Trabalho 2 - Especificação Formal de Software

Carlos Damasceno / Delcio da Silva – Universidade de São Paulo 02/10/2015

Problema 1

Escreva uma função triang que recebe três valores reais positivos e retorna um inteiro que representa se os valores formam um triângulo e qual o seu tipo. Deve retornar -1 se não for um triângulo válido, 0 se for equilátero, 1 se for isósceles e 2 se for escaleno. Use o z3 para provar que a função está correta.

Algorítmo 1: Função **triang** que dado três valores retorna o tipo de triângulo.

```
1    public static int triang(double a, double b, double c) {
2        if  (
3            !(((a+b)>c)||((b+c)>a)||((c+a)>b)||(a>0)||(b>0)||(c>0))
4            ){
5            return -1; // not valid
6        }else if    (
7                ((a == b) && (b ==c)) // equilateral
8                ){
9            return 0;
10       }else if    (
11               && ((a == b) ^ (a == c) ^ (b == c)) // isosceles
12               ){
13           return 1;
14       }else {
15           return 2; // scalene
16       }
17       return -1;
18   }
```

A partir do programa apresentado no Algorítmo 1, foi especificado o seguinte código em Z3 que prova a sua corretude. O código em Z3 que prova a corretude do Algorítmo 1 pode ser visualizado em Algorítmo 2. Entre as linhas 5-8 temos a assertiva que verifica casos onde um triângulo inválido é recebido (valores *não positivos* e onde *a soma de dois lados não é maior que o terceiro*). Entre as linhas 9-13 temos definida a assertiva que verifica triângulos equiláteros, ou seja, casos onde o triângulo é válido (linhas 10-11) e tem todos os lados iguais (linha 13). Entre as linhas 14-19 temos a assertiva responsável por verificar se o triângulo é válido, não equilátero e isósceles. Entre as linhas 20-25 temos a assertiva que verifica a validade do triângulo e a negação dos demais tipos de triângulos, restando apenas o caso

onde todos os lados são diferentes, ou seja escaleno.

Algorítmo 2: Especificação formal em Z3 da função **triang**.

```
1  (declare-fun a () Real)
2  (declare-fun b () Real)
3  (declare-fun c () Real)
4  (declare-fun triang () Int)
5  (assert (or (and
6                 (> (+ a b) c) (> (+ b c) a) (> (+ a c) b)
7                 (> a 0) (> b 0) (> c 0))
8            (= triang -1))) ; if( not valid ) return -1
9  (assert (or (not (and
10           (> (+ a b) c) (> (+ b c) a) (> (+ a c) b)
11           (> a 0) (> b 0) (> c 0)))
12           (not (and (= a b) (= a c))) ;; a==b && b==c (equilateral)
13           (= triang 0))) ; if( equilateral ) return 0
14 (assert (or (not(and
15           (> (+ a b) c) (> (+ b c) a) (> (+ a c) b)
16           (> a 0) (> b 0) (> c 0)))
17           (and (= a b) (= a c)) ;; a==b && b==c (equilateral)
18           (not (xor (= a b) (= a c) (= b c))) ;; (isosceles)
19           (= triang 1)))  ; if( isosceles ) return 1
20 (assert (or (not(and
21           (> (+ a b) c) (> (+ b c) a) (> (+ a c) b)
22           (> a 0) (> b 0) (> c 0)))
23           (and (= a b) (= a c)) ;; a==b && b==c (equilateral)
24           (xor (= a b) (= a c) (= b c)) ;; (scalene)
25           (= triang 2)))  ; if( scalene ) return 2
26 ;(assert (and (not (= a b c)) (= triang 0))) ; equilateral
27 ;(assert (and
28 ;    (not (xor (= a b) (= a c) (= b c)))
29 ;    (= triang 1) )) ; isosceles
30 ;(assert (and (or (= a b) (= a c) (= b c)) (= triang 2) )) ; scalene
31 (check-sat)
32 (get-model)
```

Nas linhas 26, 27-29 e 30 temos, respectivamente as assertivas que verificam se um triângulo é equilátero, isósceles ou escaleno. E por fim, nas linhas 31 e 32 temos a chamada das operações de checagem de satisfabilidade e visualização do modelo, se houver.

A seguir, são apresentadas as entradas e saídas para a especificação formal em Z3. Para as entradas de valores de a, b e sendo iguais a 0, a saída é um sat com retorno da função com valor -1, isto é, não existe um triângulo.

Algorítmo 3: Valores de a,b e c que são inválidos para formar um triângulo

```
1 ...
2 (assert (= a 3))
3 (assert (= b 4))
4 (assert (= c 7))
5 ...
```

Algorítmo 4: Saida do Z3

```
1  sat
2  (model
3    (define-fun triang () Int
4    - 1)
5    (define-fun c () Real
6    7.0)
7    (define-fun b () Real
8    4.0)
9    (define-fun a () Real
10   3.0)
11 )
```

As provas das saídas da especificação em Z3 para sua respectiva entrada com valores de triângulos equilátero, isósceles e escaleno são apresentadas a seguir.

Algorítmo 5: Valores de a,b e c para Triangulo Equilatero

```
1 ...
2 (assert (= a 4))
3 (assert (= b 4))
4 (assert (= c 4))
5 ...
```

Algorítmo 6: Saida do Z3

```
1  sat
2  (model
3    (define-fun triang () Int
4    0)
5    (define-fun c () Real
6    4.0)
7    (define-fun b () Real
8    4.0)
9    (define-fun a () Real
10   4.0)
11 )
```

Algorítmo 7: Valores de a,b e c para Triangulo Isosceles

```
1 ...
2 (assert (= a 4))
3 (assert (= b 4))
4 (assert (= c 6))
5 ...
```

Algorítmo 8: Saida do Z3

```
1  sat
2  (model
3    (define-fun triang () Int
4    1)
5    (define-fun c () Real
6    6.0)
7    (define-fun b () Real
8    4.0)
9    (define-fun a () Real
10   4.0)
11 )
```

Algorítmo 9: Valores de a,b e c para Triangulo Escaleno

```
1 ...
2 (assert (= a 4))
3 (assert (= b 5))
4 (assert (= c 6))
5 ...
```

Algorítmo 10: Saida do Z3

```
1  sat
2  (model
3    (define-fun triang () Int
4    2)
5    (define-fun c () Real
6    6.0)
7    (define-fun b () Real
8    5.0)
9    (define-fun a () Real
10   4.0)
11 )
```

Problema 2

Escreva uma função que calcula o MDC de dois números usando o algoritmo de Euclides. Escreva outra função que recebe três números e calcula o MDC desses números usando a função anterior. Use o z3 para gerar casos de teste (ou seja, triplas de números) que satisfaça aos critérios:

a) Todos-nós

b) Todos-usos

Algorítmo 11: Funções que calculam MDC para dois e três valores inteiros usando algorítmo de euclides recursivo.

```
1     public int gcd(int a, int b, int c) {
2         return gcd(a,gcd(b,c));
3     }
4
5     public int gcd(int a, int b) {
6         if(b==0) {
7             return a;
8         }
9         else {
10            return gcd(b, a % b);
11        }
12    }
```

Os métodosem Java apresentado no Algorítmo 11 permitem calcular o MDC de dois e três números inteiros. Para gerar o MDC de três valores inteiros, o calcula o MDC de um dos três valores com o MDC dos dois valores restante. O MDC de um par de valores inteiros, por sua vez, é calculado recursivamente usando o método de Euclides.

Para gerar testes que atendam aos critérios *Todos-Usos* e *Todos-Nós*, o trecho do Algorítmo 11 onde a função *gcd(int a, int b)* é chamada dentro da função *gcd(int a, int b, int c)* (Linha 2 do Algorítmo 11) foi modelado como apresentado no Algorítmo 12.

Entre as linhas 10-19 do Algorítmo 12 foram definidas restrições que garantem que o teste gerado cobrirá o uso predicativo da variável 'b' na linha 6 e a linha 7 do Algorítmo 11. Entre as linhas 21-30 do Algorítmo 12 o mesmo procedimento anteriormente é feito, entretanto, a fim de garantir a cobertura da linha 10 do Algorítmo 11.

Em conjunto, as assertivas apresentadas anteriormente permitem atender os dois critérios de teste estabelecidos para a geração. Ao ser executado no Z3 o código acima gerou a tripla de dados de entrada $a = 0$, $b = 1$ e $c = 0$. Um trecho da saída obtida a partir da execução do código em Z3 em questão pode ser visto no Algorítmo 13

Algorítmo 12: Especificação Formal que permite gerar casos de teste que atendem *Todos-Usos* e *Todos-Nós*.

```
(declare-const a Int) ; a value
(declare-const b Int) ; b value
(declare-const c Int) ; c value
(declare-fun gcd (Int Int) Int)
(declare-fun gcd3 (Int Int Int) Int)

; gcd(a,b,c) == gcd(a,gcd(b,c))
(assert (= (gcd3 a b c) (gcd a (gcd b c)) ) )

; REFERS TO gcd(b,c)
(declare-const a1 Int)
(declare-const b1 Int)
(assert (and (= a1 b) (= b1 c)))
(declare-const ifb0 Bool)
(assert
 (ite (= b1 0)
  (and (= ifb0 true) (= a1 (gcd a1 b1)))
  (and (= ifb0 false) (= (gcd a1 b1) (gcd b1 (mod a1 b1))))))
(assert (= ifb0 true))  ; covers "return a" coverage

; REFERS TO gcd(a,gcd(b,c))
(declare-const a2 Int)
(declare-const b2 Int)
(assert (and (= a2 a) (= b2 (gcd a1 b1))))
(declare-const ifb1 Bool)
(assert
   (ite (= b2 0)
      (and (= ifb1 true) (= a2 (gcd a2 b2)))
      (and (= ifb1 false) (= (gcd a2 b2) (gcd b2 (mod a2 b2))))))
(assert (= ifb1 false)) ; covers "gcd(a,gcd(b,c))" coverage

(assert (and ; assures gcd is divisor
   (= 0 (mod b (gcd b c)))
   (= 0 (mod c (gcd b c)))
   (= 0 (mod a (gcd3 a b c)))
   (= 0 (mod b (gcd3 a b c)))
   (= 0 (mod c (gcd3 a b c)))))

(check-sat)
(get-model)
```

Algorítmo 13: Saída obtida a partir da execução da especificação em Z3 do Algorítmo 12.

```
sat
(model
  ...
  (define-fun b () Int
    1)
  (define-fun a () Int
    0)
  ...
  (define-fun c () Int
    0)
  ...
)
```

Considerações Finais

Os dados referentes ao trabalho podem ser encontrados no diretório *efs_trabalho02_z3* no seguinte link: `https://github.com/damascenodiego/formalSpecification_usp_2015/`. O código em Z3, além de estar disponível no diretório do github, também pode ser visualizado no seguinte *permalink* do rise4fun: `http://rise4fun.com/Z3/8oYr`