TP3 - Exercício 3

Grupo 1

- Diogo Coelho da Silva A100092
- Pedro Miguel Ramôa Oliveira A97686

Problema Proposto: Exercicio 3

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 X0,X1,X2 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 $E \equiv \neg I$

Proposta Resolução

O objetivo deste problema é analisar a segurança de um modelo baseado nos três Linear Feedback Shift Registers (LFSRs) do algoritmo A5/1 utilizando um algoritmo de PDR. A abordagem utilizada foi fundamentada em BitVectors para modelar os estados dos LFSRs, com um foco na construção de um Finite State Transition System (SFOTS). A verificação de segurança foi conduzida com base no algoritmo de PDR.

1. Importar as bibliotecas importantes

In [11]:
```
from z3 import *
```

- `z3` : Importa a biblioteca Z3, um solver de satisfabilidade (SMT)

2.Definição da configuraçao inicial

In [12]:
```
# Tamanho dos LFSRs
n0, n1, n2 = 19, 22, 23
```

- `n0` , `n1` , `n2` : Tamanho dos LFSRs (19, 22 e 23 bits, respectivamente).

In [13]:
```
s0 = [1, 1, 1, 0, 0, 1] + [0] * 13
s1 = [1, 1] + [0] * 20
s2 = [1, 1, 1] + [0] * 12 + [1] + [0] * 7
```

- `s0` , `s1` , `s2` : Vetores de feedback, definidos como listas de bits. Estes vetores determinam como se atualizam os estados para cada LFSR

In [14]:
```
X0 = BitVec('X0', n0)
X1 = BitVec('X1', n1)
X2 = BitVec('X2', n2)
```

- `X0` , `X1` , `X2` : Representam os estados inicias dos três LFSRs, definidos como variáveis de 19, 22 e 23 bits utilizando a biblioteca z3 e com recurso a BitVec's.

3.Funçao de Transiçao do LSFR

In [15]:
```
def transaction(estado, feedback, bitControlo, size):
    feedback_bit = Sum([feedback[i] * ((estado >> i) & 1) for i in range(
    proximoEstado = If(bitControlo, (estado << 1) | feedback_bit, estado)
    return Extract(size - 1, 0, proximoEstado)
```

A função transaction atualiza o estado de um sistema baseado em feedback e um bit de controle. A função recebe como parametros de entrada, o `estado` atual como um número inteiro, uma lista de bits de `feedback` , um `bitControlo` que é um booleano para decidir se a transição deve ocorrer ou não e por fim um `tamanho` do estado após a transição

- 1.Cálculo do Bit de Feedback: A função itera sobre cada bit no feedback. Para cada bit, realiza uma operação binária `AND` bit por bit entre o estado deslocado i posições para a direita e 1 para extrair o bit na posição i.
- 2.Determinação do Próximo Estado: Se o `bitControlo` for verdadeiro, realiza a transição. Desloca o `estado` para a esquerda e adiciona o `feedback_bit` ao estado deslocado com a operação `OR` bit por bit. Se o `bitControlo` for falso, mantém o estado atual.
- 3.Extração do estado final: Utilizamos a função `Extract` para obter os bits do proximoEstado do bit `size - 1` até ao bit 0, garantindo que o estado final tenha o tamanho determinado.

4. Bits de Controlo e Decisao

In [16]:
```
b0 = Extract(8, 8, X0)
b1 = Extract(10, 10, X1)
b2 = Extract(10, 10, X2)


c0 = Or(b0 == b1, b0 == b2)
c1 = Or(b0 == b1, b1 == b2)
c2 = Or(b0 == b2, b1 == b2)
```

Extraímos os bits na posição 8, 10 e 10 de `X0`, `X1` e `X2` respetivamente. A seguir verificamos se os pares de bits são iguais, utilizando a operação lógica `OR` para combinar duas possibilidades de igualdade

5. Transições de Estado

In [ ]:
```
X0_next = transaction(X0, s0, c0, n0)
X1_next = transaction(X1, s1, c1, n1)
X2_next = transaction(X2, s2, c2, n2)
```

Calcula os próximos estados dos LFSRs com base nas regras de transição.

- Próximos Estados:
  - $X0\_next$, $X1\_next$, $X2\_next$: Determinam os próximos estados dos LFSRs usando a função lfsr_transition.
- SFOTS (Sistema de Transição de Estados Finitos):
  - transitions: Combina as condições para garantir que as transições entre estados sejam válidas.

  As transições do sistema foram representadas por:
  $transitions \equiv (X_0 = X_0\_next) \wedge (X_1 = X_1\_next) \wedge (X_2 = X_2\_next)$

6. Condições de Segurança

In [18]:
```
# Condições iniciais e de erro
I = And(X0 > 0, X1 > 0, X2 > 0)  # Condição inicial
E = Not(I)                        # Condição de erro

# SFOTS (Fórmulas de Transição)
transitions = And(X0 == X0_next, X1 == X1_next, X2 == X2_next)
```

- Condições Iniciais:
  - I: Condição inicial do sistema, onde todos os LFSRs devem ter estados positivos.
- Condição de Erro:
  - E: Define uma situação de erro como o complemento de I (ou seja, qualquer estado onde pelo menos um LFSR não é positivo).

7. Prova com PDR (Property Directed Reachability)

In [ ]:
```
# Prova com PDR
def pdr_prove(solver, init, inseguro):
```

```
    frame = [init]  # Frame inicial
    while True:
        # Verifica se existe um estado no frame que atinge o estado inseg
        solver.push()
        solver.add(frame[-1], inseguro)
        if solver.check() == sat:
            print("Contra-exemplo encontrado: modelo inseguro.")
            print(solver.model())
            return False
        solver.pop()

        # Expandir frame com transições válidas
        solver.push()
        solver.add(frame[-1], transitions)
        if solver.check() == unsat:
            print("Modelo seguro: não é possível atingir o estado de erro
            return True
        new_frame = solver.assertions()
        frame.append(And(*new_frame))
        solver.pop()
```

A função pdr_prove implementa o Property Directed Reachability (PDR), que é um método utilizado para verificar a segurança de sistemas dinâmicos, garantindo que estados inseguros não sejam alcançáveis a partir do estado inicial.

A função recebe como parametros de entrada uma instância do solver Z3, um predicado inicial `init` e ainda um predicado de um estado inseguro `insecure` que deve ser evitado.

Começamos por inicializar uma lista `frame` que armazena os diferentes frames (conjuntos de estados) analisados durante a execução do PDR. Começa apenas com o estado inicial.

Enquanto que a segurança não é verificada, estamos a correr o nosso código dentro de um ciclo while.

Temos que verificar se existe um estado no frame atual que pode alcançar o estado não seguro. Guardamos o estado atual do solver. A seguir adicionamos ao solver as condições do ultimo frame e do estado inseguro que foi introduzido como parametro de entrada. A seguir verificamos a satisfatibilidade das condições adicionas com o `solver.check()`. Se o resultado for `sat` existe um caminho para um estado inseguro. Imprimimos esse contra-exemplo e a função retorna `False` pois o sistema não é seguro. Caso contrário revertemos as adições feitas ao solver.

Após isto é necessário expandir o frame atual adicionando as transições do sistema para analisar estados futuros. Voltamos a criar uma instância de um solver e agora ao inves de adicionar o estado inseguro, adicionamos as transições definidas no sistema. Repetimos os passos. Se o resultado for `unsat` não foi possivel expandir o

frame sem violar as condições de segurança, por isso o sistema é seguro. Caso contrário existem transições válidas que podem levar a novos estados.

8.Resolução do Problema

In [ ]:
```
solver = Solver()
solver.add(transitions)
pdr_prove(solver, I, E)
```

```
Modelo seguro: não é possível atingir o estado de erro.
```

Out[ ]: True