POLITECNICO
MILANO 1863

Software Engineering 2

V&V Exercises

M Camilli, E Di Nitto, M Rossi

Verification & Validation

Exercises: Concolic execution, fuzzing, SBST

Exercise 1

- Consider the function `foo`, written in a C-like language:

- Run a concolic execution starting from the following input {a = 1, b = 3} to find possible test cases that guarantee:
  - No execution of the loop;
  - Execution of the loop two times. Line 3 must be executed only in the second iteration

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

Exercise 1

- First execution from input {a = 1, b = 3}

```
0: void foo(int a, int b) {
1:    for (int i=a; i<b; i++) {
2:       if (i % 5 == 0) {
3:          print(i)
4:       }
5: }
```

<0>

a	b	i	π
A	B		T
1	3		

→

<0,1>

a	b	i	π
A	B	A	A<B
1	3	1	

→

<0,1,2>

a	b	i	π
A	B	A	A<B A%5≠0
1	3	1	

→

<0,1,2,1>

a	b	i	π
A	B	A+1	~~A<B~~ A%5≠0 A+1<B
1	3	2	

→

<0,1,2,1,2>

a	b	i	π
A	B	A+1	A%5≠0 A+1<B A+1%5≠0
1	3	2	

→

<0,1,2,1,2,1>

a	b	i	π
A	B	A+2	A%5≠0 ~~A+1<B~~ A+1%5≠0 ~~A+2≥B~~ A+2 = B
1	3	3	

The test case executes the loop 2 times without executing the location 3

Exercise 1

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

- Let’s find a test case that runs the loop zero times…

<0>

a	b	i	π
A	B		T
1	3		

→

<0,1>

a	b	i	π
A	B	A	A<B
1	3	1	

Negation: ¬(A < B) => A ≥ B

Solve: A ≥ B

{a = 3, b = 1}

<0>

a	b	i	π
A	B		T
3	1		

→

<0,1>

a	b	i	π
A	B	A	A≥B
3	1	3	

The new test case {a = 3, b = 1} does not execte the loop

Exercise 1

- Let's find a test case for the other path: <0,1,2,1,2,3,1>

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

<0>

a	b	i	π
A	B		T
1	3		

→ … →

a	b	i	π
A	B	A+2	A%5≠0 ~~A+1<B~~ A+1%5≠0 ~~A+2≥B~~ A+2=B
1	3	3	

Partial negation:

A%5≠0 ∧ ¬(A+1%5≠0) ∧ A+2=B

Solve:

A%5≠0 ∧ A+1%5=0 ∧ A+2=B

{a = 4, b = 6}

<0>

a	b	i	π
A	B		T
4	6		

Exercise 1

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

- Let's find a test case for the other path: <0,1,2,1,2,3,1>

<0>

a	b	i	π
A	B		T
4	6		

→

<0,1>

a	b	i	π
A	B	A	A<B
4	6	4	

→

<0,1,2>

a	b	i	π
A	B	A	A<B A%5≠0
4	6	4	

→

<0,1,2,1>

a	b	i	π
A	B	A+1	~~A<B~~ A%5≠0 A+1<B
4	6	5	

→

<0,1,2,1,2>

a	b	i	π
A	B	A+1	A%5≠0 A+1<B A+1%5=0
4	6	5	

→

<0,1,2,1,2,3,1>

a	b	i	π
A	B	A+2	A%5≠0 A+1<B A+1%5=0 ~~A+2≥B~~ A+2 = B
4	6	6	

The new test case {a = 4, b = 6} executes the loop 2 times and runs location 3 the second time only.

Exercise 1

- Summary of test cases we found:
  - { a=3, b=1 }: no loop
  - { a=1, b=3 }: loop 2 times without executing location 3
  - { a=4, b=7 }: loop 2 times executing location 3 (2nd iteration only)

Exercise 2

- Consider the following function `foo2`, written in a C-like language:

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8: }
```

- Run concolic execution to explore all possible branches
- Write the pseudocode of a possible generational fuzzer that should test `foo2`.
- Given the fuzzer defined in the previous point
  - What is the chance of executing a path that includes location 6?
  - How much time do you have to wait on average assuming that a single execution takes ~1ms?
- Consider an SBST procedure with the objective of executing paths that reach location 6. Define proper search space, neighborhood relation, and fitness function to use such approach

Exercise 2

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8: }
```

- Concolic execution. For each conditional statement we want to execute both T and F branches

Test case {a = 1, b = 2} executes the F branch of condition 2

<0>

a	b	x	i	π
A	B			T
1	2			

→

<0,1>

a	b	x	i	π
A	B	1		T
1	2	1		

→

<0,1,2>

a	b	x	i	π
A	B	1		A>0 B>1 A≤B
1	2	1		

Partial negation: $A>0 \land B>1 \land \neg(A \leq B)$

Solve: $A>0 \land B>1 \land A>B$

{a = 4, b = 2}

<0>

a	b	x	i	π
A	B			T
4	2			

Exercise 2

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8: }
```

- Concolic execution

Test case {a = 4, b = 2} executes the T branch of condition 2

<0>

a	b	x	i	π
A	B			T
4	2			

→

<0,1>

a	b	x	i	π
A	B	1		T
4	2	1		

→

<0,1,2>

a	b	x	i	π
A	B	1		~~A>0~~, B>1, A>B
4	2	1		

→

<0,1,2,3>

a	b	x	i	π
A	B	1	0	B>1, A>B ~~0<B~~
4	2	1	0	

→

<0,1,2,3,4>

a	b	x	i	π
A	B	A	0	B>1, A>B
4	2	4	0	

→ …

Exercise 2

- Concolic execution

```
0: void foo2(int a, int b) {
1:   int x = 1;
2:   if (a>b && a>0 && b>1) {
3:     for (int i=0; i<b; i++)
4:       x = x*a;
5:     if (x < rand(1,10))
6:       print("Log message");
7:   }
8: }
```

<0,1,2,3,4>

a	b	x	i	π
A	B	A	0	B>1, A>B
4	2	4	0	

→

<0,1,2,3,4,3>

a	b	x	i	π
A	B	A	1	B>1, A>B ~~1<B~~
4	2	4	1	

→

<0,1,2,3,4,3,4>

a	b	x	i	π
A	B	A*A	1	B>1, A>B
4	2	16	1	

<0,1,2,3,4,3,4,3>

a	b	x	i	π
A	B	A*A	2	~~B>1~~, A>B ~~2≥B~~ B=2
4	2	16	2	

→

<0,1,2,3,4,3,4,3,5>

a	b	x	i	π
A	B	A*A	2	A>B B=2 A*A≥rand(1,10)
4	2	16	2	

"rand(1,10)" black-box function.
Assume it generates a random int in [1,10].
Assume we draw 10.

Exercise 2

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8: }
```

- Concolic execution

`<0,1,2,3,4,3,4,3,5>`

… →

a	b	x	i	π
A	B	A*A	2	A>B 2=B A*A≥rand(1,10)
4	2	16	2	

Test case {a = 4, b = 2} executes the T branch of condition 2 and F branch of condition 5. We still need a test case that executes T branch for condition 2 and T branch for condition 5.

Partial negation: $A>B \wedge B=2 \wedge \neg(A*A \geq rand(1,10))$

Symbolic-to-concrete step:

$A>B \wedge B=2 \wedge \neg(A*A \geq 10) \Rightarrow$

$A>B \wedge B=2 \wedge A*A<10$

Exercise 2

- Concolic execution

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8:  }
```

`<0,1,2,3,4,3,4,3,5>`

… →

a	b	x	i	π
A	B	A*A	2	A>0, A>B 2=B A*A≥rand(1,10)
4	2	16	2	

Test case {a = 4, b = 2} executes the T branch of condition 2 and F branch of condition 5. We still need a test case that executes T branch for condition 2 and T branch for condition 5.

Partial negation: A>B ∧ B=2 ∧ ¬(A*A ≥ rand(1,10))

Symbolic-to-concrete step:

A>B ∧ B=2 ∧ ¬(A*A ≥ 10)

Solve:

A>B ∧ B=2 ∧ A*A<10

Exercise 2

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8: }
```

- Concolic execution

{a = 3, b = 2}

Solve: A>B ∧ B=2 ∧ A*A<10

<0>

a	b	x	i	π
A	B			T
3	2			

<0,1>

a	b	x	i	π
A	B	1		T
3	2	1		

→

<0,1,2>

a	b	x	i	π
A	B	1		~~A>0~~, B>1, A>B
3	2	1		

→

<0,1,2,3>

a	b	x	i	π
A	B	1	0	B>1, A>B ~~0<B~~
3	2	1	0	

→

<0,1,2,3,4>

a	b	x	i	π
A	B	A	0	B>1, A>B
3	2	3	0	

→ …

Exercise 2

- Concolic execution

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8:  }
```

<0,1,2,3,4>

a	b	x	i	π
A	B	A	0	B>1, A>B
3	2	3	0	

→

<0,1,2,3,4,3>

a	b	x	i	π
A	B	A	1	~~B>1~~, A>B ~~1<B~~ B>1
3	2	3	1	

→

<0,1,2,3,4,3,4>

a	b	x	i	π
A	B	A*A	1	B>1, A>B
3	2	9	1	

<0,1,2,3,4,3,4,3>

a	b	x	i	π
A	B	A*A	2	~~B>1~~, A>B ~~2≥B~~ B=2
3	2	9	2	

→

<0,1,2,3,4,3,4,3,5,6>

a	b	x	i	π
A	B	A*A	2	B>0, A>B B=2 A*A<rand(1,10)
4	2	9	2	

Assume we draw again 10.

In this case, the test case {a=3, b=2} executes T branch for both conditions 2 and 5

Exercise 2

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8: }
```

- We can define the following generational fuzzer for the function foo2

```
int[] fuzzer( ) {
  int b = rand();
  int a = rand();
  return [a,b];
}
```

Assume "rand( )" returns a random int with no constraints

- To execute location 6, we need to generate inputs that satisfy the following condition: A>B ∧ B=2 ∧ A*A<rand(1,10)
  - Just one possibility: {a=3, b=2}

Exercise 2

- To execute location 6, we need to generate inputs that satisfy the following condition: $A>B \wedge B=2 \wedge A*A<rand(1,10)$
  - Just one possibility: {a=3, b=2}
- Assuming 32-bit encoding for integers, the range is [-2147483648, 2147483647]
  - Total chance = $P(a=3 \wedge b=2 \wedge 9<rand(1,10)) =$
    $(1/4.3*10^9)*(1/4.3*10^9)*(1/10) = 5.4*10^{-21}$

Exercise 2

- Total chance = $5.4*10^{-21}$
- #average runs
  - $1/ 5.4*10^{-21} = 1.8*10^{20}$
- Average execution time
  - $1.8*10^{20} * 0.001 = 1.8*10^{17}$s = $5.7*10^{9}$ years

Exercise 2

- SBST approach
  - By looking at the source code we need to generate values for `a` and `b` s.t. both conditions 2 and 5 are T
  - The **search space** can be all possible pairs of int values
    - a=1, b=-10
    - a=0, b=3
    - …
  - We can easily define a neighborhood relation that, given a pair, we possibly modify each element of the pair by 1
    - 8 pairs in total
    - Example <1,1> → <0,0>, <0,1>, <0,2>, <1,0>, <1,2>, <2,0>, <2,1>, <2,2>

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8: }
```

Exercise 2

- SBST approach
  - To define the fitness we can consider the distance from the branches we want to execute
  - Consider the **first condition (location 2)**
    - `if (a>b && a>0 && b>1)`
    - All sub-conditions must be T
    - Let's consider the sub-condition "a>b" →
      - a=10, b= 100 is worse than a=10, b= 10
      - If F, we can measure the distance from T as "b-a+1"
      - If T, the distance is 0
  - We can do the same for all sub-conditions →
    - Total distance is the sum, e.g., `dist1_sub1(a,b)+dist1_sub2(a)+dist1_sub3(b)`

```
0:  void foo2(int a, int b) {
1:    int x = 1;
2:    if (a>b && a>0 && b>1) {
3:      for (int i=0; i<b; i++)
4:        x = x*a;
5:      if (x < rand(1,10))
6:        print("Log message");
7:    }
8: }
```

Exercise 2

- SBST approach
  - **first condition (location 2)**
  - We can do the same for all sub-conditions →
    - Total distance is the sum, e.g., `dist1_sub1(a,b)+dist1_sub2(a)+dist1_sub3(b)`

```
0:  int dist1_sub1(int a, int b) {
1:    if (a>b)
2:      return 0;
3:    return b-a+1;
4:  }
```

```
0:  int dist1_sub2(int a) {
1:    return dist1_sub1(a,0);
2:  }
```

```
0:  int dist1_sub3(int a) {
1:    return dist1_sub1(b,1);
2:  }
```

Exercise 2

- SBST approach
  - **first condition (location 2)**
  - We can do the same for all sub-conditions →
    - Total distance is `dist1(a,b)`
    - **Note**: each distance should be normalized to avoid biased search (some distances may depend on very large values, some other distances on very small values)

```
int dist1(int a, int b) {
  return norm(dist_subb1(a,b))+ norm(dist_subb2(a)) + norm(dist_subb3(b));
}
```

Example of norm function: norm(x) = x / (1 + x) →
rescales any positive value x in the range [0,1]

Exercise 2

POLITECNICO
MILANO 1863

- SBST approach
  - **second condition (location 5)**
  - Just one condition → Total distance is `dist2(x, rand(1,10))`

```
int dist2(int a, int b) {
  return norm(dist_subb1(b,a));
}
```

```
0: void foo2(int a, int b) {
1:   int x = 1;
2:   if (a>b && a>0 && b>1) {
3:     for (int i=0; i<b; i++)
4:       x = x*a;
5:     if (x < rand(1,10))
6:       print(“Log message”);
7:   }
8: }
```

Exercise 2

- SBST approach
  - The **fitness** can be defined as the sum of dist1 and dist2
  - When the sum is zero it means we execute the T branch of both conditions

```
…
if (a>b && a>0 && b>1) {
  …
  if (x < rand(1,10))
    …
```

  - Assuming we can observe the execution of `foo2` through proper instrumentation, the test case generation shall minimize the following:
    - `fitness(a,x,rand) = dist1(a,b) + dist2(x,rand)`
    - With `a, x, rand=rand(1,10)` extracted from the execution by the instrumentation

Exercise 3

- Consider the following operation `events` exposed by a REST API of a web service
  - The operation takes as input a date and returns a list of events scheduled for that date (if any)

method	endpoint	parameter	type	Example
GET	/events	date	In path	http://my.url/api/v1/events?date=01-01-2024

- Write the pseudocode of a generational fuzzer that produces possible test cases for the operation `events`
- Write the pseudocode of an automated testing procedure that uses the previous fuzzer and defines an executable oracle that recognizes server errors. Define a testing budget that is enough to generate ~10 syntactically valid inputs

Exercise 3

- Assume you want to generate test cases using a mutational fuzzer. Define the pseudocode of the following mutators
  - **Syntactic** mutator: apply a small mutation to produce a new input with the same length. The mutator shall modify the structure or syntax of the input data without considering the meaning of the input
  - **Semantic** mutator: apply a small mutation to produce a new input with the same length. The mutator shall consider the meaning of the input and modify it so that the new date differs from the previous one by 1 month at most

Exercise 3

- Possible generational fuzzer that produces test cases (i.e., inputs for the parameter of the operation event)

```
string fuzzer( ) {
  out = "";
  for (int i=0; i<10, i++)
      out += chr(rand(32, 64));
  return out;
}
```

- We assume rand produces a random integer in the range [32,64], while chr returns the corresponding char given a integer value
- According to the ASCII table, the range includes numeric values, the separator "-" and also other chars should not appear as part of a valid date

Exercise 3

POLITECNICO
MILANO 1863

- Test generation procedure that uses the previous generational fuzzer

```
list test_procedure(int budget) {
  archive = list();
  for (int i=0; i<budget, i++) {
    input = fuzzer();
    command = “curl -X GET http://my.url/api/v1/events?date=” + input;
    result = system(command);
    if (result.status_code >= 500)
      archive.append(input);
  }
  return archive;
}
```

Oracle: check occurrence of server errors 5xx

Exercise 3

- The Testing budget should be defined according to the probability of generating syntactically valid inputs through the fuzzer
- We have to generate
  - 2 digits → (10/32)*(10/32)
  - "-" → 1/32
  - 2 digits → (10/32)*(10/32)
  - "-" → 1/32
  - 4 digits → (10/32)*(10/32)*(10/32)*(10/32)
- Total likelihood = $(10/32)^8 * (1/32)^2 = 8.8*10^{-8}$
- Average #runs for a single valid input = $1/8.8*10^{-8} = 1.1*10^7$
- Budget = $1.1*10^7 * 10 = 1.1*10^8$

Exercise 3

POLITECNICO
MILANO 1863

- Mutational fuzzer: syntactic mutator (modify the structure or syntax of the input data without considering the meaning of the input)

```
string syntactic_mutator(string s, int start=32, int end=64) {
  pos = rand(0, len(s))
  val = ord(s[pos])
  if (val > start && val < end)
    val += rand_choice([-1,1])
  else if (val == start)
    val += 1
  else if (val == end)
    val += -1
  return s[:pos] + char(val) + s[pos+1:]
}
```

ord(c) = int encoding of char c

Substring from 0 to pos-1

Substring from pos+1 to the end

Exercise 3

- Mutational fuzzer: semantic mutator (consider the meaning of the input and modify by 1 month at most)

```
string semantic_mutator(string s) {
    d = s[:2]; m = s[3:5]; y = s[6:];
    d += rand_choice([-1,1]) * rand(0, 31)
    if (d > 31) {
      d -= 31;
      m += 1;
    } else if (d < 0) {
      d = 31-d;
      m -= 1;
    }
    if (m > 12) {
      m = 1;
      y += 1;
    } else if (m < 1) {
      m = 12;
      y -= 1;
   }
   return d + "-" + m + "-" y;
}
```

Exercise 4

- Consider the following function `bar`, written in a C-like language:

```
0:  string bar(string s) {
1:    r = "";
2:    i = 0;
3:    while (i < length(s)) {
4:      c = s[i];
5:      if (c != '+')
6:        r += c;
7:      else if (i+1 < length(s))
8:        r += decode(s[i+1])
9:      i += 1;
10:   return r;
11: }
```

- Assume you have the following test objective: test the portion of bar that invokes the decode function (location 8)
- Define the following elements to reach the objective using a SBST approach
  - Search space
  - Neighborhood relation
  - Fitness function
  - Code instrumentation

Exercise 4

```
string bar(string s) {
  r = "";
  i = 0;
  while (i < length(s)) {
    c = s[i];
    if (c != '+')
      r += c;
    else if (i+1 < length(s))
      r += decode(s[i+1])
    i += 1;
   return r;
}
```

- Search space:
  - The function bar takes as input a single string
  - Neither the text nor the function include specific constraints on inputs
  - We could consider, for instance, all possible ASCII encoded strings with max length = 100
- Neighborhood:
  - Given s, we can consider all strings having edit distance = 1
  - Edit distance: `sum(abs(ord(s1[i]) - ord(s2[i])))` for all `i`
  - Neighborhood size = 2*length(s)

Exercise 4

- Objective:
  - Condition location 3: branch T
  - Condition location 5: branch F
  - Condition location 7: branch T
- Approach:
  - Derive proper distance functions for each condition above
  - Fitness = sum of normalized distance values

```
0:   string bar(string s) {
1:     r = "";
2:     i = 0;
3:     while (i < length(s)) {
4:       c = s[i];
5:       if (c != '+')
6:         r += c;
7:       else if (i+1 < length(s))
8:         r += decode(s[i+1])
9:       i += 1;
10:    return r;
11: }
```

Exercise 4

- Condition location 3: branch T
  - i < length(s)
  - If T, distance = 0
  - If F, distance = i - length(s) + 1
- Condition location 5: branch F
  - c != ‘+’
  - If F, distance = 0
  - If T, distance = 1
- Condition location 7: branch F
  - i+1 < length(s)
  - If T, distance = 0
  - If F, distance = (i+1) - length(s) + 1

```
0:  string bar(string s) {
1:    r = ””;
2:    i = 0;
3:    while (i < length(s)) {
4:      c = s[i];
5:      if (c != ’+’)
6:        r += c;
7:      else if (i+1 < length(s))
8:        r += decode(s[i+1])
9:      i += 1;
10:   return r;
11: }
```

Exercise 4

- Code instrumentation

```
string intrumented_bar(string s, int[] dist) {
  r = "";
  i = 0;
  while (eval_1(dist, 0, i, length(s))) {
    c = s[i];
    if (eval_2(dist, 1, c, '+'))
      r += c;
    else if (eval_1(dist, 2, i+1, length(s)))
      r += decode(s[i+1])
    i += 1;
   return r;
}
```

Exercise 4

- Code instrumentation

```
boolean eval_1(int[] dist, int i, int a, int b) {
  if (a < b)
    dist[i] = 0;
  dist[i] = a-b+1;
  return dist[i] == 0;
}
```

```
boolean eval_2(int[] dist, int i, int a, int b) {
  if (a != b)
    dist[i] = 0;
  dist[i] = 1;
  return dist[i] == 0;
}
```

Exercise 4

- Implementation of the fitness function

```
double fitness(string s) {
  int[] dist = [0,0,0]
  instrumented_bar(s, dist)
  return norm(dist[0]) + norm(dist[1]) + norm(dist[2]);
}
```