Lógica Computacional 24/25

Grupo 09

- João Afonso Almeida Sousa (A102462)
- Rafael Cunha Costa (A102526)

Problema 1

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



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.
```

Inicialização

Para resolver os problemas de satisfação de restrições, usamos o Z3py, uma biblioteca Python que cria uma interface para o Z3, um solver SMT. Para instalar o Z3py, basta correr o seguinte comando:

```
pip install z3-solver
```

```
from z3 import *
```

Declaração dos BitVecs

```
n = 32  # Tamanho dos BitVectors
a = BitVec('a', n)
b = BitVec('b', n)
r = BitVec('r', n)
rline = BitVec('rline', n)
s = BitVec('s', n)
sline = BitVec('sline', n)
t = BitVec('t', n)
tline = BitVec('tline', n)

# Variáveis do proximo estado
r_next = BitVec('r_next', n)
rline_next = BitVec('rline_next', n)
s_next = BitVec('s_next', n)
sline_next = BitVec('sline_next', n)
t_next = BitVec('t_next', n)
tline_next = BitVec('tline_next', n)

q = BitVec('q', n)
```

Estado Inicial

```
init = [
    r == a,
    rline == b,
    s == 1,
    sline == 0,
    t == 0,
    tline == 1,
    a > 0,
    b > 0
]
```

```
# q = r div r'
q_def = q == UDiv(r, rline)
```

Calcular as variáveis do próximo estado

```
trans = [
    r_next == rline,
    rline_next == r - q * rline,
    s_next == sline,
    sline_next == s - q * sline,
    t_next == tline,
    tline_next == t - q * tline
]
```

Invariante

```
invariant = r == a * s + b * t
```

Condição do While

```
loop_condition = rline != 0
```

Atualizar as variáveis

```
state_update = Implies(
    loop_condition,
    And(
        r == r_next,
        rline == rline_next,
        s == s_next,
        sline == sline_next,
        t == t_next,
        tline == tline_next
    )
)
```

Definir estados de erro (Overflow ou r==0)

```
error_state = Or(
    r == 0,
    r > 2**(n-1) - 1,
```

```
    rline > 2**(n-1) - 1,
    s > 2**(n-1) - 1,
    sline > 2**(n-1) - 1,
    t > 2**(n-1) - 1,
    tline > 2**(n-1) - 1,
)
```

Inicializar solver

```
solver = Solver()
solver.add(init)
solver.add(loop_condition)
solver.add(trans)
solver.add(error_state)
result = solver.check()
if result == sat:
    print("O estado de erro é alcançável.")
    print(solver.model())
else:
    print("O estado de erro não é alcançável.")
```

```
O estado de erro não é alcançável.
```

Caso base

```
solver.reset()
solver.add(init)
solver.add(Not(invariant))
result = solver.check()
if result == sat:
    print("O invariante falha no caso base.")
    print(solver.model())
else:
    print("O invariante não falha no caso base.")
```

```
O invariante não falha no caso base.
```

Passo indutivo

```
solver.reset()
solver.add(invariant)
solver.add(loop_condition)
solver.add(trans)
solver.add(Not(invariant))
```

```
result = solver.check()
if result == sat:
    print("O invariante falha no passo indutivo.")
    print(solver.model())
else:
    print("O invariante não falha no passo indutivo.")
```

```
O invariante não falha no passo indutivo.
```