# Lógica Computacional: 25/26

---

# TP3 - Ex1

$Grupo\ 05$

- Vasco Ferreira Leite (A108399)
- Gustavo da Silva Faria (A108575)
- Afonso Henrique Cerqueira Leal (A108472)

---

# Problema:

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".

1. Construa um SFOTS usando BitVector's de tamanho $n = 16$ bits que descreva o comportamento deste programa. Considere estado de erro quando $r = 0$ ou alguma das variáveis atinge o "overflow".
2. Usando a metodologia das "Constraint Horn Clauses" (CHCs) verifique se é possível determinar um invariante que garanta que nunca se atinge um estado de erro.
3. Verifique, usando a metodologia dos invariantes e interpolantes, se o modelo atinge um estado de erro. Para o cálculo do interpolante usar a metodologia das "Constraint Horn Clauses" (CHCs).

---

## Variáveis e Parâmetros do Modelo:

**Parâmetros de Configuração:**

- `N`: Um inteiro que define a largura dos `BitVecs` (neste caso, **16 bits**). Define o domínio finito onde ocorrem as operações do algoritmo.
- `EXT`: Um inteiro (**64 bits**) que define a largura estendida para cálculos intermédios. É usado para garantir que multiplicações e subtrações não sofram overflow ou underflow antes da verificação.

**Metodologia SFOTS:**

O problema é modelado através de uma classe `Ts` que encapsula:

- **Init**: Condição inicial ($a, b > 0$, identidade dos coeficientes).
- **Tr**: Relação de transição (lógica do passo do algoritmo).
- **Bad**: Condição de erro ($r = 0$ ou overflow).

---

## Célula 1: Configuração Inicial

Importa a biblioteca **Z3** e define as constantes globais:

- `N = 16`: A largura em bits das variáveis do estado.
- `EXT = 64`: Largura estendida para segurança aritmética.

In [ ]:
```python
import z3

N = 16
EXT = 64
```

## Célula 2: `solve_horn`

Define a *engine* de resolução baseada em PDR/Spacer para Cláusulas de Horn (CHCs).

- Configura o solver `HORN` com a engine `spacer`.
- Permite definir um *timeout* e limite de *unfolding* para evitar execução infinita.
- Retorna `sat` (seguro/invariante encontrado) ou `unsat` (inseguro/contra-exemplo).

In [ ]:
```python
def solve_horn(chc, max_unfold=10, timeout_ms=10000):
    z3.set_param(verbose=0)
    s = z3.SolverFor('HORN')
    s.set('engine', 'spacer')
    s.set('spacer.order_children', 2)
    s.set('timeout', timeout_ms)
    if max_unfold > 0:
        s.set('spacer.max_level', max_unfold)
    s.add(chc)
    res = s.check()

    answer = None
    if res == z3.sat:
        try:
            answer = s.model()
        except:
            pass
    elif res == z3.unsat:
        try:
```

```
            answer = s.proof()
        except:
            pass

    return res, answer
```

## Célula 3: Classe `Ts` (Transition System)

Estrutura genérica para representar um Sistema de Transição (SFOTS):

- `add_var`: Cria pares de variáveis $(v, v')$ para o estado atual e próximo.
- `Init`, `Tr`, `Bad`: Armazenam as fórmulas lógicas do sistema.

In [ ]:
```
class Ts(object):
    def __init__(self, name='Ts'):
        self.name = name
        self._vars = []
        self._inputs = []
        self.Tr = z3.BoolVal(True)
        self.Init = z3.BoolVal(True)
        self.Bad = z3.BoolVal(False)

    def add_var(self, sort, name=None):
        idx = len(self._vars)
        pre_name = str(name) if name else f'v_{idx}'
        post_name = (str(name) + "'") if name else f'v_out_{idx}'
        v_in = z3.Const(pre_name, sort)
        v_out = z3.Const(post_name, sort)
        self._vars.append((v_in, v_out))
        return (v_in, v_out)

    def add_input(self, sort, name=None):
        v = z3.Const(name if name else f'i_{len(self._inputs)}', so
        self._inputs.append(v)
        return v

    def pre_vars(self):
        return [u for (u, v) in self._vars]

    def post_vars(self):
        return [v for (u, v) in self._vars]

    def all(self):
        return self.pre_vars() + self.post_vars() + self._inputs

    def sig(self):
        return [v.sort() for (u, v) in self._vars]
```

## Célula 4: `vc_gen` e `interpolate`

Implementação das ferramentas de verificação:

- `vc_gen` : Gera as três condições de verificação (VCs) para segurança: $Init \rightarrow Inv$, $Inv \wedge Tr \rightarrow Inv'$, $Inv \wedge Bad \rightarrow \bot$.
- `interpolate` : Utiliza o solver para tentar encontrar um interpolante de Craig que separe os estados iniciais ($A = Init$) dos estados de erro ( $B = Bad$).

In [ ]:
```python
def free_arith_vars(fml):
    seen = set()
    vars_set = set()
    def visit(f):
        if f in seen: return
        seen.add(f)
        if z3.is_const(f) and f.decl().kind() == z3.Z3_OP_UNINTERPR
            vars_set.add(f)
        for child in f.children():
            visit(child)
    visit(fml)
    return vars_set

def vc_gen(T):
    Inv = z3.Function('Inv', *(T.sig() + [z3.BoolSort()]))
    InvPre = Inv(*T.pre_vars())
    InvPost = Inv(*T.post_vars())
    all_vars = T.all()

    vc_init = z3.ForAll(all_vars, z3.Implies(T.Init, InvPre))
    vc_ind = z3.ForAll(all_vars, z3.Implies(z3.And(InvPre, T.Tr), I
    vc_bad = z3.ForAll(all_vars, z3.Implies(z3.And(InvPre, T.Bad),

    return [vc_init, vc_ind, vc_bad], InvPre

def interpolate(A, B, timeout_ms=10000):
    As = free_arith_vars(A)
    Bs = free_arith_vars(B)
    shared = list(As & Bs)

    sig = [s.sort() for s in shared] + [z3.BoolSort()]
    Itp = z3.Function('Itp', *sig)
    args = shared

    left = z3.ForAll(list(As), z3.Implies(A, Itp(*args)))
    right = z3.ForAll(list(Bs), z3.Implies(Itp(*args), z3.Not(B)))

    res, ans = solve_horn([left, right], timeout_ms=timeout_ms)
    if res == z3.sat:
        return ans.eval(Itp(*args))
    return None
```

## Célula 5: `calc_checked_next`

Realiza a aritmética segura do algoritmo ($curr - q \times next$):

1. Estende os BitVectors para 64 bits para evitar wrap-around.

2. Converte para Inteiros ( `BV2Int` ) para verificar os limites de 16 bits.
3. Suporta verificação *Signed* (para coeficientes $s, t$) e *Unsigned* (para resto $r$).

In [ ]:
```python
def calc_checked_next(curr_bv, q_bv, next_bv, signed_check):
    if signed_check:
        curr_ext = z3.SignExt(EXT - N, curr_bv)
        q_ext    = z3.ZeroExt(EXT - N, q_bv)
        next_ext = z3.SignExt(EXT - N, next_bv)
    else:
        curr_ext = z3.ZeroExt(EXT - N, curr_bv)
        q_ext    = z3.ZeroExt(EXT - N, q_bv)
        next_ext = z3.ZeroExt(EXT - N, next_bv)

    term_ext = q_ext * next_ext
    res_ext  = curr_ext - term_ext

    res_int = z3.BV2Int(res_ext, is_signed=signed_check)

    if signed_check:
        min_int = z3.IntVal(-(1 << (N-1)))
        max_int = z3.IntVal((1 << (N-1)) - 1)
    else:
        min_int = z3.IntVal(0)
        max_int = z3.IntVal((1 << N) - 1)

    is_overflow = z3.Or(res_int < min_int, res_int > max_int)

    res_trunc = z3.Extract(N-1, 0, res_ext)

    return res_trunc, is_overflow
```

## Célula 6: `create_eea_system`

Constrói o SFOTS específico do Algoritmo Estendido de Euclides:

1. Define as variáveis de estado ( `r` , `s` , `t` ) e entradas ( `a` , `b` ).
2. `Init` : Configura $r = a, r' = b$ e coeficientes iniciais.
3. `Tr` : Calcula $q = r/r'$, atualiza variáveis e transita para estado de erro se ocorrer overflow.
4. `Bad` : Define o estado de erro como $r = 0$.

In [ ]:
```python
def create_eea_system(bits=16):
    T = Ts('EEA_SFOTS')
    bv = z3.BitVecSort(bits)

    r, r_out = T.add_var(bv, 'r')
    rp, rp_out = T.add_var(bv, 'rp')
    s, s_out = T.add_var(bv, 's')
    sp, sp_out = T.add_var(bv, 'sp')
    t, t_out = T.add_var(bv, 't')
    tp, tp_out = T.add_var(bv, 'tp')
```

```
    a, a_out = T.add_var(bv, 'a')
    b, b_out = T.add_var(bv, 'b')

    zero_bv = z3.BitVecVal(0, bits)
    one_bv = z3.BitVecVal(1, bits)

    T.Init = z3.And(
        z3.UGT(a, zero_bv), z3.UGT(b, zero_bv),
        r == a, rp == b,
        s == one_bv, sp == zero_bv,
        t == zero_bv, tp == one_bv
    )

    q = z3.UDiv(r, rp)

    next_rp_val, ovf_r = calc_checked_next(r, q, rp, signed_check=Fa
    next_sp_val, ovf_s = calc_checked_next(s, q, sp, signed_check=T
    next_tp_val, ovf_t = calc_checked_next(t, q, tp, signed_check=T

    any_overflow = z3.Or(ovf_r, ovf_s, ovf_t)
    loop_cond = (rp != zero_bv)

    T.Tr = z3.If(
        loop_cond,
        z3.If(
            z3.Not(any_overflow),
            z3.And(
                r_out == rp, rp_out == next_rp_val,
                s_out == sp, sp_out == next_sp_val,
                t_out == tp, tp_out == next_tp_val,
                a_out == a, b_out == b
            ),
            z3.And(
                r_out == zero_bv, rp_out == zero_bv,
                s_out == zero_bv, sp_out == zero_bv,
                t_out == zero_bv, tp_out == zero_bv,
                a_out == a, b_out == b
            )
        ),
        z3.And(
            r_out == r, rp_out == rp,
            s_out == s, sp_out == sp,
            t_out == t, tp_out == tp,
            a_out == a, b_out == b
        )
    )

    T.Bad = (r == zero_bv)

    return T
```

## Célula 7: Execução Principal

Executa o fluxo completo:

1. Cria o sistema.
2. Verifica as VCs (Invariant Safety).
3. Calcula o Interpolante (Craig).

In [ ]:

```python
def main():
    print("=" * 70)
    print(f"  VERIFICAÇÃO EEA: SFOTS COM BITVECTORS ({N}-bit)")
    print("=" * 70)

    T = create_eea_system(N)
    print(f"\nSistema Criado: {T.name}")

    vcs, inv_decl = vc_gen(T)

    print("\n" + "-" * 70)
    print("  1. VERIFICAÇÃO DE INVARIANTE DE SEGURANÇA (CHCs)")
    print("-" * 70)

    res, ans = solve_horn(vcs, timeout_ms=15000)

    print(f"\nResultado: {res}")

    if res == z3.sat:
        print("\n✓ SISTEMA SEGURO (SAT)")
        print("  O solver encontrou um invariante que prova que r n
        if ans:
            print("\nInvariante sintetizado:")
            print(ans.eval(inv_decl))
    elif res == z3.unsat:
        print("\n✗ SISTEMA INSEGURO (UNSAT)")
        print("  Existe um caminho onde r=0 ou ocorre overflow.")
        if ans:
            print("\nProva/Contra-exemplo:")
            print(ans)
    else:
        print("\n⚠ RESULTADO INCONCLUSIVO (UNKNOWN/TIMEOUT)")

    print("\n" + "-" * 70)
    print("  2. CÁLCULO DE INTERPOLANTE (Craig)")
    print("-" * 70)

    itp = interpolate(T.Init, T.Bad, timeout_ms=15000)

    if itp is not None:
        print("✓ Interpolante encontrado:")
        print(f"  {itp}")
    else:
        print("✗ Não foi possível gerar interpolante.")

    print("\n" + "=" * 70)

if __name__ == "__main__":
    main()
```