Verificação Formal - SMT Solving

Hugo Faria

Março 2021

1 Introdução

Este trabalho foi realizado com o objectivo de aprofundar os conhecimentos sobre *SMT Solvers*. Serão apresentadas as resoluções a cada um dos exercícios propostos.

2 Exercício 1 - Matriz

```
for (i=1; i<=3; i++)
    for (j=1; j<=3; j++)
        M[i][j] = i+j;
```

O código dado corresponde a um programa que preenche uma matriz 3x3 com a soma dos índices de linha e coluna de cada uma das posições.

2.1 Exercício 1.1

É pedido para escrever em C um programa equivalente com apenas uma sequência de atribuições. Foi então criado o seguinte programa:

```
void main(){
  int M[3][3];
      M[1][1] = 2;
      M[1][2] = 3;
      M[1][3] = 4;
      M[2][1] = 3;
      M[2][2] = 4;
      M[2][3] = 5;
      M[3][1] = 4;
      M[3][2] = 5;
      M[3][3] = 6;
}
```

2.2 Exercício 1.2

De seguida é pedida a codificação lógica do programa feito em cima num ficheiro em formato *SMT-LIBv2*, tendo sido gerado então o seguinte ficheiro:

```
(declare-const M0 (Array Int (Array Int Int)))
(declare-const M1 (Array Int (Array Int Int)))
(declare-const M2 (Array Int (Array Int Int)))
(declare-const M3 (Array Int (Array Int Int)))
(declare-const M4 (Array Int (Array Int Int)))
(declare-const M5 (Array Int (Array Int Int)))
(declare-const M6 (Array Int (Array Int Int)))
(declare-const M7 (Array Int (Array Int Int)))
(declare-const M8 (Array Int (Array Int Int)))
(declare-const M9 (Array Int (Array Int Int)))

(assert (= M1 (store M0 1 (store (select M0 1) 1 2))))
(assert (= M2 (store M1 1 (store (select M1 1) 2 3))))
(assert (= M3 (store M2 1 (store (select M2 1) 3 4))))
(assert (= M4 (store M3 2 (store (select M3 2) 1 3))))
(assert (= M5 (store M4 2 (store (select M4 2) 2 4))))
(assert (= M6 (store M5 2 (store (select M5 2) 3 5))))
(assert (= M7 (store M6 3 (store (select M6 3) 1 4))))
(assert (= M8 (store M7 3 (store (select M7 3) 2 5))))
(assert (= M9 (store M8 3 (store (select M8 3) 3 6))))

(check-sat)
(push)
```

2.3 Exercício 1.3

Após a codificação em *SMT-LIBv2*, é pedido para se confirmar a veracidade das seguintes afirmações:

2.3.1 a) Se i = j então M[i][j] != 3.

A afirmação corresponde a (i = j) -> (M[i][j] != 3). Ora, negando a afirmação temos:

```
(declare-fun i () Int)
(declare-fun j () Int)

(assert (not(=> (= i j) (not(= (select (select M9 i) j) 3)))))

(check-sat)
(pop)
(push)
```

Correndo no solver temos que o código é satisfazível, ou seja, a afirmação é refutável. Ora isto acontece uma vez que o i e o j não são limitados entre 1 e 3, não podendo assim afirmar que tal não se iria verificar fora desses limites.

Adicionando os seguintes limites à afirmação:

```
(declare-fun i () Int)
(declare-fun j () Int)

(assert (and (>= i 1) (<= i 3)))
(assert (and (>= j 1) (<= j 3)))
(assert (not(=> (= i j) (not(= (select (select M9 i) j) 3)))))

(check-sat)
(pop)
(push)
```

O *solver* já diz que o código é insatisfazível, ou seja, a afirmação é verdadeira.

2.3.2 b) Para quaisquer i e j entre 1 e 3, M[i][j] = M[j][i].

A negação da afirmação corresponde ao seguinte código:

```
(declare-fun i () Int)
(declare-fun j () Int)

(assert (and (>= i 1) (<= i 3)))
(assert (and (>= j 1) (<= j 3)))
(assert (not(= (select (select M9 i) j) (select (select M9 j) i))))

(check-sat)
(pop)
(push)
```

Correndo no *solver* temos que o código é insatisfazível, ou seja, a afirmação é verdadeira.

2.3.3 c) Para quaisquer i e j entre 1 e 3, se i < j então M[i][j] < 6.

A afirmação corresponde a, sabendo que i e j estão entre 1 e 3, (i<j) -> M[i][j] < 6, que negada traduz para o seguinte código:

```
(declare-fun i () Int)
(declare-fun j () Int)

(assert (and (>= i 1) (<= i 3)))
(assert (and (>= j 1) (<= j 3)))
(assert (not(=> (< i j) (< (select (select M9 i) j) 6))))
```

```
(check-sat)
(pop)
(push)
```

Correndo no *solver* temos que o código é insatisfazível, ou seja, a afirmação é verdadeira.

2.3.4 d) Para quaisquer i, a e b entre 1 e 3, se a > b então M[i][a] > M[i][b].

A afirmação corresponde a, sabendo que i,a e b estão entre 1 e 3, (a>b) -> M[i][a] > M[i][b], que negada traduz para o seguinte código:

```
(declare-fun i () Int)
(declare-fun a () Int)
(declare-fun b () Int)

(assert (and (>= i 1) (<= i 3)))
(assert (and (>= a 1) (<= a 3)))
(assert (and (>= b 1) (<= b 3)))

(assert (not(=> (> a b) (> (select (select M9 i) a) (select (select M9 i) b)))))

(check-sat)
(pop)
(push)
```

Correndo no *solver* temos que o código é insatisfazível, ou seja, a afirmação é verdadeira.

2.3.5 e) Para quaisquer i e j entre 1 e 3, M[i][j] + M[i+1][j+1] = M[i+1][j] + M[i][j+1].

A negação da afirmação corresponde ao seguinte código:

```
(declare-fun i () Int)
(declare-fun j () Int)

(assert (and (>= i 1) (<= i 3)))
(assert (and (>= j 1) (<= j 3)))
(assert (not(=(+ (select (select M9 i) j) (select (select M9 (+ i 1)) (+ j 1))) (+ (sele

(check-sat)
(pop)
(push)
```

Correndo no *solver* temos que o código é satisfazível, ou seja, a afirmação é refutável. Ora tal acontece porque quando i ou j são 3 temos tentativas de acesso a valores no índice 4, índice este que não é preenchido no código logo não podemos afirmar sobre o mesmo.

3 Exercício 2 - Puzzle solver

Neste exercício é pedido para fazer um *solver* de um dos puzzles fornecidos. Ora o seguinte *solver* foi feito *python* para o puzzle *survo*.

3.1 Input

O input para o *solver* é feito através dum ficheiro txt que tem de ter o seguinte formato:

```
 ,6, , ,30
8, , , ,18
 , ,3, ,30
27,16,10,25
```

Ou seja, as várias células da tabela têm de ser separadas por vírgulas sendo que os espaços em branco têm de ser representados por .

3.2 Preparação

Para poder começar a escrever as restrições é preciso primeiro extrair a informação sobre o puzzle. Ora, é então pedido o nome do ficheiro de texto que contém o puzzle. De seguida, percorre-se o ficheiro preenchendo a matriz **matriz** com os vários valores do mesmo, registando-se também o número de linhas e de colunas.

```
from z3 import *
file = input("Enter puzzle file name:\n")
matriz = []
with open(file) as fp:
    for line in fp:
        line = line.strip('\n')
        newLines = line.split(',')
        matriz.append(newLines)
linhas = len(matriz)-1
colunas = len(matriz[0])-1
```

A seguir, declaram-se as variáveis e guardam-se na matriz **matrizVar** para poder aceder mais tarde aos valores de cada célula

```
matrizVar = []
solver = Solver()
for i in range(0,linhas):
    linhaVar = []
    for j in range(0,colunas):
        linhaVar.append(Int("x"+str(i)+str(j)))
    matrizVar.append(linhaVar)
```

3.3 Restrições

Em seguida, começam-se a registar as restrições associadas ao puzzle, sendo estas:

3.3.1 Números todos distintos

```
dist = [ Distinct([ matrizVar[l][c]
for l in range(linhas) for c in range(colunas) ]) ]
solver.add(dist)
```

3.3.2 Números todos maior que 0

```
maior_zero = [matrizVar[l][c] > 0
for c in range(colunas) for l in range(linhas)]
solver.add(maior_zero)
```

3.3.3 A soma das linhas têm de dar um valor predefinido

```
linhas_sum = [ And(sum([ matrizVar[l][c]
for c in range(colunas) ]) == matriz[l][colunas])
for l in range(linhas)]
solver.add(linhas_sum)
```

3.3.4 A soma das colunas têm de dar um valor predefinido

```
colunas_sum = [ And(sum([ matrizVar[l][c]
for l in range(linhas) ]) == matriz[linhas][c])
for c in range(colunas)]
solver.add(colunas_sum)
```

3.3.5 Registar valores de células que podem já estar definidas à partida

```
colunas_sum = [ And(sum([ matrizVar[l][c]
for l in range(linhas) ]) == matriz[linhas][c]) for c in range(colunas)]
solver.add(colunas_sum)
```

No final verifica-se a veracidade do modelo e escreve-se em ficheiro usando a função **solucaoWrite**:

```
def solucaoWrite(matriz,valor):
    linhas = len(matriz)
    colunas = len(matriz[0])
    f = open("solucao.txt","w")

    for i in range(0,linhas):
        string = "Linha " + str(i) + " : "
```

```
        for j in range(0,colunas):
            string = string+str(matriz[i][j])+" | "
        string = string+str(valor[i][colunas])+"\n"
        f.write(string)
    string = "          "
    for j in range(0,colunas-1):
        string = string+str(valor[linhas][j])+" | "
    string = string+str(valor[linhas][j])+" |"
    string = string+"\n"
    f.write(string)

print(solver.check())
if solver.check() == sat:
    m = solver.model()
    r = [ [ m.evaluate(matrizVar[i][j]) for j in range(colunas) ]
          for i in range(linhas) ]
    print_matrix(r)
    solucaoWrite(r,matriz)
else:
    print ("failed to solve")
```

O ficheiro resultante tem como nome solucao.txt e tem o formato que se pode ver no seguinte exemplo resultante do input exemplificado acima:

```
Linha 0 : 12 | 6 | 2 | 10 | 30
Linha 1 : 8 | 1 | 5 | 4 | 18
Linha 2 : 7 | 9 | 3 | 11 | 30
          27 | 16 | 10 | 10 |
```