ex1TP2

November 12, 2022

1 Exercício 1 (Control Flow Automaton) - Trabalho Prático 2

Grupo 4: Carlos Costa-A94543Ruben Silva-A9463

2 Problema:

1. Um programa imperativo pode ser descrito por um modelo do tipo Control Flow Automaton (CFA) como ilustrado no exemplo seguinte: (Não existe imagem) Este programa implementa a multiplicação de dois inteiros a, b , fornecidos como "input", e com precisão limitada a n bits (fornecido como parâmetro do programa). Note-se que:

   - Existe a possibilidade de alguma das operações do programa produzir um erro de "overflow".
   - Os nós do grafo representam ações que actuam sobre os "inputs" do nó e produzem um "output" com as operações indicadas
   - Os ramos do grafo representam ligações que transferem o "output" de um nodo para o "input" do nodo seguinte. Esta transferência é condicionada pela satisfação da condição associada ao ramo

   1. Construa um FOTS usando BitVector de tamanho n que descreva o comportamento deste autómato. Para isso identifique as variáveis do modelo, o estado inicial e a relação de transição.
   2. Verifique se $\mathsf{P} \equiv (x * y + z = a * b)$ é um invariante deste comportamento.

3 Análise do Problema

Este é um problema sobre a criação de um "Control Flow Automaton (CFA)". * $pc \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Program Counter * $x \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Variável "a" do input do utilizador * $y \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Variável "b" do input do utilizador * $z \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Variável que dará o output * $x' \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Variável "a " que representa o próximo número do traço * $y' \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Variável "b " que representa o próximo número do traço * $z' \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Variável "z " que representa o próximo número do traço * $n \in \mathbb{N}_{\mathbb{0}}; \rightarrow$ Representa a precisão dos bits

4 Limitações e obrigações

1. **Condições iniciais:**

$$pc = 0 \land x = a \land y = b \land z = 0 \land b >= 0$$

2. **Condição para parar o ciclo:**

$$y = 0 \wedge pc = 0 \wedge pc' = 1 \wedge x' = x \wedge y' = y \wedge z' = z$$

3. **y!=0 e Par:**

$$y! = 0 \wedge mod(y) = 0 \wedge x' = 2 * x \wedge y' = y/2 \wedge z' = z \wedge pc' = 0 \wedge pc = 0$$

4. **y!=0 e Impar:**

$$y! = 0 \wedge mod(y) = 1 \wedge x' = 2 * x \wedge y' = y/2 \wedge z' = z \wedge pc' = 0 \wedge pc = 0$$

5. **Não fazer mais nada após o ciclo parar:**

$$pc = 1 \wedge pc' = 1 \wedge x' = x \wedge y' = y \wedge z' = z$$

6. **Invariante tem de ser sempre verdade para todos as iterações**

$$(x * y + z = a * b)$$

5 Implementaçao do Problema

Importar o solver 1. Instalar o z3-solver a partir da libraria do PyPi (pip) 2. Importar o z3-solver

[ ]:
```
%pip install z3-solver
from z3 import *
```

```
Requirement already satisfied: z3-solver in
c:\users\ruben\appdata\local\programs\python\python310\lib\site-packages
(4.11.2.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: BitVector in
c:\users\ruben\appdata\local\programs\python\python310\lib\site-packages (3.5.0)
Note: you may need to restart the kernel to use updated packages.
```

6 Resolver o codigo

Função "declare" Esta função é responsável pela declaração de todas as variáveis que serão utilizadas no solver. 1. Parâmetros: 1. i -> um inteiro que será responsável por dar o nr às variaveis 2. Função: 1. Inicialmente criamos um dicionário para colocar todas as variáveis necessárias. 2. Criamos 4 variáveis: pc (program counter), x, y, z (parâmetros a, b, z respetivamente). 3. Return do novo dicionário com as variáveis

[ ]:
```
def declare(i,nBits):
    state = {}
    state['pc'] = BitVec('pc'+str(i),nBits)
    state['x'] = BitVec('x'+str(i),nBits)
    state['y'] = BitVec('y'+str(i),nBits)
    state['z'] = BitVec('z'+str(i),nBits)
    return state
```

Função "init" Esta função é responsável pela inicialização do primeiro node do traço (Primeiro membro do dicionário principal da função) e algumas condições lógicas necessárias 1. Parâmetros: 1. *state* -> Primeiro membro do dicionário principal da função 2. *a* -> Variável do input do utilizador 3. *b* -> Variável do input do utilizador 2. Return de um "And" com a seguinte condição lógica: $(pc = 0 \land x = a \land y = b \land z = 0 \land b >= 0)$

```
[ ]: def init(state, a, b):
         # pc=0  =a y=b z=0 b>=0
         return And(state['pc'] == 0, state['x'] == a, state['y'] == b, state["z"]␣
     ↪== 0, b >= 0)
```

Função "trans" Esta função é responsável pela criação das conexões lógicas necessárias para o FOTS fazer sentido e ser o pretendido 1. Parâmetros: 1. *curr* -> Membro atual do dicionário principal da função 2. *prox* -> Membro seguinte ao atual do dicionário principal da função 2. Função: 1. Inicialmente calculamos o número máximo de bits para conseguirmos fazer a condição de overflow 2. Criamos as condições lógicas chamadas transita: 1. transita01: $(y = 0 \land pc = 0 \land pc' = 1 \land x' = x \land y' = y \land z' = z)$ 2. transita02: $(y! = 0 \land even(y) \land x' = 2 * x \land y' = y/2 \land z' = z \land pc' = 0 \land pc = 0)$ 3. transita03: $(y! = 0 \land odd(y) \land x' = x \land y' = y - 1 \land z' = z + x \land pc' = 0 \land pc = 0)$ 4. transita04: $(pc = 1 \land pc' = 1 \land x' = x \land y' = y \land z' = z)$ 3. Return de um "And" com a seguinte condição lógica: ($ transita01 transita02 transita03 transita04 $)

```
[ ]: def trans(curr, prox):
         # (y=0   pc=0   pc =1   x =x   y =y   z =z)
         transita01 = And(curr['y'] == 0, curr["pc"] == 0,
                          prox["x"] == curr["x"], prox["y"] == curr["y"], prox["z"]␣
     ↪== curr["z"], prox["pc"] == 1)

         # if y!=0   even(y)   even=PAR
         # (y!=0   even(y)   x =2*x   y =y/2   z =z   pc = 0   pc=0, pc  <= n)
         # Aqui temos o overflow aqui
         transita02 = And(curr['y'] != 0, (curr['y'] % 2) == 0, prox["x"] ==␣
     ↪2*curr["x"], prox["y"] ==
                          curr["y"]/2, prox["z"] == curr["z"], prox["pc"] == 0,␣
     ↪curr["pc"] == 0)

         # if y!=0   odd(y)   odd=Impar
         # (y!=0   odd(y)   x =x   y =y-1   z =z+x   pc =0   pc=0)
         transita03 = And(curr['y'] != 0, (curr['y'] % 2) == 1, prox["x"] ==␣
     ↪curr["x"], prox["y"] ==
                          curr["y"]-1, prox["z"] == curr["z"]+curr["x"], prox["pc"]␣
     ↪== 0, curr["pc"] == 0)

         # Quebrar o ciclo ACHO
         # (pc =1   pc =1   x =x   y =y   z =z)
         transita04 = And(curr["pc"] == 1, prox["pc"] == 1, prox["y"] ==
                          curr["y"], prox["x"] == curr["x"], prox["z"] == curr["z"])

         return Or(transita01, transita02, transita03, transita04)
```

Função "inv" Esta função é responsável pela testagem do invariante em cada iteração. 1. Parâmetros: 1. *state* -> Membro do dicionário principal da função 2. *a* -> Variável do input do utilizador 3. *b* -> Variável do input do utilizador 2. Return de um "And" com a seguinte condição lógica: $(a * b = x * y + z)$

```
[ ]: def inv(state, a, b):
         # a*b== x*y + z
         return And(a*b==(state["x"]*state["y"])+state["z"])
```

Função "algoritmo_encontrar_k" Esta função calcula o tamanho máximo necessário de nodes do traço para a função funcionar a perfeitamente 1. Parâmetros: 1. *n* -> Variável "*b*" do input do utilizador 2. Função: 1. Ciclo para ir dividindo ou decrementado se o número atual é par ou ímpar 3. Return do counter do ciclo

```
[ ]: def algoritmo_encontrar_k(n):
         n = abs(n)
         counter = 1
         while n:
             if (n % 2 == 0):
                 n = n/2
             else:
                 n = n-1
             counter += 1
         return counter+1
```

Função "cfa" Esta é a função principal e é a que irar juntar as funções todas e gerar o traço pretendido e com ele tabelar o output 1. Parâmetros: 1. declare -> Função declare 2. init -> Função init 3. trans -> Função trans 4. *nBits* -> Precisão de bits (input do utilizador) 5. *a* -> Variável *a* (input do utilizador) 6. *b* -> Variável *b* (input do utilizador) 2. Função: 1. Iniciamos o Solver 2. Descobrimos o número máximo de nodes necessário para criar o traço da função 3. Criar as variáveis todas que serão usadas no solver 4. Inicializar as variáveis 5. Criar a conexão lógica entre os nodes do traço todos e adicionar ao solver 6. Testar o Invariante para todas as iterações 7. Correr o Solver e Tabelar o resultado

```
[ ]: def cfa(declare, init, trans, nBits, a, b):
         s = Solver()

         # Descobrimos o número máximo de nodes necessário para criar o traço da␣
      ↪função
         k = algoritmo_encontrar_k(b)

         # Declarar todas as variaveis que serão usadas no solver
         trace = [declare(i, nBits) for i in range(k)]

         # inicializar o estado inicial a 0
         s.add(init(trace[0], a, b))

         # Criar a conexão lógica entre os nodes do traço todos e adicionar ao solver
```

```
    for i in range(k-1):
        s.add(trans(trace[i], trace[i+1]))

    #Testar o invariante para todos os estagios
    for i in range(k-1):
        s.add(inv(trace[i],a,b))

    # Correr o Solver e Tabelar o resultado
    if s.check() == sat:
        result = 0
        m = s.model()

        if (((2 ** nBits)-1) <= a*b):
            print("Overflow!!!!")
        else:
            for i in range(k):
                print("\nIteração -> "+str(i))
                for v in trace[i]:
                    print(v + '=' + str(m[trace[i][v]]))
                    result = m[trace[i][v]]

            print("\n\nPrecisão: "+str(nBits)+" bits\n" +
                  str(a) + " x "+str(b) + " = " + str(result))

    else:
        if (b < 0):
            print("Sem solução porque B é negativo!!!")
        else:
            print("Sem Solução")
```

Exemplo 1 (Overflow) Exemplo que irá gerar overflow pois 2x4 = 8 e a precisão é de 3 bits

[ ]:
```
a = 2
b = 4
k = 3   # nr max: 7
cfa(declare, init, trans, k, a, b)
```

```
Overflow!!!!
```

Exemplo 2 (Impossível/Overflow) Exemplo que possui um B negativo (isto dá sem solução, pois em (y-1) vai ciclar infinitamente se y é negativo)

[ ]:
```
a = 5
b = -4
k = 10   # nr max: 1023
cfa(declare, init, trans, k, a, b)
```

```
Sem solução porque B é negativo!!!
```

Exemplo 3 (Possível) Exemplo simples

[ ]:
```
a = 2
b = 4
k = 7
cfa(declare, init, trans, k, a, b)
```

```
Iteração -> 0
pc=0
x=2
y=4
z=0

Iteração -> 1
pc=0
x=4
y=2
z=0

Iteração -> 2
pc=0
x=8
y=1
z=0

Iteração -> 3
pc=0
x=8
y=0
z=8

Iteração -> 4
pc=1
x=8
y=0
z=8


Precisão: 7 bits
2 x 4 = 8
```

Exemplo 4 (Possível) Exemplo mais complexo

[ ]:
```
a = 100000000
b = 10
k = 32
cfa(declare, init, trans, k, a, b)
```

```
Iteração -> 0
```

```
pc=0
x=100000000
y=10
z=0

Iteração -> 1
pc=0
x=200000000
y=5
z=0

Iteração -> 2
pc=0
x=200000000
y=4
z=200000000

Iteração -> 3
pc=0
x=400000000
y=2
z=200000000

Iteração -> 4
pc=0
x=800000000
y=1
z=200000000

Iteração -> 5
pc=0
x=800000000
y=0
z=1000000000

Iteração -> 6
pc=1
x=800000000
y=0
z=1000000000


Precisão: 32 bits
100000000 x 10 = 1000000000
```