TP2_Ex2

November 10, 2025

0.1 ## Lógica Computacional: 25/26

0.2 TP2 - Ex2

Grupo 05

- Vasco Ferreira Leite (A108399)
- Gustavo da Silva Faria (A108575)
-

0.3 Afonso Henrique Cerqueira Leal (A108472)

0.4 Problema:

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.

0.4.1 Tarefa 1: Construção do FOTS

Construir um FOTS (Fair Object Transition System), usando BitVec's de tamanho n, que descreva o comportamento deste autómato. Para isso, identifique e codifique em **z3-solver** ou **pySMT**, as **variáveis** do modelo, o **estado inicial**,a **relação de transição** e o **estado de erro**

0.4.2 Tarefa 2: Verificação de Invariante (BMC)

Usando Bounded Model Checking (BMC), verifique nesse SFOTS se a propriedade abaixo é um **invariante** do seu comportamento:

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

0.4.3 Tarefa 3: Verificação de Segurança (BMC)

Sejam N, M parâmetros do problema. Usando BMC em N passos no FOTS acima e adicionando a condição $N \leq a, b \leq M$ ao estado inicial, verifique a **segurança do programa**;nomeadamente verifique que, com tal estado inicial, o estado de erro não é acessível.

0.4.4 Variáveis e Parâmetros:

Variáveis de Estado:

O estado do FOTS, s, é representado por um conjunto de variáveis simbólicas de Z3:

- `loc`: Um `z3.Int` que representa o nodo atual no CFA. Usamos inteiros para os locais (ex: `SKIP = 1`, `ERROR = 5`).
- `x`, `y`, `z`: `z3.BitVecs` de tamanho `n_bits` que representam as variáveis do programa.

Parâmetros: - `n_bits`: O tamanho dos `BitVecs` usados para as variáveis do programa.

- `a`, `b`: Variáveis simbólicas de Z3 que representam os valores de entrada.
- `k_steps_inv`: O número de passos (limite k) para a verificação do invariante.
- `n_steps_safety`: O número de passos (limite N) para a verificação de segurança.
- `n_bound`, `m_bound`: Limites numéricos (N e M) impostos sobre as entradas `a` e `b`.

0.5 Função: Definição dos Estados do CFA

Define constantes inteiras para representar os nodos do grafo.

[ ]:
```
import z3

SKIP = 1
LEFT = 2
RIGHT = 3
STOP = 4
ERROR = 5
```

0.6 Função: Predicado de Estado Inicial

Esta função define a fórmula lógica para o **estado inicial** `s` , denotada por `I(s )`. A mesma garante que, no início:

- O nodo é `SKIP`.
- As variáveis x, y, z são inicializadas com a, b e 0, respetivamente.
- A **pré-condição** `a  0` e `b  0` é satisfeita

[ ]:
```
def init_state(s, a, b):

    loc, x, y, z = s
    n_bits = x.size()

    return z3.And(
        loc == SKIP,
        x == a,
```

```
        y == b,
        z == z3.BitVecVal(0, n_bits),
        z3.UGE(a, z3.BitVecVal(0, n_bits)),
        z3.UGE(b, z3.BitVecVal(0, n_bits))
    )
```

0.7 Função: Predicado de Estado de Erro

Define a fórmula que identifica um **estado de erro**, `Error(s)`.
A fórmula é **verdadeira** se, e apenas se, o local do estado `s` for `ERROR`.

[ ]:
```
def error_state(s):
    loc, _, _, _ = s
    return loc == ERROR
```

0.8 Função: Relação de Transição

Esta é a **função central do FOTS**, e codifica todas as **transições do CFA**.

Inclui:

- Cálculo das **condições de overflow** para aritmética *unsigned*.

- Implementação das **3 transições** de `SKIP`.
- Implementação das transições de `LEFT` e `RIGHT`.
- Garantia de que os estados terminais `STOP` e `ERROR` fazem um **loop** para si mesmos.

[ ]:
```
def transition_relation(s, s_p, a, b, n_bits):
    loc, x, y, z = s
    loc_p, x_p, y_p, z_p = s_p

    msb_mask = z3.BitVecVal(1, n_bits) << (n_bits - 1)
    overflow_left = (x & msb_mask) != 0
    overflow_right = z3.ULT(z + x, z)

    trans_skip_stop = z3.And(
        loc == SKIP, y == 0,
        loc_p == STOP, x_p == x, y_p == y, z_p == z
    )

    trans_skip_left = z3.And(
        loc == SKIP, y != 0, (y & 1) == 0,
        loc_p == LEFT, x_p == x, y_p == y, z_p == z
    )

    trans_skip_right = z3.And(
        loc == SKIP, y != 0, (y & 1) != 0,
        loc_p == RIGHT, x_p == x, y_p == y, z_p == z
    )
```

```
    trans_left_skip = z3.And(
        loc == LEFT, z3.Not(overflow_left),
        loc_p == SKIP, x_p == (x << 1), y_p == z3.LShR(y, 1), z_p == z
    )

    trans_left_error = z3.And(
        loc == LEFT, overflow_left,
        loc_p == ERROR, x_p == (x << 1), y_p == z3.LShR(y, 1), z_p == z
    )

    trans_right_skip = z3.And(
        loc == RIGHT, z3.Not(overflow_right),
        loc_p == SKIP, x_p == x, y_p == (y - 1), z_p == (z + x)
    )

    trans_right_error = z3.And(
        loc == RIGHT, overflow_right,
        loc_p == ERROR, x_p == x, y_p == (y - 1), z_p == (z + x)
    )

    trans_stop_stop = z3.And(
        loc == STOP,
        loc_p == STOP, x_p == x, y_p == y, z_p == z
    )

    trans_error_error = z3.And(
        loc == ERROR,
        loc_p == ERROR, x_p == x, y_p == y, z_p == z
    )

    return z3.Or(
        trans_skip_stop,
        trans_skip_left,
        trans_skip_right,
        trans_left_skip,
        trans_left_error,
        trans_right_skip,
        trans_right_error,
        trans_stop_stop,
        trans_error_error
    )
```

0.9 Função: get_state_vecs (Auxiliar BMC)

Função **auxiliar** para o **Bounded Model Checking (BMC)**.
Cria a sequência de **k + 1 vetores de estado simbólicos** (`s` até `s`).

Cada estado `s` é uma lista de novas variáveis **Z3** (`loc` , `x` , `y` , `z`).

[ ]:
```python
def get_state_vecs(k, n_bits):

    states = []
    for i in range(k + 1):
        s_i = [
            z3.Int(f'loc_{i}'),
            z3.BitVec(f'x_{i}', n_bits),
            z3.BitVec(f'y_{i}', n_bits),
            z3.BitVec(f'z_{i}', n_bits)
        ]
        states.append(s_i)
    return states
```

0.10 Função: Verificação de Invariante

Implementa a solução da **Tarefa 2**.
Esta função usa **Bounded Model Checking (BMC)** para verificar se o invariante $P = (x \cdot y + z = a \cdot b)$ é válido para `k_steps`.

- Constrói a fórmula BMC: $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$
- Adiciona a **negação** da propriedade $\bigvee_{i=0}^{k} \neg P(s_i)$,
  onde $P(s_i) \equiv (loc_i \neq ERROR) \Rightarrow (x_i \cdot y_i + z_i = a \cdot b)$
- Pede ao Z3 para encontrar um modelo.
- Se for `unsat`, não há contra-exemplo, e o invariante é válido até k passos.

[ ]:
```python
def check_invariant(k_steps, n_bits):

    print(f"--- A verificar Invariante para {k_steps} passos ({n_bits} bits)␣
↪---")

    a = z3.BitVec('a', n_bits)
    b = z3.BitVec('b', n_bits)

    states = get_state_vecs(k_steps, n_bits)

    solver = z3.Solver()

    solver.add(init_state(states[0], a, b))

    for i in range(k_steps):
        solver.add(transition_relation(states[i], states[i+1], a, b, n_bits))

    def P(s):
        loc, x, y, z = s
```

```
        prop = (x * y + z) == (a * b)
        return z3.Implies(loc != ERROR, prop)

    counter_example = z3.Or([z3.Not(P(s)) for s in states])
    solver.add(counter_example)

    result = solver.check()

    if result == z3.unsat:
        print(f"Resultado: Invariante (x*y + z == a*b) válido até {k_steps}␣
↪passos.")
    else:
        print(f"Resultado: Invariante VIOLADO em {k_steps} passos.")
        print("Contra-exemplo (modelo):")
        print(solver.model())
    print("-" * 60)
```

0.11 Função: Verificação de Segurança

Utilisa **BMC** para verificar se é possivel alcançar um **estado de erro** em `n_steps`, dadas as restrições de entrada $N \leq a, b \leq M$

- Contrói a fórmula **BMC**, tal como no invariante.
- Adiciona as restrições de entrada $I(s_0) \wedge (N \leq a \leq M) \wedge (N \leq b \leq M)$
- Adiciona a **propriadade de erro** $\wedge \bigvee_{i=0}^{n} Error(s_i)$
- Pede ao z3 para encontrar um modelo que leve a erro. Se um for encontrado o programa é inseguro.

[ ]:
```
def check_safety(n_steps, n_bits, n_bound, m_bound):

    print(f"--- A Verificar Segurança para {n_steps} passos ({n_bits} bits)␣
↪---")
    print(f"             Restrições: {n_bound} <= a, b <= {m_bound}")

    a = z3.BitVec('a', n_bits)
    b = z3.BitVec('b', n_bits)

    states = get_state_vecs(n_steps, n_bits)

    solver = z3.Solver()

    solver.add(init_state(states[0], a, b))

    a_val_n = z3.BitVecVal(n_bound, n_bits)
    a_val_m = z3.BitVecVal(m_bound, n_bits)
    b_val_n = z3.BitVecVal(n_bound, n_bits)
```

```
    b_val_m = z3.BitVecVal(m_bound, n_bits)

    solver.add(z3.And(
        z3.UGE(a, a_val_n), z3.ULE(a, a_val_m),
        z3.UGE(b, b_val_n), z3.ULE(b, b_val_m)
    ))

    for i in range(n_steps):
        solver.add(transition_relation(states[i], states[i+1], a, b, n_bits))

    error_reached = z3.Or([error_state(s) for s in states])
    solver.add(error_reached)

    result = solver.check()

    if result == z3.unsat:
        print(f"Resultado: O estado de ERRO NÃO é alcançável em {n_steps}␣
↪passos")
        print(f"           com as restrições {n_bound} <= a, b <= {m_bound}.")
    else:
        print(f"Resultado: O estado de ERRO É ALCANÇÁVEL em {n_steps} passos!")
        print("Modelo que leva ao erro:")
        m = solver.model()
        print(m)
        print("\nTraço para o erro:")
        for i in range(n_steps + 1):
            s_i = states[i]
            loc_val = m.eval(s_i[0])
            x_val = m.eval(s_i[1])
            y_val = m.eval(s_i[2])
            z_val = m.eval(s_i[3])
            print(f"  Passo {i}: loc={loc_val}, x={x_val}, y={y_val},␣
↪z={z_val}")
            if loc_val.as_long() == ERROR:
                break

    print("-" * 60)
```

0.12 Execução

Define os parâmetros para as verificações e chama as funções `check_invariant` e `check_safety`.

[ ]:
```
if __name__ == "__main__":
    N_BITS = 8
    K_STEPS_INV = 15
    N_STEPS_SAFETY = 15
    N_BOUND = 10
```

```
    M_BOUND = 20

    check_invariant(k_steps=K_STEPS_INV, n_bits=N_BITS)

    check_safety(
        n_steps=N_STEPS_SAFETY,
        n_bits=N_BITS,
        n_bound=N_BOUND,
        m_bound=M_BOUND
    )

    print("\n--- Teste de Segurança 2: Forçando um Overflow (a=128, b=3) ---")
    check_safety(
        n_steps=5,
        n_bits=8,
        n_bound=128,
        m_bound=128
    )
```