TP3-Exercicio1

December 13, 2022

1 TP3

1.1 Grupo 15

Carlos Eduardo Da Silva Machado A96936

Gonçalo Manuel Maia de Sousa A97485

1.2 Problema 1

1.2.1 Descrição do Problema

1. Pretende-se construir uma implementação simplificada do algoritmo "model checking" orientado aos interpolantes. Para tal seguimos a estrutura apresentada nos apontamentos onde, no passo (n, m) na impossibilidade de encontrar um interpolante invariante se dá ao utilizador a possibilidade de incrementar um dos índices n e m à sua escolha. Pretende-se aplicar este algoritmo ao problema da da multiplicação de inteiros positivos em `BitVec` (apresentado no TP2).

1.2.2 Abordagem do Problema

Numa abordagem inicial decidimos aplicar o algoritmo com a solução apresentado no TP2 sem qualquer alteração. No entanto, rapidamente descobrimos que, sendo a variavel *pc* do tipo de dados `Int` o *solver* não é capaz de encontrar um interpolante. Como modo de resolver este problema decidimos alterar ligeiramente a solução original de forma a que *pc* seja agora de tipo `BitVec` mantendo a mesma lógica da solução original. Além disso removemos todas as transições para estados de erro, pois estas não são necessárias.

1.3 Código Python

Esta secção de codigo serve para importar as bibliotecas necessárias para a realização do trabalho.

[3]:
```
from pysmt.shortcuts import *
import itertools
from pysmt.typing import INT
import random as rn
```

Aqui estão apresentadas as funções `genState` e `init_fixed`, em tudo análogas às funções `declare` e `init` do TP2. Como forma de poupar espaço *pc* é declarado com tamanho 2 visto que é suficiente para representar todos os valores que *pc* possa tomar na execução do algoritmo. Além disso, implementamos duas versões adicionais da função `init`: Uma delas, a função `init_unbounded` não

limita o valor inicial de x e y de modo a que o solver teste se é possivel alcançar um estado de erro a partir de x e y arbitrários. (O resultado será sempre unsafe pois para qualquer tamanho é possivel encontrar dois valoras cujo produto resulte em "overflow") A outra, a função `init_bounded` limita o valor de x e y superiormente de modo a que o solver teste se é possivel alcançar um estado de erro a partir x, y menores ou iguais a um certo limite.

[4]:
```
def genState(vars,s,i,n):
    state = {}
    for v in vars:
        if v == 'pc':
            state[v] = Symbol(v+'!'+s+str(i),types.BVType(2))
        else:
            state[v] = Symbol(v+'!'+s+str(i),types.BVType(n+1))
    return state

def init_fixed(state,a,b,n):
    if b > a:
        a,b = b,a

    tPc = Equals(state['pc'],BVZero(2)) # Program counter a zero
    tZ  = Equals(state['z'],BVZero(n+1)) # Z a zero
    tX  = Equals(state['x'], BV(a,n+1)) # x inicilizado com valor de a
    tY  = Equals(state['y'], BV(b,n+1)) # y inicilizado com valor de b
    return And(tPc,tX,tY,tZ)

def init_unbounded(state,a,b,n):
    tPc = Equals(state['pc'],BVZero(2)) # Program counter a zero
    tZ  = Equals(state['z'],BVZero(n+1)) # Z a zero
    return And(tPc,tZ)

def init_bounded(state,upper_a,upper_b,n):

    tPc = Equals(state['pc'],BVZero(2)) # Program counter a zero
    tZ  = Equals(state['z'],BVZero(n+1)) # Z a zero
    tX  = BVULE(state['x'], BV(upper_a,n+1)) # x inicilizado com valor de a
    tY  = BVULE(state['y'], BV(upper_b,n+1)) # y inicilizado com valor de b
    return And(tPc,tZ,tX,tY)
```

Aqui é apresentada a função de transição(`trans`), análoga à apresentada no TP2 a menos da remoção de todas as transições para estados de erro e da mudança do tipo de dados do *pc* para `BitVec`. Além disso apresentamos uma função de transição extra, `trans_more_states` que procura seguir de forma mais rigida o modelo dado no enunciado do TP2. Nota: após alguns testes notamos que esta nova versão é mais lenta do que a versão anterior.

Função `trans` apresentada no TP2.

[5]:
```
def BVFirst(n):
    return BV(2**(n-1),n)
```

```
def tEven(curr,prox,n):
    tPcZero = Equals(curr['pc'],BVZero(2))
    tYLast  = Equals(BVAnd(curr['y'],BVOne(n+1)),BVZero(n+1))#ultimo bit = 0
    tYGt    = BVUGT(curr['y'],BVZero(n+1))#y > 0
    tX      = Equals(prox['x'], BVLShl(curr['x'],BVOne(n+1)))#2*x
    tY      = Equals(prox['y'], BVLShr(curr['y'],BVOne(n+1)))#y/2
    tZ      = Equals(prox['z'],curr['z'])#z
    tPc     = Equals(prox['pc'],BVZero(2))
    return And(tPcZero,tYLast,tYGt,tX,tY,tZ,tPc)

def tOdd(curr,prox,n):
    tPcZero = Equals(curr['pc'],BVZero(2))
    tYLast  = Equals(BVAnd(curr['y'],BVOne(n+1)),BVOne(n+1))
    tX      = Equals(prox['x'], curr['x'])
    tY      = Equals(prox['y'],BVSub(curr['y'],BVOne(n+1)))
    tZ      = Equals(prox['z'],BVAdd(curr['x'],curr['z']))
    tPc     = Equals(prox['pc'],BVZero(2))
    return And(tPcZero,tYLast,tX,tY,tZ,tPc)

def tStop(curr,prox,n):
    tPcZero = Equals(curr['pc'],BVZero(2))
    tYZero  = Equals(curr['y'],BVZero(n+1))#y=0
    tX      = Equals(prox['x'],curr['x'])
    tY      = Equals(prox['y'],curr['y'])
    tZ      = Equals(prox['z'],curr['z'])
    tPc     = Equals(prox['pc'],BVOne(2))
    return And(tYZero,tPcZero,tX,tY,tZ,tPc)

def tEnd(curr,prox,n):
    tPcOne = Equals(curr['pc'],BVOne(2))
    tX     = Equals(prox['x'],curr['x'])
    tY     = Equals(prox['y'],curr['y'])
    tZ     = Equals(prox['z'],curr['z'])
    tPc    = Equals(prox['pc'],BVOne(2))
    return And(tPcOne,tX,tY,tZ,tPc)


def trans(curr,prox,n):
    tToStop = tStop(curr,prox,n)
    tToEven = tEven(curr,prox,n)
    tToOdd  = tOdd(curr,prox,n)
    tToEnd  = tEnd(curr,prox,n)
    return Or(tToStop,tToEven,tToOdd,tToEnd)
```

Versão nova da função.

[6]:
```
def initToEnven(curr,prox,n):
    tPcZero = Equals(curr['pc'],BVZero(2))
    tYLast = Equals(BVAnd(curr['y'],BVOne(n+1)),BVZero(n+1))#ultimo bit = 0
    tYGt = BVUGT(curr['y'],BVZero(n+1))#y > 0
    tX = Equals(prox['x'], curr['x'])
    tY = Equals(prox['y'], curr['y'])#y/2
    tZ = Equals(prox['z'],curr['z'])#z
    tPc = Equals(prox['pc'],BV(2,2))
    return And(tPcZero,tYLast,tYGt,tX,tY,tZ,tPc)


def initToOdd(curr,prox,n):
    tPcZero = Equals(curr['pc'],BVZero(2))
    tYLast = Equals(BVAnd(curr['y'],BVOne(n+1)),BVOne(n+1))#ultimo bit = 1
    tX = Equals(prox['x'], curr['x'])
    tY = Equals(prox['y'],curr['y'])
    tZ = Equals(prox['z'],curr['z'])
    tPc = Equals(prox['pc'],BVOne(2))
    return And(tPcZero,tYLast,tX,tY,tZ,tPc)

def OddToInit(curr,prox,n):
    cpc = Equals(curr['pc'], BVOne(2))
    cxp = Equals(curr['x'],prox['x'])
    cyp = Equals(prox['y'],BVSub(curr['y'],BVOne(n+1)))
    czp = Equals(prox['z'],BVAdd(curr['x'],curr['z']))
    ppc = Equals(prox['pc'], BVZero(2))
    return And(cpc,cxp,cyp,czp,ppc)

def EvenToInit(curr,prox,n):
    cpc = Equals(curr['pc'], BV(2,2))
    cxp = Equals(prox['x'],BVLShl(curr['x'],BVOne(n+1)))
    cyp = Equals(prox['y'],BVLShr(curr['y'],BVOne(n+1)))
    czp = Equals(curr['z'],prox['z'])
    ppc = Equals(prox['pc'], BVZero(2))
    return And(cpc,cxp,cyp,czp,ppc)

def initToStop(curr,prox,n):
    tPcZero = Equals(curr['pc'],BVZero(2))
    tYZero = Equals(curr['y'],BVZero(n+1))#y=0
    tZFirst = Equals(BVAnd(curr['z'],BVFirst(n+1)),BVZero(n+1))#primriro bit de␣
 ↪z = 0
    tX = Equals(prox['x'],curr['x'])
    tY = Equals(prox['y'],curr['y'])
    tZ = Equals(prox['z'],curr['z'])
    tPc = Equals(prox['pc'],BV(3,2))
    return And(tYZero,tZFirst,tPcZero,tX,tY,tZ,tPc)
```

```
def stopToStop(curr,prox,n):
    tPcOne = Equals(curr['pc'],BV(3,2))
    tX = Equals(prox['x'],curr['x'])
    tY = Equals(prox['y'],curr['y'])
    tZ = Equals(prox['z'],curr['z'])
    tPc = Equals(prox['pc'],BV(3,2))
    return And(tPcOne,tX,tY,tZ,tPc)

def trans_more_states(curr,prox,n):
    ite = initToEnven(curr,prox,n)
    ito = initToOdd(curr,prox,n)
    oti = OddToInit(curr,prox,n)
    eti = EvenToInit(curr,prox,n)
    its = initToStop(curr,prox,n)
    sts = stopToStop(curr,prox,n)
    return Or(ite,ito,oti,eti,its,sts)
```

Aqui apresentamos a condição de erro na forma de uma função `error()` que, dado um estado e o tamanho máximo de *bits* para uma variável, devolve uma condição de erro para esse estado. Para tal apenas é necessário verificar se o bit mais á esquerda do `BitVec` declarado é 1.

[7]:
```
def error(state,n):
    tYFirst = Equals(BVAnd(state['y'],BVFirst(n+1)),BVFirst(n+1))
    tXFirst = Equals(BVAnd(state['x'],BVFirst(n+1)),BVFirst(n+1))
    tZFirst = Equals(BVAnd(state['z'],BVFirst(n+1)),BVFirst(n+1))
    return Or(tYFirst,tXFirst,tZFirst)
```

Função que gera um traço de tamanho n. Esta função serve apenas para fins ilustrativos.

[8]:
```
def genTrace_fixed(vars,init,trans,error,n,tam,a,b):
    with Solver(name="z3") as s:

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

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

Aqui apresentamos algumas funções auxiliares utilizadas na função de "model checking".

[9]:
```
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 ] # Descobrir os␣
↪pares # symbol_name dá o nome aka string da var
    return form.substitute(dict(pairs)) # recebe um dicionário e substitui as␣
↪chaves do dicionário pelo o que está no value

def same(state1,state2): # ver se as duas vars têm o mesmo valor
    return And([Equals(state1[x],state2[x]) for x in state1])

def invert(trans,n):
    return (lambda u, v: trans(v,u,n))
```

1.3.1 Função "Model checking".

Nesta secção do relatório apresentamos duas implementações semelhantes de "model checking".

Uma delas percorre todos os pares (n, m) possíveis à procura de um interpolante invariante não necessita, por isso input do utilizador.

Outra testa inicialmente o par $(1, 1)$ e, caso não seja capaz de encontrar um interpolante invariante pede ao utilizador para incrementar uma das variáveis n ou m e um valor pelo qual incrementar.

Versão sem input.

[10]:
```
def model_checking(vars,init,trans,error,N,M,a,b,tamanho):
    with Solver(name="msat") as s:

        # Criar todos os estados que poderão vir a ser necessários.
        X = [genState(vars,'X',i,tamanho) for i in range(N+1)] # Com a função␣
↪genState, criar todos os estados que possam ser necessário, TODOS. # X SFOTS␣
↪original
        Y = [genState(vars,'Y',i,tamanho) for i in range(M+1)] # Y SFOTS␣
↪invertido

        # Estabelecer a ordem pela qual os pares (n,m) vão surgir. Por exemplo:
        order = sorted([(a,b) for a in range(1,N+1) for b in␣
↪range(1,M+1)],key=lambda tup:tup[0]+tup[1]) # Estabelecer ordem que criamos o␣
↪n e o m # ideia da stora: usar o sorted,
                                                                              ␣
↪                       # gerar todos os pares possíveis
                                                                              ␣
↪                       # e ter como critério de ordenação as soma dos␣
↪elementos dos pares

        for (n,m) in order: # Seguir o algoritmo
            # completar
```

```
            I = init(X[0],a,b,tamanho) # o X é uma lista de estados
            Tn = And([trans(X[i],X[i+1],tamanho) for i in range(n)])
            Rn = And(I,Tn) # estados acessíveis em n transições
            #print(X[0])
            E = error(Y[0],tamanho)
            Bm = And([invert(trans,tamanho)(Y[i],Y[i+1]) for i in range(m)])
            Um = And(E,Bm) # estados inseguros em m transições

            Vnm = And(Rn,same(X[n],Y[m]),Um) # temos de testar se dois estados␣
↪estão iguais e, portanto, usamos a função same dada acima

            #pprint(Vnm.serialize())

            if s.solve([Vnm]):
                print("unsafe")
                return

            # Se for insatisfazível, temos de criar o interpolante para n␣
↪fórmulas
            C = binary_interpolant(And(Rn,same(X[n],Y[m])), Um)
            if C is None:
                print("Interpolante None")
                continue

            C0 = rename(C,X[0]) # Rename do C com o estado envolvido, neste caso␣
↪o X[0]
            C1 = rename(C,X[1])
            # Trabalhamos com X[0] e X[1] porque T pode ser escrito como T =␣
↪(X0,X1)

            T = trans(X[0],X[1],tamanho)

            if not s.solve([C0,T,Not(C1)]):
                print(n,m)
                print("Safe")
                return
            else:
                    #### gerar o S (Parte que descreve o Majorante S)

                #Passo 1:
                S = rename(C,X[n])
                while True:
                    #Passo 2:
                    A = And(S,trans(X[n],Y[m],tamanho))
                    if s.solve([A,Um]):
                        #N = int(input("new N"))
                        #M = int(input("new M"))
```

```
                        print("Não foi possível encontrar o majorante.")
                        break
                    else:
                        CNew = binary_interpolant(A,Um) # as duas formulas têm␣
↪de ser inconsistentes para que faça sentido para usar binary_interpolant
                        Cn = rename(CNew,X[n])

                        if s.solve([Cn,Not(S)]):
                            S = Or(S,Cn)
                        else:
                            print(n,m)
                            print("Safe")
                            return
            #new_n = int(input("new n POGCHAMP"))
            #new_m = int(input("new m POGCHAMP"))
            #order.append((new_n,new_m))

        print("unknown")
```

Versão com input.

[11]:
```
def model_checking_input(vars,init,trans,error,N,M,a,b,tamanho):
    with Solver(name="msat") as s:

        # Criar todos os estados que poderão vir a ser necessários.
        X = [genState(vars,'X',i,tamanho) for i in range(N+1)] # Com a função␣
↪genState, criar todos os estados que possam ser necessário, TODOS. # X SFOTS␣
↪original
        Y = [genState(vars,'Y',i,tamanho) for i in range(M+1)] # Y SFOTS␣
↪invertido

        # Estabelecer a ordem pela qual os pares (n,m) vão surgir. Por exemplo:
        #order = sorted([(a,b) for a in range(1,N+1) for b in␣
↪range(1,M+1)],key=lambda tup:tup[0]+tup[1]) # Estabelecer ordem que criamos o␣
↪n e o m # ideia da stora: usar o sorted,
                                                                              ␣
↪                       # gerar todos os pares possíveis
                                                                              ␣
↪                       # e ter como critério de ordenação as soma dos␣
↪elementos dos pares
        (n,m) = 1,1

        while(n <= N and m <= M):
            # completar
            I = init(X[0],a,b,tamanho) # o X é uma lista de estados
            Tn = And([trans(X[i],X[i+1],tamanho) for i in range(n)])
            Rn = And(I,Tn) # estados acessíveis em n transições
```

```
            #print(X[0])
            E = error(Y[0],tamanho)
            Bm = And([invert(trans,tamanho)(Y[i],Y[i+1]) for i in range(m)])
            Um = And(E,Bm) # estados inseguros em m transições

            Vnm = And(Rn,same(X[n],Y[m]),Um) # temos de testar se dois estados␣
↪estão iguais e, portanto, usamos a função same dada acima

            #pprint(Vnm.serialize())

            if s.solve([Vnm]):
                print("unsafe")
                return

            # Se for insatisfazível, temos de criar o interpolante para n␣
↪fórmulas
            C = binary_interpolant(And(Rn,same(X[n],Y[m])), Um)
            if C is None:
                print("Interpolante None")
                continue

            C0 = rename(C,X[0]) # Rename do C com o estado envolvido, neste caso␣
↪o X[0]
            C1 = rename(C,X[1])
            # Trabalhamos com X[0] e X[1] porque T pode ser escrito como T =␣
↪(X0,X1)

            T = trans(X[0],X[1],tamanho)

            if not s.solve([C0,T,Not(C1)]):
                print("Safe")
                return
            else:
                    #### gerar o S (Parte que descreve o Majorante S)

                #Passo 1:
                S = rename(C,X[n])
                while True:
                    #Passo 2:
                    A = And(S,trans(X[n],Y[m],tamanho))
                    if s.solve([A,Um]):
                        #N = int(input("new N"))
                        #M = int(input("new M"))
                        print("Não foi possível encontrar o majorante.")

                        #n = int(input("new n"))
                        #m = int(input("new m"))
```

```
                    break
                else:
                    CNew = binary_interpolant(A,Um) # as duas formulas têm␣
↪de ser inconsistentes para que faça sentido para usar binary_interpolant
                    Cn = rename(CNew,X[n])

                    if s.solve([Cn,Not(S)]):
                        S = Or(S,Cn)
                    else:
                        print("Safe")
                        return

            var = input("Qual é a variável que pretende incrementar.").lower()
            new_val = int(input("Quantidade"))
            (n,m) = (n+new_val if var == 'n' else n, m+new_val if var == 'm'␣
↪else m)
            #new_m = int(input("new m POGCHAMP"))
            #order.append((new_n,new_m))

        print("unknown")
```

2 Exemplos

Nesta secção apresentamos alguns exemplos.

2.0.1 Exemplo 1

Nestes exemplos mostramos varias execuções da função com x e y fixos à partida.

[10]:
```
N = 100
M = 100
a = 10
b = 10
tamanho = 16
```

[11]:
```
model_checking(['pc','x','y','z'], init_fixed, trans, error, N, M, a, b, tamanho)
```

```
Não foi possível encontrar o majorante.
1 2
Safe
```

[12]:
```
model_checking_input(['pc','x','y','z'], init_fixed, trans, error, N, M, a, b,␣
↪tamanho)
```

```
Não foi possível encontrar o majorante.
Qual é a variável que pretende incrementar. m
```

```
Quantidade 2
Safe
```

[13]:
```
model_checking(['pc','x','y','z'], init_fixed, trans_more_states, error, N, M,␣
↪a, b, tamanho)
```

```
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
1 6
Safe
```

[14]:
```
model_checking_input(['pc','x','y','z'], init_fixed, trans_more_states, error,␣
↪N, M, a, b, tamanho)
```

```
Não foi possível encontrar o majorante.
Qual é a variável que pretende incrementar. m
Quantidade 6
Safe
```

2.0.2 Exemplo 2

Nestes exemplos mostramos várias execuções da função com x e y arbitrários. Nota: Nestas condições o resultado será sempre "unsafe" pois para qualquer tamanho é sempre possível encontrar "x" e "y" de modo a que a sua multiplicação resulte em $overflow$.

[15]:
```
model_checking(['pc','x','y','z'], init_unbounded, trans, error, N, M, a, b,␣
↪tamanho)
```

```
unsafe
```

[16]:
```
model_checking_input(['pc','x','y','z'], init_unbounded, trans, error, N, M, a,␣
↪b, tamanho)
```

```
unsafe
```

[17]:
```
model_checking(['pc','x','y','z'], init_unbounded, trans_more_states, error, N,␣
↪M, a, b, tamanho)
```

```
unsafe
```

[18]:
```
model_checking_input(['pc','x','y','z'], init_unbounded, trans_more_states,␣
↪error, N, M, a, b, tamanho)
```

```
unsafe
```

2.0.3 Exemplo 3

Nestes exemplos mostramos várias execuções da função com x e y iniciais limitados superiormente. Nota: Este modo de execução é o que, na nossa experiência, requer mais tempo de execução.

[19]:
```
model_checking(['pc','x','y','z'], init_bounded, trans, error, N, M, a, b,␣
↪tamanho)
```

```
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
3 3
Safe
```

[20]:
```
model_checking_input(['pc','x','y','z'], init_bounded, trans, error, N, M, a, b,␣
↪tamanho)
```

```
Não foi possível encontrar o majorante.
Qual é a variável que pretende incrementar. n
Quantidade 2
Não foi possível encontrar o majorante.
Qual é a variável que pretende incrementar. n
Quantidade 3
Safe
```

[21]:
```
model_checking(['pc','x','y','z'], init_bounded, trans_more_states, error, N, M,␣
↪a, b, tamanho)
```

```
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
```

```
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
1 10
Safe
```

[22]:
```
model_checking_input(['pc','x','y','z'], init_bounded, trans_more_states, error, N, M, a, b, tamanho)
```

```
Não foi possível encontrar o majorante.
Qual é a variável que pretende incrementar. m
Quantidade 9
```

```
Safe
```

2.0.4 Exemplo 4

Aqui mostramos mais exemplos da execução da função `model_checking`

[10]:
```
N = 100
M = 100
a = 10
b = 10
tamanho = 4
```

[11]:
```
model_checking(['pc','x','y','z'], init_fixed, trans, error, N, M, a, b, tamanho)
```

```
unsafe
```

[20]:
```
N = 100
M = 100
a = 2
b = 15
tamanho = 5
```

[21]:
```
model_checking(['pc','x','y','z'], init_bounded, trans, error, N, M, a, b,␣
↪tamanho)
```

```
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
Não foi possível encontrar o majorante.
1 6
Safe
```