Ingenierá del Software II

Taller #3 – Ejecución Simbólica Dinámica usando Z3

Deadline: 13 de mayo de 2021 a las 23:59 hs

Z3 Solver

Z3 es un demostrador de teoremas moderno de Microsoft Research. Puede ser usado para verificar la satisfactibilidad de fórmulas lógicas sobre una o más teorías.

¿Cómo escribo fórmulas lógicas para Z3? El formato de entrada de Z3 es una extensión del formato definido por el estándar SMT-LIB 2.0 (`http://smtlib.cs.uiowa.edu/language.shtml`).

- El comando `declare-const` declara una constante de un tipo dado.
- El comando `declare-fun` declara una función.
- El comando `assert` agrega un axioma al conjunto de axiomas de Z3.

Un conjunto de fórmulas es satisfactible si existe una interpretación (para las constantes y funciones declaradas por el usuario) que hace verdaderas a todas las fórmulas asertadas. Cuando el comando `check-sat` devuelve sat, el comando `get-model` puede ser usado para conseguir una interpretación (i.e. valuación) que hace verdaderas a todas las fórmulas escritas.

Z3 está disponible para Windows, OSX, Linux (Ubuntu, Debian) y FreeBSD y su código fuente se puede obtener de `https://github.com/Z3Prover/z3`.

Su última versión se puede descargar de `https://github.com/Z3Prover/z3/releases/tag/z3-4.8.10`. Sin embargo, para este taller recomendamos utilizar la interfaz web disponible en `http://www.rise4fun.com/z3`, la cual permite cargar fórmulas y resolverlas directamente online sin tener que descargarlo.

Ejercicio 1

Podemos comprobar aserciones de lógica proposicional en Z3 usando funciones que representan las proposiciones x e y. Por ejemplo, la siguiente especificación comprueba si existen al menos un par de valores booleanos de x, y tales que $\neg(x \wedge y) \equiv (\neg x \vee \neg y)$:

```
; ejemplo1
(declare-const x Bool)
(declare-const y Bool)
(assert ( = (not(and x y)) (or (not x)(not y))))
(check-sat)
```

Si copiamos y pegamos la especificación en la interfaz web de Z3 y apretamos el botón **Play**, Z3 intentará encontrar un par de valores booleanos tales que hagan verdadera la fórmula $\neg(x \wedge y) \equiv (\neg x \vee \neg y)$. Dado que tales valores existen y Z3 es lo suficientemente inteligente para encontrarlos, Z3 retorna **sat** (i.e. statisfacible).

Del mismo modo, si deseamos ejecutar Z3 por línea de comando usamos la opción `-file` para indicar el archivo donde se ubica la especificación Z3 que queremos analizar (por ejemplo, `ejemplo1.smt`:

```
% bin/z3 -file smt/ejemplo1.smt
sat
```

En cambio, si buscamos un par de valores x, y tales que $x = true \wedge y = false \wedge x = y$ usando la siguiente especificación:

```
; ejemplo2
(declare-const x Bool)
(declare-const y Bool)
(assert ( = x true))
(assert ( = y false))
(assert ( = x y))
(check-sat)
```

Al apretar **Play**, Z3 concluirá que no existen valores de x, y que puedan satisfacer esa fórmula, y por lo tanto devolverá **unsat** (i.e. insatisfacible). Del mismo modo, podemos ejecutar Z3 por línea de comando y retornará el mismo resultado:

```
% bin/z3 -file smt/ejemplo2.smt
unsat
```

Debido a que la verificación de algunas especificiaciones es indecidible, Z3 puede también devolver **unknown** cuando su procedimiento de decisión no es lo suficientemente poderoso para determinar si una fórmula es satisfactible o no.

Resolver

Ejecutar Z3 para comprobar si las siguientes fórmulas de la lógica proposicional son satisfacibles (i.e. si existe al menos un par de valores de x e y que las haga verdaderas).

a. $\neg(x \vee y) \equiv (\neg x \wedge \neg y)$

b. $(x \wedge y) \equiv \neg(\neg x \vee \neg y)$

c. $\neg(x \wedge y) \equiv \neg(\neg x \wedge \neg y)$

- Adjuntar una especificación Z3 para comprobar la satisfacibilidad de cada una de ellas. Nombrar a los archivos `ejercicio1a.smt`, `ejercicio1b.smt` y `ejercicio1c.smt`.
- Indicar el resultado encontrado por Z3 (i.e. **sat**, **unsat**, o **unknown**) para cada una de ellas.

Ejercicio 2

Además de booleanos, Z3 puede analizar la satisfacibilidad de fórmulas con constantes y funciones con números enteros. Por ejemplo, si escribimos la siguiente especificación y apretamos **Play**:

```
; ejemplo3
(declare-const x Int)
(declare-const y Int)
(assert (= (+ x y) 10))
(assert (= (+ x (* 2 y)) 20))
(check-sat)
```

Z3 responderá que la fórmula es **sat** ya que encontró valores enteros para x e y tales que $x + y = 10 \wedge x + 2 * y = 20$. En particular, podemos pedirle a Z3 que nos diga cuáles son estos valores si agregamos debajo del comando `check-sat` el comando `get-model` de la siguiente forma:

```
; ejemplo4
(declare-const x Int)
(declare-const y Int)
(assert (= (+ x y) 10))
(assert (= (+ x (* 2 y)) 20))
(check-sat)
(get-model)
```

Ahora, cuando volvemos a apretar **Play**, no sólo reporta `sat`, sino también:

```
sat
(model
  (define-fun y () Int
    10)
  (define-fun x () Int
    0)
)
```

Esto nos está diciendo que la valuación que encontró Z3 que hace verdadera a la fórmula fue $x = 0$, $y = 10$.

Observación: Z3 internamente trata todas las constantes como funciones sin argumentos, por lo tanto, la constante x se convierte en la función $x()$ sin argumentos. Cualquier constante puede ser reescrita usando funciones sin argumentos. Por ejemplo, la siguiente especificación es equivalente a la anterior:

```
; ejemplo5
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= (+ x y) 10))
(assert (= (+ x (* 2 y)) 20))
(check-sat)
(get-model)
```

Resolver

Usando Z3, encontrar una solución para x e y en las siguientes ecuaciones:

a. $3x + 2y = 36$

b. $5x + 4y = 64$

c. $x * y = 64$

- Adjuntar una especificación Z3 para comprobar la satisfacibilidad de cada una de ellas. Nombrar a los archivos `ejercicio2a.smt`, `ejercicio2b.smt` y `ejercicio2c.smt`.
- Indicar el resultado encontrado por Z3 (i.e. **sat**, **unsat**, o **unknown**) para cada una de ellas. En caso de ser **sat**, indicar los valores de x e y reportados por Z3.

Ejercicio 3

Z3 también soporta la división, y los operadores división entera, módulo y resto. Por ejemplo, dada la siguiente especificación de una fórmula:

```
; ejemplo6
(declare-const a Int)
(declare-const r1 Int)
(declare-const r2 Int)
(declare-const r3 Int)
(declare-const b Real)
(declare-const c Real)
(assert (= a 10))
(assert (= r1 (div a 4))); integer division
(assert (= r2 (mod a 4))); mod
(assert (= r3 (rem a 4)));remainder
(assert (>= b (/ c 3.0)))
(assert (>= c 20.0))
(check-sat)
(get-model)
```

Z3 indica que la fórmula es satisfacible y existe la siguiente valuación para cada una de sus constantes, indicando que $c = 20.0$, $b = \frac{20}{3}$, $r3 = 2$, $r2 = 2$, $r1 = 2$, $a = 10$.

```
sat
(model
  (define-fun c () Real
    20.0)
  (define-fun b () Real
    (/ 20.0 3.0))
  (define-fun r3 () Int
    2)
  (define-fun r2 () Int
    2)
  (define-fun r1 () Int
    2)
  (define-fun a () Int
    10)
)
```

Resolver

Crear una **única** especifición Z3 que almacene en las constantes **reales** $a1$, $a2$, $a3$ el resultado de calcular las siguientes expresiones:

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

Adjuntar la especificación escrita en un archivo `ejercicio3.smt` e indicar la interpretación (i.e. la salida) de Z3 al analizar la especificación.

Dynamic Symbolic Execution (DSE)

Dynamic Symbolic Execution es una técnica que permite explorar el arbol de ejecución de un programa con el objetivo de encontrar errores en el mismo. Un ejemplo de su ejecución se muestra a continuación.

```
int f(int z) {
  if (z == 12) {
    return 1;
  } else {
    return 2;
  }
}
```

Iteración	Input Concreto	Condición de Ruta	Especificación para Z3	Resultado Z3
1	z=0	$z_0 \neq 12$	(assert (= z 12))	$z_0 = 12$
2	z=12	$z_0 = 12$	END	END

Ejercicio 4

Sea el siguiente programa `triangle` que clasifica lados de un triángulo:

```
int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }
    return 3;  // scalene
}
```

a. Completar la siguiente tabla con la ejecución simbólica dinámica del programa `triangle` de forma manual, indicando para cada iteración:

- El input concreto utilizado
- La condición de ruta (i.e. "path condition") que se produce de ejecutar el input concreto, asumiendo que el valor simbólico inicial es $a = a_0$, $b = b_0$, $c = c_0$.
- La especificación que se envía a Z3 de acuerdo al algoritmo de ejecución simbólica dinámica.
- El resultado que produjo Z3.

Iteración	Input Concreto	Condición de Ruta	Especificación para Z3	Resultado Z3
1	a=0, b=0, c=0	...	...	...
2	...	...	...	...
...	...	...	...	...

Observación: En caso de necesitarse más de una invocación a Z3 por iteración, agregar una nueva línea dejando en blanco el Input y la condición de ruta.

Recomendación: Poner nombres a cada condición y utilizarlos en la condición de ruta y especificación para Z3.

b. Sea el test suite compuesto por los tests generados usando ejecución simbólica dinámica en el punto anterior, ¿cuál es el branch coverage que obtiene el test suite sobre le programa `triangle`?

c. Dibujar el árbol de cómputo explorado con la ejecución simbólica dinámica.

Ejercicio 5

Sea el siguiente programa `magicFunction`:

```
int magicFunction(double k) {
  double[] array = { 5.0, 1.0, 3.0 };
  int c = 0;
  for (int i = 0; i < 3; i++) {
    if (array[i] + k == 0) {
      c++;
    }
  }
  return c;
}
```

a. Completar la siguiente tabla con la ejecución simbólica dinámica del programa `magicFunction` indicando para cada iteración. Seguir los mismos lineamientos que los presentados en el ejercicio anterior para cada columna de la tabla. Utilizar más de una línea en caso de ser necesario cuando haya más de un llamado a Z3.

Iteración	Input Concreto	Condición de Ruta	Especificación para Z3	Resultado Z3
1	k=0.0	...	...	...
2	...	...	...	...
...	...	...	...	...

Recomendación: Poner nombres a cada condición y utilizarlos en la condición de ruta y especificación para Z3.

b. Sea el test suite compuesto por los inputs generados únicamente con ejecución simbólica dinámica, ¿cuál es el branch coverage que obtiene el test suite generado?

c. Dibujar el árbol de cómputo explorado con la ejecución simbólica dinámica.

Formato de Entrega

El taller debe ser subido al campus. Debe ser un archivo zip con el siguiente contenido.

1. Un archivo `answers.pdf` con las respuestas a los ejercicios.
2. Los archivos adicionales solicitados en cada ejercicio.