tp2pt2

November 5, 2024

0.0.1 Outubro, 8, 2024

0.1 TP2 - Grupo 20

Afonso Martins Campos Fernandes - A102940

Luís Filipe Pinheiro Silva - A105530

0.1.1 Exercicio 2 - Multiplicação de Inteiros

Considere o problema descrito no documento "+Lógica Computacional: Multiplicação de Inteiros" . Nesse documento usa-se um "Control Flow Automaton" como modelo do programa imperativo que calcula a multiplicação de inteiros positivos representados por vetores de bits.

Pretende-se:

a. Construir um SFOTS, usando BitVec's de tamanho n , que descreva o comportamento deste autómato; para isso identifique e codifique em `Z3` ou `pySMT`, as variáveis do modelo, o estado inicial , a relação de transição e o estado de erro.

[6]:
```
from pysmt.shortcuts import *
from z3 import *
```

0.1.2 Função states(i, n)

Esta função cria a i-nésima cópia das variáveis de estados, acessíveis apartir de um dicionário

[7]:
```
def states(i,n):
    state = {}

    state['c'] = Int('c'+str(i))
    state['x'] = BitVec('x' + str(i),n)
    state['y'] = BitVec('y' + str(i),n)
    state['z'] = BitVec('z' + str(i),n)

    return state
```

0.1.3 Função init(state, a, b)

- state: Dicionário das variáveis de estado
- a: Valor do 1º número a ser multiplicado
- b: Valor do 2º número a ser multiplicado

Esta função devolve um predicado do Solver que testa se um dado estado, acessado apartir do dicionário state, pode ser um estado inicial do programa

[8]:
```
def init(state,a,b):

    return And(state['c'] == 0, state['x'] == a, state['y'] == b, state['z'] ==␣
↪0)
```

0.1.4 Função trans(atual, prox, n)

- atual: Estado das variáveis atuais
- prox: Estado das variáveis na próxima iteração
- n: número de bits

Esta função devolve um predicado do Solver que testa, entre os estados possíveis, se é possível transitar entre dois deles

[9]:
```
def trans(atual,prox,n):

    #transicões entre estados Ex: t0_1 -> transição entre o estado 0 e o estado 1

    valores_constantes = And(prox['x'] == atual['x'],prox['y'] ==␣
↪atual['y'],prox['z'] == atual['z'])

    t0_1 = And(atual['c'] == 0, prox['c'] == 1, valores_constantes)

    t1_5 = And(atual['c'] == 1, prox['c'] == 5, atual['y'] == 0,␣
↪valores_constantes)

    t1_2 = And(atual['c'] == 1, prox['c'] == 2, Not(atual['y'] == 0),␣
↪URem(atual['y'],2) == 1, valores_constantes)

    t1_3 = And(atual['c'] == 1, prox['c'] == 3, Not(atual['y'] == 0),␣
↪URem(atual['y'],2) == 0, valores_constantes)

    t3_next = And(atual['c'] == 3, prox['x'] == atual['x'] << BitVecVal(1,n),␣
↪prox['y'] == atual['y'] >> BitVecVal(1,n), prox['z'] == atual['z'],␣
↪Or(And(ULT(atual['x'], BitVecVal(2**(n-1),n)),prox['c'] ==␣
↪1),And(UGT(atual['x'] >> BitVecVal(1, n), prox['x']),prox['c'] == 4)))

    t2_next = And(atual['c'] == 2, prox['x'] == atual['x'], prox['y'] ==␣
↪atual['y'] - 1, prox['z'] == atual['z'] + atual['x'], Or(And(ULT(atual['x'],␣
↪BitVecVal(2**(n-1),n)),prox['c'] == 1),And(UGT(atual['x'] >> BitVecVal(1, n),␣
↪prox['x']),prox['c'] == 4)))

    stop_transition = And(prox['c'] == atual['c'], valores_constantes,␣
↪Or(And(atual['c'] == 4, prox['c'] == 4), And(atual['c'] == 5, prox['c'] == 5)))
```

```
    return Or(t0_1,t1_5,t1_2,t1_3,t3_next,t2_next,stop_transition)
```

0.1.5 Função error(state, a, b)

- state: Dicionário das variáveis de estado
- a: Valor do 1º número a ser multiplicado
- b: Valor do 2º número a ser multiplicado

Esta função devolve um predicado do Solver que testa se um dado estado é estado de erro

[10]:
```
def error(state,a,b):

    return Or(state['c'] == 4, And(state['z'] != a*b, state['c'] == 5))
```

b. Usando k-indução verifique nesse SFOTS se a propriedade

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

é um invariante do seu comportamento.

0.1.6 Função gera_traco(states, init, trans, error, inv, k, n, a, b)

- states: Cria variáveis de estado
- init: Condições de estado inicial
- trans: Função transição
- error: Condições de estado de erro
- inv: Invariante a comprovar
- k: valor do traço
- n: número de bits
- a: 1º valor a ser multiplicado
- b: 2º valor a ser multiplicado

A função gera_traco serve para testar, para dados n, a e b, se a propriedade inv = (x*y + z = a*b), é um invariante do seu comportamento em k passos

[11]:
```
def gera_traco(states,init,trans,error,inv,k,n,a,b):

    if a < 0 or b < 0:
        print("As variáveis 'a' e 'b' tem que ser maiores que 0.")
        return

    solver = Solver()

    estados = [states(i,n) for i in range(k)]

    solver.add(init(estados[0],a,b))

    for i in range(k-1):
```

```
        solver.add(trans(estados[i],estados[i+1],n))

    for i in range(k):
        solver.push()
        solver.add(error(estados[i],a,b))
        if solver.check() == sat:
            m = solver.model()
            print(f">0 passo",i,"é um estado de erro")
            return
        solver.pop()

    for i in range(k):
        solver.push()
        solver.add(Not(inv(estados[i],a,b,n)))
        if solver.check() == sat:
            m = solver.model()
            print(f">0 invariante não se verifica nos k estados inicais.")
            for i in range(k):
                print("x, c, inv: ", m[estados[i]['x']], "|"␣
↪,m[estados[i]['c']], "|" ,m[estados[i]['x']]*m[estados[i]['y']] +␣
↪m[estados[i]['z']], "=\=", a*b)
            return
        solver.pop()


    check = solver.check()

    if check == sat:
        m = solver.model()
        for i in range(k):
            print("Passo ",i, end=" | ")
            if m[estados[i]['c']] == 4:
                print("Estado de Erro")
            else:
                for v in estados[i]:
                    print(v,"=", m[estados[i][v]] , end=" | ")
                print("")

        if not m[estados[i]['c']] == 4:
            print(f"> 0 invariante verifica-se por k-indução (k={k}).")
    else:
        print(check)
```

0.1.7 A função check_inv(state, a, b, n)

- state: Dicionário das variáveis de estado
- a: Valor do 1º número a ser multiplicado

- b: Valor do 2º número a ser multiplicado
- n: Número de bits

A função devolve um predicado do Solver que testa se no estado 'state' o invariante comprova-se

[12]:
```
def check_inv(state,a,b,n):

    return (((state['x'] * state['y']) + state['z']) == BitVecVal(a, n) *␣
 ↪BitVecVal(b, n))

gera_traco(states,init,trans,error,check_inv,20,9,120,3)
```

```
Passo  0 | c = 0 | x = 120 | y = 3 | z = 0 |
Passo  1 | c = 1 | x = 120 | y = 3 | z = 0 |
Passo  2 | c = 2 | x = 120 | y = 3 | z = 0 |
Passo  3 | c = 1 | x = 120 | y = 2 | z = 120 |
Passo  4 | c = 3 | x = 120 | y = 2 | z = 120 |
Passo  5 | c = 1 | x = 240 | y = 1 | z = 120 |
Passo  6 | c = 2 | x = 240 | y = 1 | z = 120 |
Passo  7 | c = 1 | x = 240 | y = 0 | z = 360 |
Passo  8 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  9 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  10 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  11 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  12 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  13 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  14 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  15 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  16 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  17 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  18 | c = 5 | x = 240 | y = 0 | z = 360 |
Passo  19 | c = 5 | x = 240 | y = 0 | z = 360 |
> O invariante verifica-se por k-indução (k=20).
```

c. Usando k-indução no FOTS acima e adicionando ao estado inicial a condição

$$(a < 2^{n/2}) \wedge (b < 2^{n/2})$$

verifique a segurança do programa; nomeadamente prove que, com tal estado inicial, o estado de erro nunca é acessível.

0.1.8 Função gera_traco_cond(states, init, trans, error, k, n, a,b)

- states: Cria variáveis de estado
- init: Condições de estado inicial
- trans: Função transição
- error: Condições de estado de erro

- k: valor do traço
- n: número de bits
- a: 1º valor a ser multiplicado
- b: 2º valor a ser multiplicado

Esta função é uma alteração da função gera_traco em que não existe invariante a ser comprovado e foi adicionada uma pré-condição, neste caso (a < 2^(n/2) e b < 2^(n/2)), que garante que o estado de erro é inacessível.

[13]:
```
def gera_traco_cond(states,init,trans,error,k,n,a,b):

    if a < 0 or b < 0:
        print("As variáveis 'a' e 'b' tem que ser maiores que 0.")
        return

    if a >= 2**(n/2) or b >= 2**(n/2):
        print("As variáveis 'a' e 'b' tem que ser menores que 2**(n/2).")
        return

    solver = Solver()

    estados = [states(i,n) for i in range(k)]

    solver.add(init(estados[0],a,b))

    for i in range(k-1):
        solver.add(trans(estados[i],estados[i+1],n))

    for i in range(k):
        solver.push()
        solver.add(error(estados[i],a,b))
        if solver.check() == sat:
            m = solver.model()
            print(f">0 passo",i,"é um estado de erro")
            return
        solver.pop()


    check = solver.check()

    if check == sat:
        m = solver.model()
        for i in range(k):
            print("Passo ",i, end=" | ")
            for v in estados[i]:
                print(v,"=", m[estados[i][v]] , end=" | ")
            print("")
```

```
        if not m[estados[i]['c']] == 4:
            print(f"> O estado de erro é inacessível em",k, "passos")
    else:
        print(check)

gera_traco_cond(states,init,trans,error,20,9,16,3)
```

```
Passo  0 | c = 0 | x = 16 | y = 3 | z = 0 |
Passo  1 | c = 1 | x = 16 | y = 3 | z = 0 |
Passo  2 | c = 2 | x = 16 | y = 3 | z = 0 |
Passo  3 | c = 1 | x = 16 | y = 2 | z = 16 |
Passo  4 | c = 3 | x = 16 | y = 2 | z = 16 |
Passo  5 | c = 1 | x = 32 | y = 1 | z = 16 |
Passo  6 | c = 2 | x = 32 | y = 1 | z = 16 |
Passo  7 | c = 1 | x = 32 | y = 0 | z = 48 |
Passo  8 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  9 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  10 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  11 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  12 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  13 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  14 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  15 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  16 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  17 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  18 | c = 5 | x = 32 | y = 0 | z = 48 |
Passo  19 | c = 5 | x = 32 | y = 0 | z = 48 |
> O estado de erro é inacessível em 20 passos
```