# Ingeniería del Software II

Taller #3 – Ejecución Simbólica Dinámica

*LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.*

**Fecha de entrega:** 23 de Abril de 2025 **(16:59hs)**
**Fecha de re-entrega:** 7 de Mayo de 2025 **(16:59hs) (no hay extensiones)**

## 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.
- Todas las líneas que comienzan con con punto y coma (`;`) son interpretadas como comentarios.

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`. Otra opción es, si se utiliza sistema Ubuntu, instalarlo a través de apt: `sudo apt install z3`.

También se puede instalar como librería de Python corriendo: `pip install z3-solver`.

O usar desde la interfaz web disponible en `https://jfmc.github.io/z3-play/` o `https://compsys-tools.ens-lyon.fr/z3/`.

## Set up

Descargar el proyecto y abrirlo en la IDE de preferencia.
Luego correr: `pip install -r requirements.txt`

- Todos los ejemplos del enunciado se encuentran en un archivo .smt con el nombre correspondiente.
- Si se ejecuta `run_parte_1` se correrá el solver de Z3 para todos los archivos dentro de la carpeta `parte_1`, dando un output con los resultados.

## Parte 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 llamamos al solver, 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).

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 **Execute**, Z3 concluirá que no existen valores de $x, y$ que puedan satisfacer esa fórmula, y por lo tanto devolverá **unsat** (i.e. insatisfacible).

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.

**Ejercicio 1**

Completar los archivos `ejercicio_1_*.smt` con la especificación para Z3 necesaria 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. $\neg(x \wedge y) \equiv \neg(\neg x \wedge \neg y)$

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

```
; 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)
```

Observe que el output no sólo reporta `Resultado: sat`, sino también `Modelo: [y = 10, x = 0]`. Si lo ejecuta online o localmente "directo" desde Z3 se verá:

```
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)
```

**Ejercicio 2**

Completar los archivos `ejercicio_2_*.smt` con la especificación para Z3 necesaria para encontrar una solución para $x$ e $y$ en las siguientes ecuaciones:

a. $3x + 2y = 36$

b. $5x + 4y = 64$

c. $x * y = 64$

• 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:
`[b = 20/3, c = 20, a = 10, r2 = 2, r3 = 2, r1 = 2]`

**Ejercicio 3**

Crear una **única** especifición Z3, en el archivo `ejercicio_3.smt`, que almacene en las constantes **reales** $a1$, $a2$, $a3$ el resultado, respectivamente, de calcular las siguientes expresiones:

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

**Observación:** puede corroborar que la primera parte de su taller es correcta corriendo los tests en `test_parte1.py`

## Parte 2: Dynamic Symbolic Execution (DSE)

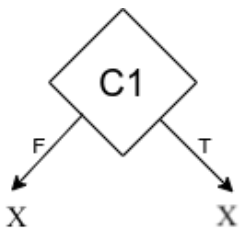
Dynamic Symbolic Execution es una técnica que permite explorar el árbol de cómputo de un programa con el objetivo de encontrar errores en el mismo. En este árbol, cada nodo interno representa una decisión tomada durante la ejecución del programa. Por otro lado, las hojas puden representar, o bien el resultado de satisfabilidad arrojado por el *constraint solver* cuando es `UNSAT` o `UNKNOWN`, o bien el input concreto que llevó a esa ejecución cuando es `SAT`. Además, utilizamos la siguiente convención: El arco izquierdo que sale de un nodo representa la rama que se toma cuando la condición es falsa, y el arco derecho representa la rama que se toma cuando la condición es verdadera.

Se muestra a continuación un ejemplo de ejecución de DSE:

```
int f(int z) {
  if (z == 12) { // C1
    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 |

Y su correspondiente árbol de cómputo es:



**Ejercicio 4**

Arme el árbol de cómputo del siguiente programa suponiendo que todas las queries al *constraint solver* devuelven SAT:

```
def foo(a: int, b: int) -> int:
    z = a + b
    if z == 42:  # C1
        z = 0
    else:
        z = 1
    if a == b:  # C2
        z = z + 1
    else:
        z = z - 1
    return z
```

Adjunte una imágen de dicho árbol en un archivo `ejercicio4.png`. Puede utilizar el sitio web `https://draw.io/` para confeccionar el diagrama.

**Ejercicio 5**

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

```
def triangle(a: int, b: int, c: int) -> str:
    if a <= 0 or b <= 0 or c <= 0:  # C1
        return "invalid"
    if not (a + b > c and a + c > b and b + c > a):  # C2
        return "invalid"
    if a == b and b == c:  # C3
        return "equilateral"
    if a == b or b == c or a == c:  # C4
        return "isosceles"
    return "scalene"
```

a. Completar el archivo `src/parte_2/ejercicio_5/ejercicio_5.csv` 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 $a = a_0$, $b = b_0$, $c = c_0$.
  Escribir la condición con sintaxis de Python (not, and, or), **usar los renombres definidos en la columna "Renombres"**. Ejemplo: not c1 and not c2 and c3
- El nombre del archivo con la especificación que se envía a Z3 para esta iteración de acuerdo al algoritmo de ejecución simbólica dinámica, debe estar en la misma carpeta (`src/parte_2/ejercicio_5/` en este caso).
- El resultado que produjo Z3 por cada consulta a Z3.
- Puede corroborar que su sintaxis es correcta corriendo los tests en `test_syntax_parte2.py`

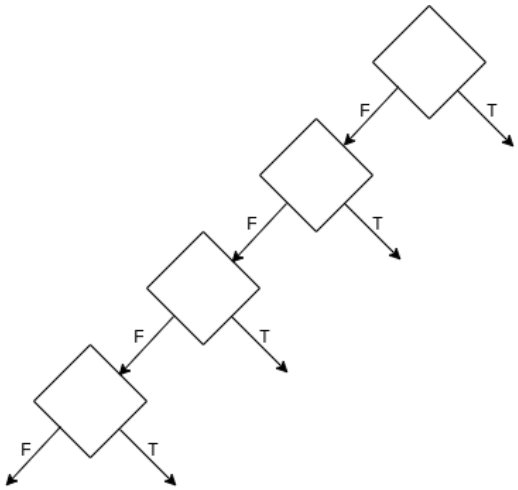
Por ejemplo:

| Iteración | Input Concreto | Condición de Ruta | Especificación para Z3 | Resultado Z3 |
|---|---|---|---|---|
| 1 | a=0, b=0, c=0 | c1 | iteracion1.smt | a=1, b=1, c=1 |
| 2 | ... | ... | ... | ... |
| ... | ... | ... | ... | ... |

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

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. Completar los nodos y hojas del siguiente árbol de cómputo con el resultado de la ejecución simbólica dinámica. Entregar como un archivo `ejercicio5c.png`. Puede utilizar como base el archivo `ejercicio5c-completar.svg` provisto en el campus.




**Ejercicio 6**

Sea el siguiente programa `magicFunction`:

```
def magicFunction(k: float) -> int:
    array: List[float] = [5.0, 1.0, 3.0]
    c: int = 0
    for i in range(3):  # C1  (range(3) == [0,1,2])
        if array[i] + k == 0:  # C2
            c += 1
    return c
```

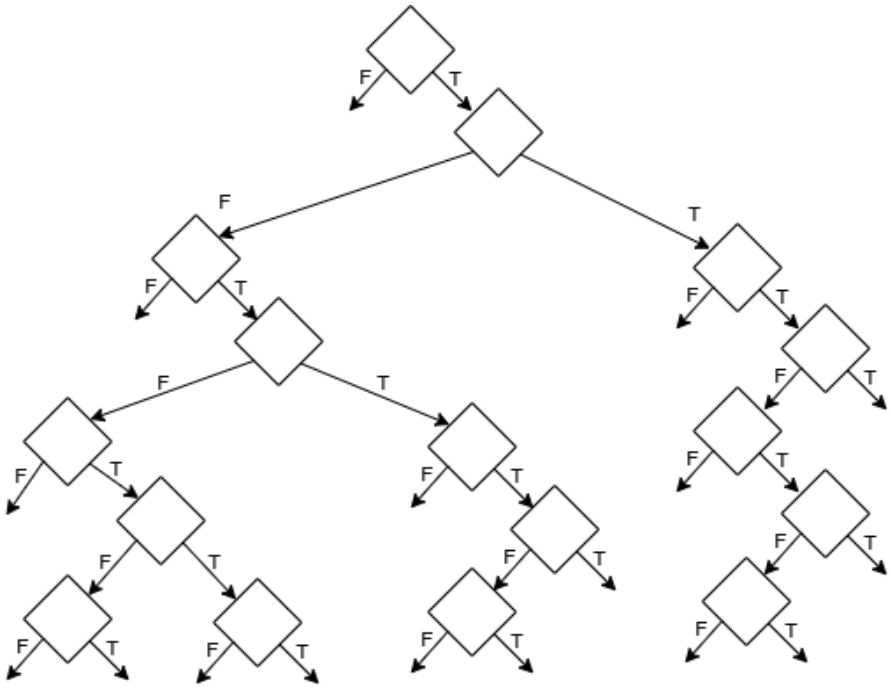
a. Completar el archivo `src/parte_2/ejercicio_6/ejercicio_6.csv` con la ejecución simbólica dinámica del programa `magicFunction`. Seguir los mismos lineamientos que los presentados en el ejercicio anterior para cada columna de la tabla.

Recordar utilizar más de una línea en caso de ser necesario cuando haya más de un llamado a Z3.

Asumir que el cuerpo del ciclo se puede ejecutar a lo sumo tres veces (i.e., $u(c1) \leq 4$),

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. Completar los nodos y hojas del siguiente árbol de cómputo con el resultado de la ejecución simbólica dinámica. Entregar como un archivo `ejercicio6c.png`. Puede utilizar como base el archivo `ejercicio6c-completar.svg` provisto en el campus.



**Ejercicio 7**

Dado el siguiente programa:

```
def bar(n: int) -> int:
  i: int = 0
  while i < n:  # C1
      i = i + 1
  return i
```

a. Completar el archivo `src/parte_2/ejercicio_7/ejercicio_7.csv` con la ejecución simbólica dinámica del programa `bar`. Seguir los mismos lineamientos que los presentados en el ejercicio 5 para cada columna de la tabla. Recordar utilizar más de una línea en caso de ser necesario cuando haya más de un llamado a Z3.

   Asumir:

   - El cuerpo del ciclo del while (es decir, la guarda True del mismo) se puede ejecutar a lo sumo dos veces (i.e., $u(c_1) \leq 2$).
   - Entre dos valores posibles para una solución, Z3 devolverá el **menor** valor posible.

b. Armar el árbol de cómputo con el resultado de la ejecución simbólica dinámica. Entregar como un archivo `ejercicio7b.png`.

## Formato de Entrega

El taller debe ser entregado en el campus de la materia. La entrega debe incluir un archivo `entrega.zip`. Este archivo debe incluir:

- Un zip del proyecto entero, revisar que estén todos los archivos .smt
- Los archivos `ejercicio4.png`, `ejercicio5c.png`, `ejercicio6c.png` y `ejercicio7b.png` con los árboles de cómputo solicitados en los ejercicios 4, 5c, 6c y 7b.
- Un archivo `RESPUESTAS` con las respuestas a las preguntas que no entren en otro formato.