Lógica Computacional 24/25

Grupo 09

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

Descrição do Problema

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

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

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

```
from z3 import *
from random import getrandbits
```

Implementação do SFOTS

Definição das Variáveis e Estados:

```
# Função para criar os estados dos LFSRs
def criar_estado(indice):
    estado = {
        'X0': BitVec(f'X0_{indice}', 19),
        'X1': BitVec(f'X1_{indice}', 22),
        'X2': BitVec(f'X2_{indice}', 23),
    }
    return estado
```

Condição Inicial

A condição inicial assegura que todos os registros LFSRs têm valores positivos.

```
# Função que define a condição inicial
def condicao_inicial(estado):
    return And(estado['X0'] > 0, estado['X1'] > 0, estado['X2'] > 0)
```

Condição de Erro

A condição de erro é o complemento da inicial.

```
# Função que define a condição de erro
def condicao_erro(estado):
    return Not(condicao_inicial(estado))
```

Transições do Sistema

Definição das transições entre os estados dos três LSFRs no modelo. Cada LFSR possui um estado representado por um BitVec e a função seguinte descreve como esses estados mudam com base em condições específicas.

```
# Função para definir as transições entre estados
def trans(atual, proximo):
    # Bits de controlo
    bit_controlo1 = Extract(8, 8, atual['X0'])  # Extrai o 8º bit de
X0 (o bit na posição 8).
    bit_controlo2 = Extract(10, 10, atual['X1']) # Extrai o 10º bit de
X1.
    bit_controlo3 = Extract(10, 10, atual['X2']) # Extrai o 10º bit de
X2.

    # Transição para X0
    transicao1_ativa = And(
        proximo['X0'] == Concat(  # o próximo estado de X0 é
determinado por uma operação XOR (entre 4 bits específicos de X0) e
concatenado com o bit 18 de X0.
            Extract(18, 18, atual['X0']) ^ Extract(17, 17,
atual['X0']) ^
            Extract(16, 16, atual['X0']) ^ Extract(13, 13,
atual['X0']),
            Extract(18, 1, atual['X0'])
        ),
        Or(bit_controlo1 == bit_controlo2, bit_controlo1 ==
bit_controlo3)  # a transição só acontece se o bit de controlo 1 for
igual ao bit de controlo 2 ou ao bit de controlo 3.
    )
    transicao1_inativa = And(
        proximo['X0'] == atual['X0'], # Ocorre quando o proximo estado
de X0 é igual ao estado atual de X0.
        Not(Or(bit_controlo1 == bit_controlo2, bit_controlo1 ==
bit_controlo3)) # não occer se o bit de controlo 1 for igual ao bit de
controlo 2 ou ao bit de controlo 3.
    )

    # Transição para X1
    transicao2_ativa = And(
        proximo['X1'] == Concat(
            Extract(21, 21, atual['X1']) ^ Extract(20, 20,
atual['X1']),
            Extract(21, 1, atual['X1'])
```

```
        ),
        Or(bit_controlo1 == bit_controlo2, bit_controlo2 ==
bit_controlo3)
    )
    transicao2_inativa = And(
        proximo['X1'] == atual['X1'],
        Not(Or(bit_controlo1 == bit_controlo2, bit_controlo2 ==
bit_controlo3))
    )

    # Transição para X2
    transicao3_ativa = And(
        proximo['X2'] == Concat(
            Extract(22, 22, atual['X2']) ^ Extract(21, 21,
atual['X2']) ^
            Extract(20, 20, atual['X2']) ^ Extract(7, 7, atual['X2']),
            Extract(22, 1, atual['X2'])
        ),
        Or(bit_controlo3 == bit_controlo2, bit_controlo1 ==
bit_controlo3)
    )
    transicao3_inativa = And(
        proximo['X2'] == atual['X2'],
        Not(Or(bit_controlo3 == bit_controlo2, bit_controlo1 ==
bit_controlo3))
    )

    # Retorna a conjunção de todas as transições possíveis
    return And(
        Or(transicao1_ativa, transicao1_inativa),
        Or(transicao2_ativa, transicao2_inativa),
        Or(transicao3_ativa, transicao3_inativa)
    )
```

Geração e Verificação do SFOTS

O sistema gera uma sequência de estados a partir da condição inicial, garantindo que nenhum estado satisfaça a condição de erro.

```
# Função para gerar e testar o SFOTS
def gerar_sfots(cond_inicial, cond_erro, transicao, passos):
    solver = Solver()
    estados = [criar_estado(i) for i in range(passos)]

    # Adicionar condições iniciais
    solver.add(cond_inicial(estados[0]))

    # Garantir que nenhum estado é de erro
    for i in range(passos):
```

```
        solver.add(Not(cond_erro(estados[i])))

    # Adicionar as transições entre estados
    for i in range(passos - 1):
        solver.add(transicao(estados[i], estados[i + 1]))

    # Verificar se o sistema é resolvível
    if solver.check() == sat:
        print("Caminho encontrado")
        modelo = solver.model()
        for i in range(passos):
            print(f"Passo {i + 1}:")
            for nome, valor in estados[i].items():
                estado_bin = format(modelo.eval(valor).as_long(),
f'0{modelo[valor].size()}b')
                print(f"  {nome} = {estado_bin} =
{modelo.eval(valor)}")
            print("--------------------------------")
    else:
        print("Caminho não encontrado")


# Exemplo de execução do SFOTS com 8 passos
gerar_sfots(condicao_inicial, condicao_erro, trans, 8)
```

```
Caminho encontrado
Passo 1:
  X0 = 0000000000001010010 = 82
  X1 = 00100100000011010101 = 590677
  X2 = 01001101000010010011000 = 2524312
--------------------------------
Passo 2:
  X0 = 0000000000000101001 = 41
  X1 = 00010010000000110101010 = 295338
  X2 = 01001101000010010011000 = 2524312
--------------------------------
Passo 3:
  X0 = 0000000000000010100 = 20
  X1 = 00001001000000011010101 = 147669
  X2 = 01001101000010010011000 = 2524312
--------------------------------
Passo 4:
  X0 = 0000000000000001010 = 10
  X1 = 00000100100000001101010 = 73834
  X2 = 01001101000010010011000 = 2524312
--------------------------------
Passo 5:
  X0 = 0000000000000000101 = 5
  X1 = 00000010010000000110101 = 36917
  X2 = 01001101000010010011000 = 2524312
```

```
--------------------------------
Passo 6:
  X0 = 0000000000000000010 = 2
  X1 = 00000001001000000011010 = 18458
  X2 = 01001101000010010011000 = 2524312
--------------------------------
Passo 7:
  X0 = 0000000000000000001 = 1
  X1 = 00000000100100000001101 = 9229
  X2 = 01001101000010010011000 = 2524312
--------------------------------
Passo 8:
  X0 = 0000000000000000001 = 1
  X1 = 00000000010010000000110 = 4614
  X2 = 00100110100001001001100 = 1262156
--------------------------------
```

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.

Verificação de Segurança com PDR

Utilizamos Property Directed Reachability (PDR) para garantir que o sistema nunca alcança um estado de erro.

Criar as variáveis de estado do modelo:

```python
def estados_pdr():
    return {
        'X0': BitVec('X0', 19),
        'X1': BitVec('X1', 22),
        'X2': BitVec('X2', 23),
    }
```

Obter Cubo Inválido no Algoritmo PDR

Um cubo inválido representa um conjunto de valores que viola a condição de segurança no estado atual.

```python
def obter_cubo_invalido(cond_erro, frames, k, solver):
    estado_atual = estados_pdr()
    solver.push()
    for frame in frames:
        solver.add(Not(frame))
    solver.add(cond_erro(estado_atual))

    if solver.check() == sat:
        modelo = solver.model()
        cubo = {var: modelo.eval(var, model_completion=True) for var
```

```
in estado_atual.values()}
        solver.pop()
        return cubo
    solver.pop()
    return None
```

Tentar bloquear um cubo inválido:

A função `bloquear_cubo` é uma parte essencial do algoritmo PDR. Ela tenta bloquear um cubo inválido adicionando restrições que evitam que ele reapareça em frames futuros.

```
def bloquear_cubo(cubo, frames, k, transicao, solver):
    for i in range(k, 0, -1):
        solver.push()
        estado_anterior = estados_pdr()
        estado_atual = estados_pdr()

        solver.add(transicao(estado_anterior, estado_atual))
        for j in range(i):
            solver.add(Not(frames[j]))

        cond_bloqueio = True
        for var, val in cubo.items():
            cond_bloqueio = And(cond_bloqueio, var != val)

        solver.add(cond_bloqueio)
        if solver.check() == unsat:
            solver.pop()
            print(f"Cubo bloqueado no frame {i}")
            frames[i] = cond_bloqueio
            return True
        solver.pop()

    return False
```

Função verificar_pdr

A função `verificar_pdr` implementa o algoritmo de **Property Directed Reachability (PDR)**, que é utilizado para verificar a segurança de sistemas dinâmicos modelados com **satisfatibilidade booleana (SAT)**. O PDR é um algoritmo de verificação de modelo incremental, que tenta provar que um sistema nunca alcança um estado "inválido" (erro), com base em uma sequência de condições de segurança e transições.

```
def verificar_pdr(cond_inicial, transicao, cond_erro):
    solver = Solver()
    frames = [Not(cond_inicial(estados_pdr()))]
    k = 0

    while True:
```

```
        print(f"Iteração {k}")
        cubo_invalido = obter_cubo_invalido(cond_erro, frames, k,
solver)

        if cubo_invalido is None:
            if k > 0 and frames[k] == frames[k - 1]: # Se já houve uma
transição anterior e o frame se mantem igual, ou seja, o sistema não
pode evoluir mais para um estado de erro
                print("Sistema é seguro")
                return
            else:
                frames.append(False) # Adiciona um novo frame à lista
e k é incrementado
                k += 1
        else:
            bloqueado = bloquear_cubo(cubo_invalido, frames, k,
transicao, solver) # Tenta bloquear o cubo invalido
            if not bloqueado:
                print("Sistema não é seguro") # Se não for possível
bloquear o cubo, o sistema não é seguro
                return
```

Execução do PDR

```
# Execução do PDR
verificar_pdr(condicao_inicial, trans, condicao_erro)
```

```
Iteração 0
Iteração 1
Iteração 2
Sistema é seguro
```