estado pc: %s"%(solver.get_value(s['pc']).bv_s
                    print("Valor R: %s"%(solver.get_value(s['r']).bv_sign
                    print("Valor R_: %s"%(solver.get_value(s['r_']).bv_si
                    print("Valor S: %s"%(solver.get_value(s['s']).bv_sign
                    print("Valor S_: %s"%(solver.get_value(s['s_']).bv_si
                    print("Valor T: %s"%(solver.get_value(s['t']).bv_sign
                    print("Valor T_: %s"%(solver.get_value(s['t_']).bv_si
                    print("Valor Q: %s"%(solver.get_value(s['q']).bv_sign
                    print("")
            else:
                print("> Not feasible.")
```

A função $gera\_traco$, cria uma lista "traço" com os estados que foram aplicados à função init: $init(traco[0])$ no primeiro estado e para todos os estados consecutivos

S e $S´$ que foram aplicados à função transição: $\forall_{i=0}^{k-2}\ trans(traco[i], traco[i+1])$

3. Prova que o programa nunca atinge um estado de erro

In [100…

```
def r_dif_zero(state):
    return Not(Equals(state['r'],SBV(0,n)))
```

O código define uma função r_dif_zero que verifica se a variável r no estado atual não é igual a zero.

In [101…

```
def no_overflow(state):
    N= BV(2**n-1,width=n)
    A=Or(
        BVUGT(state['r'], N),
        BVUGT(state['s'], N),
        BVUGT(state['t'], N),
        BVUGT(state['r_'], N),
        BVUGT(state['s_'], N),
        BVUGT(state['t_'], N),
        BVUGT(state['q'], N)
    )
    return Not(A)
```

A função $no\_overflow$ verifica se não ocorre overflow nos valores de várias variáveis no estado atual. A função verifica se todas as variáveis indicadas (r, s, t, r_, s_, t_, q) estão dentro do limite representável por n bits. Se nenhuma delas está em overflow, a função retorna $True$,caso contrário, retorna $False$. Também devemos considerar como erro um estado em que alguma variável atinge o overflow, ou seja, em que alguma variável seja maior do que o maior número representável com 32 bits. Foi introduzido um predicado que dado um estado S, diz se alguma variável atingiu overflow.

In [102…

```
def nao_erro(state):
    return And(r_dif_zero(state),no_overflow(state))
```

Combinando r_dif_zero e $no\_overflow$, temos que um estado S, nao é estado de erro, quando ambos os predicados validam S.

Daí, o predicado nao_erro.

4. Definição do interpolante

In [103…

```
def interpolante(Rn, Um):
    return And(Rn, Um)

def rename(C, state):
    return substitute(C, {Symbol("r" + str(i), BVType(n)): state['r'] for
```

```python
def provaInterpolante(solver, X, Y, a, b, order, error):
    for (n, m) in order:
        I = init(X[0], a, b)
        Tn = And([trans(X[i], X[i+1]) for i in range(n)])
        Rn = And(I, Tn)

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

        C = interpolante(Rn, Um)

        print("> Interpolante C:", C)

        if C is None:
            print("> O interpolante é não existe.")
            break

        # Verificar se o interpolante é invariante
        C0 = rename(C, X[0])
        T = trans(X[0], X[1])
        C1 = rename(C, X[1])

        if not solver.solve([C0, T, Not(C1)]):
            # Se não encontrar contraexemplo, o sistema é seguro
            print("> O sistema é seguro.")
            return
        else:
            # Caso contrário, tentamos encontrar um maiorante
            S = rename(C, X[n])
            while True:
                T = trans(X[n], Y[m])
                A = And(S, T)
                if solver.solve([A, Um]):
                    print("> Não foi encontrado majorante.")
                    break
                else:
                    # Calculamos novamente o interpolante
                    C = interpolante(A, Um)
                    print("> Novo interpolante C:", C)
                    Cn = rename(C, X[n])
                    if not solver.solve([Cn, Not(S)]):
                        print("> O sistema é seguro.")
                        return
                    else:
                        S = Or(S, Cn)

def runInterpolante(declare, init, trans, error, a, b, order, k, solver):
    X = [declare(i) for i in range(k)]
    Y = [declare(i + k) for i in range(k)]
    provaInterpolante(solver, X, Y, a, b, order, error)
```

Este pedaço de código implementa um método de verificação baseado em interpolantes para provar a segurança de um sistema descrito por transições de estados. A função interpolante_prova constroi um interpolante entre duas partes do sistema. A primeira descreve as transições válidas até um estado n (representado por R_n) e a segunda descreve os estados que levam ao erro a partir de m (representado por U_m). O interpolante C funciona como um separador lógico entre os estados seguros e inseguros. Se C for um invariante, prova a segurança do sistema. Caso contrário, o algoritmo tenta melhorar o interpolante iterativamente. Se falhar, conclui-se que a prova de segurança não é possível.

A função $rename$ realiza a substituição simbólica de variáveis numa fórmula C. Isto é útil para adaptar o interpolante C gerado num contexto geral para o estado específico do sistema que está a ser analisado.

A função $verifica\_sistema\_com\_interpolacao$ organiza os elementos necessários, como os estados, transições e condições de erro, para chamar a função de interpolação e realizar a verificação formal.

5. Prova por K-Indução

In [104…

```
def k_induction_always(declare,init,trans,inv,k,a,b):
    if(a>0 and b>0):
        with Solver() as solver:
            s = [declare(i) for i in range(k)]
            solver.add_assertion(init(s[0],a,b))
            for i in range(k-1):
                solver.add_assertion(trans(s[i],s[i+1]))

            for i in range(k):
                solver.push()
                solver.add_assertion(Not(inv(s[i])))
                if solver.solve():
                    print(f"> Contradição! O invariante não se verifica n
                    for st in s:
                        print(f" pc = {solver.get_value(st['pc']).bv_sign
                        print(f" r = {solver.get_value(st['r']).bv_signed
                        print(f" s = {solver.get_value(st['s']).bv_signed
                        print(f" t = {solver.get_value(st['t']).bv_signed
                        print(f" r_ = {solver.get_value(st['r_']).bv_sign
                        print(f" s_ = {solver.get_value(st['s_']).bv_sign
                        print(f" t_ = {solver.get_value(st['t_']).bv_sign
                        print(f" q = {solver.get_value(st['q']).bv_signed
                        print("-----------------------------------------
                        print()

                    return
```

```
            solver.pop()

        s2 = [declare(i+k) for i in range(k+1)]

        for i in range(k):
            solver.add_assertion(inv(s2[i]))
            solver.add_assertion(trans(s2[i],s2[i+1]))

        solver.add_assertion(Not(inv(s2[-1])))

        if solver.solve():
            print(f"> Contradição! O passo indutivo não se verifica."
            return
        print(f"> A propriedade verifica-se por k-indução (k={k}).")
```

Pretendemos verificar que o programa nunca atinge um estado de erro, para verificar esta propriedade, utilizamos o método da k_indução que consiste em:

- ϕ é válido nos estados iniciais, ou seja, $\mathit{init}(s) \to \phi(s)$
- Para qualquer estado, assumindo que ϕ é verdade, se executarmos uma transição, ϕ continua a ser verdade no próximo estado, ou seja, $\phi(s) \wedge \mathit{trans}(s, s') \to \phi(s')$.

Um processo parecido com este seria generalizar a indução assumindo no passo indutivo que o invariante é válido nos k estados anteriores.

6. Execução do código e apresentação dos resultados

In [105…

```
gera_traco(declare,init,trans,7,3,6)
```

```
> is sat
Estado pc: 0
Valor R: 3
Valor R_: 6
Valor S: 1
Valor S_: 0
Valor T: 0
Valor T_: 1
Valor Q: 0

Estado pc: 1
Valor R: 3
Valor R_: 6
Valor S: 1
Valor S_: 0
Valor T: 0
Valor T_: 1
Valor Q: 0
```

```
Estado pc: 2
Valor R: 3
Valor R_: 6
Valor S: 1
Valor S_: 0
Valor T: 0
Valor T_: 1
Valor Q: 0

Estado pc: 1
Valor R: 6
Valor R_: 3
Valor S: 0
Valor S_: 1
Valor T: 1
Valor T_: 0
Valor Q: 0

Estado pc: 2
Valor R: 6
Valor R_: 3
Valor S: 0
Valor S_: 1
Valor T: 1
Valor T_: 0
Valor Q: 0

Estado pc: 1
Valor R: 3
Valor R_: 0
Valor S: 1
Valor S_: -2
Valor T: 0
Valor T_: 1
Valor Q: 2

Estado pc: 3
Valor R: 3
Valor R_: 0
Valor S: 1
Valor S_: -2
Valor T: 0
Valor T_: 1
Valor Q: 2
```

In [106…

```
with Solver() as solver:
    order = [(2, 2)]  # Adjust (n, m) pairs as needed
    runInterpolante(declare, init, trans, nao_erro, 3, 6, order, 7, solve
```

```
> Interpolante C: ((((0_32 u< r0) & (0_32 u< r_0) & (pc0 = 0_32) & (r0 = 3
_32) & (s0 = 1_32) & (t0 = 0_32) & (r_0 = 6_32) & (s_0 = 0_32) & (t_0 = 1_
32) & (q0 = 0_32)) & (((... & ... & ... & ... & ... & ... & ... & ... &
...) | (... & ... & ... & ... & ... & ... & ... & ... & ...) | (... & ...
& ... & ... & ... & ... & ... & ... & ...) | (... & ... & ... & ... & ...
& ... & ... & ... & ... & ...) | (... & ... & ... & ... & ... & ... & ...
& ... & ...)) & ((... & ... & ... & ... & ... & ... & ... & ... & ...) |
(... & ... & ... & ... & ... & ... & ... & ... & ...) | (... & ... & ... &
... & ... & ... & ... & ... & ...) | (... & ... & ... & ... & ... & ... &
... & ... & ... & ...) | (... & ... & ... & ... & ... & ... & ... & ... &
...)))) & (((! (... = ...)) & (! (... | ... | ... | ... | ... | ... |
...))) & (((... & ... & ... & ... & ... & ... & ... & ... & ...) | (... &
... & ... & ... & ... & ... & ... & ... & ...) | (... & ... & ... & ... &
... & ... & ... & ... & ...) | (... & ... & ... & ... & ... & ... & ... &
... & ... & ...) | (... & ... & ... & ... & ... & ... & ... & ... & ...))
& ((... & ... & ... & ... & ... & ... & ... & ... & ...) | (... & ... &
... & ... & ... & ... & ... & ... & ...) | (... & ... & ... & ... & ... &
... & ... & ... & ...) | (... & ... & ... & ... & ... & ... & ... & ... &
... & ...) | (... & ... & ... & ... & ... & ... & ... & ... & ...)))))
> O sistema é seguro.
```

In [107…

```
k_induction_always(declare,init,trans,nao_erro,7,1,5)
```

```
> A propriedade verifica-se por k-indução (k=7).
```