1	2	3	4
B	B	M	B

Ingeniería del Software II

Parcial #1 – Generación Automática de Tests

A

Cada ejercicio se evaluará como **Bien**, **Regular** o **Mal**. Para aprobar el examen es necesario tener al menos 2 ejercicios Bien y al menos 1 ejercicio Regular. Bien = 2,5p, Regular = 1,25p, Mal = 0p.

Ejercicio 1

Sea el siguiente programa:

```
1  def test_me(j: int, n: int) -> int:
2    r:int=0
3    if n>=0 and j>0:
4      i:int=0;
5      while i < n:
6        if i%j ==0:
7          r=r+i
8        i=i+1
9    return r
```

Usando el operador de mutación ROR[1], exhibir dos mutantes (indicando la línea que cambió) tales que:

a. Un mutante con un test que lo mate. Indique cual es el test correspondiente.

b. Un mutante equivalente tal que no exista test que lo detecte.

Ejercicio 2

Sea el programa `test_me` del ejercicio #1 haciendo dos unrolls del while (i.e., se reemplaza el while por un if).

a. Escribir el programa resultante después de hacer el unroll del while. Asignar a cada condición en el nuevo programa un alias (e.g., "C1", etc.)

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

- El input concreto utilizado
- La condición de ruta (i.e. "path condition") que se produce al ejecutar el input concreto, asumiendo que el valor simbólico inicial es $n = n_0$. $j = j_0$
- La fórmula lógica (no es necesario escribirla en SMTLib) que se envía al demostrador de teoremas de acuerdo al algoritmo de ejecución simbólica dinámica.
- El resultado posible que podría producir un demostrador de teoremas (ej: Z3).

c. Describir el árbol de cómputo del programa explorado durante la ejecución simbólica dinámica del programa

Iteración	Input Concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0,n=0	...	...	...
2	...	...	...	...
...	...	...	...	...

[1] modifica una comparación artimética reemplazandola por $<, >, <=, >=, ==, !=, false, true$

Ejercicio 3

Sea el programa `test_me` del ejercicio #1 y el siguiente test suite:

```
class TestSuite(unittest.TestCase):
  def test_1(self):
    self.assertEqual(0, test_me(1,1))

  def test_2(self):
    self.assertEqual(1, test_me(1,2))

  def test_3(self):
    self.assertEqual(3, test_me(1,3))
```

a. Sea K=5, ¿cuál es el valor de la distancia de branch no normalizada para cada decisión si ejecutamos el test suite?

branch	distancia true	distancia false
3: `if n>=0 and j>0`		
5: `while i<n`		
6: `if i%j==0`		

b. ¿Cuál es el cubrimiento de branches que logra el test suite?

Ejercicio 4

Sea el siguiente programa:

```
1  def target_function(input_string: str) -> int:
2      tokens = input_string.split() # Split on whitespace
3      stack = []
4      for token in tokens:
5          if token.isdigit():
6              stack.append(int(token))
7          else:
8              try:
9                  arg2 = stack.pop()
10                 arg1 = stack.pop()
11                 if token == "+":
12                     result = arg1 + arg2
13                 elif token == "-":
14                     result = arg1 - arg2
15                 elif token == "*":
16                     result = arg1 * arg2
17                 elif token == "/":
18                     result = arg1 / arg2
19                 else:
20                     raise ValueError("Invalid operator: %s" %token)
21                 stack.append(result)
22             except:
23                 raise ValueError("Invalid expression")
24     return stack[0]
```

Asumiendo que tenemos un boosted greybox fuzzer con exponente a=3. Sea el siguiente conjunto inicial de inputs: `"42","42 42 +","13 13 +","13 24 +","1 7 +","7 8 +","9 10 +","10 10 -"`. ¿Cuál es la probabilidad que el fuzzer elija el input `"10 10 -"` para mutar?