Laboratorio de Programación Multiparadigma (LPM)

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

Práctica 4: Satisfacibilidad y la herramienta Z3 de Microsoft

Santiago Escobar

Julia Sapiña

Índice

1. Introducción

En esta parte vamos a describir la herramienta Z3 de Microsoft para comprobaciones de satisfacibilidad en un conjunto de desigualdades o inecuaciones en distintos dominios.

2. La herramienta Z3 de Microsoft

Z3 es un asistente para la prueba de teoremas y propiedades desarrollado por Microsoft Research. Sin embargo las pruebas se basan sólo en satisfacibilidad de soluciones, es decir, dado un conjunto de propiedades de un dominio específico devuelve sí, no, o no lo sé; indicado que las propiedades tienen al menos una solución, que las propiedades no tienen solución, o que no puede responder. Hay que tener en cuenta que siempre terminará su ejecución devolviendo sí, no, o no lo sé, por lo tanto no hay peligro de no-terminación.

Z3 se puede utilizar para comprobar la satisfacibilidad de formulas lógicas en una o más teorías (o dominios). Z3 está diseñado más como un componente de análisis y verificación para ser integrado en otras herramientas, por lo que su interfaz es muy simple y bastante engorrosa.

La herramienta Z3 está disponible online en la siguiente url: `https://rise4fun.com/Z3`

2.1. Comandos básicos

El lenguaje de entrada de Z3 es una extensión del estándar internacional SMT-LIB 2.0 definido para herramientas de satisfacibilidad.

El usuario introduce varias declaraciones del problema que quiere resolver usando dos comandos: `declare-const` y `declare-fun` para declarar constantes y funciones con argumentos, respectivamente. Por ejemplo, las siguientes instrucciones definen una constante a y una función binaria f que toma como argumentos un número entero y un booleano y devuelve un número entero.

```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
```

El usuario puede introducir tantas propiedades como desee verificar utilizando la palabra reservada `assert`. Por ejemplo podemos decir que la constante a definida anteriormente es mayor que 10 y que la función f aplicada sobre la constante a y la constante predefinda *true* devuelve un número entero menor que 100.

```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
(assert (> a 10))
(assert (< (f a true) 100))
```

Una vez hemos introducido todas las propiedades deseables, se comprueba si existe alguna solución usando la instrucción `check-sat`.

```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
(assert (> a 10))
(assert (< (f a true) 100))
(check-sat)
```

Z3 devuelve `sat` si existe una solución y `unsat` si no existe. En el caso de que exista al menos una solución, se puede solicitar a Z3 que proporcione una de las soluciones con el comando `get-model`.

```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
(assert (> a 10))
(assert (< (f a true) 100))
(check-sat)
(get-model)
```

La solución mostrada (cuyo término más correcto sería la interpretación mostrada) utiliza la misma sintaxis de entrada de Z3.

```
sat
(model
  (define-fun a () Int
    11)
  (define-fun f ((x!1 Int) (x!2 Bool)) Int
    (ite (and (= x!1 11) (= x!2 true)) 0
      0))
)
```

de forma que asigna el valor 11 a la constante a y la función f se define con una expresión `if-then-else`, que en Z3 se escribe `ite`, de tal forma que si el argumento entero de f, denominado `x!1`, es igual a 11 y el argumento booleano de f, denominado `x!2`, es *true*, entonces devuelve un 0 y, si las condiciones no son ciertas, se devuelve también un 0.

Los mecanismos para generar soluciones son bastante complejos y no vamos a explicarlos en esta práctica, pero hay que tener en cuenta que intentan devolver la solución más simple, aunque no sea la más obvia.

2.2. Aritmética

Z3 admite propiedades descritas para los enteros y reales. Estos dos tipos de datos no se deben confundir con los enteros a nivel máquina (con 32 ó 64 bits) ni con los números en punto fijo, sino que hacen referencia a los enteros y reales desde un punto de vista más abstracto o matemático. Por ejemplo, podemos declarar constantes de tipo entero y real.

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Real)
(declare-const e Real)
```

El usuario puede introducir expresiones que utilicen varios operadores aritméticos, como la suma ($+$), resta ($-$), menor ($<$), menor o igual ($<=$), mayor ($>$), mayor o igual ($>=$) e igualdad ($=$). El siguiente ejemplo define cuatro propiedades $a > (b + 2)$, $a = (2 * c) + 10$, $(c + b) \leq 1000$ y $d \geq e$. Todas se escriben en formato prefijo y no se admiten símbolos infijos o sufijos.

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Real)
(declare-const e Real)
(assert (> a (+ b 2)))
(assert (= a (+ (* 2 c) 10)))
(assert (<= (+ c b) 1000))
(assert (>= d e))
(check-sat)
(get-model)
```

2.3. Vectores

Otra de las teorías soportadas por Z3 es los vectores en notación extensional en vez de intensional, es decir caracterizados con las palabras `select` y `store` en vez de por las posiciones con sus valores. La expresión `select a i` denota devolver el elemento en la posición i del vector a. La expresión `store a i v` denota reemplazar el elemento del vector a en la posición i por el valor v.

Por ejemplo, podemos definir un vector $a1$ de enteros donde establecemos dos restricciones: que el elemento en la posición x de $a1$ debe ser justamente el valor x y que al reemplazar el elemento en $a1$ en la posición x por el valor y, el vector $a1$ se queda igual.

```
(declare-const x Int)
(declare-const y Int)
(declare-const a1 (Array Int Int))
(assert (= (select a1 x) x))
(assert (= (store a1 x y) a1))
(check-sat)
(get-model)
```

La solución es muy simple ya que x e y deben tener el mismo valor.

```
sat
(model
  (define-fun y () Int
    1)
  (define-fun a1 () (Array Int Int)
    (_ as-array k!0))
  (define-fun x () Int
    1)
  (define-fun k!0 ((x!1 Int)) Int
    (ite (= x!1 1) 1
      0))
)
```

sin embargo, Z3 no devolverá la solución genérica con $x = y$ sino que instancia x e y con valores concretos.

De hecho, podemos añadir justamente la condición $x \neq y$ y no hay solución.

```
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(declare-const a1 (Array Int Int))
(declare-const a2 (Array Int Int))
(declare-const a3 (Array Int Int))
(assert (= (select a1 x) x))
(assert (= (store a1 x y) a1))
(assert (not (= x y)))
(check-sat)
```

```
unsat
```

3. Objetivo de la práctica

El objetivo de esta práctica consiste en codificar los ejercicios adjuntos en Z3 y responder a las preguntas.

3.1. Vectores

Dada las siguientes definiciones de propiedades sobre vectores, ¿qué devuelve en cada caso?

1. ¿Qué devuelve la siguiente definición ?

   ```
   (declare-const x Int)
   (declare-const i Int)
   (declare-const a1 (Array Int Int))
   (assert (not (= (select a1 i) x)))
   (check-sat)
   (get-model)
   ```

2. ¿Qué devuelve la siguiente definición?

   ```
   (declare-const x Int)
   (declare-const a1 (Array Int Int))
   (assert (= (store a1 1 x) a1))
   (assert (= (store a1 2 x) a1))
   (assert (= (store a1 3 x) a1))
   (assert (= (store a1 4 x) a1))
   (assert (= (store a1 5 x) a1))
   (assert (= (store a1 6 x) a1))
   (assert (= (store a1 7 x) a1))
   (assert (= (store a1 8 x) a1))
   (assert (= (store a1 9 x) a1))
   (assert (= (store a1 10 x) a1))
   (check-sat)
   (get-model)
   ```

3.2. Aritmética

Dada la siguientes propiedades aritméticas, ¿qué devuelve en cada caso?

1. ¿Qué devuelve la siguiente definición ?

```
(declare-const x Int)
(declare-const y Int)
(assert (<= (+ x y) 1 ))
(check-sat)
(get-model)
```

2. ¿Qué devuelve la siguiente definición?

```
(declare-const y Int)
(assert (<= (+ 0 0) (+ y 1) ))
(check-sat)
(get-model)
```

3. ¿Qué devuelve la siguiente definición?

```
(declare-const x Int)
(declare-const y Int)
(assert (<= (+ x y) (+ x 1) ))
(check-sat)
(get-model)
```

4. ¿Qué devuelve la siguiente definición?

```
(declare-const x Int)
(declare-const y Int)
(assert (= (+ x y) 1))
(assert (<= x y))
(check-sat)
(get-model)
```