Trabalho Prático 3

Grupo 04 - Renato Garcia (A101987) & Bernardo Moniz (A102497)

Problema 3

Enunciado

Considere de novo o 1º problema do trabalho TP2 relativo à descrição da cifra $A5/1$ e o FOTS usando BitVec's que aí foi definido para a componente do gerador de chaves. Ignore a componente de geração final da chave e restrinja o modelo aos três LFSR's. Sejam X_0, X_1, X_2 as variáveis que determinam os estados dos três LFSR's que ocorrem neste modelo. Como condição inicial e condição de erro use os predicados

$I \equiv (X_0 > 0) \wedge (X_1 > 0) \wedge (X_2 > 0)$ e $E \equiv \neg I$

a. Codifique em "z3" o SFOTS assim definido.

b. Use o algoritmo PDR "property directed reachability" (codifique-o ou use uma versão pré-existente) e, com ele, tente provar a segurança deste modelo.

Resolução

Alínea a)

In [1]:
```python
from pysmt.shortcuts import *
from pysmt.typing import *
import random as rn

def gerar_estado_aleatorio(n):
    return BV(rn.getrandbits(n), n)

def bsel(lfsr, pos):
    return BVExtract(lfsr, start=pos, end=pos)

lfsr_sizes = [19, 22, 23]
lfsr_bitclock_idx = [8, 10, 10]
lfsr_tapping_idx = [[13, 16, 17, 18], [20, 21], [7, 20, 21, 22]]

def tapping(x, indices):
    r = bsel(x, indices[0])
    for i in range(1, len(indices)):
        r = BVXor(r, bsel(x, indices[i]))
    return BVZExt(r, x.bv_width() - 1)

def declare(n):
    result = {}
    for i in range(3):
        result[f"lfsr{i}"] = Symbol(f"lfsr{i}_{n}", types.BVType(lfsr_sizes[i]))
    return result

def trans(curr, prox):
    lfsr0_b = bsel(curr['lfsr0'], 8)
    lfsr1_b = bsel(curr['lfsr1'], 10)
    lfsr2_b = bsel(curr['lfsr2'], 10)

    majority = BVOr(BVAnd(lfsr0_b, lfsr1_b),
                    BVAnd(lfsr0_b, lfsr2_b),
                    BVAnd(lfsr1_b, lfsr2_b))
    lfsr0_tapping = tapping(curr['lfsr0'], lfsr_tapping_idx[0])
    lfsr0 = Ite(Equals(majority, lfsr0_b), Equals(prox['lfsr0'], BVXor(BVLShl(curr['lfsr0'], 1), lfsr0_tapping)), Equals(prox['lfsr0'], curr['lfsr0']))
    lfsr1_tapping = tapping(curr['lfsr1'], lfsr_tapping_idx[1])
    lfsr1 = Ite(Equals(majority, lfsr1_b), Equals(prox['lfsr1'], BVXor(BVLShl(curr['lfsr1'], 1), lfsr1_tapping)), Equals(prox['lfsr1'], curr['lfsr1']))
    lfsr2_tapping = tapping(curr['lfsr2'], lfsr_tapping_idx[2])
    lfsr2 = Ite(Equals(majority, lfsr2_b), Equals(prox['lfsr2'], BVXor(BVLShl(curr['lfsr2'], 1), lfsr2_tapping)), Equals(prox['lfsr2'], curr['lfsr2']))
    return And(lfsr0, lfsr1, lfsr2)

def init(s):
    # Condição inicial I: X0 > 0 ∧ X1 > 0 ∧ X2 > 0
    init_cond = And(
        BVUGT(s['lfsr0'], BV(0, lfsr_sizes[0])),
        BVUGT(s['lfsr1'], BV(0, lfsr_sizes[1])),
        BVUGT(s['lfsr2'], BV(0, lfsr_sizes[2]))
    )
    return And(
        init_cond,
        Equals(s['lfsr0'], gerar_estado_aleatorio(lfsr_sizes[0])),
        Equals(s['lfsr1'], gerar_estado_aleatorio(lfsr_sizes[1])),
        Equals(s['lfsr2'], gerar_estado_aleatorio(lfsr_sizes[2]))
    )

def error_condition(s):
    # Condição de erro E: ¬I (negação da condição inicial)
    return Not(And(
        BVUGT(s['lfsr0'], BV(0, lfsr_sizes[0])),
        BVUGT(s['lfsr1'], BV(0, lfsr_sizes[1])),
        BVUGT(s['lfsr2'], BV(0, lfsr_sizes[2]))
    ))

with Solver(name="z3") as solver:
    k = 8
    states = [declare(i) for i in range(k+1)]
    solver.add_assertion(init(states[0]))

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

    solver.push()
    solver.add_assertion(error_condition(states[k]))
    is_error_reachable = solver.solve()
    solver.pop()

    print(f"Condição de Erro Alcançável: {is_error_reachable}")

    if solver.solve():

        print("  ", end="")
        for ii in range(3):
            lfsr_header = f"LFSR{ii}".center(lfsr_sizes[ii] + 15)
            print(f'{lfsr_header}', end=" ")
        print()


        for ii in range(k+1):
            print(f"{ii:2d}", end=" ")
            for j in range(3):
                value = states[ii]['lfsr'+str(j)]
                binary_str = solver.get_value(value).bv_bin_str()
                decimal_value = solver.get_value(value).constant_value()

                formatted_output = f'{binary_str} ({decimal_value})'.ljust(lfsr_sizes[j] + 15)
                print(f'{formatted_output}', end=" ")
            print()
```

```
Condição de Erro Alcançável: False
              LFSR0                           LFSR1                              LFSR2
 0 0110101001100110001 (217905)       0001001001110101010000 (302416)       00011111001001000000101 (1020421)
 1 1101010011001100010 (435810)       0010010011101010100000 (604832)       00011111001001000000101 (1020421)
 2 1010100110011000101 (347333)       0100100111010101000000 (1209664)      00111110010010000001010 (2040842)
 3 1010100110011000101 (347333)       1001001110101010000001 (2419329)      01111100100100000010101 (4081685)
 4 0101001100110001010 (170378)       0010011101010100000011 (644355)       11111001001000000101010 (8163370)
 5 1010011001100010101 (340757)       0100111010101000000110 (1288710)      11111001001000000101010 (8163370)
 6 1010011001100010101 (340757)       1001110101010000001101 (2577421)      11110010010000001010101 (7938133)
 7 0100110011000101011 (157227)       0011101010100000011011 (960539)       11110010010000001010101 (7938133)
 8 1001100110001010110 (314454)       0111010101000000110110 (1921078)      11100100100000010101011 (7487659)
```

Alínea b)

In [2]:
```python
class PDR:
    def __init__(self, system):
        self.system = system
        self.frames = [system.init]
        self.solver = Solver(name="z3")

        self.prime_map = {
            v: Symbol(f"{v.symbol_name().split('_')[0]}_prime", v.symbol_type())
            for v in system.symbols
        }

    def check_property(self, prop):
        """Property Directed Reachability approach."""
        print("A iniciar verificação da propriedade...")
        frame_num = 0
        while True:
            print(f"\n[Frame {frame_num}] A verificar estados que violam a propriedade...")
            cube = self.get_bad_state(prop)
            if cube is not None:
                print(f"  -> Estado mau encontrado no frame {frame_num}: {cube}")
                if self.recursive_block(cube):
                    print("  [ERRO] Sistema inseguro! Bug encontrado.")
                    return False
                else:
                    print(f"  [Frame {frame_num}] Estado mau bloqueado com sucesso.")
            else:
                # Checking if the last two frames are equivalent i.e., are inductive
                print(f"  [Frame {frame_num}] Nenhum estado mau encontrado. A verificar indutividade...")
                if self.inductive():
                    print(f"  [SUCESSO] O sistema é seguro após {frame_num} frames!")
                    return True
                else:
                    print(f"  [Frame {frame_num}] Adicionando novo frame para continuar.")
                    self.frames.append(TRUE())
            frame_num += 1

    def get_bad_state(self, prop):
        """Extracts a reachable state that intersects the negation of the property"""
        formula = And(self.frames[-1], Not(prop))
        self.solver.push()
        self.solver.add_assertion(formula)
        is_sat = self.solver.solve()

        if is_sat:
            # Create a cube representing the bad state
            bad_state_cube = And([
                Equals(v, self.solver.get_value(v))
                for v in self.system.symbols
            ])
            print(f"    [DEBUG] Estado mau descoberto: {[self.solver.get_value(v) for v in self.system.symbols]}")
            self.solver.pop()
            return bad_state_cube

        self.solver.pop()
        return None

    def solve(self, formula):
        """Provides a satisfiable assignment to the state variables"""
        self.solver.push()
        self.solver.add_assertion(formula)
        is_sat = self.solver.solve()

        if is_sat:
            cube = And([
                Equals(v, self.solver.get_value(v))
                for v in self.system.symbols
            ])
            self.solver.pop()
            return cube

        self.solver.pop()
        return None

    def recursive_block(self, cube):
        """Blocks the cube at each frame, if possible."""
        print("  [Bloqueio] Tentar bloquear estado mau nos frames anteriores...")
        for i in range(len(self.frames)-1, 0, -1):
            print(f"    Tentar bloquear no frame {i}...")
            cubeprime = cube.substitute({
                v: self.prime_map[v] for v in self.system.symbols
            })

            cubepre = self.solve(And(
                self.frames[i-1],
                self.system.trans,
                Not(cube),
                cubeprime
            ))

            if cubepre is None:
                print(f"    [SUCESSO] Estado mau bloqueado até ao frame {i}.")
                for j in range(1, i+1):
                    self.frames[j] = And(self.frames[j], Not(cube))
                return False

            print(f"    [DEBUG] Estado predecessor encontrado no frame {i}: {cubepre}")
            cube = cubepre

        print("  [ERRO] Estado mau não pode ser bloqueado!")
        return True

    def inductive(self):
        """Checks if last two frames are equivalent"""
        if len(self.frames) > 1:
            is_inductive = self.solve(Not(EqualsOrIff(self.frames[-1], self.frames[-2]))) is None
            if is_inductive:
                print("  [DEBUG] Frames são indutivos. Propriedade mantida.")
            else:
                print("  [DEBUG] Frames não são indutivos. Continuando...")
            return is_inductive
        return False

class LFSRSystem:
    def __init__(self):

        self.symbols = [
            Symbol("lfsr0", BVType(lfsr_sizes[0])),
            Symbol("lfsr1", BVType(lfsr_sizes[1])),
            Symbol("lfsr2", BVType(lfsr_sizes[2]))
        ]


        self.init = self.init_condition()

        self.trans = self.transition()

    def init_condition(self):

        init_vars = dict(zip(["lfsr0", "lfsr1", "lfsr2"], self.symbols))
        return And(
            BVUGT(init_vars['lfsr0'], BV(0, lfsr_sizes[0])),
            BVUGT(init_vars['lfsr1'], BV(0, lfsr_sizes[1])),
            BVUGT(init_vars['lfsr2'], BV(0, lfsr_sizes[2])),
            Equals(self.symbols[0], BV(rn.getrandbits(lfsr_sizes[0]), lfsr_sizes[0])),
            Equals(self.symbols[1], BV(rn.getrandbits(lfsr_sizes[1]), lfsr_sizes[1])),
            Equals(self.symbols[2], BV(rn.getrandbits(lfsr_sizes[2]), lfsr_sizes[2]))
        )

    def transition(self):

        curr_vars = {
            'lfsr0': self.symbols[0],
            'lfsr1': self.symbols[1],
            'lfsr2': self.symbols[2]
        }
        prox_vars = {
            'lfsr0': Symbol("lfsr0_next", BVType(lfsr_sizes[0])),
            'lfsr1': Symbol("lfsr1_next", BVType(lfsr_sizes[1])),
            'lfsr2': Symbol("lfsr2_next", BVType(lfsr_sizes[2]))
        }

        lfsr0_b = bsel(curr_vars['lfsr0'], 8)
        lfsr1_b = bsel(curr_vars['lfsr1'], 10)
        lfsr2_b = bsel(curr_vars['lfsr2'], 10)

        majority = BVOr(BVAnd(lfsr0_b, lfsr1_b),
                        BVAnd(lfsr0_b, lfsr2_b),
                        BVAnd(lfsr1_b, lfsr2_b))

        lfsr0_tapping = tapping(curr_vars['lfsr0'], lfsr_tapping_idx[0])
        lfsr0 = Ite(Equals(majority, lfsr0_b),
                    Equals(prox_vars['lfsr0'], BVXor(BVLShl(curr_vars['lfsr0'], 1), lfsr0_tapping)),
                    Equals(prox_vars['lfsr0'], curr_vars['lfsr0']))

        lfsr1_tapping = tapping(curr_vars['lfsr1'], lfsr_tapping_idx[1])
        lfsr1 = Ite(Equals(majority, lfsr1_b),
                    Equals(prox_vars['lfsr1'], BVXor(BVLShl(curr_vars['lfsr1'], 1), lfsr1_tapping)),
                    Equals(prox_vars['lfsr1'], curr_vars['lfsr1']))

        lfsr2_tapping = tapping(curr_vars['lfsr2'], lfsr_tapping_idx[2])
        lfsr2 = Ite(Equals(majority, lfsr2_b),
                    Equals(prox_vars['lfsr2'], BVXor(BVLShl(curr_vars['lfsr2'], 1), lfsr2_tapping)),
                    Equals(prox_vars['lfsr2'], curr_vars['lfsr2']))

        return And(lfsr0, lfsr1, lfsr2)

if __name__ == "__main__":
    system = LFSRSystem()

    #Proposição 1: Os LFSRs devem ser maiores que zero
    safety_prop = And(
         BVUGT(system.symbols[0], BV(0, lfsr_sizes[0])),
         BVUGT(system.symbols[1], BV(0, lfsr_sizes[1])),
         BVUGT(system.symbols[2], BV(0, lfsr_sizes[2]))
    )

    # Proposição 2: Os LFSRs devem ser menores ou iguais ao máximo permitido
    # safety_prop = And(
    #     BVULE(system.symbols[0], BV(2**lfsr_sizes[0] - 1, lfsr_sizes[0])),
    #     BVULE(system.symbols[1], BV(2**lfsr_sizes[1] - 1, lfsr_sizes[1])),
    #     BVULE(system.symbols[2], BV(2**lfsr_sizes[2] - 1, lfsr_sizes[2]))
    # )

    # Proposição 3: Os LFSRs devem ser maiores ou iguais a zero
    #safety_prop = And(
    #     BVUGE(system.symbols[0], BV(0, lfsr_sizes[0])),
    #     BVUGE(system.symbols[1], BV(0, lfsr_sizes[1])),
    #     BVUGE(system.symbols[2], BV(0, lfsr_sizes[2]))
    #)

    # Proposição 4: Os LFSRs devem ser iguais a zero
    # safety_prop = And(
    #     Equals(system.symbols[0], BV(0, lfsr_sizes[0])),  # LFSR0 deve ser zero
    #     Equals(system.symbols[1], BV(0, lfsr_sizes[1])),  # LFSR1 deve ser zero
    #     Equals(system.symbols[2], BV(0, lfsr_sizes[2]))   # LFSR2 deve ser zero
    # )

     #Proposição 5: Os LFSRs devem ser menores que 1
    #safety_prop = And(
    #     BVULT(system.symbols[0], BV(1, lfsr_sizes[0])),  # LFSR0 deve ser menor que 1
    #     BVULT(system.symbols[1], BV(1, lfsr_sizes[1])),  # LFSR1 deve ser menor que 1
    #     BVULT(system.symbols[2], BV(1, lfsr_sizes[2]))   # LFSR2 deve ser menor que 1
    #)

    pdr = PDR(system)
    result = pdr.check_property(safety_prop)
```

```
A iniciar verificação da propriedade...

[Frame 0] A verificar estados que violam a propriedade...
  [Frame 0] Nenhum estado mau encontrado. A verificar indutividade...
  [Frame 0] Adicionando novo frame para continuar.

[Frame 1] A verificar estados que violam a propriedade...
    [DEBUG] Estado mau descoberto: [0_19, 0_22, 0_23]
  -> Estado mau encontrado no frame 1: ((lfsr0 = 0_19) & (lfsr1 = 0_22) & (lfsr2 = 0_23))
  [Bloqueio] Tentar bloquear estado mau nos frames anteriores...
    Tentar bloquear no frame 1...
```

[DEBUG] Estado predecessor encontrado no frame 1: ((ifxr0 = 205839_19) & (ifxr1 = 2212669_22) & (ifxr2 = 8369499_23))
[ERRO] Estado mau não pode ser bloqueado!
[ERRO] Sistema inseguro! Bug encontrado.