DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Taller Nº 4

Ejecución Simbólica Dinámica usando Z3

Ingeniería de Software II
Segundo cuatrimestre de 2020

Integrante	LU	Correo electrónico
Manuel Mena	313/14	manuelmena1993@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Introducción

En este taller nos enfocaremos en la técnica de Dynamic Symbolic Execution, la cual tiene como objetivo testar el código mezclando las técnicas de Symbolic Execution con Random Testing, con el fin de obtener lo mejor de ambas técnicas y cubrir todos los caminos posibles.

Del lado de Symbolic Execution, toma la idea de ejecutar con valores simbólicos para poder vincularlos con las condiciones para tratar de tomar ambos caminos. En cada condicional, se almacena el camino tomado y se agrega a un conjunto de condiciones de ruta, para luego resolver con un *Theorem Prover* los condicionales necesarios para tomar otra ruta y así recorrer mayor parte del código. Del lado de random testing, toma valores elegidos de manera aleatoria para iniciar la ejecución y va guardando valores concretos para poder tomar las distintas rutas del programa. Usando esto y un *Theorem Prover* incompleto, es que esta técnica logra llegar a rutas que usando las otras técnicas, debido a sus debilidades, son más difíciles explorar.

1.

1.1.

```
(declare-const x Bool)
(declare-const y Bool)
(assert (= (not (or x y)) (and (not x) (not y))))
(check-sat)
```

1.2.

```
(declare-const x Bool)
(declare-const y Bool)
(assert (=  (and x y) (not (or (not x) (not y)))))
(check-sat)
```

1.3.

```
(declare-const x Bool)
(declare-const y Bool)
(assert (=  (not (and x y)) (not (and (not x) (not y)))))
(check-sat)
```

Si añadimos el comando $(get - model)$ obtenemos valores posibles para satisfacer las formulas. En los dos primeros casos la respuesta está vacia ya que al tratarse de tautologías, se satisfacirán las formulas para cualquier valor de x e y. Pero para el tercer punto, Z3 nos indica que deben ser $x = false$, $y = true$.

2.

2.1.

```
(declare-const x Int)
(declare-const y Int)
(assert (= (+ (* 3 x) (* 2 y)) 36))
(check-sat)
(get-model)
```

La formula se satisface con $x = 12$, $y = 0$.

2.2.

```
(declare-const x Int)
(declare-const y Int)
(assert (= (+ (* 5 x) (* 4 y)) 64))
(check-sat)
(get-model)
```

La formula se satisface con $x = 12$, $y = 1$.

2.3.

```
(declare-const x Int)
(declare-const y Int)
(assert (= (* x y) 64))
(check-sat)
(get-model)
```

La formula se satisface con $x = 64$, $y = 1$.

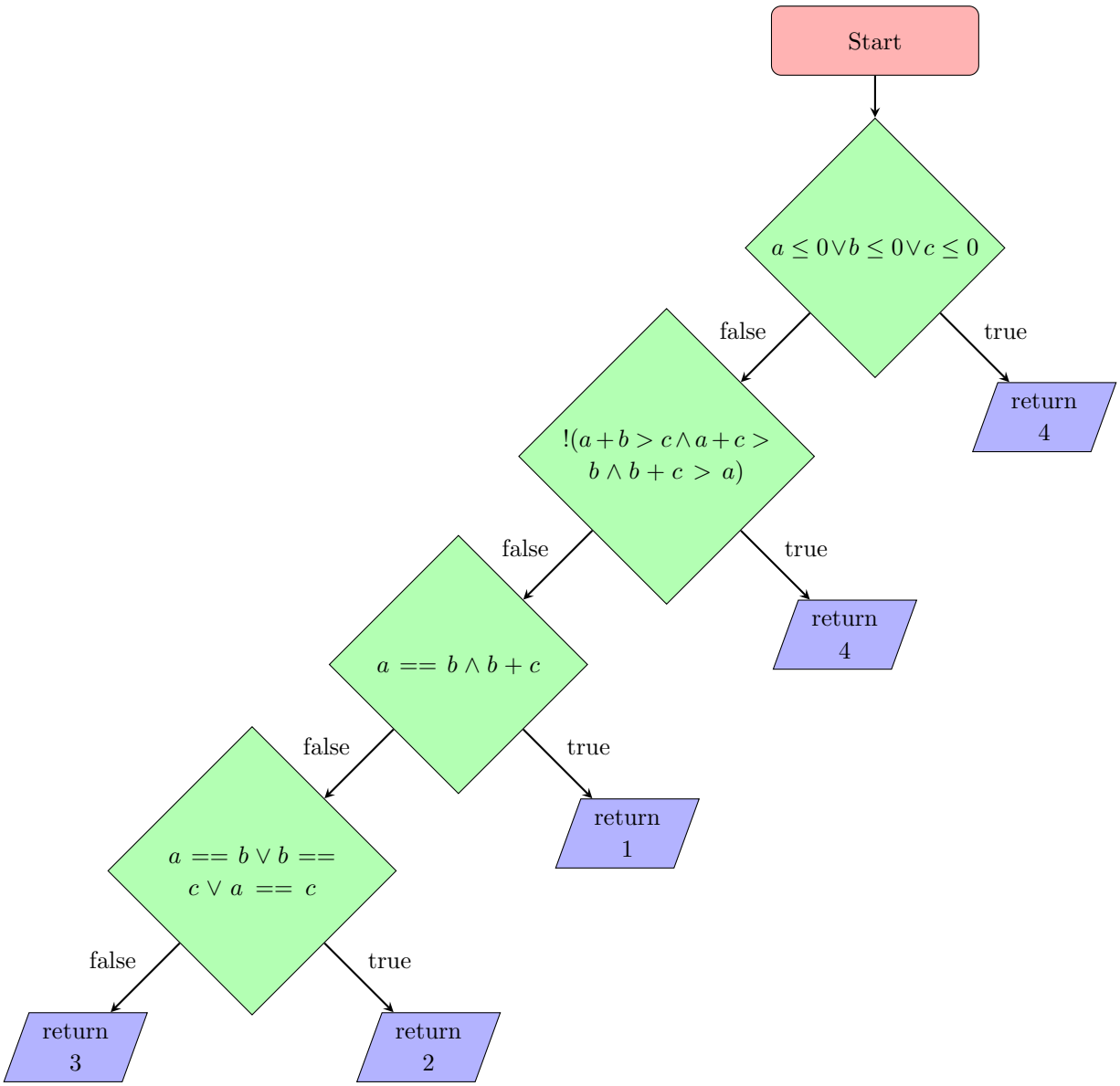
3.

```
(declare-const a1 Real)
(declare-const a2 Real)
(declare-const a3 Real)
(assert (= a1 (mod 16 2)))
(assert (= a2 (/ 16 4)))
(assert (= a3 (rem 16 5)))
(check-sat)
(get-model)
```

La salida de Z3 es $a_1 = 4{,}0$, $a_2 = 0{,}0$, $a_3 = 1{,}0$.

4.

4.1.



4.2.

El line coverage es de 100 %, no el branch coverage, que es de 91 %. Hay dos branches que no pudieron ser alcanzadas.

El compilador de java traduce las cadenas de && en ifs anidados y las cadenas de ||| en ifs consecutivos. Es importante tener esto en cuenta para entender por qué esas branches pueden no haber sido alcanzadas.

Las líneas donde jacoco reporta que faltó una rama en cada una fueron las de las ultimas dos condiciones de triangle. En $a == b \&\& b == c$ reporta que falto una rama de entre 4. Como el compilador transforma esta condición en

```
if (a == b) {
if (b == c) {
        return 1;
}
}
```

Jacoco muestra que llegó al return, por lo que el caso donde $a == b == c$ está cubierto, pero es muy probable que al haber puesto poco tiempo de ejecución, randoop no haya podido cubrir el caso cuando $a! = b$ o $b! = c$, dado que los valores son aleatorios, y teniendo en cuenta que por mas que en la mayor parte los casos es de esperarse que si pueda generar a, b y c distintos, para llegar a ese punto en la ejecución debe haber pasado las dos primeras condiciones antes, lo cual reduce enormemente la probabilidad de encontrar los valores para ese caso.

Para la cuarta condición jacoco indica que falta una branch de entre 6. El compilador transforma $(a == b||||b == c||||a == c)$ en

```
if (a==b) {
return 2;
}
if (b==c) {
return 2;
}
if (a==c) {
return 2;
}
```

por lo que al igual con la condición anterior, la probabilidad de que randoop no haya podido generar alguno de los casos, por ejemplo que $a! = c$, ya habiendo evadido las condiciones anteriores, es muy alta.

4.3.

Cuadro 1: Tabla ejercicio 4c

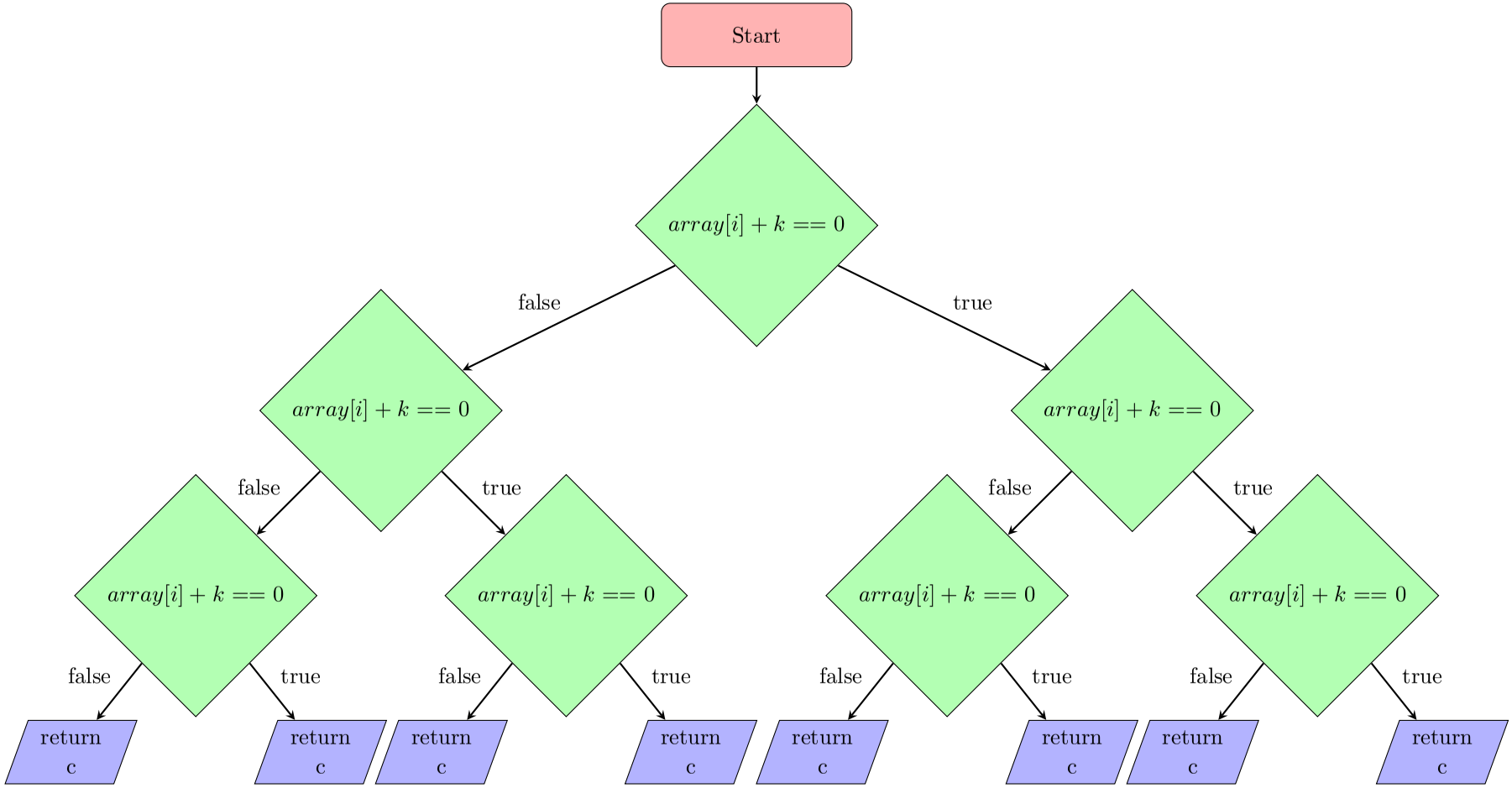
Iteracion	Input concreto	Condicion de ruta	Escpecificacion para Z3	Resultado Z3
1	a = 0, b = 0, c = 0	a <= 0 ∨ b <= 0 ∨ c <= 0	(assert (not (or (or (<= a 0) (<= b 0)) (<= c 0))))	a = 1, b = 1, c =1
2	a = 1, b = 1, c =1	!(a <= 0 ∨ b <= 0 ∨ c <= 0) && (a+b>c&&a+c>b∧b+c >a) ∧ (a == b ∧ b == c ∧ a == c)	(assert (not (or (or (<= a 0) (<= b 0)) (<= c 0)))) (assert (and (>(+ a b) c) (and (>(+ a c) b) (>(+ b c) a)))) (assert (not (and (= a b) (= b c))))	a = 2, b = 3, c = 4
3	a = 2, b = 3, c = 4	!(a <= 0 ∨ b <= 0 ∨ c <= 0) ∧ (a+b>c∧a+c>b∧b+c >a) ∧ !(a == b ∧ b == c ∧ a == c) ∧ !(a == b ∨ b == c ∨ a == c)	(assert (not (or (or (<= a 0) (<= b 0)) (<= c 0)))) (assert (and (>(+ a b) c) (and (>(+ a c) b) (>(+ b c) a)))) (assert (not (and (= a b) (= b c)))) (assert (or (or (= a b) (= b c)) (= a c)))	a = 2, b = 1, c = 2
4	a = 2, b = 1, c = 2	!(a <= 0 ∨ b <= 0 ∨ c <= 0) ∧ (a+b>c∧a+c>b∧b+c >a) ∧ !(a == b ∧ b == c ∧ a == c) ∧ (a == b ∨ b == c ∨ a == c)	(assert (not (or (or (<= a 0) (<= b 0)) (<= c 0)))) (assert (not (and (>(+ a b) c) (and (>(+ a c) b) (>(+ b c) a)))))	a = 1, b = 1, c = 2
5	a = 1, b = 1, c = 2	!(a <= 0 ∨ b <= 0 ∨ c <= 0) ∧ !(a+b>c∧a+c>b∧b+c >a)	END	END

4.4.

Jacoco indica una covertura del 100 % de las líneas, pero una caída a 68 % de las branches cubiertas. Lo que sucede es que si bien con DSE se cubren todos los paths posibles, hay muchos branches distintos que son parte de paths equivalentes. Esto es debido a la forma de separar las condiciones en ifs distintos del compilador de java.

5.

5.1.


Start
array[i] + k == 0
false
true
array[i] + k == 0
array[i] + k == 0
false
true
false
true
array[i] + k == 0
array[i] + k == 0
array[i] + k == 0
array[i] + k == 0
false
true
false
true
false
true
false
true
return c
return c
return c
return c
return c
return c
return c
return c

5.2.

La cobertura es de 100 % tanto de lineas como de branches. A diferencia del ejercicio anterior, en este es muy probable poder cubrir todas las lineas ya que solamente se requiere cumplir la condición una vez. Como hay una única condición, cubrir todas las branches se reduce a poder entrar o no en ella.

Lo que no va a poder cubrirse son todos los paths, ya que solo puede cumplirse la condición una unica vez debido a que ni k ni $array$ cambian. Hay muchos paths que no van a poder recorrerse.

5.3.

Cuadro 2: Tabla de ejercicio 5.c

Iteración	Input concreto	Condición de ruta	Especificación de ruta	Resultado Z3
1	k = 0.0	5 + k != 0 && 1 + k != 0 && 3 + k != 0	(assert (not (= (+ 5 k) 0))) (assert (not (= (+ 1 k) 0))) (assert (= (+ 3 k) 0))	k = -3.0
2	k = -3.0	5 + k != 0 && 1 + k != 0 && 3 + k == 0	(assert (not (= (+ 5 k) 0))) (assert (= (+ 1 k) 0)) (assert (not (= (+ 3 k) 0)))	k = -1.0
3	k = -1.0	5 + k != 0 && 1 + k == 0 && 3 + k != 0	(assert (not (= (+ 5 k) 0))) (assert (= (+ 1 k) 0)) (assert (= (+ 3 k) 0))	UNSAT
4			(assert (= (+ 5 k) 0)) (assert (not (= (+ 1 k) 0))) (assert (not (= (+ 3 k) 0)))	k = -5.0
5	k = -5.0	5 + k == 0 && 1 + k != 0 && 3 + k != 0	(assert (= (+ 5 k) 0)) (assert (not (= (+ 1 k) 0))) (assert (= (+ 3 k) 0))	UNSAT
6			(assert (= (+ 5 k) 0)) (assert (= (+ 1 k) 0)) (assert (not (= (+ 3 k) 0)))	UNSAT
7			(assert (= (+ 5 k) 0)) (assert (= (+ 1 k) 0)) (assert (not (= (+ 3 k) 0)))	UNSAT
8			END	END

5.4.

Jacoco indica una cobertura de 100 % al igual que con los tests de randoop. Por como está escrito el código, con uno alcanzar dos de los paths posibles se puede garantizar la cobertura total de la función.