Implementação

Começamos por implementar as bibliotecas que necessitamos para realizar o problema. O **pysmt.shortcuts** oferece uma API simplificada que disponibiliza as funcionalidades para a utilização usual de um SMT solver e chamamos o tipo Int de **pysmt.typing**. Também chamamos a biblioteca **itertools**

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

import itertools
```

Parâmetros do problema 1 do TP2

- **a** e **b** : dois valores inteiros que serão multiplicados
- **n** : precisão limitada em bits

[2]:
```
a = 4
b = 3
n = 3
```

Função genState

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 `!`.

[3]:
```
def genState(vars,s,i):
    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+s+str(i), BVType(n))
    return state
```

SFOTS (Baseado no problema do TP2)

Assim como diz no enunciado pretendemos modelar este problema para poder testar a sua segurança. Fizemos algumas alterações no exercicio 1 do trabalho pratico 2, de forma a modelar um **SFOTS**. Observando a definição de ser seguro ou inseguro, chegamos à conclusão que este modelo em expecifico é unsafe.

[4]:
```
def init2(state):
    return And(Equals(state['pc'], BV(0, n)), Equals(state['x'], BV(a, n)),␣
↪Equals(state['y'], BV(b, n)), Equals(state['z'], BVZero(n)))

def error2(state):
    return Equals(state['pc'], BV(7,n))

def trans2(curr,prox):
    t0_1 = 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_2 = And(
            Equals(curr['pc'], BV(1,n)),
            Equals(prox['pc'], BV(2,n)),
            NotEquals(curr['y'], BVZero(n)),
            Equals(BVZero(1), BVExtract(curr['y'], start=0, end=0)),
            Equals(prox['x'], curr['x']),
            Equals(prox['y'], curr['y']),
            Equals(prox['z'], curr['z'])
        )

    t2_3 = And(
            Equals(curr['pc'], BV(2,n)),
            Equals(prox['pc'], BV(3,n)),
            Equals(prox['x'], BVMul(curr['x'], BV(2, n))),
            Equals(prox['y'], BVUDiv(curr['y'], BV(2, n))),
            Equals(prox['z'], curr['z']),
            BVUGE(prox['x'], curr['x'])
        )

    t2_7 = And(
            Equals(curr['pc'], BV(2,n)),
            Equals(prox['pc'], BV(7,n)),
            Equals(prox['x'], BVMul(curr['x'], BV(2, n))),
            Equals(prox['y'], BVUDiv(curr['y'], BV(2, n))),
            Equals(prox['z'], curr['z']),
            Not(BVUGE(prox['x'], curr['x']))
        )

    t3_1 = And(
            Equals(curr['pc'], BV(3,n)),
            Equals(prox['pc'], BV(1,n)),
            Equals(prox['x'], curr['x']),
            Equals(prox['y'], curr['y']),
            Equals(prox['z'], curr['z'])
        )

    t1_4 = And(
            Equals(curr['pc'], BV(1,n)),
            Equals(prox['pc'], BV(4,n)),
            NotEquals(curr['y'], BVZero(n)),
            Equals(BVOne(1), BVExtract(curr['y'], start=0, end=0)),
```

```
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z'])
    )

t4_5 = And(
        Equals(curr['pc'], BV(4,n)),
        Equals(prox['pc'], BV(5,n)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], BVSub(curr['y'], BV(1, n))),
        Equals(prox['z'], BVAdd(curr['z'], curr['x'])),
        BVUGE(prox['z'], curr['z'])
    )

t4_7 = And(
        Equals(curr['pc'], BV(4,n)),
        Equals(prox['pc'], BV(7,n)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], BVSub(curr['y'], BV(1, n))),
        Equals(prox['z'], BVAdd(curr['z'], curr['x'])),
        Not(BVUGE(prox['z'], curr['z']))
    )

t5_1 = And(
        Equals(curr['pc'], BV(5,n)),
        Equals(prox['pc'], BV(1,n)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z']),
    )

t1_6 = And(
        Equals(curr['pc'], BV(1,n)),
        Equals(prox['pc'], BV(6,n)),
        Equals(curr['y'], BVZero(n)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['z'], curr['z'])
)

t6_6 = And(
            Equals(curr['pc'], BV(6,n)),
            Equals(prox['pc'], BV(6,n)),
            Equals(prox['x'], curr['x']),
            Equals(prox['y'], curr['y']),
            Equals(prox['z'], curr['z'])
        )
```

```
    t7_7 = And(
                Equals(curr['pc'], BV(7,n)),
                Equals(prox['pc'], BV(7,n)),
                Equals(prox['x'], curr['x']),
                Equals(prox['y'], curr['y']),
                Equals(prox['z'], curr['z'])
            )

    return Or(t0_1, t1_2, t2_3, t2_7, t3_1, t1_4, t4_5, t4_7, t5_1, t1_6, t6_6,␣
↪t7_7)
```

Função genTrace

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.

Esta função é uma auxilar, pois nos permite visualizar os estados.

[5]:
```
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):
                print("Estado:",i)
                for v in X[i]:
                    print("          ",v,'=',s.get_value(X[i][v]))

genTrace(['pc', 'x', 'y', 'z'], init2, trans2, error2, 60)
```

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 $\mathsf{U}_0 = \mathsf{E}$.
2. 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**.
3. 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 $\mathsf{U}_m \to \neg C(Y_m)$.

4. 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**.
5. 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.
6. 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.

Para encontrar um majorante S: A parte crítica é o passo 5. Várias estratégias são possíveis (veremos algumas mais tarde). Uma solução possível é um algoritmo iterativo que tenta encontrar um invariante S pelos passos seguintes

1. S é inicializado com $C(X_n)$
2. Faz-se $A \equiv S(X_n) \wedge \mathsf{T}(X_n, Y_m)$ e verifica-se se $A \wedge U_m$ é insatisfazível. Se for satisfazível então não é possível encontrar o majorante e esta rotina termina sem sucesso.
3. Se $A \wedge U_m$ for insatisfazível calcula-se um novo interpolante $C(Y_m)$ deste par (A, U_m).
4. Se $C(X_n) \rightarrow S$ for tautologia, o invariante pretendido está encontrado.
5. Se $C(X_n) \rightarrow S$ não é tautologia, actualiza-se S com $S \vee C(X_n)$ e repete-se o processo a partir do passo (1).

Para auxiliar na implementação deste algoritmo, começamos por definir 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.

[6]:
```
def invert(trans):
    return( lambda u, v : trans(v,u) )

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])
```

Versão Manual do model_checking

[7]:
```
def model_checking1(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)]

```

```
        n = 1
        m = 1

        while n < N and m < M:
            print(f'm = {m}, n = {n}')

            I  = init(X[0])
            Tn = And([trans(X[i],X[i+1]) for i in range(n) ])
            Rn = And (I,Tn)

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

            Vnm = And( Rn, same(X[n],Y[m]), Um )

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

            C = binary_interpolant(And (Rn, same(X[n],Y[m])), Um)

            if C is None:
                print(" interpolante None ")
                n += int(input("Deseja incrementar o n em quanto? (0 ou mais): "))
                m += int(input("Deseja incrementar o m em quanto? (0 ou mais): "))
                continue

            C0 = rename(C,X[0])
            C1 = rename(C,X[1])
            T  = trans(X[0],X[1])

            if not s.solve([C0,T, Not(C1) ]):
                print("safe")
                return

            else:
                # gerar o S
                S = rename(C,X[n])

                while(True):
                    A = And(S, trans(X[n],Y[n]) )
                    if s.solve( [A,Um] ):
                        print("Não encontramos o majorante")
```

```
                        n += int(input("Deseja incrementar o n em quanto? (0 ou␣
↪mais): "))
                        m += int(input("Deseja incrementar o m em quanto? (0 ou␣
↪mais): "))
                        break
                    else:
                        Cnew = binary_interpolant( A, Um )
                        Cn   = rename(Cnew, X[n])

                        if s.solve( [Cn,Not(S)] ):
                            S = Or(S,Cn)

                        else:
                            print("safe")
                            return
        print("unknown")

model_checking1(['x','y','z', 'pc'], init2, trans2, error2, 50, 50)
```

Versão Automatica do model_checking

[8]:
```
def model_checking2(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)]

        # 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])

        for (n,m) in order:

            I  = init(X[0])
            Tn = And([trans(X[i],X[i+1]) for i in range(n) ])
            Rn = And (I,Tn)

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

            Vnm = And( Rn, same(X[n],Y[m]), Um )

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

```
            C = binary_interpolant(And (Rn, same(X[n],Y[m])), Um)

            if C is None:
                print(" interpolante None ")
                continue

            C0 = rename(C,X[0])
            C1 = rename(C,X[1])
            T  = trans(X[0],X[1])

            if not s.solve([C0,T, Not(C1) ]):
                print("safe")
                return

            else:
                # gerar o S
                S = rename(C,X[n])

                while(True):
                    A = And(S, trans(X[n],Y[n]) )
                    if s.solve( [A,Um] ):
                        print("Não encontramos o majorante")
                        break
                    else:
                        Cnew = binary_interpolant( A, Um )
                        Cn   = rename(Cnew, X[n])

                        if s.solve( [Cn,Not(S)] ):
                            S = Or(S,Cn)

                        else:
                            print("safe")
                            return
        print("unknown")

#####

model_checking2(['x','y','z', 'pc'], init2, trans2, error2, 50, 50)
```