TP3 - Implementação Simplificada do Algoritmo "model checking"

Dezembro, 2022

Bruno Miguel Ferreira Fernandes - a95972

Hugo Filipe de Sá Rocha - a96463

Definir valores de input do problema

In [1]:
```
from pysmt.shortcuts import *
from pysmt.typing import *

import itertools

global a
global b
global n
a = 2
b = 3
n = 4
#8 passos

#caso overflow:
# a = 4
# b = 6
# n = 4
#7 passos
```

Para modelar este programa como um SFOTS teremos o conjunto X de variáveis do estado dado pela lista `['pc','x','y','z']`, e definimos a função `genState` que recebe a lista com o nome das variáveis do estado, uma etiqueta e um inteiro, e cria a i-ésima cópia das variáveis do estado para essa etiqueta. As variáveis lógicas começam sempre com o nome de base das variáveis dos estado, seguido do separador `!`.

In [2]:
```
def genState(vars,s,i):
    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+s+str(i), BVType(n))
    return state
```

Função `init1` dado um estado do programa (um dicionário de variáveis), devolve um predicado do pySMT que testa se esse estado é um possível estado inicial do programa.

In [3]:
```
def init1(state):
    return And(Equals(state['pc'],BV(0,n)), Equals(state['x'], BV(a,n)),
               Equals(state['y'],BV(b,n)), Equals(state['z'],BV(0,n)))
```

A função `error1` dado um estado do programa, devolve um predicado do pySMT que testa se esse estado é um possível estado de erro do programa.

In [4]:
```
def error1(state):
    return Or(Equals(state['pc'],BV(5,n)), Equals(state['pc'],BV(6,n)))
```

A função `trans1` que, dados dois estados do programa, devolve um predicado do pySMT que testa se é possível transitar do primeiro para o segundo estado.

In [5]:
```python
def trans1(curr, prox):

    t0 = And(Equals(curr['pc'],BV(0,n)),
             Equals(prox['pc'],BV(1,n)),
             Equals(prox['x'],curr['x']),
             Equals(prox['y'],curr['y']),
             Equals(prox['z'],curr['z']))

    t1 = And(Equals(curr['pc'],BV(1,n)),
             NotEquals(curr['y'],BV(0,n)),
             Equals(BVURem(curr['y'], BV(2,n)), BV(0,n)),
             Equals(prox['pc'],BV(2,n)),
             Equals(prox['x'],curr['x']),
             Equals(prox['y'],curr['y']),
             Equals(prox['z'],curr['z']))

    t2 = And(Equals(curr['pc'],BV(1,n)),
             NotEquals(curr['y'],BV(0,n)),
             NotEquals(BVURem(curr['y'], BV(2,n)), BV(0,n)),
             Equals(prox['pc'],BV(3,n)),
             Equals(prox['x'],curr['x']),
             Equals(prox['y'],curr['y']),
             Equals(prox['z'],curr['z']))

    #atinge estado final 4
    t3 = And(Equals(curr['pc'],BV(1,n)),
             Equals(curr['y'] ,BV(0,n)),
             Equals(prox['pc'],BV(4,n)),
             Equals(prox['x'],curr['x']),
             Equals(prox['y'],curr['y']),
             Equals(prox['z'],curr['z']))


    #OVERFLOW (atinge estado final 5)
    t4 = And(Equals(curr['pc'],BV(2,n)),
             BVUGT(curr['x'], prox['x']),
             Equals(prox['pc'],BV(5,n)),
             Equals(prox['x'], BVMul(curr['x'], BV(2,n))),
             Equals(prox['y'], BVUDiv(curr['y'], BV(2,n))),
             Equals(prox['z'], curr['z'])
            )

    t5 = And(Equals(curr['pc'],BV(2,n)),
             BVULE(curr['x'], prox['x']),
             Equals(prox['pc'],BV(1,n)),
             Equals(prox['x'], BVMul(curr['x'], BV(2,n))),
             Equals(prox['y'], BVUDiv(curr['y'], BV(2,n))),
             Equals(prox['z'], curr['z'])
            )

    t6 = And(Equals(curr['pc'],BV(3,n)),
             BVULE(curr['z'], prox['z']),
             Equals(prox['pc'],BV(1,n)),
             Equals(prox['x'],curr['x']),
             Equals(prox['y'],BVSub(curr['y'], BV(1,n))),
             Equals(prox['z'],BVAdd(curr['z'],curr['x']))
            )

    #OVERFLOW DIREITA (atinge estado final 6)
    t7 = And(Equals(curr['pc'],BV(3,n)),
             BVUGT(curr['z'], prox['z']),
```

```
                Equals(prox['pc'],BV(6,n)),
                Equals(prox['x'],curr['x']),
                Equals(prox['y'],BVSub(curr['y'], BV(1,n))),
                Equals(prox['z'],BVAdd(curr['z'],curr['x'])))

    return Or(t0,t1,t2,t3,t4,t5,t6,t7)
```

Seguindo esta notação, a fórmula $\mathsf{I} \wedge \mathsf{T}^n$ denota um traço finito com n transições em Σ, $\mathsf{X}_0, \cdots, \mathsf{X}_n$, que descrevem estados acessíveis com n ou menos transições. Inspirada nesta notação, a seguinte função `genTrace` gera um possível traço de execução com n transições.

In [6]:

```
def genTrace(vars,init,trans,error,n):
    with Solver(name="z3") as s:

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

        if s.solve([I,And(Tks)]):      # testa se I /\ T^n  é satisfazível
            for i in range(n+1):
                print("Passo:",i)
                for v in X[i]:
                    print("          ",v,'=',s.get_value(X[i][v]))
        else:
            print('unsat')

genTrace(['pc','x','y','z'],init1,trans1,error1,8)
```

```
Passo: 0
           pc = 0_4
           x = 2_4
           y = 3_4
           z = 0_4
Passo: 1
           pc = 1_4
           x = 2_4
           y = 3_4
           z = 0_4
Passo: 2
           pc = 3_4
           x = 2_4
           y = 3_4
           z = 0_4
Passo: 3
           pc = 1_4
           x = 2_4
           y = 2_4
           z = 2_4
Passo: 4
           pc = 2_4
           x = 2_4
           y = 2_4
           z = 2_4
Passo: 5
           pc = 1_4
           x = 4_4
           y = 1_4
           z = 2_4
Passo: 6
           pc = 3_4
           x = 4_4
           y = 1_4
           z = 2_4
Passo: 7
```

```
        pc = 1_4
        x = 4_4
        y = 0_4
        z = 6_4
Passo: 8
        pc = 4_4
        x = 4_4
        y = 0_4
        z = 6_4
```

Definimos uma função de ordem superior `invert` que recebe a função python que codifica a relação de transição e devolve a relação e transição inversa. Para auxiliar na implementação deste algoritmo, definimos ainda duas funções. A função `rename` renomeia uma fórmula (sobre um estado) de acordo com um dado estado. A função `same` testa se dois estados são iguais.

In [7]:
```
def baseName(s):
    return ''.join(list(itertools.takewhile(lambda x: x!='!', s)))

def rename(form,state):
    vs = get_free_variables(form)
    pairs = [ (x,state[baseName(x.symbol_name())]) for x in vs ]
    return form.substitute(dict(pairs))

def same(state1,state2):
    return And( [Equals(state1[x],state2[x]) for x in state1])

def invert(trans):
    return (lambda c, p: trans(p,c))
```

O algoritmo de "model-checking"

O algoritmo de “model-checking” manipula as fórmulas $\mathsf{R}_n \equiv \mathsf{I} \wedge \mathsf{T}^n$ e $\mathsf{U}_m \equiv \mathsf{E} \wedge \mathsf{B}^m$ fazendo crescer os índices n, m de acordo com as seguintes regras

---

1. Inicia-se $n = 0$, $\mathsf{R}_0 = \mathsf{I}$ e $\mathsf{U}_0 = \mathsf{E}$.

1. No estado (n, m) tem-se a certeza que em todos os estados anteriores não foi detectada nenhuma justificação para a insegurança do SFOTS. Se $\mathsf{V}_{n,m} \equiv \mathsf{R}_n \wedge (X_n = Y_m) \wedge \mathsf{U}_m$ é satisfazível o sistema é inseguro e o algoritmo termina com a mensagem **unsafe**.

1. Se $\mathsf{V}_{n,m} \equiv \mathsf{R}_n \wedge (X_n = Y_m) \wedge \mathsf{U}_m$ for insatisfazível calcula-se C como o interpolante do par $(\mathsf{R}_n \wedge (X_n = Y_m)\,,\, \mathsf{U}_m)$. Neste caso verificam-se as tautologias $\mathsf{R}_n \to C(X_n)$ e $\mathsf{U}_m \to \neg C(Y_m)$.

1. Testa-se a condição $\mathsf{SAT}(C \wedge \mathsf{T} \wedge \neg C') = \emptyset$ para verificar se C é um invariante de T; se for invariante então, pelo resultado anterior, sabe-se que $\mathsf{V}_{n',m'}$ é insatisfazível para todo $n' \geq n$ e $m' \geq n$. O algoritmo termina com a mensagem **safe**.

1. Se C não for invariante de T procura-se encontrar um majorante $S \supseteq C$ que verifique as condições do resultado referido: seja um invariante de T disjunto de U_m.

1. Se for possível encontrar tal majorante S então o algoritmo termina com a mensagem **safe**. Se não for possível encontrar o majorante pelo menos um dos índices n, m é incrementado, os valores das fórmulas R_n, U_m são actualizados e repete-se o processo a partir do passo 2.

Definimos uma função de ordem superior `model-checking` que dada a lista de nomes das variáveis do sistema, um predicado que testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, um predicado que testa se um estado é de erro, e dois números positivos N e M que são os limites máximos para os indices n e m. Implementando o algoritmo iterativo que manipula as fórmulas $\mathsf{R}_n \equiv \mathsf{I} \wedge \mathsf{T}^n$ e $\mathsf{U}_m \equiv \mathsf{E} \wedge \mathsf{B}^m$ fazendo crescer os índices n,m de acordo com as regras acima apresentadas.

In [8]:
```
def model_checking(vars,init,trans,error,N,M):
    with Solver(name="z3") as s:


        # Criar todos os estados que poderão vir a ser necessários.
        X = [genState(vars,'X',i) for i in range(N+1)]
        Y = [genState(vars,'Y',i) for i in range(M+1)]

        n1 = 1
        m1 = 1

        while(n1 <= N and m1 <= M):
            print(f'(n,m) = ({n1},{m1})')
            Tn = And([trans(X[i], X[i+1]) for i in range(n1)])
            I = init(X[0])
            Rn = And(I, Tn)

            Bm = And([invert(trans)(Y[i], Y[i+1]) for i in range(m1)])
            E = error(Y[0])
            Um = And(E, Bm)

            Vnm = And(Rn, same(X[n1], Y[m1]), Um)

            if s.solve([Vnm]):
                print("Unsafe")
                return
            else:                        # Vnm é instatisfazível
                C = binary_interpolant(And(Rn, same(X[n1], Y[m1])), Um)
                if C is None:
                    print("Interpolant None")
                C0 = rename(C, X[0])
                C1 = rename(C, X[1])
                T = trans(X[0], X[1])
                if not s.solve([C0, T, Not(C1)]):   # C é invariante de T
                    print("Safe")
                    return
                else:

                    S = rename(C, X[n1])
                    while True:
                        A = And(S, trans(X[n1], Y[m1]))
                        if s.solve([A,Um]):
                            print("Não é possível encontrar um majorante")
                            break
                        else:
                            Cnew = binary_interpolant(A, Um)
                            Cn = rename(Cnew, X[n1])
                            if s.solve([Cn, Not(S)]):# Se Cn -> S não é tautologia
                                S = Or(S, Cn)
                            else:           # S foi encontrado
                                print("Safe")
                                return
                    print("Quer incrementar o n ou o m?\n")
                    inp = input()
                    if inp =='m':
                        m1+=1
                    else:
                        n1+=1
```

```
model_checking(['x', 'y', 'z','pc'], init1, trans1, error1, 50, 50)
```

```
(n,m) = (1,1)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?

n
(n,m) = (2,1)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?

m
(n,m) = (2,2)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?


(n,m) = (3,2)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?


(n,m) = (4,2)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?


(n,m) = (5,2)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?


(n,m) = (6,2)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?


(n,m) = (7,2)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?


(n,m) = (8,2)
Não é possível encontrar um majorante
Quer incrementar o n ou o m?


(n,m) = (9,2)
Safe
```