TP3 - Grupo 14

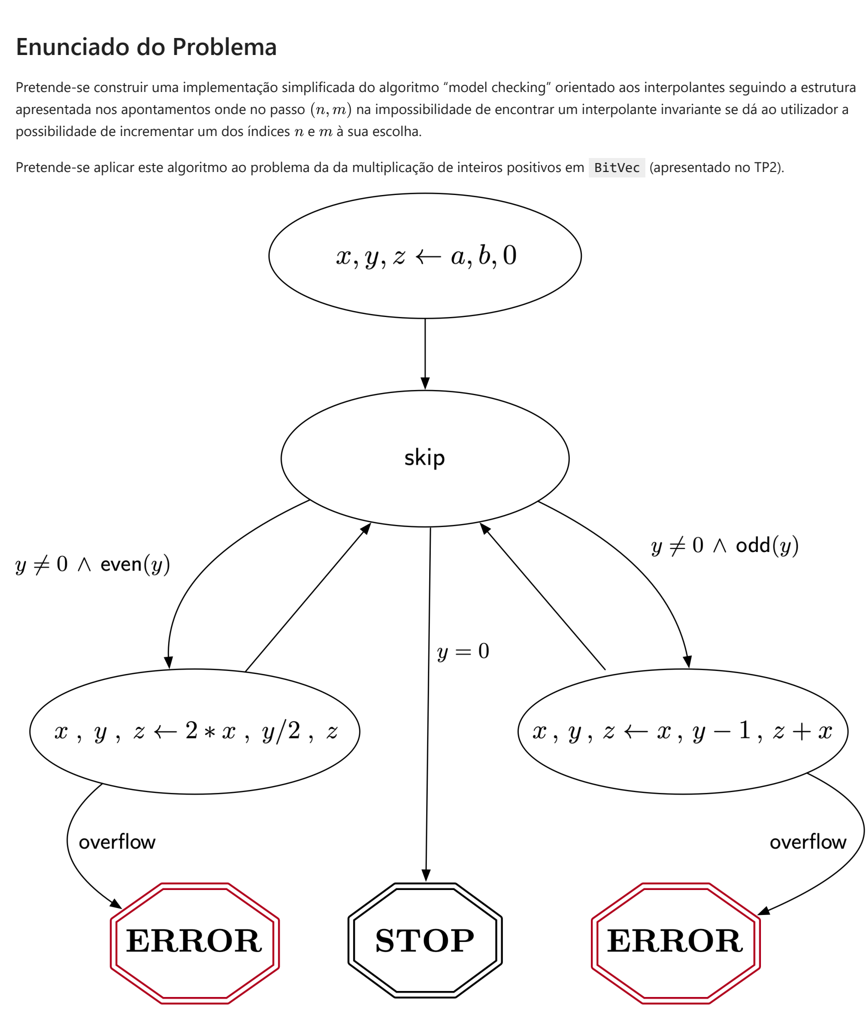
André Lucena Ribas Ferreira - A94956

Paulo André Alegre Pinto - A97391

Enunciado do Problema

Pretende-se construir uma implementação simplificada do algoritmo “model checking” orientado aos interpolantes seguindo 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).



Análise

A Análise deste problema consiste em definir as diferenças entre o problema resolvido no TP2 e este. Principalmente, a utilização de SFOTS para o descrever, tendo em conta as diferenças nas condições de erro e na notação, tal como visto nas aulas práticas.

Definição do SFOTS

Dessa forma, o sistema dinâmico será um tuplo:

$$\sum \equiv < \mathcal{T}, X, next, I, T, E >$$

Onde se verifica o seguinte, para representar o sistema em específico:

1. $\mathcal{T}$ representa um SMT apropriada, que pertence à FOL, que vamos representar no nosso Solver;
2. X é o conjunto das variáveis base do problema;
3. $next$ é um operador que gera "clones" das variáves em X;
4. I é um predicado unário que determina quais os estados iniciais;
5. T é um predicado binário que determina as transições entre dois estados;
6. E é um predicado unário que determina os estados de erro.

Nesse sentido, o modelo terá 3 variáveis do tipo BitVector, x, y e z, este último que terminará com o resultado da execução. Para além destes, também terá uma variável inteira p que representa o estado de execução. Definiu-se um inteiro para cada um dos estados, nomeadamente:

- 0 para Estado central (loop). Nele testam-se os casos.
- 1 para Estado de execução se y for par (não zero).
- 2 para Estado de execução se y for ímpar (não zero).
- 3 para Estado Final.

As cópias destas variáveis serão dadas pelo operador $next$, cuja notação se pode expandir para incluir qualquer predicado P que tenha X como o conjunto das variáveis livres. Assim, $next(X) \equiv X'$ e $next(\mathsf{P}) \equiv \mathsf{P}' \equiv \mathsf{P}\,\{X \,/\, next(X)\}$ Esta notação segue a convenção das aulas teóricas, o que lhe permite ficar "livre de variáveis".

Considerando o estado inicial, manter-se-á a optimização possível à execução do autómato, isto é, definir o valor b o menor dos dois. Isto diminui o número de operações totais necessárias, no pior caso. O estado inicial será então defindo pelo predicado seguinte:

$$I \quad \equiv \quad p = 0 \wedge x = a \wedge y = b \wedge z = 0$$

O predicado de transição não terá as transições para os estados de erro, sendo indêntica à anterior nos restantes aspetos.

$$\begin{gathered} T \equiv \\ (p = 0 \wedge y \equiv 0 \pmod 2) \wedge y \neq 0 \wedge p' = 1 \wedge x' = x \wedge y' = y \wedge z' = z) \\ \vee \\ (p = 0 \wedge y \equiv 1 \pmod 2) \wedge y \neq 0 \wedge p' = 2 \wedge x' = x \wedge y' = y \wedge z' = z) \\ \vee \\ (p = 1 \wedge 2x < 2^n \wedge p' = 0 \wedge x' = 2x \wedge y' = y/2 \wedge z' = z) \\ \vee \\ (p = 2 \wedge 2^n - 1 - z \geq x \wedge p' = 0 \wedge x' = x \wedge y' = y - 1 \wedge z' = z + x) \\ \vee \\ (p = 0 \wedge y = 0 \wedge p' = 3 \wedge x' = x \wedge y' = y \wedge z' = z) \\ \vee \\ (p = 3 \wedge p' = 3 \wedge x' = x \wedge y' = y \wedge z' = z) \end{gathered}$$

Como condição de erro, considera-se as retiradas do predicado em cima descrito, mas sem a consideração do estado atual. No entanto, deve-se limitar a esta condição o valor de y ser diferente de 0, já que não se considera erro occorrer algum possível overflow se este não for efetivamenteo calculado.

$$\begin{gathered} E \equiv \\ (y \neq 0 \wedge 2x \leq 2^n \wedge y \equiv 0 \pmod 2)) \\ \vee \\ (y \neq 0 \wedge 2^n - 1 - z < x \wedge y \equiv 1 \pmod 2)) \end{gathered}$$

Um traço de execução é uma sequência de estados, onde dois estados consequentes validam um predicado de transição. Tal como no TP2, o número de estados é finito, já que os valores que a e b podem tomar estão limitados pela precisão n, e porque as operações executadas tendem para um dos estados de *loop*, qualquer traço de execução deste problema é limitado. Dessa forma, pode-se sempre calcular o traço até ao momento em que um estado transiciona para outro que já ocorreu no traço, descrevendo assim um loop.

Segurança e Acessibilidade

Num SFOTS $\Sigma \;\equiv < \mathcal{T}, X, next, I, T, E >$ a verificação deriva das noções de acessibilidade e insegurança:

- Um estado r é acessível em Σ quando $r \in \mathsf{I}$ ou quando existe uma transição $(s, r) \in \mathsf{T}$ em que s é acessível em Σ.
- Um estado u é inseguro em Σ quando $u \in \mathsf{E}$ ou quando existe uma transição $(u, v) \in \mathsf{T}$ em que v é inseguro em Σ.
- O SFOTS Σ é inseguro se existe algum estado s que seja simultaneamente acessível e inseguro. Em caso contrário o sistema Σ é seguro.

Para ajudar na definição dos estados inseguros, define-se um SFOTS $\Sigma^T \;\equiv < \mathcal{T}, Y, next, E, B, I >$, onde Y é um clone das variáveis de X e $B \equiv T^{-1}$.

Desses dois sistemas, definem-se os predicados $\mathsf{R}_n \;\equiv\; \mathsf{I} \wedge \mathsf{T}^n$ e $\mathsf{U}_m \;\equiv\; \mathsf{E} \wedge \mathsf{B}^m$ que representam traços finitos com n, respetivamente m, transições cujos estados são acessíveis, respetivamente inseguros.

Para avaliar a segurança eventual do sistema Σ é necessário determinar se nenhum estado é simultâneamente acessível e inseguro. Para isso tem que se avaliar se, para todo o n, m, a fórmula $V_{n,m} \;\equiv\; \mathsf{R}_n \wedge (X_n = Y_m) \wedge \mathsf{U}_m$ é insatisfazível, onde $X_n = \mathsf{top}(\mathsf{R}_n)$ e $Y_m = \mathsf{top}(\mathsf{U}_m)$.

Algoritmo de Interpolantes e Invariantes

O Algoritmo de Interpolantes e Invariantes utilizado neste Trabalho Prático é o mesmo que o descrito e implementado nas aulas práticas, nomeadamente na `ficha9`. O seguinte resultado informa a sua implementação:

> Se existe um predicado unário S que é invariante de T e, para algum par de índices (n, m) verifica-se que $\mathsf{R}_n(\overline{X}_n) \rightarrow S(X_n)$ e $\mathsf{U}_m(\overline{Y}_m) \rightarrow \neg S(Y_m)$ são tautologias , então $\mathsf{V}_{n',m'}$ é insatisfazível para todo $n' \geq n$ e $m' \geq m$.

Implementação

Para a resolução do problema em questão, decidiu-se usar o módulo `pysmt.shortcuts`, com as funcionalidades possíveis para a utilização de um SMT Solver. Importam-se também os tipos deste Solver, a partir do módulo `pysmt.typing`. Para modelar este problema, usar-se-ão variáveis do tipo `BitVector`. Como auxiliar, também é importado o módulo `itertools`.

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

import itertools
```

Como número máximo da precisão dos valores, determina-se o valor de `n` para representar os n bits do valor.

In [2]:
```
n = 16
```

A função `bv_sel(z,i)` seleciona o i-ésimo bito do BitVector z.

In [3]:
```
def bv_sel(z,i):                    # seleciona o bit i do BitVec "z"
    return BVExtract(z,start=i,end=i)
```

Para gerar os estados, a função `genState(vars,s,i)` recebe as variáveis do problema e, para cada uma, cria-se uma variável binária. Também se marca a variável com uma letra `s` que representa o conjunto de variáveis, nomeadamente X ou Y, tal como um número `i` que representa a i-ésima cópia da variável.

In [4]:
```
def genState(vars,s,i):
    state = {}
    for v in vars:
        state[v] = Symbol(v+'!'+s+str(i),BVType(n))
    return state
```

As variáveis do problema são x, y, z e p, todos BitVectores.

In [5]:
```
vars = ['x','y','z','p']
```

A função `init1(state)` devolve o predicado do estado inicial, com a optimização de gerar sempre $b \leq a$. A função `init_ab(state,a,b)` não deixa ao critério do *solver* qual os valores para a e b, recebendo-os como parâmetro.

In [6]:
```
def init1(state):
    return And(Equals(state['z'],BVZero(n)), Equals(state['p'], BVZero(n)),
               Or(BVULT(state['y'], state['x']), Equals(state['y'],state['x'])))

def init_ab(state, a, b):
    if a < b:
        a,b = b,a
    return And(Equals(state['x'], BV(a,n)), Equals(state['y'], BV(b,n)), Equals(state['z'],BVZero(n)),
               Equals(state['p'], BVZero(n)))
```

A função `error1(state)` devolve o predicado do estado de erro.

In [7]:
```
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)
```

Por fim, definiu-se a função `trans(curr, prox)`, de acordo com a especificação na Análise. Notavelmente, observa-se o seguinte:

1. $2x \geq 2^n \equiv x_{n-1} = 1$
2. $y \geq 0 \pmod 2) \equiv y_0 = 0$
3. $y \geq 1 \pmod 2) \equiv y_0 = 1$
4. $2x = x << 1$
5. $2y = y >> 1$

In [8]:
```
def trans1(curr, prox):
    tend = And(Equals(curr['p'], BVZero(n)), Equals(prox['p'], BV(3,n)), Equals(curr['y'], BVZero(n)),
               Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

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

    todd = And(Equals(curr['p'], BVZero(n)), Equals(prox['p'], BV(2,n)), Equals(bv_sel(curr['y'],0), BVOne(1)),
               Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

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

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

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

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

A função `genTrace` recebe todas estas funções, exceto a do predicado do erro, mais o número `n`, tamanho máximo do traço, e devolve um traço execução sem qualquer consideração pelo estado de erro.

In [9]:
```
def genTrace(vars,init,trans,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]))
```

In [10]:
```
genTrace(vars, init1, trans1, 10)
```

```
Estado: 0
           x = 512_16
           y = 18_16
           z = 0_16
           p = 0_16
Estado: 1
           x = 512_16
           y = 18_16
           z = 0_16
           p = 1_16
Estado: 2
           x = 1024_16
           y = 9_16
           z = 0_16
           p = 0_16
Estado: 3
           x = 1024_16
           y = 9_16
           z = 0_16
           p = 2_16
Estado: 4
           x = 1024_16
           y = 8_16
           z = 1024_16
           p = 0_16
Estado: 5
           x = 1024_16
           y = 8_16
           z = 1024_16
           p = 1_16
Estado: 6
           x = 2048_16
           y = 4_16
           z = 1024_16
           p = 0_16
Estado: 7
           x = 2048_16
           y = 4_16
           z = 1024_16
           p = 1_16
Estado: 8
           x = 4096_16
           y = 2_16
           z = 1024_16
           p = 0_16
Estado: 9
           x = 4096_16
           y = 2_16
           z = 1024_16
           p = 1_16
```

Verificação

Algoritmo de "model-checking" usando interpolantes e invariantes

A seguinte implementação do algoritmo de model-checking é a mesma que a realizada nas aulas práticas, nomeadamente da resolução da `ficha9`. A diferença particular é a definida pelo enunciado, onde o incremento ao n e ao m é feito por interpolação ao utilizador.

In [11]:
```
def invert(trans):
    return (lambda curr,prox: trans(prox, curr))
```

In [12]:
```
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])
```

In [48]:
```
def model_checking(vars,init,trans,error,N,M,a,b):
    with Solver(name="msat") 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)]

        #Testar para o caso n=0 e m=0
        I = init_ab(X[0],a,b)
        E = error(Y[0])
        if s.solve([I,E,same(X[0], Y[0])]):
            print("Unsafe!")

            print("Estado X0:")
            for v in X[0]:
                print("          ",v,'=',s.get_value(X[0][v]))
            print("Estado Y0:")
            for v in Y[0]:
                print("          ",v+'\'','=',s.get_value(Y[0][v]))
            return

        n = 1
        m = 1

        while n<=N and m <= M:
            print("n =",n,"m =",m)
            #I = init_ab(X[0],a,b)
            #E = error(Y[0])
            Tn = And([trans(X[i], X[i+1]) for i in range(n)])
            Bm = And([invert(trans)(Y[j], Y[j+1]) for j in range(m)])
            Rn = And(I, Tn)
            Um = And(E, Bm)
            Vnm = And(Rn, Um, same(X[n], Y[m]))

            #1º Passo
            if s.solve([Vnm]):
                print("Unsafe!")
                for i in range(n+1):
                    print("Estado X%d:" % i)
                    for v in X[i]:
                        print("          ",v,'=',s.get_value(X[i][v]))
                for i in range(m+1):
                    print("Estado Y%d:" % i)
                    for v in Y[i]:
                        print("          ",v+'\'','=',s.get_value(Y[i][v]))
                return

            #2º Passo
            C = binary_interpolant(And(Rn, same(X[n], Y[m])), Um)
            if C is None:
                print("Interpolante None!")
                continue
            #3* Passo
            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("C:", serialize(C))
                print("Safe, o interpolante é invariante!")
                return

            #4º Passo - gerar o S
            S = rename(C, X[n])
            while True:
                A = And(S, trans(X[n], Y[m]))
                if s.solve([A, Um]):
                    print("Interpolante None!")
                    break
                Cnew = binary_interpolant(A, Um)
                #if Cnew is None:
                #    print("Interpolante None!")
                #    break
                Cn = rename(Cnew, X[n])
                if s.solve([Cn, Not(S)]):
                    S = Or(S, Cn)
                else:
                    print("Safe, o majorante é invariante!")
                    return
            nm = ""
            while nm != "n" and nm != "m":
                nm = input("Introduza 'n' ou 'm' para incrementar um dos valores.")
            inp = 0
            while True:
                try:
                    inp = int(input("Quanto pretende que seja o seu incremento?"))
                except ValueError:
                    continue
                if inp < 0:
                    print("Valor inválido")
                    continue
                if nm == "n":
                    n += inp
                else:
                    m += inp
                break

        print("unknown")
```

In [49]:
```
model_checking(vars, init1, trans1, error1, 50, 50,10,10)
```

```
n = 1 m = 1
Interpolante None!

n = 11 m = 1
Safe, o interpolante é invariante!
```

A alteração feita para alterar manualmente o incremento tanto do n como do m tem uma implicação interessante nas conclusões que se podem tirar. Tal como diz o resultado em cima enunciado, quando se prova que o sistema é **seguro** para qualquer $n' >= n, m' >= m$, dados n e m, não se tem em consideração os valores anteriores. O algoritmo original percorria todos os valores possíveis de n e de m, então dava a certeza que, quando se chegasse a dado n, todos os anteriores tinham sido visitados. Tal não ocorre com esta implementação, causando casos como o seguinte (claramente *inseguro*).

In [50]:
```
model_checking(vars, init1, trans1, error1, 50, 50,30000,10)
```

```
n = 1 m = 1
Interpolante None!

n = 5 m = 1
Safe, o interpolante é invariante!
```

In [51]:
```
model_checking(vars, init1, trans1, error1, 50, 50,30000,10)
```

```
n = 1 m = 1
Interpolante None!

n = 4 m = 1
Unsafe!
Estado X0:
           x = 30000_16
           y = 10_16
           z = 0_16
           p = 0_16
Estado X1:
           x = 30000_16
           y = 10_16
           z = 0_16
           p = 1_16
Estado X2:
           x = 60000_16
           y = 5_16
           z = 0_16
           p = 0_16
Estado X3:
           x = 60000_16
           y = 5_16
           z = 0_16
           p = 2_16
Estado X4:
           x = 60000_16
           y = 4_16
           z = 60000_16
           p = 0_16
Estado Y0:
           x' = 60000_16
           y' = 4_16
           z' = 60000_16
           p' = 1_16
Estado Y1:
           x' = 60000_16
           y' = 4_16
           z' = 60000_16
           p' = 0_16
```

Será então importante ter em conta estes saltos, eventualmente implementando uma lista de visitados para avisar o utilizador que ainda existem casos não visitados, se se verificar a *segurança* do sistema.

Para testar que o sistema chega ao estado final, isto é $p = 3$, apenas basta definir um predicado "de erro" e considerar que se for "unsafe" quer dizer que esse estado é acessível, ou seja, que o estado inicial é inseguro para o erro "terminar". Note-se a coincidência do estado $X6$ e $Y5$ (para $n = 6$ e $m = 5$).

In [16]:
```
def error2(state):
    return(Equals(state['p'],BV(3,n)))
```

In [54]:
```
model_checking(vars, init1, trans1, error2, 50, 50,10,10)
```

```
n = 1 m = 1
Interpolante None!

n = 5 m = 1
Interpolante None!

n = 5 m = 5
Interpolante None!

n = 6 m = 5
Unsafe!
Estado X0:
           x = 10_16
           y = 10_16
           z = 0_16
           p = 0_16
Estado X1:
           x = 10_16
           y = 10_16
           z = 0_16
           p = 1_16
Estado X2:
           x = 20_16
           y = 5_16
           z = 0_16
           p = 0_16
Estado X3:
           x = 20_16
           y = 5_16
           z = 0_16
           p = 2_16
Estado X4:
           x = 20_16
           y = 4_16
           z = 20_16
           p = 0_16
Estado X5:
           x = 20_16
           y = 4_16
           z = 20_16
           p = 1_16
Estado X6:
           x = 40_16
           y = 2_16
           z = 20_16
           p = 0_16
Estado Y0:
           x' = 80_16
           y' = 0_16
           z' = 100_16
           p' = 3_16
Estado Y1:
           x' = 80_16
           y' = 0_16
           z' = 100_16
           p' = 0_16
Estado Y2:
           x' = 80_16
           y' = 1_16
           z' = 20_16
           p' = 2_16
Estado Y3:
           x' = 80_16
           y' = 1_16
           z' = 20_16
           p' = 0_16
Estado Y4:
           x' = 40_16
           y' = 2_16
           z' = 20_16
           p' = 1_16
Estado Y5:
           x' = 40_16
           y' = 2_16
           z' = 20_16
           p' = 0_16
```

In [ ]: