Lógica Computacional - TP2 Exercício 1 - G01

Bruno Dias da Gião A96544, João Luis da Cruz Pereira A95375, David Alberto Agra A95726

October 25, 2023

1 Exercício 1 - Enunciado

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', r', t − q × t' OUTPUT r, s, t ""
   1. Construa um FOTS usando BitVector de tamanho n que descreva o comportamento deste programa: identifique as variáveis do modelo, o estado inicial e a relação de transição.
   2. Considere estado de erro quando $r = 0$ ou alguma das variáveis atinge o "overflow". Prove que o programa nunca atinge o estado de erro
   3. Prove que a relação de Bézout $a * s + b * t = r$ é um invariante do algoritmo.

2 Exercício 1 - Solução

2.1 Exercício 1 - Restrições e Metodologia

Os inputs serão do seguinte formato, i, bv_width, K, a, b sendo inputs numéricos que remetem para o número do estado inicial, o número de bit no BitVector, o número de estados e os números para os quais vamos calcular o máximo divisor comum.

Como, por exemplo, o seguinte conjunto de *input*:

```
i = 0
bv_width = 32
K = 20
a = 5
b = 2
```

De forma a encontrarmos solução para este problema, utilizaremos o Solver pysmt. Utilizaremos também vários shortcuts do pysmt para resolver o problema.

```
from pysmt.shortcuts import Symbol, Equals, GT, And, Or, Solver, Int, Not, BVUDiv, BVType, BVSu
```

Começaremos por definir uma função que cria um dicionário onde iremos guardar as diferentes variáveis do problema, O solver preencherá cada variável com o nome da variável e o número do estado, o valor guardado é um `BVType` que irá criar e guardar o type sempre que necessário.

```
def declare(i, bv_width):
```

```
    bv_type = BVType(bv_width)
    state = {}
    state['s'] = Symbol('s' + str(i), bv_type)
    state['t'] = Symbol('t' + str(i), bv_type)
    state['r'] = Symbol('r' + str(i), bv_type)
    state['s_prox'] = Symbol('s_prox' + str(i), bv_type)
    state['t_prox'] = Symbol('t_prox' + str(i), bv_type)
    state['r_prox'] = Symbol('r_prox' + str(i), bv_type)
    state['q'] = Symbol('q' + str(i), bv_type)
    state['pc'] = Symbol('pc' + str(i), bv_type)

    return state
```

De seguida criamos a função `init`, onde iremos inicializar as variáveis. Inicialmente só inicializavamos `a` e `b` como maior que `0` mas devido ao invariante pedidos (`r > 0` e `a*s + b*t = r`) tivemos que adicionar os valores que essas variáveis iriam assumir no estado 1.

```
def init(state, a, b, bv_width):

    A = Equals(state['pc'], BV(0, bv_width))
    B = GT(Int(a), Int(0))
    C = GT(Int(b), Int(0))
    D = Equals(state['r'], BV(a, bv_width))
    E = Equals(state['s'], BV(1, bv_width))
    F = Equals(state['t'], BV(0, bv_width))

    return And(A, B, C, D, E, F)
```

Foi definida uma função `trans` para calcular as transições de cada estado.
Podemos ver o estado de cada linha do problema:

```
0 : r, r', s, s', t, t' = a, b, 1, 0, 0, 1
1 : while r' != 0
2 : q = r div r'
3 : r, r', s, s', t, t' = r', r − q × r', s', s − q × s', r', t − q × t'
4 : stop
```

As transições são as seguintes: * 0 -> 1 * 1 -> 2 * 2 -> 3 * 3 -> 1 * 4 -> 4

É necessário uma transição `4 -> 4` pois como não sabemos o número de estados à partida se o programa chegar ao fim antes do último estado precisamos de uma transição para manter os valores das variáveis igual até chegarmos ao último estado e o programa terminar.

Em cada uma das transições alteramos o valor da variável no próximo estado conforme o pedido em cada linha do código. Se não são efetuadas alterações no valor da variável mantemos o mesmo valor.

```
def trans(curr, prox, a, b, bv_width):

    t01 = And(
        Equals(curr['pc'], BV(0, bv_width)),
        Equals(prox['pc'], BV(1, bv_width)),
```

```
    Equals(prox['r'], BV(a, bv_width)),
    Equals(prox['r_prox'], BV(b, bv_width)),
    Equals(prox['s'], BV(1, bv_width)),
    Equals(prox['s_prox'], BV(0, bv_width)),
    Equals(prox['t'], BV(0, bv_width)),
    Equals(prox['t_prox'], BV(1, bv_width)),
    Equals(curr['q'], prox['q'])
)

t14 = And(
    Equals(curr['pc'], BV(1, bv_width)),
    Equals(prox['pc'], BV(4, bv_width)),
    Equals(curr['r_prox'], BV(0, bv_width)),
    Equals(curr['r'], prox['r']),
    Equals(curr['s'], prox['s']),
    Equals(curr['s_prox'], prox['s_prox']),
    Equals(curr['t'], prox['t']),
    Equals(curr['t_prox'], prox['t_prox']),
    Equals(curr['q'], prox['q'])
)

t12 = And(
    Equals(curr['pc'], BV(1, bv_width)),
    Equals(prox['pc'], BV(2, bv_width)),
    Not(Equals(curr['r_prox'], BV(0, bv_width))),
    Equals(curr['r_prox'], prox['r_prox']),
    Equals(curr['r'], prox['r']),
    Equals(curr['s'], prox['s']),
    Equals(curr['s_prox'], prox['s_prox']),
    Equals(curr['t'], prox['t']),
    Equals(curr['t_prox'], prox['t_prox']),
    Equals(curr['q'], prox['q'])
)

t23 = And(
    Equals(curr['pc'], BV(2, bv_width)),
    Equals(prox['pc'], BV(3, bv_width)),
    Equals(prox['q'], BVUDiv(curr['r'], curr['r_prox'])),
    Equals(curr['r'], prox['r']),
    Equals(curr['r_prox'], prox['r_prox']),
    Equals(curr['s'], prox['s']),
    Equals(curr['s_prox'], prox['s_prox']),
    Equals(curr['t'], prox['t']),
    Equals(curr['t_prox'], prox['t_prox'])
)

t31 = And(
    Equals(curr['pc'], BV(3, bv_width)),
```

```
        Equals(prox['pc'], BV(1, bv_width)),
        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'])
    )

    t44 = And(
        Equals(curr['pc'], BV(4, bv_width)),
        Equals(prox['pc'], BV(4, bv_width)),
        Equals(curr['r'], prox['r']),
        Equals(curr['r_prox'], prox['r_prox']),
        Equals(curr['s'], prox['s']),
        Equals(curr['s_prox'], prox['s_prox']),
        Equals(curr['t'], prox['t']),
        Equals(curr['t_prox'], prox['t_prox']),
        Equals(curr['q'], prox['q'])
    )

    return Or(t01, t14, t12, t23, t31, t44)
```

Para gerarmos o traço criamos a função `gera_traco`. Tal como fizemos na aula prática, começamos por gerar uma lista onde iremos usar a função `declare` para cada um dos estados pedidos pelo input, de seguida, inicializamos as variáveis para o estado com a função `init`, após isso calculamos as transições entre cada um dos estados. No final, verificamos que o `pc` (linha atual) no último estado é `4` (stop). No caso de haver solução, exibimos ao utilizador o valor de `r` (máximo divisor comum), `s` (valor a multiplicar por `a`), `t` (valor a multiplicar por `b`) em EXA. Caso contrário, informamos que não é possível chegar a uma solução.

```
def gera_traco(i, a, b, K, bv_width):

    with Solver() as solver:
        states = [declare(i, bv_width) for i in range(K)]
        solver.add_assertion(init(states[i], a, b, bv_width))
        solver.add_assertion(Equals(states[-1]['pc'], BV(4, bv_width)))

        for k in range(K):
            if k>0:
                solver.add_assertion(trans(states[k-1], states[k], a, b, bv_width))

        if solver.solve():
            print(f"r: {solver.get_value(states[-1]['r']).bv_signed_value()}, s: {solver.get_va
        else:
            print("> Not feasible.")
```

Para provarmos invariantes usamos a função `bmc_always` como foi feita na aula prática. O processo

é idêntico a `gera_traco` mas temos uma restrição extra onde verificamos se o invariante não é satisfeito em algum dos estados.
No caso de ser solúvel, o invariante não é provado sendo exibido o contra-exemplo ao utilizador. Caso contrário informamos que o invariante é satisfeito para todos os estados.

```
def bmc_always(declare,init,trans,inv, i, a, b, K, bv_width):

    with Solver() as solver:
        states = [declare(i, bv_width) for i in range(K+1)]
        solver.add_assertion(init(states[i], a, b, bv_width))
        solver.add_assertion(Equals(states[-1]['pc'], BV(4, bv_width)))

        for k in range(K):
            if k>0:
                solver.add_assertion(trans(states[k-1], states[k], a, b, bv_width))

            solver.push()
            solver.add_assertion(Not(inv(states[k], bv_width)))

            if solver.solve():
                print(f"> Invariant does not hold for {k+1} first states. Counter-example:")
                for i,state in enumerate(states[:k+1]):
                    print(f"> State {i}: pc = {solver.get_value(state['pc']).bv_signed_value()
                return
            else:
                if k==K-1:
                    print(f"> Invariant holds for the first {K} states.")
                else:
                    solver.pop()
```

Finalmente, definimos duas funções. Uma para cada invariante que queremos provar. Para provarmos que nunca atingimos estado de erro (`r = 0`, `s` atingir *overflow* ou `t` atingir *overflow* usamos a `nonzerooverflow`. Para provar que a relação de Bézout é um invariante do programa utilizamos a função `bezout`.

```
def nonzerooverflow(state, bv_width):
    zero = SBV(0, bv_width)
    max_value = BV(2**bv_width - 1, bv_width)
    return And(BVUGT(state['r'], zero), Not(BVUGT(state['s'], max_value)), Not(BVUGT(state['t'

def bezout(state, bv_width):
    a_bv = BV(a, bv_width)
    b_bv = BV(b, bv_width)
    return Equals(BVAdd(BVMul(a_bv, state['s']), BVMul(b_bv, state['t'])), state['r'])
```

2.2 Exercício 1 - Testes

Para estes exemplos utilizaremos sempre o estado inicial 0, o número de bits do BitVector 32 e o número de estados 20.

Quanto ao valor de a e b iremos dar exemplos para quando `a = b`, `a` e `b` são primos entre si e quando `a` e `b` tem um divisor comum.

Com a = 21, b = 21 obtemos:
r: 21, s: 0, t: 1
> Invariant holds for the first 20 states.
> Invariant holds for the first 20 states.

Com a = 132, b = 67 obtemos:
r: 1, s: 33, t: -65
> Invariant holds for the first 20 states.
> Invariant holds for the first 20 states.

Com a = 154, b = 32 obtemos:
r: 2, s: 5, t: -24 > Invariant holds for the first 20 states.
> Invariant holds for the first 20 states.

2.3 Exercício 1 - Anexo

Corra a célula abaixo para inicializar o programa.

[4]:
```
from pysmt.shortcuts import Symbol, Equals, GT, And, Or, Solver, Int, Not,␣
↪BVUDiv, BVType, BVSub, SBV, BVUGT, BVAdd, BVMul, BV

def declare(i, bv_width):

    bv_type = BVType(bv_width)
    state = {}
    state['s'] = Symbol('s' + str(i), bv_type)
    state['t'] = Symbol('t' + str(i), bv_type)
    state['r'] = Symbol('r' + str(i), bv_type)
    state['s_prox'] = Symbol('s_prox' + str(i), bv_type)
    state['t_prox'] = Symbol('t_prox' + str(i), bv_type)
    state['r_prox'] = Symbol('r_prox' + str(i), bv_type)
    state['q'] = Symbol('q' + str(i), bv_type)
    state['pc'] = Symbol('pc' + str(i), bv_type)

    return state

def init(state, a, b, bv_width):

    A = Equals(state['pc'], BV(0, bv_width))
    B = GT(Int(a), Int(0))
    C = GT(Int(b), Int(0))
    D = Equals(state['r'], BV(a, bv_width))
    E = Equals(state['s'], BV(1, bv_width))
    F = Equals(state['t'], BV(0, bv_width))

    return And(A, B, C, D, E, F)
```

```python
def trans(curr, prox, a, b, bv_width):

    t01 = And(
        Equals(curr['pc'], BV(0, bv_width)),
        Equals(prox['pc'], BV(1, bv_width)),
        Equals(prox['r'], BV(a, bv_width)),
        Equals(prox['r_prox'], BV(b, bv_width)),
        Equals(prox['s'], BV(1, bv_width)),
        Equals(prox['s_prox'], BV(0, bv_width)),
        Equals(prox['t'], BV(0, bv_width)),
        Equals(prox['t_prox'], BV(1, bv_width)),
        Equals(curr['q'], prox['q'])
    )

    t14 = And(
        Equals(curr['pc'], BV(1, bv_width)),
        Equals(prox['pc'], BV(4, bv_width)),
        Equals(curr['r_prox'], BV(0, bv_width)),
        Equals(curr['r'], prox['r']),
        Equals(curr['s'], prox['s']),
        Equals(curr['s_prox'], prox['s_prox']),
        Equals(curr['t'], prox['t']),
        Equals(curr['t_prox'], prox['t_prox']),
        Equals(curr['q'], prox['q'])
    )

    t12 = And(
        Equals(curr['pc'], BV(1, bv_width)),
        Equals(prox['pc'], BV(2, bv_width)),
        Not(Equals(curr['r_prox'], BV(0, bv_width))),
        Equals(curr['r_prox'], prox['r_prox']),
        Equals(curr['r'], prox['r']),
        Equals(curr['s'], prox['s']),
        Equals(curr['s_prox'], prox['s_prox']),
        Equals(curr['t'], prox['t']),
        Equals(curr['t_prox'], prox['t_prox']),
        Equals(curr['q'], prox['q'])
    )

    t23 = And(
        Equals(curr['pc'], BV(2, bv_width)),
        Equals(prox['pc'], BV(3, bv_width)),
        Equals(prox['q'], BVUDiv(curr['r'], curr['r_prox'])),
        Equals(curr['r'], prox['r']),
        Equals(curr['r_prox'], prox['r_prox']),
        Equals(curr['s'], prox['s']),
```

```
        Equals(curr['s_prox'], prox['s_prox']),
        Equals(curr['t'], prox['t']),
        Equals(curr['t_prox'], prox['t_prox'])
    )

    t31 = And(
        Equals(curr['pc'], BV(3, bv_width)),
        Equals(prox['pc'], BV(1, bv_width)),
        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'])
    )

    t44 = And(
        Equals(curr['pc'], BV(4, bv_width)),
        Equals(prox['pc'], BV(4, bv_width)),
        Equals(curr['r'], prox['r']),
        Equals(curr['r_prox'], prox['r_prox']),
        Equals(curr['s'], prox['s']),
        Equals(curr['s_prox'], prox['s_prox']),
        Equals(curr['t'], prox['t']),
        Equals(curr['t_prox'], prox['t_prox']),
        Equals(curr['q'], prox['q'])
    )

    return Or(t01, t14, t12, t23, t31, t44)

def gera_traco(i, a, b, K, bv_width):

    with Solver() as solver:
        states = [declare(i, bv_width) for i in range(K)]
        solver.add_assertion(init(states[i], a, b, bv_width))
        solver.add_assertion(Equals(states[-1]['pc'], BV(4, bv_width)))
        for k in range(K):
            if k>0:
                solver.add_assertion(trans(states[k-1], states[k], a, b,␣
↪bv_width))
        if solver.solve():
```

```python
            print(f"r: {solver.get_value(states[-1]['r']).bv_signed_value()}, s: {solver.get_value(states[-1]['s']).bv_signed_value()}, t: {solver.get_value(states[-1]['t']).bv_signed_value()}")
        else:
            print("> Not feasible.")

def bmc_always(inv, i, a, b, K, bv_width):

    with Solver() as solver:
        states = [declare(i, bv_width) for i in range(K)]
        solver.add_assertion(init(states[i], a, b, bv_width))
        solver.add_assertion(Equals(states[-1]['pc'], BV(4, bv_width)))
        for k in range(K):
            if k>0:
                solver.add_assertion(trans(states[k-1], states[k], a, b, bv_width))
            solver.push()
            solver.add_assertion(Not(inv(states[k], bv_width)))
            if solver.solve():
                print(f"> Invariant does not hold for {k+1} first states. Counter-example:")
                for i,state in enumerate(states[:k+1]):
                    print(f"> State {i}: pc = {solver.get_value(state['pc']).bv_signed_value()}\nq = {solver.get_value(state['q']).bv_signed_value()}\ns = {solver.get_value(state['s']).bv_signed_value()}\nt = {solver.get_value(state['t']).bv_signed_value()}\nr = {solver.get_value(state['r']).bv_signed_value()}\ns' = {solver.get_value(state['s_prox']).bv_signed_value()}\nt' = {solver.get_value(state['t_prox']).bv_signed_value()}\nr' = {solver.get_value(state['r_prox']).bv_signed_value()}")
                return
            else:
                if k==K-1:
                    print(f"> Invariant holds for the first {K} states.")
                else:
                    solver.pop()

def nonzerooverflow(state, bv_width):
    zero = SBV(0, bv_width)
    max_value = BV(2**bv_width - 1, bv_width)
    return And(BVUGT(state['r'], zero), Not(BVUGT(state['s'], max_value)), Not(BVUGT(state['t'], max_value)))

def bezout(state, bv_width):
    a_bv = BV(a, bv_width)
    b_bv = BV(b, bv_width)
```

```
    return Equals(BVAdd(BVMul(a_bv, state['s']), BVMul(b_bv, state['t'])),␣
↪state['r'])
```

Correr célula abaixo para definir o valor do estado inicial, o número de bits do BitVector e o número de estados.

```
[ ]: i = int(input('Insira o número do estado inicial: '))
bv_width = int(input('Insira o número de bits par ao BitVector: '))
K = int(input('Insira o número de estados: '))
```

Correr célula abaixo para definir o valor de `a` e `b` (valores para os quais vai ser cálculado o EXA)

```
[ ]: a = int(input('Insira o valor de a: '))
b = int(input('Insira o valor de b: '))
```

Correr a célula abaixo para correr o programa com os valores definidos anteriormente.

```
[ ]: gera_traco(i, a, b, K, bv_width)
bmc_always(nonzerooverflow, i, a, b, K, bv_width)
bmc_always(bezout, i, a, b, K, bv_width)
```