TP2

Daniel Francisco Texeira Andrade - A100057

Pedro André Ferreira Malainho - A100050

Problema 2

Enunciado

Considere o proble 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 _(x * y + z = a * b)_é um invariante do seu comportamento.

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

Importes

In [2]:
```
from pysmt.shortcuts import *
```

Implementação

BV_Sel Function

Selects the i-th bit from a BitVector z.

`Parameters:`

- `v:` The BitVec from which the bit is to be selected.
- `i:` The index of the bit to select.

`Returns:` A BitVec representing the selected bit.

GenState Function

Generates a state dictionary for the given variables and state label.

`Parameters:`

- `vars:` List of variables names to include in the state.
- `s:` The label for the state.
- `i:` The index of the state.

`Returns:` A dictionary mapping variable names to their corresponding BitVec symbols.

Init1 function

Defines the initial conditions for the state.

`Parameters:`

- `state:` The current state containing variables.

`Returns:` A boolean expression representing the initial conditions.

Init_AB function

Initializes the state with specified values for x, y, z, and pc.

`Parameters:`

- `state:` The state to initialize.
- `a:` The value to assign to x.
- `b:` The value to assign to y.

`Returns:` A boolean expression representing the initialization conditions.

Error1 function

Defines the error conditions based on the state.

`Parameters:`

- `state:` The current state containing variables.

`Returns:` A boolean expression representing the error conditions.

Trans1 function

Defines the transition relations between the current and next state.

`Parameters:`

- `curr:` The current state.
- `prox:` The next state.

`Returns:` A boolean expression representing the valid transitions.

GenTrace function

Generates a trace of the system based on initial conditions and transitions.

`Parameters:`

- `vars:` List of variable names.
- `init:` The initial condition function.
- `trans:` The transition function.
- `n:` The number of transitions to simulate.

`Returns:` None: Prints the state of the system at each step if satisfiable.

In [22]:
```
# Alinea A)

n= 8


def bv_sel(v,i): # seleciona o bit i do BitVec "v"

    return BVExtract(v,start=i,end=i)


def genState(vars, s, i):

    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+s+str(i), BVType(n))

    return state


vars = ['x', 'y', 'z', 'pc']


def init1(state):
    return And(
        Equals(state['z'], BVZero(n)),
        Equals(state['pc'], BVZero(n)),
        Or(
            BVULT(state['y'], state['x']),
            Equals(state['y'], state['x'])
        ),
```

```
        Not(Equals(state['x'], BVZero(n))),
        Not(Equals(state['y'], BVZero(n)))
    )


def init_ab(state, a, b):

    assert not (a == 0 and b == 0)

    if a < b:
        a,b = b,a
    return And(
        Equals(state['x'], BV(a,n)), Equals(state['y'], BV(b,n)), Equals(state['
               Equals(state['pc'], BVZero(n))
    )


def error1(state):

    err_odd = And(
        Not(
            Equals(
                state['y'], BVZero(n)
            )
        ),
        Equals(
            bv_sel(state['y'],0), BVOne(1)
        ),
        state['x'] > BVSub(BV(2**n-1,n), state['z'])
    )

    err_even = And(
        Not(Equals(state['y'], BVZero(n))),
        Equals(bv_sel(state['y'],0), BVZero(1)),
        Equals(bv_sel(state['x'], n-1), BVOne(1))
    )

    return Or(err_odd, err_even)


def trans1(curr, prox):

    tend = And(
        Equals(curr['pc'], BVZero(n)),
        Equals(prox['pc'], BV(3,n)),
        Equals(curr['x'], prox['x']),
        Equals(curr['y'], BVZero(n)),
        Equals(curr['y'], prox['y']),
        Equals(curr['z'], prox['z'])
    )

    tendl = And(
        Equals(curr['pc'], BV(3,n)),
        Equals(prox['pc'], BV(3,n)),
        Equals(curr['x'], prox['x']),
        Equals(curr['y'], prox['y']),
        Equals(curr['z'], prox['z'])
    )

    todd = And(
```

```python
        Equals(curr['pc'], BVZero(n)),
        Equals(prox['pc'], BV(2,n)),
        Equals(curr['x'], prox['x']),
        Equals(bv_sel(curr['y'],0), BVOne(1)),
        Equals(curr['y'], prox['y']),
        Equals(curr['z'], prox['z'])
    )

    toddt = And(
        Equals(curr['pc'], BV(2,n)),
        Equals(prox['pc'], BVZero(n)),
        Equals(prox['x'], curr['x']),
        Not(curr['x'] > BVSub(BV(2**n-1,n), curr['z'])),
        Equals(prox['y'], curr['y'] - BVZExt(BVOne(1), n-1)),
        Equals(prox['z'], curr['z'] + curr['x'])
    )

    teven = And(
        Equals(curr['pc'], BVZero(n)),
        Equals(prox['pc'], BV(1,n)),
        Equals(curr['x'], prox['x']),
        Not(Equals(curr['y'], BVZero(n))),
        Equals(bv_sel(curr['y'],0), BVZero(1)),
        Equals(curr['y'], prox['y']),
        Equals(curr['z'], prox['z'])
    )

    tevent = And(
        Equals(curr['pc'], BV(1,n)),
        Equals(prox['pc'], BVZero(n)),
        Equals(prox['x'], BVLShl(curr['x'], BVZExt(BVOne(1), n-1))),
        Not(Equals(bv_sel(curr['x'], n-1), BVOne(1))),
        Equals(prox['y'], BVLShr(curr['y'], BVZExt(BVOne(1), n-1))),
        Equals(curr['z'], prox['z'])
    )

    return Or(tend, tendl, todd, toddt, teven, tevent)


def genTrace(vars,init,trans,steps):

    with Solver(name="z3") as s:

        X = [genState(vars,'X',i) for i in range(steps+1)]   # cria n+1 estados
        I = init(X[0])
        Tks = [ trans(X[i],X[i+1]) for i in range(steps) ]

        if s.solve([I,And(Tks)]):      # testa se I /\ T^n  é satisfazível
            for i in range(steps):
                print("Estado:",i)
                for v in X[i]:
                    print("          ",v,'=',s.get_value(X[i][v]))


steps = 15
genTrace(vars, init1, trans1, steps)


def invert(trans):
    return lambda curr, prox: trans(prox, curr)
```

```
Estado: 0
	x = 16_8
	y = 15_8
	z = 0_8
	pc = 0_8
Estado: 1
	x = 16_8
	y = 15_8
	z = 0_8
	pc = 2_8
Estado: 2
	x = 16_8
	y = 14_8
	z = 16_8
	pc = 0_8
Estado: 3
	x = 16_8
	y = 14_8
	z = 16_8
	pc = 1_8
Estado: 4
	x = 32_8
	y = 7_8
	z = 16_8
	pc = 0_8
Estado: 5
	x = 32_8
	y = 7_8
	z = 16_8
	pc = 2_8
Estado: 6
	x = 32_8
	y = 6_8
	z = 48_8
	pc = 0_8
Estado: 7
	x = 32_8
	y = 6_8
	z = 48_8
	pc = 1_8
Estado: 8
	x = 64_8
	y = 3_8
	z = 48_8
	pc = 0_8
Estado: 9
	x = 64_8
	y = 3_8
	z = 48_8
	pc = 2_8
Estado: 10
	x = 64_8
	y = 2_8
	z = 112_8
	pc = 0_8
Estado: 11
	x = 64_8
	y = 2_8
	z = 112_8
	pc = 1_8
```

```
Estado: 12
           x = 128_8
           y = 1_8
           z = 112_8
           pc = 0_8
Estado: 13
           x = 128_8
           y = 1_8
           z = 112_8
           pc = 2_8
Estado: 14
           x = 128_8
           y = 0_8
           z = 240_8
           pc = 0_8
```

Invariant Function

Defines an invariant condition based on the state and constants a and b.

`Parameters:`

- `state:` The current state containing variables.
- `a:` The first constant for the invariant.
- `b:` The second constant for the invariant.

`Returns:` A boolean expressiong representing the invariant condition.

BMC_Always Function

Performs bounded model checking to verify ans invariant holds for n transitions.

`Parameters:`

- `vars:` List of variable names.
- `init:` The initial condition function.
- `trans:` The transition function.
- `inv:` The invariant function.
- `n:` The number of transitions to check.

`Returns:` Prints wether the invariant holds or is violated.

In [4]:
```
# Alinea B)

# x*y+z = a*b
def invariant(state, a, b):

    a_val = BV(a, n)
    b_val = BV(b, n)

    return Equals(
        BVAdd(
            BVMul(
```

```
                state['x'],
                state['y'],
            ),
            state['z']
        ),
        BVMul(a_val, b_val)
    )


def bmc_always(vars, init_ab, trans, inv, steps):
    with Solver(name="z3") as s:

        X = [genState(vars, 'X', i) for i in range(steps + 1)]

        a = 10
        b = 20

        s.add_assertion(init_ab(X[0], a, b))

        for k in range(steps):

            s.add_assertion(trans(X[k], X[k + 1]))

            s.push()

            s.add_assertion(Not(inv(X[k + 1], a, b)))

            if s.solve():
                print(f"> Invariant does not hold for {k+1} first states. Counte

                for i,ss in enumerate(X[:k+1]):
                    print(f"> State {i}: x = {s.get_value(ss['x'])}, pc= {s.get_
                return

            s.pop()

        print(f"> Invariant holds for the first {steps} transitions.")


steps = 10
bmc_always(vars, init_ab, trans1, invariant, steps)
```

```
> Invariant holds for the first 10 transitions.
```

Init_AB2 Function

Initializes the state with specific values for x, y, z, and pc, enforcing a constraint on a and b.

`Parameters:`

- `state:` The state to initialize.
- `a:` The value to assign to x (must be less than 2^(n/2))
- `b:` The value to assign to y (must be less than 2^(n/2))

`Returns:` A boolean expression representing the initialization conditions.

`Raises:` ValueError if a or b is not less than 2^(n/2).

BMC_Always2 Function

Performs bounded model checking with additional constraints on the values of a and b.

`Parameters:`

- `vars:` List of variable names.
- `init:` The initial condition function.
- `trans:` The transition function.
- `inv:` The invariant function.
- `n:` The number of transitions to check.

`Returns:` Prints whether the invariant holds or is violated.

In [6]:
```
# Alinea C)

def init_ab2(state, a, b):

    assert not (a == 0 and b == 0)

    if not ( a < 2**(n/2) and b < 2**(n/2) ):
        raise ValueError("b must be less than 2^(n/2)", a, b)

    if a < b:
        a, b = b, a

    return And(
        Equals(state['x'], BV(a, n)),
        Equals(state['y'], BV(b, n)),
        state['x'] < 2**(n//2),
        state['y'] < 2**(n//2),
        Equals(state['z'], BVZero(n)),
        Equals(state['pc'], BVZero(n))
    )


def bmc_always2(vars, init_ab, trans, inv, steps):
    with Solver(name="z3") as s:
        # Generate states for each step
        X = [genState(vars, 'X', i) for i in range(steps + 1)]

        a = 10
        b = 15

        s.add_assertion(init_ab(X[0], a, b))

        for k in range(steps):

            s.add_assertion(trans(X[k], X[k + 1]))

            s.push()

            s.add_assertion(Not(inv(X[k + 1], a, b)))
```

```python
            if s.solve():
                print(f"> Invariant does not hold for {k+1} first states. Counte

                for i,ss in enumerate(X[:k+1]):
                    print(f"> State {i}: x = {s.get_value(ss['x'])}, pc= {s.get_
                return

            s.pop()

        print(f"> Invariant holds for the first {steps} transitions.")
        print(f"> The error state is never accessible")


steps = 10
bmc_always2(vars, init_ab2, trans1, invariant, steps)
```

```
> Invariant holds for the first 10 transitions.
> The error state is never accessible
```

Exemplos

Exemplo 1

N-bits: 8:

```
n = 8
2**(n/2) = 2**4 = 16
a < 16
b < 16
a = 10
b = 30
```

In [7]:
```python
# Alinea C) # Para dar erro pelo valor de b se maior que 2^(n/2)

def init_ab2(state, a, b):

    assert not (a == 0 and b == 0)

    if not ( a < 2**(n/2) and b < 2**(n/2) ):
        raise ValueError("b must be less than 2^(n/2)", a, b)

    if a < b:
        a, b = b, a

    return And(
        Equals(state['x'], BV(a, n)),
        Equals(state['y'], BV(b, n)),
        state['x'] < 2**(n//2),
        state['y'] < 2**(n//2),
        Equals(state['z'], BVZero(n)),
        Equals(state['pc'], BVZero(n))
    )
```

```python
def bmc_always2(vars, init_ab, trans, inv, steps):
    with Solver(name="z3") as s:
        X = [genState(vars, 'X', i) for i in range(steps + 1)]

        a = 10
        b = 30

        s.add_assertion(init_ab(X[0], a, b))

        for k in range(steps):

            s.add_assertion(trans(X[k], X[k + 1]))

            s.push()

            s.add_assertion(Not(inv(X[k + 1], a, b)))

            if s.solve():
                print(f"> Invariant does not hold for {k+1} first states. Counte

                for i,ss in enumerate(X[:k+1]):
                    print(f"> State {i}: x = {s.get_value(ss['x'])}, pc= {s.get_
                return

            s.pop()

        print(f"> Invariant holds for the first {steps} transitions.")
        print(f"> The error state is never accessible")


steps = 10
bmc_always2(vars, init_ab2, trans1, invariant, steps)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[7], line 54
     50         print(f"> The error state is never accessible")
     53 steps = 10
---> 54 bmc_always2(vars, init_ab2, trans1, invariant, steps)

Cell In[7], line 30, in bmc_always2(vars, init_ab, trans, inv, steps)
     27 a = 10
     28 b = 30
---> 30 s.add_assertion(init_ab(X[0], a, b))
     32 for k in range(steps):
     34     s.add_assertion(trans(X[k], X[k + 1]))

Cell In[7], line 8, in init_ab2(state, a, b)
      5 assert not (a == 0 and b == 0)
      7 if not ( a < 2**(n/2) and b < 2**(n/2) ):
----> 8     raise ValueError("b must be less than 2^(n/2)", a, b)
     10 if a < b:
     11     a, b = b, a

ValueError: ('b must be less than 2^(n/2)', 10, 30)
```