Métodos Formais em Engenharia de Software (2021/22)

SMT solving

Comece por instalar alguns SMT solvers. Recomendamos o `z3` e o `cvc4`. Como alternativa de resurso, pode utilizar a versão online destes solvers indicada na página de MFES.

1 SMT-LIB 2: um exemplo simples

Comecemos por usar um SMT solver para nos ajudar a responder à seguinte pergunta:

> *Sejam x, y e z inteiros positivos, distintos entre si. Se o valor de y não poder exceder 3, que valores poderão ter as variáveis x, y e z para que a sua soma dê 8?*

Vamos usar a lógica QF_LIA (*quantifier-free linear integer arithmetic*) para este caso. O ficheiro `equacoes.smt2` no formato SMT-LIB 2, contém a descrição das restrições impostas pelo problema:

```
(set-logic QF_LIA)

(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)

(assert (> x 0))
(assert (> y 0))
(assert (> z 0))
(assert (distinct x y z))
(assert (= (+ x y z) 8))
(assert (<= y 3))

(check-sat)
(get-model)
; (get-value (x y))
```

Notas:
- Constantes, funções, proposições e predicados são todos declarados da mesma forma, com `declare-fun`. Apenas o tipo varia.
- As restrições do problema são declaradas com `assert`.
- O comando `(check-sat)` testa a satisfazibilidade das restrições.
- Finalmente, `(get-model)` imprime o modelo obtido caso a resposta seja SAT.
- Se desejarmos podemos, em vez de imprimir o modelo na sua totalidade, imprimir apenas algumas variáveis, substituindo `(get-model)`, por exemplo, por `(get-value (x y))`.
- O ';' é o início de uma linha de comentário.

Podemos agora invocar um solver com este ficheiro. Por exemplo, o `z3`:

```
$ z3 equacoes.smt2
sat
(model
  (define-fun z () Int
    2)
  (define-fun y () Int
    1)
  (define-fun x () Int
    5)
)
```

A solução (modelo) calculada consiste em definições das 3 constantes de tipo inteiro e dá-nos uma resposta para o problema: $x = 5,\ y = 1,\ z = 2$. Haverá outras soluções?

Podemos tentar ver a solução que o `cvc4` propõe:

```
$ cvc4 equacoes.smt2
sat
(error "Cannot get model when produce-models options is off.")
```

Note que existem diferenças na forma como é feita a invocação dos diferentes solvers. Incluindo na invocação a opção referida,

```
$ cvc4 --produce-models equacoes.smt2
sat
(model
(define-fun x () Int 1)
(define-fun y () Int 3)
(define-fun z () Int 4)
)
```

vemos que o modelo calculado pelo `cvc4` é $x = 1,\ y = 3,\ z = 4$. Uma solução diferente da que foi obtida com o `z3`. Portanto, parece haver várias soluções possíveis.

Se quisermos obter os vários modelos alternativos para o problema, teremos que ir (gradualmente) incluindo restrições que excluam a solução apresentada pelo solver.

Usando esta técnica, calcule agora todas as soluções possíveis do problema.

2 Z3Py: API do Z3 para Phython

Z3Py é a biblioteca Python de interface para o solver Z3. O notebook Colab `Z3Python.ipynb` faz uma breve introdução à utilização do Z3 em Python. Corra esse notebook e resolva os exercícios propostos.

3 Unicorn puzzle

Vamos trabalhar com a lógica mais simples, QF_UF, que inclui apenas a teoria Core (lógica Booleana) com funções não-interpretadas (UF) e proíbe a utilização de quantificadores (QF).

Para isso, recorde o enigma do unicórnio que consideramos na aula sobre SAT:

- If the unicorn is mythical, then it is immortal.
- If the unicorn is not mythical, then it is a mortal mammal.
- If the unicorn is either immortal or a mammal, then it is horned.
- The unicorn is magical if it is horned.
- Is the unicorn magical? Is it horned? Is it mythical?

Resolvemos o problema com o auxílio de 5 variáveis proposicionais, correspondentes a 5 propriedades dos unicórnios. Também o podemos fazer com um SMT solver. O ficheiro `unicornpuzzle.smt2` no formato SMT-LIB 2, contém a descrição das restrições acima:

```
(set-logic QF_UF)

(declare-fun mythical () Bool)
(declare-fun immortal () Bool)
(declare-fun mammal () Bool)
(declare-fun horned () Bool)
(declare-fun magical () Bool)

(assert (=> mythical immortal))
(assert (=> (not mythical) (and (not immortal) mammal)))
(assert (=> (or immortal mammal) horned))
(assert (=> horned magical))
```

Note que não é necessária a conversão para CNF! Podemos utilizar directamente as conectivas `or`, `and`, `not`, e `=>` (sintaxe prefixa).

Temos agora várias perguntas a colocar sobre neste contexto:

- Is the unicorn magical? Is it horned? Is it mythical?

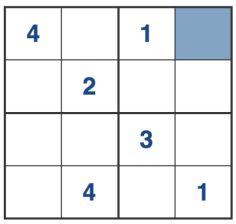
É muito usual queremos explorar vários problemas semelhantes que compartilham várias definições e asserções. Podemos usar os comandos `(push)` e `(pop)` para fazer isso. Os SMT solvers mantém uma stack global de declarações e asserções. O comando `(push)` cria um novo escopo, salvando o tamanho atual da stack. O comando `(pop)` remove qualquer afirmação ou declaração executada entre ele e o `(push)` correspondente. O comando `(check-sat)` opera no conteúdo da stack global.

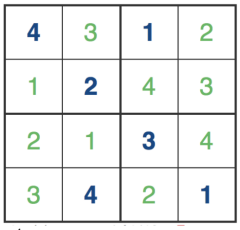
Analise o restante conteúdo do ficheiro `unicornpuzzle.smt2` e teste-o em diferentes SMT solvers (por exemplo, o `z3` e o `cvc4`).

4 Sudoku

Os puzzles Sudoku são problemas de colocação de números inteiros entre 1 e N^2 numa matriz quadrada de dimensão N^2, por forma a que cada coluna e cada linha contenha todos os números, sem repetições. Além disso, cada matriz contém N^2 sub-matrizes quadradas disjuntas, de dimensão N, que deverão também elas conter os números entre 1 e N^2.

Cada problema é dado por uma matriz parcialmente preenchida, cabendo ao jogador completá-la. Exemplo de um problema para $N = 2$, e uma possível solução:





O problema pode ser codificado através de um conjunto de N^4 constantes de tipo inteiro, correspondentes às posições da matriz, e escrevendo:

- $2 \times N^4$ desigualdades para os limites inferior e superior das constantes;
- N^2 restrições do tipo "todos diferentes", uma para cada linha da matriz;
- N^2 restrições do tipo "todos diferentes", uma para cada coluna da matriz;
- N^2 restrições do tipo "todos diferentes", uma para cada sub-matriz da matriz.

Acrescem ainda as restrições (igualdades) correspondentes à definição de um tabuleiro concreto.

1. Tendo isto em conta, complete a definição do notebook `sudoku.ipyhb` para criar um programa para resolver estes puzzles.

2. No ficheiro `sudoku.smt2` encontratrá uma implementação em SMT-LIB 2 incompleta do problema para $N = 2$ com a matriz dada acima como exemplo. Complete-a.

5 Manipulação de arrays

Uma teoria muito útil para a verificação de programas é a teoria de arrays funcionais, com extensionalidade. O mais comum é utilizá-la no contexto de uma lógica com aritmética (linear) inteira (necessária para as operações sobre os índices) e funções não interpretadas (AUFLIA ou, de preferência, QF_AUFLIA). Mas é também possível a combinação da teoria de arrays com a teoria de bitvectors ou de reais.

Os arrays funcionais são descritos com base em duas funções de escrita e leitura, `store` e `select`. A atribuição de um valor `v` à posição `i` de um array `a` é representada por um novo array `(store a i v)` (daí o nome funcional). O conteúdo do array resultado é igual ao primeiro, excepto na posição `i` que passa a conter o valor `v`.

O ponto essencial a ter em conta para captar o comportamento de um programa imperativo é que uma atribuição como `a[i] = x` terá de ser captado pela fórmula `(= a1 (store a0 i x))`. Ou seja, terão de ser utilizadas duas variáveis para o array, captando os estados anterior e posterior à atribuição, sendo o segundo dado por uma operação `store` sobre o primeiro.

Considere agora o programa sobre inteiros (sintaxe C):

```
x = a[i];
y = y + x;
a[i] = 5 + a[i];
a[i+1] = a[i-1] - 5;
```

Complete o ficheiro `arrays.smt2` por forma a estabelecer a validade das seguintes afirmações sobre o programa:

1. No final da execução, verifica-se a seguinte propriedade: `x + a[i-1] = a[i] + a[i+1]`.
2. No final da execução, a soma dos valores guardados em `a[i-1]` e `a[i]` é sempre positiva.
3. Se o valor inicial de `y` for inferior a 5, então no final da execução, o valor de `a[i]` é superior ao de `y`.

Sugestão: começe por codificar o programa. Depois faça uso dos comandos `(push)` e `(pop)` para ir colocando as perguntas sobre as suas propriedades, e tire as suas conclusões. No caso da propriedade não se verificar, analise a resposta do SMT solver e, com base nela, indique um contra-exemplo.

6 Teoria de bitvectors: o problema das N-rainhas

Uma das teorias mais úteis para a verificação de programas é a teoria FixedSizeBitVectors, que descreve vectores de bits de um comprimento arbitrário (mas fixo, dado à partida). O interesse desta teoria é a modelação de números inteiros tal como eles são de facto representados em máquina, ao invés da teoria matemática de números inteiros. Em particular, a aritmética de bitvectors é modular, captando perfeitamente o *overflow* típico da aritmética implementada em computador.

Note-se que um tratamento possível para os vectores de bits é simplesmente codificar cada vector de n bits através de um conjunto de n variáveis proposicionais. Este tratamento é conhecido por *bit-blasting*, e permite a utilização directa de um SAT solver, sem necessidade de qualquer procedimento de decisão para a teoria (as operações lógicas e aritméticas são descritas directamente por circuitos ao nível proposicional).

Vamos introduzir esta teoria num contexto diferente, o do problema das N rainhas. Recorde as restrições deste problema, relativo ao posicionamento de rainhas num tabuleiro de xadrez generalizado ($N \times N$):

- haverá no máximo uma rainha em cada linha, coluna, ou linha diagonal do tabuleiro;
- haverá pelo menos uma raínha em cada linha e em cada coluna do tabuleiro.

Vamos agora resolvê-lo com a ajuda de um SMT solver, recorrendo à teoria de vectores de bits (*bitvectors*). Utilizaremos a lógica QF_BV (*closed quantifier-free formulas over the theory of fixed-size bitvectors*), a mais simples contendo esta teoria. A ideia será representar cada tabuleiro por um conjunto de N vectores de N bits.

Para resolver o problema com $N = 4$ declaramos 4 constantes do tipo `(_ BitVec 4)`, como se segue (`4queens.smt2`):

```
(set-logic QF_BV)

; The 4 rows are represented by 4 bitvectors of length 4
(declare-fun r1 () (_ BitVec 4))
(declare-fun r2 () (_ BitVec 4))
(declare-fun r3 () (_ BitVec 4))
(declare-fun r4 () (_ BitVec 4))
```

`(_ BitVec n)` é o tipo de bitvectors cujo comprimento é `n`. As constantes podem ser definidas usando notação binária, decimal ou hexadecimal. Nos casos de notação binária ou hexadecimais, o tamanho bitvector é inferido a partir do número de caracteres. Por exemplo, o numeral 10 pode ser representado por: `#b01010` – bitvector de tamanho 5 em formato binário; `#x00a` – bitvector de tamanho 12 em formato hexadecimal; ou `(_ bv10 32)` – bitvector de tamanho 32 em formato decimal.

O problema das N-raínhas pode ser codificado de forma compacta tirando partido de algumas operações sobre bitvectors disponiveis, nomeadamente:

```
(bvand #b110 #b011)          ; bitwise and
(bvxor #x6 #x3)              ; bitwise xor
(bvsub #b00000111 #b0000011) ; subtraction
(bvshl #x07 #x03)            ; shift left
(bvlshr #xf0 #x03)           ; unsigned (logical) shift right
```

Na codificação das restrições tenha em atenção o seguinte:

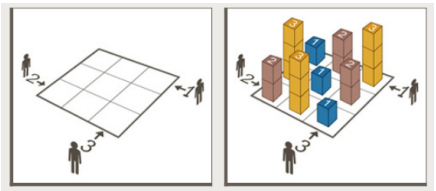
- podemos verificar se um bitvector r tem apenas um bit a 1, testando se a conjunção bitwise de r com $r-1$ retorna 0; isto pode ser útil para as restrições nas linhas;
- as restrições nas colunas podem ser feitas com o auxilio de um *xor*;
- para as restrições nas diagonais serão úteis as operações de shift.

1. Tendo em conta as explicações dadas acima, exprima as restrições necessárias para resolver o problema num tabuleiro 4×4.
2. Explore a biblioteca Z3Py para lidar com bitvectors e escreva um programa em Python resolver o problema das N-raínhas para tabuleiros de qualquer dimensão N.

7 Skyscrapers puzzle

O Skyscrapers é um puzzle lógico que tem por objectivo organizar arranha-céus num tabuleiro $(N \times N)$ de forma a que o seu horizonte seja visível de acordo com as pistas (números colocados nas bordas do tabuleiro que indicam quantos arranha-céus é possível ver daquela posição). Adicionalmente, exige-se que:

- Todos os arranha-céus têm altura entre 1 e N.
- Não podem existir arranha-céus da igual altura numa mesma coluna ou linha.
- O tabuleiro está inicialmente vazio



1. Tendo em conta as explicações dadas acima, exprima no ficheiro `skyscrapers.smt2` as restrições necessárias e resolva o problema para tabuleiros de dimensão 3×3, seguindo as seguintes sugestões:

   - Modele o problema na lógica QF_UFLIA (*quantifier-free linear integer arithmetic with uninterpreted sort and function symbols*).
   - Defina uma função lógica que recebe três argumentos (uma fila de arranha-céus) a_1, a_2 e a_3 e devolve o número de prédios visíveis quando olhamos assim: $\rightarrow a_1\ a_2\ a_3$
     Lembre-se que pode usar expressões `ite` (*if-then-else*). Por exemplo, pode definir a função que calcula o máximo de dois inteiros, assim:

     ```
     (define-fun maximo ((x Int) (y Int) Int)
        (ite (> x y) x y)  )
     ```

   - Depois de codificar as regras do puzzle, acrescente as restrições correspondentes à definição de um tabuleiro concreto (por exemplo, o da figura). Faça uso dos comandos `(push)` e `(pop)` para ir gerando soluções para vários tabuleiros.

2. Complete agora a notebook `skyscrapers.ipynb` com a implementação deste jogo em Python.

8 Scheduling

Resolva o problema de *scheduling*, apresentado nos slides das aulas teóricas, com o auxílio de um SMT solver e usando a lógica que entender adquada.

Modele o problema relatado nos slides e faça uso dos comandos `push` e `pop` para ir colocando as seguintes questões:

- Podemos fazer todos os trabalhos com max = 10?
- Ainda é possível fazer todos os trabalhos em menos de 10 unidades de tempo?
- Ainda é possível em 8 unidades de tempo?
- Ainda é possível em menos de 8 unidades de tempo?