

Taller #4 - Ejecución Simbólica Dinámica usando Z3

Ingeniería de Software II

Alumno: Lucas Raposeiras
LU: 34/15

Ejercicio 1

a.

Input de Z3:

```
(declare-const x Bool) ; Declaramos la constante x de tipo Bool
(declare-const y Bool) ; Declaramos la constante y de tipo Bool

(assert (= (not (or x y)) (and (not x) (not y)))) ; Escribimos la fórmula lógica en la sintaxis de Z3

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
```

Output de Z3:

```
sat
```

Es decir, la fórmula $\neg(x \vee y) \equiv (\neg x \wedge \neg y)$ **es satisfacible**.

b.

Input de Z3:

```
(declare-const x Bool) ; Declaramos la constante x de tipo Bool
(declare-const y Bool) ; Declaramos la constante y de tipo Bool

(assert (= (and x y) (not (or (not x) (not y))))) ; Escribimos la fórmula lógica en la sintaxis de Z3

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
```

Output de Z3:

```
sat
```

Es decir, la fórmula $(x \wedge y) \equiv \neg(\neg x \vee \neg y)$ **es satisfacible**.

c.

Input de Z3:

```
(declare-const x Bool) ; Declaramos la constante x de tipo Bool
(declare-const y Bool) ; Declaramos la constante y de tipo Bool

(assert (= (not (and x y)) (not (and (not x) (not y))))) ; Escribimos la fórmula lógica en la sintaxis de Z3

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
```

Output de Z3:

```
sat
```

Es decir, la fórmula $\neg(x \wedge y) \equiv \neg(\neg x \wedge \neg y)$ **es satisfacible**.

Ejercicio 2

a.

Input de Z3:

```
(declare-const x Int) ; Declaramos la constante x de tipo Int
(declare-const y Int) ; Declaramos la constante y de tipo Int

(assert (= (+ (* 3 x) (* 2 y)) 36)) ; Escribimos la ecuación en la sintaxis de Z3

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
(get-model) ; Le pedimos a Z3 que nos devuelva el modelo que satisface lo pedido
```

Output de Z3:

```
sat
(model
  (define-fun y () Int
    0)
  (define-fun x () Int
    12)
)
```

Es decir, la ecuación $3x + 2y = 36$ **se satisface con $y = 0$, $x = 12$.**

b.

Input de Z3:

```
(declare-const x Int) ; Declaramos la constante x de tipo Int
(declare-const y Int) ; Declaramos la constante y de tipo Int

(assert (= (+ (* 5 x) (* 4 y)) 64)) ; Escribimos la ecuación en la sintaxis de Z3

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
(get-model) ; Le pedimos a Z3 que nos devuelva el modelo que satisface lo pedido
```

Output de Z3:

```
sat
(model
  (define-fun y () Int
    1)
  (define-fun x () Int
    12)
)
```

Es decir, la ecuación $5x + 4y = 64$ **se satisface con $y = 1$, $x = 12$.**

c.

Input de Z3:

```
(declare-const x Int) ; Declaramos la constante x de tipo Int
(declare-const y Int) ; Declaramos la constante y de tipo Int

(assert (= (* x y) 64)) ; Escribimos la ecuación en la sintaxis de Z3

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
(get-model) ; Le pedimos a Z3 que nos devuelva el modelo que satisface lo pedido
```

Output de Z3:

```
sat
(model
  (define-fun y () Int
    1)
  (define-fun x () Int
    64)
)
```

Es decir, la ecuación $5x + 4y = 64$ **se satisface con $y = 1$, $x = 64$.**

Ejercicio 3

Input de Z3:

```
(declare-const a1 Real) ; Declaramos la constante a1 de tipo Real
(declare-const a2 Real) ; Declaramos la constante a2 de tipo Real
(declare-const a3 Real) ; Declaramos la constante a3 de tipo Real

(assert (= a1 (mod 16 2))) ; Primer expresión
(assert (= a2 (/ 16 4))) ; Segunda expresión
(assert (= a3 (rem 16 5))) ; Tercera expresión

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
(get-model) ; Le pedimos a Z3 que nos devuelva el modelo que satisface lo pedido
```

Output de Z3:

```

sat
(model
  (define-fun a2 () Real
    4.0)
  (define-fun a1 () Real
    0.0)
  (define-fun a3 () Real
    1.0)
)

```

Es decir, los resultados de calcular las expresiones:

- 16 mod 2
- 16 dividido por 4
- El resto de la división entera de 16 por 5

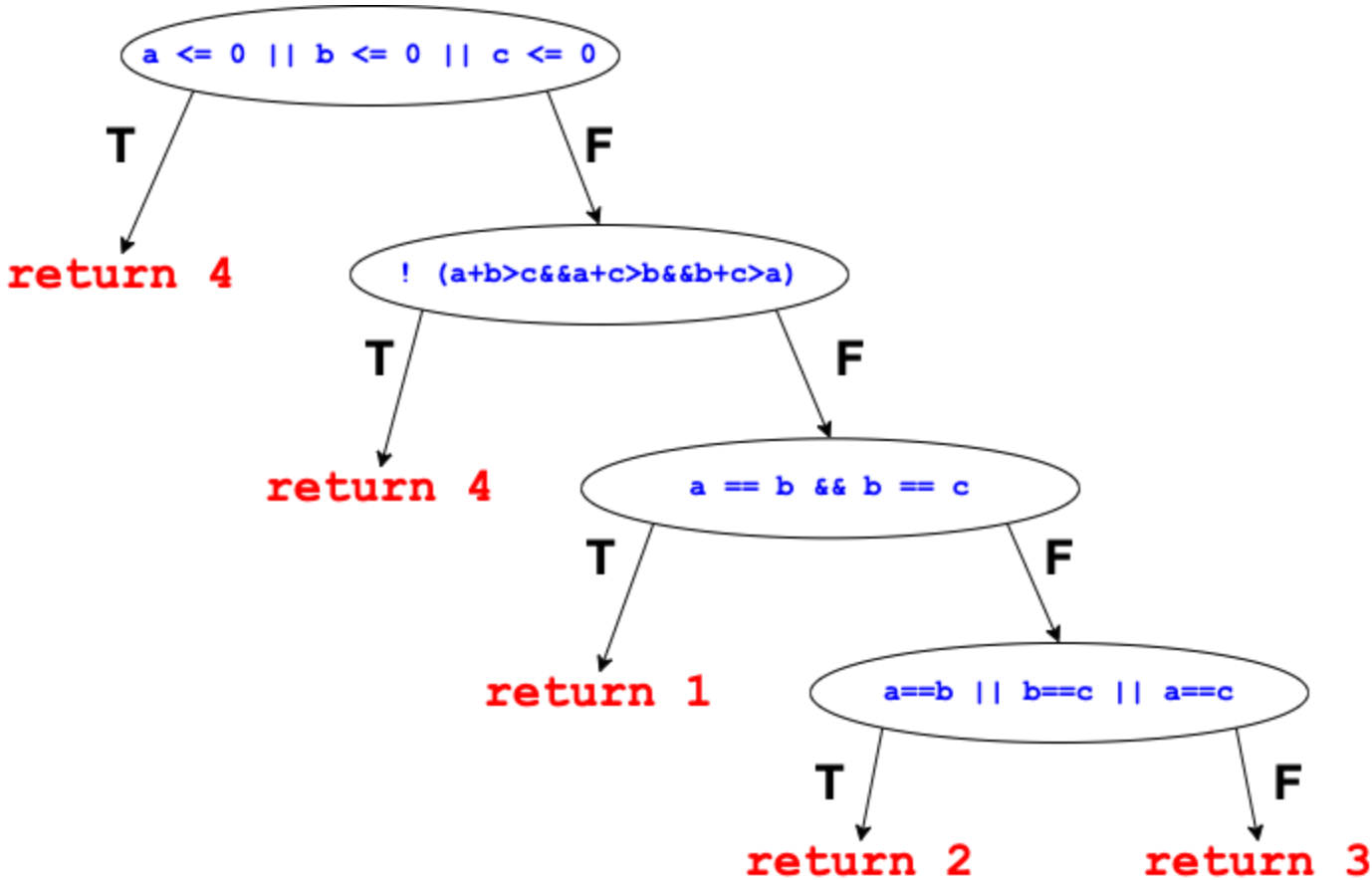
son:

- 0
- 4
- 1

respectivamente.

Ejercicio 4

a. Árbol de cómputo:



b. Al ejecutar Randoop durante dos segundos (utilizando las instrucciones de ejecución del Taller #3), se nos generan un total de 26 tests. Luego, si corremos el programa JaCoCo vemos que nos genera el siguiente reporte:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.autotest	<div><div></div></div>	90%	<div><div></div></div>	82%	3	13	1	10	0	2	0	1
Total	5 of 49	90%	4 of 22	82%	3	13	1	10	0	2	0	1

En el mismo podemos ver que el line coverage es de 9/10, es decir, un 90%. Si analizamos detalladamente el coverage de cada línea, vemos lo siguiente:

```

1. public int triangle(int a, int b, int c) {
2.     if (a <= 0 || b <= 0 || c <= 0) {
3.         return 4; // invalid
4.     }
5.     if (!(a + b > c && a + c > b && b + c > a)) {
6.         return 4; // invalid
7.     }
8.     if (a == b && b == c) {
9.         return 1; // equilateral
10.    }
11.    if (a == b || b == c || a == c) {
12.        return 2; // isosceles
13.    }
14.    return 3; // scalene
15. }

```

En la línea 7, la rama verdadera del if no se cumple en ningún test (por eso la línea 8 está pintada de rojo). Esto quiere decir que **Randoop no genera ningún test donde a, b y c sean iguales**. En la línea 10, la rama verdadera del if se cumple en algún test (por eso la línea 11 está pintada de verde), sin embargo está pintada de amarillo ya que Randoop no cubre las 6 combinaciones del condicional. Esto se puede explicar a que el tiempo de ejecución de Randoop es de tan solo dos segundos, los cuales no alcanzaron para generar un test suite lo suficientemente grande como para cubrir ese caso improbable.

c.

Iteración	Input Concreto	Condición de Ruta	Especificación para Z3	Resultado Z3
1	a = 0, b = 0, c = 0	$a_0 \leq 0 \mid \mid b_0 \leq 0 \mid \mid c_0 \leq 0$	(assert (not primer_if))	$a_0 = 1,$ $b_0 = 1,$ $c_0 = 1$
2	a = 1, b = 1, c = 1	$! (a_0 \leq 0 \mid \mid b_0 \leq 0 \mid \mid c_0 \leq 0) \ \&\&$ $! (! (a_0 + b_0 > c_0 \ \&\& a_0 + c_0 > b_0 \ \&\& b_0 + c_0 > a_0)) \ \&\&$ $(a_0 == b_0 \ \&\& b_0 == c_0)$	(assert (and (not primer_if) (and (not segundo_if) (not tercer_if))))	$a_0 = 2,$ $b_0 = 3,$ $c_0 = 4$
3	a = 2, b = 3, c = 4	$! (a_0 \leq 0 \mid \mid b_0 \leq 0 \mid \mid c_0 \leq 0) \ \&\&$ $! (! (a_0 + b_0 > c_0 \ \&\& a_0 + c_0 > b_0 \ \&\& b_0 + c_0 > a_0)) \ \&\&$ $! (a_0 == b_0 \ \&\& b_0 == c_0) \ \&\&$ $! (a_0 == b_0 \mid \mid b_0 == c_0 \mid \mid a_0 == c_0)$	(assert (and (not primer_if) (and (not segundo_if) (and (not tercer_if) cuarto_if))))	$a_0 = 2,$ $b_0 = 1,$ $c_0 = 2$
4	a = 2, b = 1, c = 2	$! (a_0 \leq 0 \mid \mid b_0 \leq 0 \mid \mid c_0 \leq 0) \ \&\&$ $! (! (a_0 + b_0 > c_0 \ \&\& a_0 + c_0 > b_0 \ \&\& b_0 + c_0 > a_0)) \ \&\&$ $! (a_0 == b_0 \ \&\& b_0 == c_0) \ \&\&$ $(a_0 == b_0 \mid \mid b_0 == c_0 \mid \mid a_0 == c_0)$	(assert (and (not primer_if) segundo_if))	$a_0 = 1,$ $b_0 = 1,$ $c_0 = 2$
5	a = 1, b = 1, c = 2	$! (a_0 \leq 0 \mid \mid b_0 \leq 0 \mid \mid c_0 \leq 0) \ \&\&$ $(! (a_0 + b_0 > c_0 \ \&\& a_0 + c_0 > b_0 \ \&\& b_0 + c_0 > a_0))$	END	END

Nota: las funciones primer_if, segundo_if y tercer_if son funciones auxiliares que definí para simplificar la especificación de Z3. A continuación detallo el programa completo de Z3 para este ejercicio:

```
(declare-const a Int); Declaramos la constante a de tipo Int
(declare-const b Int) ; Declaramos la constante b de tipo Int
(declare-const c Int) ; Declaramos la constante c de tipo Int

(define-fun primer_if () Bool ; Declaramos la función auxiliar primer_if
  (or (<= a 0) (or (<= b 0) (<= c 0)))
)

(define-fun segundo_if () Bool ; Declaramos la función auxiliar segundo_if
  (not (and (> (+ a b) c) (and (> (+ a c) b) (> (+ b c) a))))
)

(define-fun tercer_if () Bool ; Declaramos la función auxiliar tercer_if
  (and (= a b) (= b c))
)

(define-fun cuarto_if () Bool ; Declaramos la función auxiliar cuarto_if
  (or (= a b) (or (= b c) (= a c)))
)

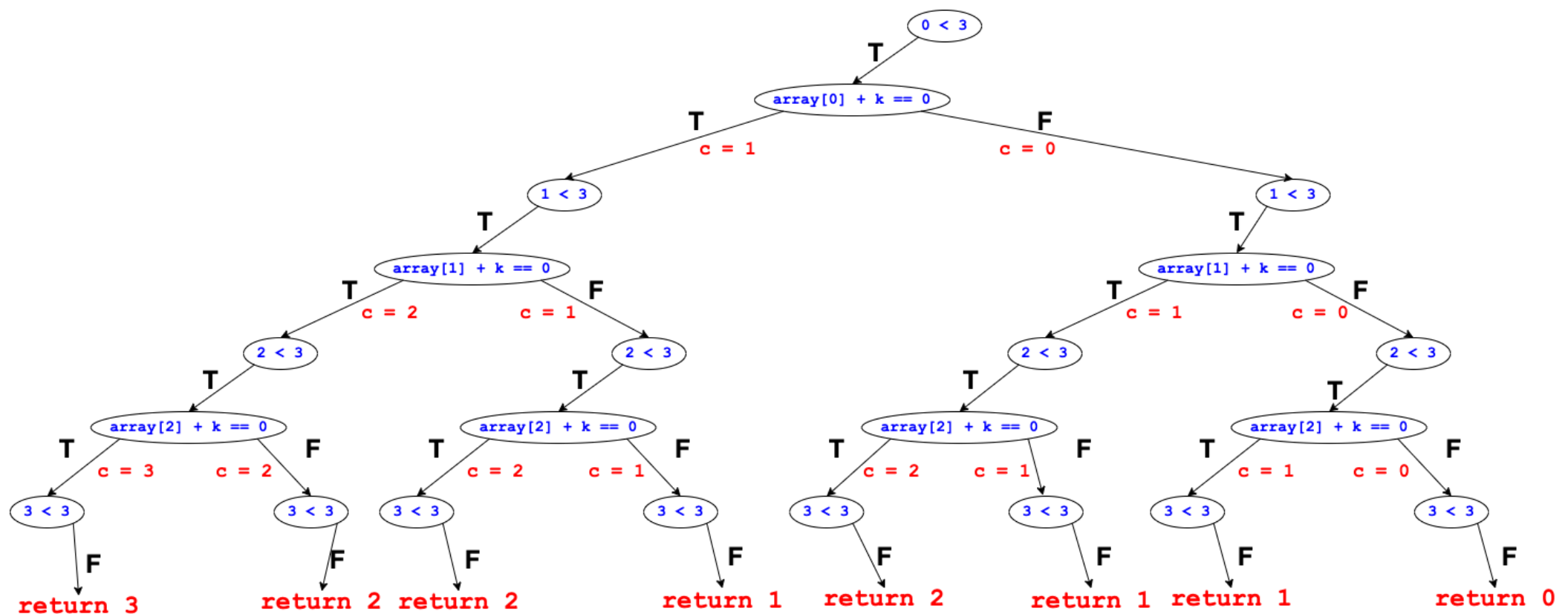
< Aquí insertamos el assert de la columna “Especificación para Z3”>

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
(get-model) ; Le pedimos a Z3 que nos devuelva el modelo que satisface lo pedido
```


d. El line coverage que se obtiene con el test suite generado en el ítem anterior es de 100% ya que hemos llegado a END.

Ejercicio 5

a. Árbol de cómputo:



b. Al ejecutar Randoop durante dos segundos, se nos generan un total de 28 tests. Luego, si corremos el programa JaCoCo vemos que nos genera el siguiente reporte:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 org.autotest	<div><div></div></div>	100%	<div><div></div></div>	100%	0	4	0	7	0	2	0	1
Total	0 of 38	100%	0 of 4	100%	0	4	0	7	0	2	0	1

En el mismo podemos ver que el line coverage es de 7/7, es decir, un 100%. Esto se debe a que Randoop genera tests que sólo varían en el valor de k. Toda línea de código que no dependa de k debería ser ejecutada en cualquier test. La **única** línea que depende de k para ser ejecutada es la línea correspondiente a la rama verdadera de `if (array[i] + k == 0)`. Para cubrir esa rama, se requiere que k valga -5.0, -1.0 o -3.0. En dos segundos, Randoop logró generar un test que instancia a k en -1.0, con lo cual se cubre esa rama.

C.

Iteración	Input Concreto	Condición de Ruta	Especificación para Z3	Resultado Z3
1	k = 0.0	! (5.0 + k ₀ == 0) && ! (1.0 + k ₀ == 0) && ! (3.0 + k ₀ == 0)	(assert (and (not cinco_mas_k_igual_cero) (and (not uno_mas_k_igual_cero) tres_mas_k_igual_cero)))	k = -3.0
2	k = -3.0	! (5.0 + k ₀ == 0) && ! (1.0 + k ₀ == 0) && (3.0 + k ₀ == 0)	(assert (and (not cinco_mas_k_igual_cero) uno_mas_k_igual_cero))	k = -1.0
3	k = -1.0	! (5.0 + k ₀ == 0) && (1.0 + k ₀ == 0) && (3.0 + k ₀ == 0)	(assert (not (not cinco_mas_k_igual_cero)))	k = -5.0
4	k = -5.0	(5.0 + k ₀ == 0) && ! (1.0 + k ₀ == 0) && ! (3.0 + k ₀ == 0)	END	END

Nota: las funciones cinco_mas_k_igual_cero, uno_mas_k_igual_cero y tres_mas_k_igual_cero son funciones auxiliares que definí para simplificar la especificación de Z3. A continuación detallo el programa completo de Z3 para este ejercicio:

```
(declare-const k Real) ; Declaramos la constante k de tipo Real

(define-fun k_mas_j_igual_cero ((j Real)) Real ; Declaramos la función auxiliar k_mas_j_igual_cero
  (= (+ j k) 0)
)

(define-fun cinco_mas_k_igual_cero () Real ; Declaramos la función auxiliar cinco_mas_k_igual_cero
  (k_mas_j_igual_cero 5.0)
)

(define-fun uno_mas_k_igual_cero () Real ; Declaramos la función auxiliar uno_mas_k_igual_cero
  (k_mas_j_igual_cero 1.0)
)
```

```
(define-fun tres_mas_k_igual_cero () Real ; Declaramos la función auxiliar tres_mas_k_igual_cero
  (k_mas_j_igual_cero 3.0)
)

< Aquí insertamos el assert de la columna “Especificación para Z3”>

(check-sat) ; Le pedimos a Z3 que verifique la satisfacibilidad
(get-model) ; Le pedimos a Z3 que nos devuelva el modelo que satisface lo pedido
```

- d. El line coverage que se obtiene con el test suite generado en el ítem anterior es de 100% ya que hemos llegado a END.