#TP3

Paulo Freitas - A100053

Pedro Santos - A100110

Enunciado

Considere-se de novo o algoritmo estendido de Euclides apresentado no TP2 mas usando o tipo dos inteiros e um parâmetro

$$N > 0$$

```
INPUT  a, b : Int
assume  a > 0 and b > 0 and a < N and b < N
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
```

Exercício 1

Este exercício é dirigido às provas de segurança do algoritmo acima.

1. Construa um FOTS

   $$\Sigma \equiv \langle X, I, T \rangle$$

   usando este modelo nos inteiros.

2. Considere como propriedade de segurança: safety = (r > 0) and (r < N) and (r = a*`s + b*t`) Prove usando k-indução que esta propriedade se verifica em qualquer traço do FOTS

3. Prove usando "Model-Checking" com interpelantes e invariantes prove também que esta propriedade é um invariante em qualquer traço de Σ.

Nota: De momento o uso de interpolantes não é possível em `z3` e requer um dos solvers `msat` ou `yices`. A experiência mostra que o `pysmt` com Python 3.11 ou 3.12 não instala qualquer destes "solvers"; para isso exige-se uma instalação com o Python 3.10.

Import das Bibliotecas

Para este exercício serão usadas as bibliotecas do pysmt e os sover z3 e msat.

```
%%capture
!yes | pip install pysmt
!apt-get install libgmp3-dev
!yes | pysmt-install --z3 --msat
```

```
file =
'/usr/local/lib/python3.10/dist-packages/pysmt/smtlib/parser/__init__.
py'
with open(file, 'r') as f:
    code = f.read()
    new_code = code.replace('USE_CYTHON = True', 'USE_CYTHON = False')

with open(file, 'w') as f:
  f.write(new_code)

from pysmt.shortcuts import *
from pysmt.typing import INT,BVType
from pysmt.logics import QF_NIA, QF_BV
import itertools
```

EXERCÍCIO 1

Exercício 1.1

Construa um FOTS

$$\Sigma \equiv \langle X , I , T \rangle$$

usando este modelo nos inteiros.

Variáveis

Este FOTS é uma adaptação do FOTS realizado no trabalho passado, sendo que a maior alteração é a mudança de tipos onde BVtype passa a Int.

Tomemos de exemplo os inteiros de input n = 40, a = 20 e b = 5.

```
  n -> Input que indica o número máximo de iterações.
  a -> Input para a resolução
  b -> Input para a resolução

  r -> variável com o resultado do algoritmo extendido de euclides
  s -> variável para cálculo do invariante
  t -> variável para cálculo do invariante

  r_,s_,t_ -> variáveis que representam r,s,t no proximo passo.
  q -> variável auxiliar para a divisão

n = 40

a=20
b=15

#mdc(a,b)
```

Declaração

A função *declare* declara as variáveis existentes em cada estado.

```
def declare(i):
    state = {}
    state['pc'] = Symbol('pc'+str(i),INT)

    state['r'] = Symbol('r'+str(i),INT)
    state['r_'] = Symbol('r_'+str(i),INT)
    state['s'] = Symbol('s'+str(i),INT)
    state['s_'] = Symbol('s_'+str(i),INT)
    state['t'] = Symbol('t'+str(i),INT)
    state['t_'] = Symbol('t_'+str(i),INT)
    state['q'] = Symbol('q' +str(i),INT)

    return state
```

Inicialização

A função *init* inicializa as variáveis de um dado estado. Segue a inicialização por base das informações presentes no enunciado.

```
  r = a;
  r_= b;
  s = 1;
  s_= 0;
  t = 0;
  t_= 1;
  q = 0;
  pc = 0.

def init(state):
    E = Equals(state['pc'],Int(0))

    R = Equals(state['r'],Int(a))
    R_ = Equals(state['r_'],Int(b))
    S = Equals(state['s'], Int(1))
    S_ = Equals(state['s_'],Int(0))
    T = Equals(state['t'],Int(0))
    T_ = Equals(state['t_'],Int(1))
    Q = Equals(state['q'],Int(0))

    return And(E,R,R_,S,S_,T,T_)
```

Transição

Seja o ciclo presente no enuniado, com a alteração em que cada estado do ciclo tem um número associado, ou seja:

```
0:while r' != 0
1:  q = r div r'
    r, r', s, s', t, t' = r', r − q × r', s', s − q × s', t', t − q ×
t'
2: Outup
```

A função trans recebe dois estados e verifica a possibilidade de um estado pode transitar para outro respeitando o ciclo do enunciado.

Seja a fase de verificação da condição o equivalente ao "program counter" = 0, o que está dentro do ciclo while o equivalente ao "program counter" = 1 e o fim do ciclo (neste caso o output) o "program counter" = 2 temos que:

$$\left(pc=0 \wedge r_{!}=0 \wedge pc'=1 \wedge vars=vars\right)$$
$$\vee$$
$$\left(pc=0 \wedge r_{¿}=0 \wedge pc'=2 \wedge vars=vars\right)$$
$$\vee$$
$$¿$$

(onde vars representa todas as variaveis)

```python
def trans(curr,prox):

    t01 = And(Equals(curr['pc'],Int(0)),
Not(Equals(curr['r_'],Int(0))), Equals(prox['pc'],Int(1)),
              Equals(prox['r'],curr['r']),
Equals(prox['r_'],curr['r_']),
              Equals(prox['s'],curr['s']),
Equals(prox['s_'],curr['s_']),
              Equals(prox['t'],curr['t']),
Equals(prox['t_'],curr['t_']),
              Equals(prox['q'],curr['q']))

    t02 = And(Equals(curr['pc'],Int(0)), Equals(curr['r_'],Int(0)),
Equals(prox['pc'],Int(2)),
              Equals(prox['r'],curr['r']),
Equals(prox['r_'],curr['r_']),
              Equals(prox['s'],curr['s']),
Equals(prox['s_'],curr['s_']),
              Equals(prox['t'],curr['t']),
Equals(prox['t_'],curr['t_']),
              Equals(prox['q'],curr['q']))

    t10 = And(Equals(curr['pc'],Int(1)), Equals(prox['pc'],Int(0)),
              Equals(curr['q'], Div(curr['r'],curr['r_'])),
              Equals(prox['r'],curr['r_']), Equals(prox['r_'],
Minus(curr['r'],Times(curr['q'],curr['r_']))),
              Equals(prox['s'],curr['s_']), Equals(prox['s_'],
Minus(curr['s'],Times(curr['q'],curr['s_']))),
```

```
                Equals(prox['t'],curr['t_']), Equals(prox['t_'],
Minus(curr['t'],Times(curr['q'],curr['t_']))))

    t22 = And(Equals(curr['pc'],Int(2)), Equals(prox['pc'],Int(2)),
                Equals(prox['r'],curr['r']),
Equals(prox['r_'],curr['r_']),
                Equals(prox['s'],curr['s']),
Equals(prox['s_'],curr['s_']),
                Equals(prox['t'],curr['t']),
Equals(prox['t_'],curr['t_']),
                Equals(prox['q'],curr['q']))

    return Or(t01,t02,t10,t22)
```

Euclides

Nesta porção de código, o programa, com o solver, invoca a função *declare* para declarar as variáveis para cada passo do algoritmo. No solver são adicionadas as condições de inicialização do primeiro estado e a função *trans* para todos os estados consecutivos. Por fim, o programa termina imprimindo as variáveis r,s e t do ultimo estado.

```
def euclides(declare,init,trans,k):
    with Solver(name="z3") as solver:
      states = [declare(i) for i in range(k)]

      solver.add_assertion(init(states[0]))
      solver.add_assertion(And(trans(states[i],states[i+1]) for i in
range(k-1)))

      if solver.solve():
        print("r =",solver.get_value(states[-1]["r"])," s
=",solver.get_value( states[-1]["s"])," t =",solver.get_value(states[-
1]["t"]))
        print(f'mdc({a},{b}) = {solver.get_value(states[-1]["r"])}')

euclides(declare,init,trans,n)

r = 5  s = 1  t = -1
mdc(20,15) = 5
```

Exercício 1.2

1. Considere como propriedade de segurança: safety = (r > 0) and (r < N) and (r = a*s* + *b*t)
   Prove usando k-indução que esta propriedade se verifica em qualquer traço do FOTS.

Invariante

Começemos por transformar a propriedade safety em um conjunto de invariantes. Esta condição deverá ser verdadeira por k interações.

```
def inv(state):

  p1 = GT(state['r'],Int(0))
  p2 = LT(state['r'],Int(n))
  p3 = Equals(state['r'], Plus(Times(Int(a),state['s']), Times(Int(b),state['t'])))

  return And(p1,p2,p3)
```

K-Indução

A k-indução procura provar que se para um dado k interações, o problema for satisfazivel a uma dado invariante, então ele permanecerá assim para todo o problema.

```
def kinduction_always(declare,init,trans,inv,k):
    with Solver(name="z3") as solver:
        states = [declare(i) for i in range(k)]

        solver.add_assertion(init(states[0]))

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

        for i in range(k):
            solver.push()
            solver.add_assertion(Not(inv(states[i])))
            if solver.solve():
                print(f"> Contradição! O invariante não se verifica nos k estados iniciais.")
                for st in states:
                    print("x, pc, inv: ", solver.get_value(st['x']), solver.get_value(st['pc']))
                return
            solver.pop()

        state2 = [declare(i+k) for i in range(k+1)]

        for i in range(k):
            solver.add_assertion(inv(state2[i]))
            solver.add_assertion(trans(state2[i],state2[i+1]))

        solver.add_assertion(Not(inv(state2[-1])))

        if solver.solve():
            print(f"> Contradição! O passo indutivo não se verifica.")
            for i,state in enumerate(states):
                print(f"> State {i}: x = {solver.get_value(state['x'])}, pc= {solver.get_value(state['pc'])}.")
            return
```

```
        print(f"> A propriedade verifica-se por k-indução (k={k}).")

kinduction_always(declare,init,trans,inv,3)
```

```
> A propriedade verifica-se por k-indução (k=3).
```

Exercício 1.3

Prove usando “Model-Checking” com interpelantes e invariantes prove também que esta propriedade é um invariante em qualquer traço de Σ.

Para a realização deste exercício, será necessário transformar o FOTS em SFOTS. Isso implica o desenvolvimento de um codigo novo baseado no FOTS já desenvolvido. Como existe uma incompatibilidade com a função *Div* e os interpolantes, usaremos o *FOTS* desenvolvido no TP2 neste exercicio, com ligeiras alterações.

Variáveis

Consideraremos a = 30 , b = 10 como variáveis de input do problema a fim de testar-lo.

Seja max o número máximo que a,b e r pode tomar (100) e n o tamanho dos bitvetores

```
max = 100
n = 40
a=30
b=10
```

Auxiliares

Para o cálculo do Model-checking, serão necessárias funções auxiliares para renomear, comparar e inverter os estadoes e funções.

```
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])

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

Declaração

Seja a função declare do exercico 1 do TP2, agora adaptada para receber um simbolo e o tamanho dos bitvetores a declarar. A função declare1 declara para cada variável o seu tipo, para cada estado.

```
def declare1(i,s,tam):
    state = {}

    state['pc'] = Symbol('pc'+ '!' + s + str(i), INT)

    state['a']  = Symbol('a' + '!' + s + str(i), BVType(tam))
    state['b']  = Symbol('b' + '!' + s + str(i), BVType(tam))
    state['r']  = Symbol('r' + '!' + s + str(i), BVType(tam))
    state['r_'] = Symbol('r_'+ '!' + s + str(i), BVType(tam))
    state['s']  = Symbol('s' + '!' + s + str(i), BVType(tam))
    state['s_'] = Symbol('s_'+ '!' + s + str(i), BVType(tam))
    state['t']  = Symbol('t' + '!' + s + str(i), BVType(tam))
    state['t_'] = Symbol('t_'+ '!' + s + str(i), BVType(tam))
    state['q']  = Symbol('q' + '!' + s + str(i), BVType(tam))

    return state
```

Inicialização

Semelhante a init, esta função possui as diferenças que recebe o tamanho e o valor máximo para inicializar um estado. Esta função atribui valores inciais a um estado, onde que as variáveis a e b têm valor entre 0 e 100.

```
def init1(state,tam,max):
    E  = Equals(state['pc'], Int(0))

    A  = And(BVSGT(state['a'], BV(0,tam)),BVSLT(state['a'],
BV(max,tam)))
    B  = And(BVSGT(state['b'], BV(0,tam)),BVSLT(state['b'],
BV(max,tam)))

    R  = Equals(state['r'],  state['a'])
    R_ = Equals(state['r_'], state['b'])

    S  = Equals(state['s'],  BV(1,tam))
    S_ = Equals(state['s_'], BV(0,tam))
    T  = Equals(state['t'],  BV(0,tam))
    T_ = Equals(state['t_'], BV(1,tam))
    Q  = Equals(state['q'],  BV(0,tam))

    return And(E,A,B,R,R_,S,S_,T,T_)
```

Transição

A função trans1 indica as mudanças de estado que ocorrem durante o algoritmo de Euclides extendido. Recebe o estado de origem, o estado de destino e o tamanho dos bitvetores.

As transições são dadas por:

$$\left(pc=0 \land r_{!}=0 \land pc'=1 \land vars=vars\right)$$
$$\lor$$
$$\left(pc=0 \land r_{¿}=0 \land pc'=2 \land vars=vars\right)$$
$$\lor$$
$$¿$$

```
def trans1(curr,prox,tam):

    t01 = And(Equals(curr['pc'],Int(0)),
Not(Equals(curr['r_'],BV(0,tam))), Equals(prox['pc'],Int(1)),
              Equals(prox['r'],curr['r']),
Equals(prox['r_'],curr['r_']),
              Equals(prox['s'],curr['s']),
Equals(prox['s_'],curr['s_']),
              Equals(prox['t'],curr['t']),
Equals(prox['t_'],curr['t_']),
              Equals(prox['q'],curr['q']),
Equals(prox['a'],curr['a']), Equals(prox['b'],curr['b']))

    t02 = And(Equals(curr['pc'],Int(0)), Equals(curr['r_'],BV(0,tam)),
Equals(prox['pc'],Int(2)),
              Equals(prox['r'],curr['r']),
Equals(prox['r_'],curr['r_']),
              Equals(prox['s'],curr['s']),
Equals(prox['s_'],curr['s_']),
              Equals(prox['t'],curr['t']),
Equals(prox['t_'],curr['t_']),
              Equals(prox['q'],curr['q']),
Equals(prox['a'],curr['a']), Equals(prox['b'],curr['b']))

    t10 = And(Equals(curr['pc'],Int(1)), Equals(prox['pc'],Int(0)),
              Equals(curr['q'], BVSDiv(curr['r'],curr['r_'])),
              Equals(prox['r'],curr['r_']), Equals(prox['r_'],
BVSub(curr['r'],BVMul(curr['q'],curr['r_']))),
              Equals(prox['s'],curr['s_']), Equals(prox['s_'],
BVSub(curr['s'],BVMul(curr['q'],curr['s_']))),
              Equals(prox['t'],curr['t_']), Equals(prox['t_'],
BVSub(curr['t'],BVMul(curr['q'],curr['t_']))),
              Equals(prox['a'],curr['a']),
Equals(prox['b'],curr['b']))

    t22 = And(Equals(curr['pc'],Int(2)), Equals(prox['pc'],Int(2)),
              Equals(prox['r'],curr['r']),
```

```
Equals(prox['r_'],curr['r_']),
              Equals(prox['s'],curr['s']),
Equals(prox['s_'],curr['s_']),
              Equals(prox['t'],curr['t']),
Equals(prox['t_'],curr['t_']),
              Equals(prox['q'],curr['q']),
Equals(prox['a'],curr['a']), Equals(prox['b'],curr['b']))

    return Or(t01,t02,t10,t22)
```

Invariante

Esta função verifica o invariante para um dado estado da máquina. Seja o invariante dado por:

*(r > 0) and (r < N) and (r = a.*s + b.t)*

Temos que a função recebe um dado estado, o tamanho e o valor máximo.

```
def inv1(state,tam,max):

  p1 = BVSGT(state['r'], BV(0,tam))
  p2 = BVSLT(state['r'], BV(max,tam))

  p3 = Equals(state['r'], BVAdd(BVMul(state['a'],state['s']),
BVMul(state['b'],state['t'])))

  return And(p1,p2,p3)
```

Model-Checking

Função principal desta alínea.

Esta função não está totalmente certa. Pois ocorre um erro na atribuição do interpolador.

```
def model_checking(declare,init,trans,inv,N,M):
    tam = 32
    with Solver(logic= QF_BV,name="msat") as solver:

        # Criar todos os estados que poderão vir a ser necessários.
        X = [declare(i,'X',tam) for i in range(N+1)]
        Y = [declare(i,'Y',tam) 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)],key=lambda tup:tup[0]+tup[1])

        # Step 1 implícito na ordem de 'order' e nas definições de Rn,
Um.
```

```python
        for (n,m) in order:
            # Step 2.
            I = init(X[0],tam,max)
            Tn = And([trans(X[i], X[i+1],tam) for i in range(N)])
            Rn = And(I, Tn)

            E = inv(Y[0],tam,max)
            Bm = And([transt(Y[i], Y[i+1],tam) 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. (1)")
                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. (2)")
                    break

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

                if not solver.solve([C0, T, Not(C1)]):
                    # C é invariante de T.
                    print("> O sistema é seguro.(3)")
                    return
                else:
                    # Step 5.1.
                    S = rename(C, X[n])
                    while True:
                        # Step 5.2.
                        T = trans(X[n], Y[m],tam)
                        A = And(S, T)
                        if solver.solve([A, Um]):
                            print("> Não foi encontrado majorante.
(4)")

                            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. (5)")
                                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.
(6)")

model_checking(declare1, init1, trans1, inv1, 3, 3)
```

```
> O sistema é inseguro. (1)
```