tp2pt1

November 5, 2024

0.0.1 Outubro, 8, 2024

0.0.2 TP2 - Grupo 20

Afonso Martins Campos Fernandes - A102940

Luís Felipe Pinheiro Silva - A105530

Exercicio 1: ##### Considere a descrição da cifra A5/1 que consta no documento +Lógica Computacional: a Cifra A5/1 Informação complementar pode ser obtida no artigo da Wikipedia.

a) Definir e codificar, em z3 e usando o tipo BitVec para modelar a informação, uma FSM que descreva o gerador.

[190]:
```
from z3 import *
import random
from random import getrandbits

# Definir os LFSRs como BitVecs com os tamanhos adequados
LFSR1 = BitVec('LFSR1', 19)
LFSR2 = BitVec('LFSR2', 22)
LFSR3 = BitVec('LFSR3', 23)

# Funções de atualização para cada LFSR
def lfsr1_seguinte(LFSR1):
    f = Extract(18, 18, LFSR1) ^ Extract(17, 17, LFSR1) ^ Extract(16, 16, LFSR1)␣
 ↪^ Extract(13, 13, LFSR1)
    return Concat(f, Extract(18, 1, LFSR1))

def lfsr2_seguinte(LFSR2):
    f = Extract(21, 21, LFSR2) ^ Extract(20, 20, LFSR2)
    return Concat(f, Extract(21, 1, LFSR2))

def lfsr3_seguinte(LFSR3):
    f = Extract(22, 22, LFSR3) ^ Extract(21, 21, LFSR3) ^ Extract(20, 20, LFSR3)␣
 ↪^ Extract(7, 7, LFSR3)
    return Concat(f, Extract(22, 1, LFSR3))

# Bits de clock para cada LFSR
cBit1 = Extract(8, 8, LFSR1)
```

```
cBit2 = Extract(10, 10, LFSR2)
cBit3 = Extract(10, 10, LFSR3)

# Função de clock majoritário
def majority(b1, b2, b3):
    return If(b1 + b2 + b3 > 1, BitVecVal(1, 1), BitVecVal(0, 1))

# Calcular o bit majoritário
majority_bit = majority(cBit1, cBit2, cBit3)

# Atualizar os LFSRs com base no bit de clock majoritário
next_LFSR1 = If(cBit1 == majority_bit, lfsr1_seguinte(LFSR1), LFSR1)
next_LFSR2 = If(cBit2 == majority_bit, lfsr2_seguinte(LFSR2), LFSR2)
next_LFSR3 = If(cBit3 == majority_bit, lfsr3_seguinte(LFSR3), LFSR3)

# Solver para testar a transição
solver = Solver()
solver.add(LFSR1 == BitVecVal(getrandbits(19), 19))
solver.add(LFSR2 == BitVecVal(getrandbits(22), 22))
solver.add(LFSR3 == BitVecVal(getrandbits(23), 23))

# Verificação de estados possíveis
if solver.check() == sat:
    modelo = solver.model()
    print("Estado inicial:")
    print("LFSR1:", modelo[LFSR1])
    print("LFSR2:", modelo[LFSR2])
    print("LFSR3:", modelo[LFSR3])

    # Simulação de uma transição
    print("\nPróximo Estado:")
    print("Next_LFSR1:", modelo.evaluate(next_LFSR1))
    print("Next_LFSR2:", modelo.evaluate(next_LFSR2))
    print("Next_LFSR3:", modelo.evaluate(next_LFSR3))
else:
    print("Nenhuma solução foi encontrada.")
```

```
Estado inicial:
LFSR1: 429273
LFSR2: 290254
LFSR3: 6649777

Próximo Estado:
Next_LFSR1: 429273
Next_LFSR2: 145127
Next_LFSR3: 7519192
```

0.0.3 b) Considere as seguintes propriedades de erro:

0.0.4 i) Ocorrência de um "burst" 0^t(t-zeros) que ocorre em 2^t passos ou menos.

Tente codificar estas propriedade e cerificar se são acessíveis a partir de um estado inicial aleatoriamente gerado

[191]:
```
from z3 import *
from random import getrandbits

# Parâmetros da propriedade de erro
t = 3  # Número de zeros consecutivos (tamanho do "burst")
passos = 2 ** t  # Número máximo de passos permitidos para encontrar o "burst"

# Configurar o solver
solver = Solver()

# Inicializar os LFSRs com estados aleatórios
curLFSR1 = BitVecVal(getrandbits(19), 19)
curLFSR2 = BitVecVal(getrandbits(22), 22)
curLFSR3 = BitVecVal(getrandbits(23), 23)

countZeros = BitVecVal(0, 32)  # Inicializar o contador de zeros consecutivos
outputs = []

# Loop para simular os passos e procurar por um "burst" de zeros
for passo in range(passos):
    # Calcular o bit majoritário
    majority_bit = majority(Extract(8, 8, curLFSR1), Extract(10, 10, curLFSR2),␣
↪Extract(10, 10, curLFSR3))

    # Atualizar os LFSRs com base no bit de clock majoritário
    next_LFSR1 = If(Extract(8, 8, curLFSR1) == majority_bit,␣
↪lfsr1_seguinte(curLFSR1), curLFSR1)
    next_LFSR2 = If(Extract(10, 10, curLFSR2) == majority_bit,␣
↪lfsr2_seguinte(curLFSR2), curLFSR2)
    next_LFSR3 = If(Extract(10, 10, curLFSR3) == majority_bit,␣
↪lfsr3_seguinte(curLFSR3), curLFSR3)

    # Atualizar o estado atual dos LFSRs
    curLFSR1, curLFSR2, curLFSR3 = next_LFSR1, next_LFSR2, next_LFSR3

    # Extrair o bit menos significativo de cada LFSR e calcular o bit de saída
    output_bit = Extract(0, 0, curLFSR1) ^ Extract(0, 0, curLFSR2) ^ Extract(0,␣
↪0, curLFSR3)
    outputs.append(output_bit)

    # Verificar se o bit de saída é zero e atualizar o contador de zeros
```

```
    isZero = output_bit == BitVecVal(0, 1)
    countZeros = If(isZero, countZeros + 1, BitVecVal(0, 32))  # Reinicia o␣
↪contador se não for zero

    # Condição de "burst" de zeros: `t` zeros consecutivos
    solver.add(countZeros <= t)

# Restrição para garantir pelo menos um bit zero e um bit um no output
solver.add(Or([output == BitVecVal(0, 1) for output in outputs]))
solver.add(Or([output == BitVecVal(1, 1) for output in outputs]))

# Verificar se o "burst" de zeros é atingível
if solver.check() == sat:
    print("Foi encontrado um burst de zeros dentro do limite.")
else:
    print("Não foi encontrado nenhum burst de zeros dentro do limite.")
```

```
Não foi encontrado nenhum burst de zeros dentro do limite.
```

0.0.5 ii) Ocorrência de um "burst" de tamanho t que repete um "burst" anterior no mesmo output em 2^(t/2) passos ou menos.

Tente codificar estas propriedade e cerificar se são acessíveis a partir de um estado inicial aleatoriamente gerado

[192]:
```
from z3 import *
from random import getrandbits

# Definir os parâmetros da propriedade de erro
t = 10  # Tamanho do "burst"
limite_passos = 2 ** (t // 2)  # Número máximo de passos permitidos entre␣
↪repetições do "burst"

# Configurar o solver
solver = Solver()

# Inicializar os LFSRs com estados aleatórios
curLFSR1 = BitVecVal(getrandbits(19), 19)
curLFSR2 = BitVecVal(getrandbits(22), 22)
curLFSR3 = BitVecVal(getrandbits(23), 23)

output_history = []
found_repetition = False

# Loop para simular os passos e procurar repetição de um "burst"
for passo in range(limite_passos):
    # Calcular o bit majoritário
```

```
    majority_bit = majority(Extract(8, 8, curLFSR1), Extract(10, 10, curLFSR2),␣
↪Extract(10, 10, curLFSR3))

    # Atualizar os LFSRs com base no bit de clock majoritário
    next_LFSR1 = If(Extract(8, 8, curLFSR1) == majority_bit,␣
↪lfsr1_seguinte(curLFSR1), curLFSR1)
    next_LFSR2 = If(Extract(10, 10, curLFSR2) == majority_bit,␣
↪lfsr2_seguinte(curLFSR2), curLFSR2)
    next_LFSR3 = If(Extract(10, 10, curLFSR3) == majority_bit,␣
↪lfsr3_seguinte(curLFSR3), curLFSR3)

    # Atualizar o estado atual dos LFSRs
    curLFSR1, curLFSR2, curLFSR3 = next_LFSR1, next_LFSR2, next_LFSR3

    # Extrair o bit menos significativo para formar a saída
    output_bit = Extract(0, 0, curLFSR1) ^ Extract(0, 0, curLFSR2) ^ Extract(0,␣
↪0, curLFSR3)

    # Adicionar o bit ao histórico de saída
    output_history.append(output_bit)

    # Verificar se uma sequência de comprimento `t` se repete no histórico
    if len(output_history) >= t * 2: # Garantir que há pelo menos duas␣
↪sequências para comparar
        bits = []
        for bit in output_history:
            bits.append(simplify(bit)) # retirar o bit da representação␣
↪simbólica do Extract
        # Comparar a sequência atual de comprimento `t` com as anteriores
        burst_sequence = bits[-t:]  # Últimos `t` bits
        for i in range(len(bits) - t*2 + 1):
            if burst_sequence == bits[i:i + t]:
                # Encontramos uma repetição do burst
                print(burst_sequence)
                found_repetition = True
                solver.add(output_bit == burst_sequence[0])  # Exemplo de␣
↪condição de restrição
                break
    if found_repetition:
        break

# Verificar se o solver encontrou uma solução para a repetição
if solver.check() == sat and found_repetition:
    print("Repetição de burst encontrada dentro do limite de passos.")
else:
```

```
    print("Não foi encontrada nenhuma repetição de burst dentro do limite de␣
↪passos.")
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Repetição de burst encontrada dentro do limite de passos.
```