Trabalho TP2 - Multiplicação de Inteiros

Grupo 27

LCC 2024/2025

Rafaela Antunes Pereira A102527

Gonçalo Gonçalves Barroso A102931

Ricardo Eusebio Cerqueira A102878

```
%pip install pysmt
%pip install z3-solver
```

```
Defaulting to user installation because normal site-packages is not
writeable
Requirement already satisfied: pysmt in
/home/utilizador/.local/lib/python3.10/site-packages (0.9.6)

Note: you may need to restart the kernel to use updated packages.
Defaulting to user installation because normal site-packages is not
writeable
Requirement already satisfied: z3-solver in
/home/utilizador/.local/lib/python3.10/site-packages (4.13.0.0)
Note: you may need to restart the kernel to use updated packages.
```

```
from pysmt.shortcuts import *
from pysmt.typing import BVType
```

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.

- b. Usando k-indução verifique nesse SFOTS se a propriedade $\space x \cdot y + z = a \cdot b \space$ é um invariante do seu comportamento.

- c. Usando k-indução no FOTS acima e adicionando ao estado inicial a condição $\space a < 2^{n/2} \land b < 2^{n/2}$, verifique a segurança do programa; nomeadamente prove que, com tal estado inicial, o estado de erro nunca é acessível.

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.

Para representar o programa como um SFOTS (Sistema Finito de Transições de Estado Simples), consideramos o conjunto de variáveis de estado X=[’x’, ’y’, ’z’, ’pc’]X=[’x’, ’y’, ’z’, ’pc’].

Para criar as variáveis do modelo, é definida uma função chamada gState. Esta função recebe como argumentos uma lista com os nomes das variáveis de estado, uma etiqueta, um índice inteiro e o número de bits. Com isso, ela gera a ii-ésima versão das variáveis de estado, rotulada pela etiqueta fornecida. Cada variável lógica segue um formato padrão, onde o nome base da variável de estado é seguido por um delimitador !, criando uma nomenclatura uniforme para as diferentes instâncias.

```
def gState(vars, prefix, i, nBits):
    state = {}
    for v in vars:
        state[v] = Symbol(f'{v}!{prefix}_{i}', BVType(nBits))
    return state
```

Para especificar o estado inicial, é criada uma função chamada init. Esta função recebe como parâmetros um estado potencial do programa (representado por um dicionário de variáveis), dois inteiros aa e bb, e o número de bits. Ela retorna um predicado do pySMT que verifica se as condições $ x=a, y=b, z=0, a≥0, b≥0 $ são todas satisfatórias, ou seja, se o estado fornecido pode ser considerado um estado inicial válido do programa

```
def init(state, a, b, nBits):
    return And(
        Equals(state['pc'], BV(0, nBits)),
        Equals(state['x'], BV(a, nBits)),
        Equals(state['y'], BV(b, nBits)),
```

```
        Equals(state['z'], BV(0, nBits)),
        BVUGE(BV(a, nBits), BV(0, nBits)),
        BVUGE(BV(b, nBits), BV(0, nBits))
    )
```

A função *error* é responsável por verificar se um determinado estado do programa contém valores que excedem os limites representáveis com um número fixo de bits, *nBits*.

Esta retorna um resultado que indica a ocorrência de um erro se qualquer uma das variáveis x, y, ou z estiver fora do intervalo permitido, ajudando assim a garantir a integridade e a segurança do programa ao prevenir o uso de valores inválidos ou excessivos.

```
def error(s, nBits):
    max_value = BV((1 << nBits) - 1, nBits)  # Valor máximo
representável com nBits
    return Or(
        BVUGT(s['x'], max_value),  # BVUGT verifica se s['x'] é maior
que max_value
        BVUGT(s['y'], max_value),
        BVUGT(s['z'], max_value)
    )
```

A função *trans* é responsável por estabelecer as relações de transição entre dois estados do programa, retornando um predicado do pySMT que determina se a transição do primeiro estado para o segundo é válida.

Dentro da função, diversas condições são avaliadas para descrever como as variáveis de estado mudam entre os dois estados. Para os casos em que a verificação de paridade é necessária, a função utiliza even(y), que verifica se a variável y é um número par, onde analisa o bit menos significativo, retornando par caso seja 0, ou 1 caso seja impar.

Assim, a função trans inclui a verificação de paridade nas suas condições, o que garante que as transições entre estados seguem as regras do programa. Isso ajuda a manter os estados que vêm a seguir consistentes e válidos durante a execução do sistema.

```
def even(value, nBits):
    return Equals(BVExtract(value, 0, 0), BV(0, 1))

def trans(curr, prox, nBits):
    t01 = And(
        Equals(curr['pc'], BV(0, nBits)),
        NotEquals(curr['y'], BV(0, nBits)),
        Equals(prox['pc'], BV(1, nBits)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z'])
    )
    t12 = And(
        Equals(curr['pc'], BV(1, nBits)),
```

```
        even(curr['y'], nBits),
        Equals(prox['pc'], BV(2, nBits)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z'])
    )
    t20 = And(
        Equals(curr['pc'], BV(2, nBits)),
        Equals(prox['pc'], BV(0, nBits)),
        Equals(prox['x'], BVMul(curr['x'], BV(2, nBits))),
        Equals(prox['y'], BVUDiv(curr['y'], BV(2, nBits))),
        Equals(prox['z'], curr['z'])
    )
    t13 = And(
        Equals(curr['pc'], BV(1, nBits)),
        Not(even(curr['y'], nBits)),
        Equals(prox['pc'], BV(3, nBits)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z'])
    )
    t34 = And(
        Equals(curr['pc'], BV(3, nBits)),
        Equals(prox['pc'], BV(4, nBits)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z'])
    )
    t40 = And(
        Equals(curr['pc'], BV(4, nBits)),
        Equals(prox['pc'], BV(0, nBits)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], BVSub(curr['y'], BV(1, nBits))),
        Equals(prox['z'], BVAdd(curr['z'], curr['x']))
    )
    t05 = And(
        Equals(curr['pc'], BV(0, nBits)),
        Equals(curr['y'], BV(0, nBits)),
        Equals(prox['pc'], BV(5, nBits)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z'])
    )

    return Or(t01, t12, t13, t20, t34, t40, t05)
```

Usando a função *genTrace*, geramos um possível estado de execução com *n* transições

```
def genTrace(vars, init, trans, error, n, a, b, nBits):
    max_value = (1 << nBits) - 1
    product = a * b

    # Verificar se o produto a*b é maior que o máximo representável
    if product > max_value:
        print(f"Erro: O produto a * b ({product}) é maior que o valor
máximo representável ({max_value}) com {nBits} bits.")
        return

    s = Solver()
    X = [gState(vars, 'X', i, nBits) for i in range(n + 1)]  # Cria
estados de 0 a n (total n+1 estados)

    # Inicializar o estado inicial
    I = init(X[0], a, b, nBits)

    # A primeira iteração começa com o estado inicial
    for i in range(n):  # Muda o loop para ir de 0 a n-1
        # Transição e verificação de erro
        transition = trans(X[i], X[i + 1], nBits)
        error_condition = error(X[i + 1], nBits)

        # Verificar se a transição é satisfatível
        if s.solve([I, transition, Not(error_condition)]):
            print(f"Estado {i} válido:")  # Modificado para começar do
estado 0
            for v in X[i + 1]:  # Mostrar valores do próximo estado
                print(f"    {v} = {s.get_value(X[i + 1][v])}")
            # Atualiza o estado inicial para a próxima iteração
            I = And(I, transition)  # Atualiza I com a transição
realizada
        else:
            print(f"Erro encontrado após a iteração {i}:")  #
Modificado para começar do estado 0
            # Verificando os valores para o estado atual
            for v in X[i]:
                val = s.get_value(X[i][v]) if s.solve([I]) else
"Modelo não encontrado."
                print(f"    {v} = {val}")
            break
    else:
        # Condição final: z no último estado deve ser igual a a * b
        final_check = Equals(X[n]['z'], BV(a * b, nBits))
        if s.solve([I, final_check]):
            print(f"Condição final satisfeita: z = a * b = {a * b}.")
            for v in X[n]:
                print(f"    {v} = {s.get_value(X[n][v])}")
            return True
        else:
```

```
            print("Nenhuma solução que satisfaça a condição final (z
== a * b) encontrada.")
            return False
```

Resultados

```
#Exemplo 1 (Não é satisfeita a condição)
vars = ['pc', 'x', 'y', 'z']
n = 10    # numero de passos
nbits = 8 # numeros com 8 bits
a = 4
b = 6
genTrace(vars, init, trans, error, n,a,b, nbits)
```

```
Estado 0 válido:
    pc = 1_8
    x = 4_8
    y = 6_8
    z = 0_8
Estado 1 válido:
    pc = 2_8
    x = 4_8
    y = 6_8
    z = 0_8
Estado 2 válido:
    pc = 0_8
    x = 8_8
    y = 3_8
    z = 0_8
Estado 3 válido:
    pc = 1_8
    x = 8_8
    y = 3_8
    z = 0_8
Estado 4 válido:
    pc = 3_8
    x = 8_8
    y = 3_8
    z = 0_8
Estado 5 válido:
    pc = 4_8
    x = 8_8
    y = 3_8
    z = 0_8
Estado 6 válido:
    pc = 0_8
    x = 8_8
    y = 2_8
    z = 8_8
Estado 7 válido:
```

```
    pc = 1_8
    x = 8_8
    y = 2_8
    z = 8_8
Estado 8 válido:
    pc = 2_8
    x = 8_8
    y = 2_8
    z = 8_8
Estado 9 válido:
    pc = 0_8
    x = 16_8
    y = 1_8
    z = 8_8
Nenhuma solução que satisfaça a condição final (z == a * b)
encontrada.

False
```

```
#Exemplo 2 (È satisfeita a condição nos passos propostos)
vars = ['pc', 'x', 'y', 'z']
n = 10    # numero de passos
nbits = 8 # numeros com 8 bits
a = 2
b = 4
genTrace(vars, init, trans, error, n,a,b, nbits)
```

```
Estado 0 válido:
    pc = 1_8
    x = 2_8
    y = 4_8
    z = 0_8
Estado 1 válido:
    pc = 2_8
    x = 2_8
    y = 4_8
    z = 0_8
Estado 2 válido:
    pc = 0_8
    x = 4_8
    y = 2_8
    z = 0_8
Estado 3 válido:
    pc = 1_8
    x = 4_8
    y = 2_8
    z = 0_8
Estado 4 válido:
    pc = 2_8
    x = 4_8
```

```
    y = 2_8
    z = 0_8
Estado 5 válido:
    pc = 0_8
    x = 8_8
    y = 1_8
    z = 0_8
Estado 6 válido:
    pc = 1_8
    x = 8_8
    y = 1_8
    z = 0_8
Estado 7 válido:
    pc = 3_8
    x = 8_8
    y = 1_8
    z = 0_8
Estado 8 válido:
    pc = 4_8
    x = 8_8
    y = 1_8
    z = 0_8
Estado 9 válido:
    pc = 0_8
    x = 8_8
    y = 0_8
    z = 8_8
Condição final satisfeita: z = a * b = 8.
    pc = 0_8
    x = 8_8
    y = 0_8
    z = 8_8

True
```

```
# Exemplo 3 (Multiplicação de a por b passa o limite de 8 bits)
vars = ['pc', 'x', 'y', 'z']
n = 10    # numero de passos
nbits = 8 # numeros com 8 bits
a = 51
b = 7
genTrace(vars, init, trans, error, n,a,b, nbits)
```

```
Erro: O produto a * b (357) é maior que o valor máximo representável
(255) com 8 bits.
```

b. Usando k-indução, verifique nesse SFOTS se a propriedade $(x*y+z=a*b)$ é um invariante do seu coportamento.

Sendo *inv* a função que define a propriedade que teremos que verificar se é invariante:

```
def inv(state, a, b, n):
    return Equals(BVAdd(BVMul(state['x'], state['y']), state['z']),
BVMul(BV(a,n), BV(b,n)))
```

Usamos a função kinduction_always, disponibilizada das fichas, com algumas alterações, que nos verifica se a propriedade *inv* é invariante por k-indução:

```
def kinduction_always(init, trans, inv, k, a, b, nbits):
    with Solver() as solver:
        # Gera uma lista de estados iniciais marcados com 'X'
        s = [gState(vars, 'X', i, nbits) for i in range(k)]

        # Adiciona a asserção do estado inicial
        solver.add_assertion(init(s[0], a, b, nbits))

        # Verifica as transições entre os estados
        for i in range(k-1):
            solver.add_assertion(trans(s[i], s[i+1], nbits))

        # Verifica se o invariante é mantido nos k primeiros estados
        for i in range(k):
            solver.push()  # Cria novo contexto para cada estado
            solver.add_assertion(Not(inv(s[i], a, b, nbits)))  #
Verifica se o invariante é violado
            if solver.solve():  # Se encontrar contradição, o
invariante não se mantém
                print(f"> Contradição! O invariante não se verifica
nos k estados iniciais.")
                for st in s:
                    print("x, pc, inv: ", solver.get_value(st['x']),
solver.get_value(st['pc']))
                return  # Interrompe a execução ao detectar erro
            solver.pop()  # Restaura o contexto após verificação

        # Gera estados adicionais marcados com 'Y' para o passo
indutivo
        s2 = [gState(vars, 'Y', i + k, nbits) for i in range(k + 1)]

        # Verifica se o invariante é mantido nos estados em 's2'
        for i in range(k):
            solver.add_assertion(inv(s2[i], a, b, nbits))  # Verifica
se o invariante se mantém
            solver.add_assertion(trans(s2[i], s2[i+1], nbits))  #
Verifica a transição entre os estados

        # Verifica se o invariante é violado no último estado de 's2'
        solver.add_assertion(Not(inv(s2[-1], a, b, nbits)))

        if solver.solve():  # Se encontrar contradição, a segurança
```

```
falha no passo indutivo
            print(f"> Contradição! O passo indutivo não se verifica.")
            for i, state in enumerate(s):
                print(f"> Estado {i}: x =
{solver.get_value(state['x'])}, pc= {solver.get_value(state['pc'])}.")
            return  # Interrompe a execução ao detectar erro

        # Se nenhuma contradição for encontrada, a propriedade é
válida por k-indução
        print(f"> A propriedade verifica-se por k-indução (k={k}).")
```

Resultados

```
a=4
b=0
nbits=8
k=3
kinduction_always(init, trans, inv, k, a, b, nbits)

> A propriedade verifica-se por k-indução (k=3).

a=4
b=6
nbits=8
k=5
kinduction_always(init, trans, inv, k, a, b, nbits)

> A propriedade verifica-se por k-indução (k=5).

a=4
b=0
nbits=8
k=1
kinduction_always(init, trans, inv, k, a, b, nbits)

> Contradição! O passo indutivo não se verifica.
> Estado 0: x = 4_8, pc= 0_8.
```

c. Usando k-indução no FOTS acima e adicionando ao estado inicial a condição $\space a < 2^{n/2} \land b < 2^{n/2}$, verifique a segurança do programa; nomeadamente prove que, com tal estado inicial, o estado de erro nunca é acessível.

A função *init_c* atua como um verificador para estados iniciais, garantindo que as variáveis do estado do programa estejam adequadamente definidas e dentro dos limites desejados, assegurando assim a integridade do estado inicial do programa antes de sua execução.

```
def init_c(state, a, b, nbits):

    x = Equals(state['x'], BV(a, nbits))
    y = Equals(state['y'], BV(b, nbits))
```

```
    z = Equals(state['z'], BV(0, nbits))
    pc = Equals(state['pc'], BV(0, nbits))
    apos = BVUGE(BV(a, nbits), BV(0, nbits))
    bpos = BVUGE(BV(b, nbits), BV(0, nbits))
    cond = And(BVULT(BV(a, nbits), BV(2 ** (nbits//2), nbits)),
BVULT(BV(b, nbits), BV(2 ** (nbits//2), nbits)))

    return And(x, y, z, pc, apos, bpos, cond)
```

A função *kinduction_s* permite verificar se o programa pode ser executado de forma segura, sem chegar a um estado de erro, desde que o estado inicial siga as condições definidas.

```
def kinduction_s(vars, init, trans, error, k, a, b, nbits):
    # Verifica se os valores de 'a' e 'b' são válidos de acordo com a
condição (a < 2^(n/2)) e (b < 2^(n/2))
    # Passo 1
    if a >=  2 ** (nbits//2) or b >= 2 ** (nbits//2):
        print(f"Erro: a ou b ({a}, {b}) é maior ou igual que 2 ^
({nbits//2}) = {2 ** (nbits//2)}")
        return False  # Retorna falso caso a condição não seja
atendida

    with Solver() as solver:
        # Gera uma lista de estados iniciais (marcados com 'X')
        s = [gState(vars, 'X', i, nbits) for i in range(k)]

        # Adiciona a asserção do estado inicial
        solver.add_assertion(init(s[0], a, b, nbits))

        # Verifica as transições entre estados para os primeiros 'k'
estados
        # Passo 2
        for i in range(k-1):
            solver.add_assertion(trans(s[i], s[i+1], nbits))


        # Verifica se algum estado atinge a condição de erro nos
primeiros 'k' passos
        # Passo 3
        for i in range(k):
            solver.push()  # Cria um novo contexto para cada estado
            solver.add_assertion(error(s[i], nbits))  # Adiciona
asserção de erro
            if solver.solve():  # Se o solver encontra um erro
                print(f"> Propriedade de segurança violada nos
primeiros {k} estados!")
                print(f"> Erro encontrado no passo {i}")
                return False  # Se encontrar erro, retorna falso
            solver.pop()  # Restaura o contexto após a verificação
```

```
        # Gera estados adicionais marcados com 'Y' para o passo
indutivo
        s2 = [gState(vars, 'Y', i + k, nbits) for i in range(k + 1)]

        # Verifica se os estados em 's2' são válidos e não geram erro
        for i in range(k):
            solver.add_assertion(Not(error(s2[i], nbits)))  # Verifica
se o estado não é um erro
            if i < k-1:
                solver.add_assertion(trans(s2[i], s2[i+1], nbits))  #
Verifica a transição entre estados

        # Verifica se o último estado de 's2' gera um erro
        solver.add_assertion(error(s2[-1], nbits))


        # Se o solver encontra um erro, a segurança é violada
        if solver.solve():
            print(f"> Propriedade de segurança violada no passo
indutivo!")
            return False

        # Se nenhum erro for encontrado, a segurança é garantida
        print(f"> Programa seguro! Provado por k-indução. (k={k})")
        return True  # Retorna verdadeiro indicando que o programa é
seguro
```

Para provar que, a partir de um estado inicial específico, o estado de erro nunca é acessível usando a técnica de k-indução com a função *kinduction_s*, precisamos garantir que:

- O estado inicial satisfaz as condições requeridas(neste caso, $\space a < 2^{n/2} \land b < 2^{n/2}\space$)
- As transições entre estados não levam a um estado de erro, desde que o estado inicial seja válido.
- No passo indutivo, se os primeiros *k* estados não acessam um estado de erro, então os estados subsequentes também não devem.

Resultados

```
vars = ['x', 'y', 'z', 'pc']  # Lista de variáveis de estado
a = 2
b = 17
nbits = 8
k = 3

kinduction_s(vars, init_c, trans, error, k, a, b, nbits)
```

```
Erro: a ou b (2, 17) é maior ou igual que 2 ^ (4) = 16
```

```
False
```

```
vars = ['x', 'y', 'z', 'pc']  # Lista de variáveis de estado
a = 2
b = 4
nbits = 8
k = 3

kinduction_s(vars, init_c, trans, error, k, a, b, nbits)
```

```
> Programa seguro! Provado por k-indução. (k=3)
```

```
True
```