UNIVERSIDADE DO MINHO

Verificação Formal

TP2 - Questão de Avaliação sobre SMT solving

Rafael Lourenço
(A86266)

Conteúdo

Capítulo 1

SMT Solving

1.1 Contextualização

No âmbito da unidade curricular de Verificação formal, foi proposto desenvolver dois exercícios . Este relatório visa detalhar duas implementação em *SMT* de uma possível solução para os problemas propostos. Para o primeiro exercício foi apenas usado o programa *z3* através de bash, já para o segundo, para além do z3 foi também utilizada a linguagem de programação *(python)* para auxiliar no processo de implementação.

1.2 Exercício 1 - Matriz

Neste exercício foi nos apresentado um programa em **C**,onde foi pedido para transformar esse programa numa sequência de atribuições. De seguida, era pedido para codificar em lógica, num formato *SMT-LIBv2*, esse programa.Por fim , foi pedido para verificar a validade de várias propriedades.
Apresento de seguida o programa em C:

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

Figura 1.1: Programa em C a ser codificado.

Para **pergunta 1.1**, era necessário transformar o programa numa sequência de atribuições, como já foi dito anteriormente.Na figura abaixo podemos verificar a reposta a mesma :

```
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;
```

Figura 1.2: Sequência de atribuições em C

Para **pergunta 1.2**, que era pedido para codificar em lógica, foi então usado um conjunto de funções que permitiram esta implementação . Passo agora a descrever as funções mais importantes:

- (**set-logic** type) - Esta função permite definir o tipo da lógica, nomeadamente se **type**=*QF_AUFLIA*, que permite a utilização de fórmulas lineares livres de quantificadores fechados sobre a teoria de *arrays*.
- (**declare-const** name type) - Esta função permite declarar variáveis, sendo passado como argumento o nome e o tipo da variável.
- (**select** est pos) - Esta função recebe uma estrutura e a posição que se pretende obter, sendo retornado essa posição.Por exemplo, no caso de um *matriz* estar definida como 2 arrays, (*select* matriz 1) que retornaria um *array* correspondente à linha 1.
- (**store** est pos value) - Esta função recebe uma estrutura, a posição que se pretende alterar e o valor que será inserido, sendo retornado a estrutura alterada. De forma similar, no caso de um *matriz*, (*store* matriz 1 a), sendo que **a** é um *array* , a linha 1 da matriz será substituída por esse *array*.
- (**check-sat**) - verifica se um modelo é satisfazível.
- (**get-model**) - Se for satisfazível , esta retorna um possível modelo.
- (**get-value** var) - obtém um valor de uma variável.

Apresento de seguida, a forma como foi codificado logicamente o programa.Inicialmente foi definido o tipo de lógica, tendo de seguida declarado as variáveis necessárias.Estas só podem ser atribuídas uma vez, podendo ser vistas como **constantes**.

```
(set-logic QF_AUFLIA)
(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)) )
```

Figura 1.3: Declaração de variaveis

Após este procedimento foi então atribuído as "variáveis"declaradas acima, os valores respetivos, executando os seguintes *asserts*.

```
(assert (= m1
        (store m0 1
                (store
                    (store
                        (store (select m0 1) 1 2)
                    2 3)
                3 4) )))

(assert (= m2
        (store m1 2
                (store
                    (store
                        (store (select m1 2) 1 3)
                    2 4)
                3 5) )))

(assert (= m3
        (store m2 3
                (store
                    (store
                        (store (select m2 2) 1 4)
                    2 5)
                3 6) )))
```

Figura 1.4: Atribuição dos valores às variáveis

Para **pergunta 1.3**, foi proposto verificar várias propriedades, e isto é feito através da colocação da negação da formula que queremos provar. Se o modelo ficar *Unsat* a propriedade é valida, caso contrário não podemos garantir a veracidade da mesma.

- **Alínea a** - Se i = j entao M[i][j] $\neq$ 3.

Nesta questão se assumir que a variável i não está quantificada esta afirmação é **falsa**.Caso contrário, se assumir que esta está quantificada entre 1 e 3, esta afirmação é correta.

```
(push)
(declare-const p1 Int)

;(assert (and (> p1 0) (< p1 4)))
(assert (= (select (select m3 p1) p1) 3))

(echo "Alinea a:")
(check-sat)
(echo "")
(pop)
```

Figura 1.5: Alínea a

Se remover a linha que está comentada, a variável i torna-se quantificada entre 1 e 3.

Um contra exemplo neste caso seria realizar um *select* para um valor não definido, ou seja , p1=0 ou p1>3.Dado estes valores conterem "lixo", não se pode garantir que seja diferente de 0.De seguida, mostro a instrução:

```
(check-sat)
(select (select m3 0) 0))
```

Figura 1.6: Contra-exemplo da alínea a

- **Alínea b** - Para quaisquer i e j entre 1 e 3, M[i][j] = M[j][i].

Esta afirmação é verdadeira, pois o resultado após acrescentar a negação da mesma o resultado do *z3* é **unsat**.

```
(push)
(declare-const p1 Int)
(declare-const p2 Int)

(assert (and (> p1 0) (< p1 4)))
(assert (and (> p2 0) (< p2 4)))
(assert (not (= (select (select m3 p1) p2) (select (select m3 p2) p1))))

(echo "Alinea b:")
(check-sat)
(echo "")
(pop)
```

Figura 1.7: Alínea b

- **Alínea c** - Para quaisquer i e j entre 1 e 3,se i<j então M[i][j]<6.

Esta afirmação é verdadeira, pelo mesmo motivo explicado na alínea anterior.

```
(push)
(declare-const p1 Int)
(declare-const p2 Int)

(assert (and (> p1 0) (< p1 4)))
(assert (and (> p2 0) (< p2 4)))
(assert (not (=> (< p1 p2 ) (< (select (select m3 p1) p2) 6)   ) ))
(echo "Alinea c:")

(check-sat)
(echo "")
(pop)
```

Figura 1.8: Alínea c

- **Alínea d** - Para quaisquer i, a e b entre 1 e 3, se a > b então M[i][a] > M[i][b].

Esta afirmação é verdadeira, pelo mesmo motivo explicado na alínea b.

```
(push)
(declare-const i Int)
(declare-const p1 Int)
(declare-const p2 Int)
(assert (and (> i 0) (< i 4)))
(assert (and (> p1 0) (< p1 4)))
(assert (and (> p2 0) (< p2 4)))
(assert (not (=> (> p1 p2 ) (> (select (select m3 i) p1) (select (select m3 i) p2) )   ) ))

(echo "Alinea d:")
(check-sat)
(echo "")
(pop)
```

Figura 1.9: Alínea d

- **Alínea 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].

Esta alínea não se pode garantir que seja verdade, dado que o resultado do *SMT-solver* dar **sat**, após a adição da negação da afirmação.

```
(push)
(declare-const p1 Int)
(declare-const p2 Int)

(assert (and (> p1 0) (< p1 4)))
(assert (and (> p2 0) (< p2 4)))
(assert (not (=
        (+ (select (select m3 p1) p2)
                (select (select m3 (+ p1 1)) (+ p2 1)) )
        (+ (select (select m3 (+ p1 1)) p2)
                (select (select m3 p1) (+ p2 1)) )
        )))

(echo "Alinea e:")
(check-sat)
(echo "")
(pop)
```

Figura 1.10: Alínea e

Um Possível contra-exemplo é para i=3 e j=3, a matriz não está definida para M[3][3] + M[4][4] = M[4][3] + M[3][4].Para verificar isto no z3, adicionamos estas instruções:

```
(push)
(declare-const v1 Int); i=3 j=3
(declare-const v2 Int)
(assert (= (+ (select (select m3 3) 3)  (select (select m3 4) 4) ) v1))
(assert (= (+ (select (select m3 4) 3)  (select (select m3 3) 4) ) v2))
(check-sat)
(get-value (v1))
(get-value (v2))
(pop)
```

Figura 1.11: Contra-exemplo da alínea e

1.3 Exercício 2 - Survo

O puzzle **Survo** consiste em preencher um matriz n×m com inteiros entre 1..m*n, sendo que cada um destes valores só pode aparecer uma vez. A soma das linhas tem de corresponder ao valor mais à direita da matriz e as ultimas células de cada coluna correspondem à soma das colunas. De seguida, são apresentadas as restrições que foram necessárias implementar para o z3 resolver o problema:

- **Células únicas** - Todas as células deste puzzle têm um numero único.
- **Valor das células** - O valor de cada célula tem de estar contido entre 1 e n*m sendo**n** o número de linhas e **m** o número de colunas.
- **restrição das linhas/colunas** - A soma de cada linha tem de corresponder a um valor que está no puzzle inicial. O mesmo acontece com as colunas.

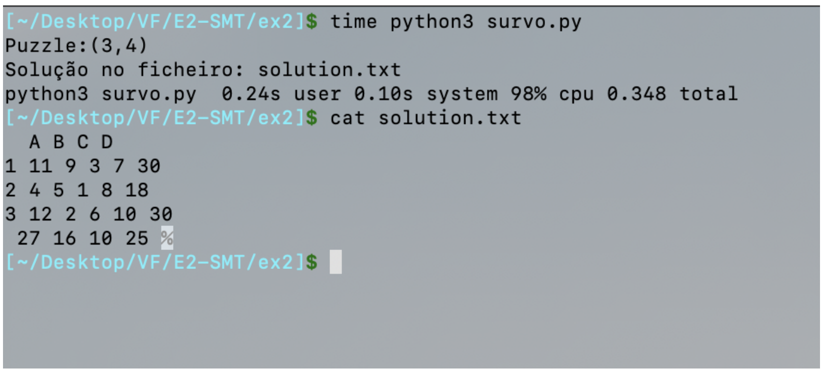
A resolução automática deste puzzle foi implementada em *python* utilizando o modulo do *z3*.Para a sua execução basta escrever em **bash** *python3 survo.py*,porém tem de existir previamente um ficheiro chamado *sample.txt* com o tabuleiro inicial.A estrutura deste tem de conter espaços entre as células com é apresentado de seguida:

```
   A B C D
1 . . . . 30
2 . . . . 18
3 . . . . 30
 27 16 10 25
```

Figura 1.12: Estrutura do ficheiro

1.3.1 Resultados obtidos:

De forma a realizar uma breve análise aos tempos de execução,o programa foi testado com vários puzzles, sendo fácil de perceber que quanto maior for a matriz mais tempo irá demorar a resolver o puzzle, podendo mesmo demorar várias horas, visto apresentar um crescimento exponencial.De seguida, mostro alguns exemplos:



```
[~/Desktop/VF/E2-SMT/ex2]$ time python3 survo.py
Puzzle:(3,4)
Solução no ficheiro: solution.txt
python3 survo.py  0.24s user 0.10s system 98% cpu 0.348 total
[~/Desktop/VF/E2-SMT/ex2]$ cat solution.txt
  A B C D
1 11 9 3 7 30
2 4 5 1 8 18
3 12 2 6 10 30
 27 16 10 25 %
[~/Desktop/VF/E2-SMT/ex2]$
```

Figura 1.13: Puzzle 3×4

Para o exemplo mostrado anteriormente,a obtenção da resolução do problema foi relativamente rápida, visto demorar 0.20 segundos.

```
[~/Desktop/VF/E2-SMT/ex2]$ time python3 survo.py
Puzzle:(5,5)
Solução no ficheiro: solution.txt
python3 survo.py  9.40s user 0.12s system 99% cpu 9.534 total
[~/Desktop/VF/E2-SMT/ex2]$ cat solution.txt
A B C D E
1 25 21 7 3 10 66
2 20 5 2 16 14 57
3 6 19 4 11 1 41
4 22 24 23 9 13 91
5 15 17 18 12 8 70
 88 86 54 51 46%
[~/Desktop/VF/E2-SMT/ex2]$
```

Figura 1.14: Puzzle 5×5

Neste puzzle o tempo de execução aumentou substancialmente, passou a ser de **9** segundos, pois passamos a ter mais variáveis proposicionais.

```
[~/Desktop/VF/E2-SMT/ex2]$ time python3 survo.py
Puzzle:(5,5)
Solução no ficheiro: solution.txt
python3 survo.py  42.12s user 0.15s system 99% cpu 42.331 total
[~/Desktop/VF/E2-SMT/ex2]$ cat solution.txt
  A B C D E
1 5 17 22 8 16 68
2 13 15 1 14 19 62
3 2 7 12 20 6 47
4 24 3 10 11 23 71
5 25 21 18 9 4 77
 69 63 63 62 68%
[~/Desktop/VF/E2-SMT/ex2]$
```

Figura 1.15: Puzzle 5×5

Neste caso, o tabuleiro não foi aumentado comparativamente ao anterior no entanto o tempo de execução aumentou para 42 segundos, mostrando assim que o valor da soma das colunas também influencia o tempo para a resolução do problema.

```
[~/Desktop/VF/E2-SMT/ex2]$ time python3 survo.py
Puzzle:(6,6)
Solução no ficheiro: solution.txt
python3 survo.py  135.49s user 0.72s system 99% cpu 2:17.08 total
[~/Desktop/VF/E2-SMT/ex2]$ cat solution.txt
  A B C D E F
1 9 35 2 25 6 32 109
2 8 11 33 10 3 28 93
3 5 4 31 27 26 21 114
4 15 22 36 7 16 18 114
5 34 19 1 29 30 14 127
6 13 12 24 17 20 23 109
 84 103 127 115 101 136%
[~/Desktop/VF/E2-SMT/ex2]$
```

Figura 1.16: Puzzle 6×6

Por ultimo, neste caso temos um puzzle 6 ×6, e como se pode verificar pela figura, o tempo aumentou para 2 minutos e 17 segundos.Logo pode-se concluir que para uma matriz N × M para N,M >7, pode não ser resolúvel em tempo útil,isto com máquinas que operam com bits *booleanos* (0 ou 1) .