

Software Engineering 2

V&V Exercises



Verification & Validation

Exercises: Concolic execution, fuzzing, SBST



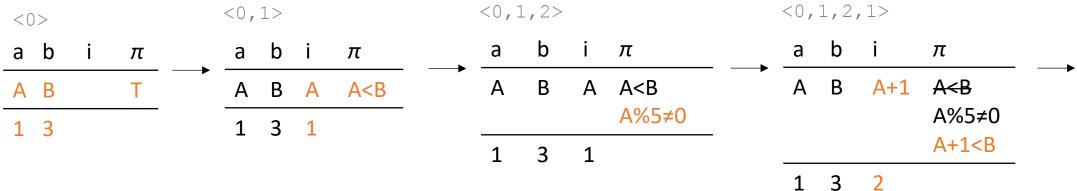
- Consider the function $f \circ \circ$, written in a C-like language:
- Run a concolic execution starting from the following input {a = 1, b = 3} to find possible test cases that guarantee:
 - No execution of the loop;
 - Execution of the loop two times. Line 3 must be executed only in the second iteration

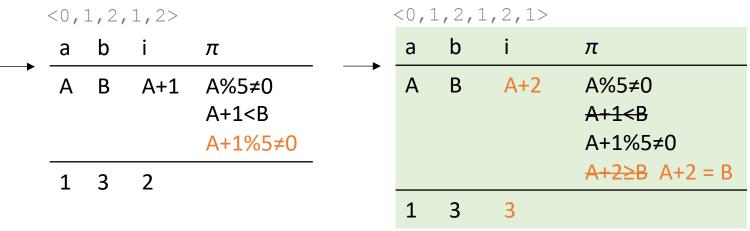
```
0: void foo(int a, int b) {
1:    for (int i=a; i<b; i++) {
2:       if (i % 5 == 0) {
         print(i)
4:    }
5: }</pre>
```



• 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: }</pre>
```



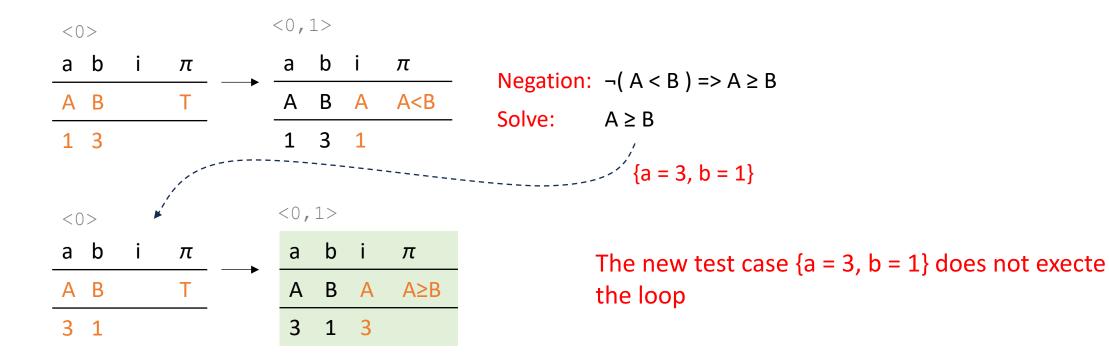


The test case executes the loop 2 times without executing the location 3



• Let's find a test case that runs the loop zero times...

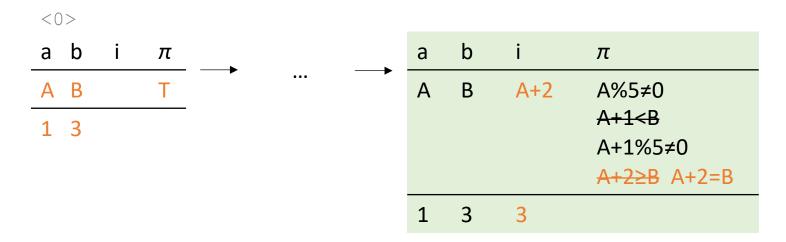
```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }</pre>
```





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: }</pre>
```



Partial negation:

A%5≠0 ∧ ¬(A+1%5≠0) ∧ A+2=B

Solve:

A%5≠0 ∧ A+1%5=0 ∧ A+2=B

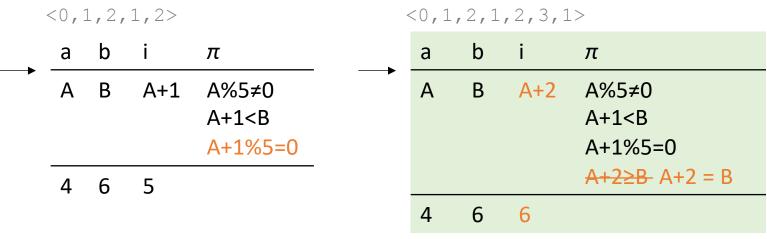
$$<0>$$
a b i π
A B T
4 6



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) {
            print(i)
4:       }
5:    }</pre>
```

```
<0,1,2,1>
                                       <0,1,2>
                 <0,1>
< 0 >
                                           b
                                                  π
                                                                    b
                     b i
                           π
                                                                 a
                                                                             π
  b
         π
                                           В
                                                                    В
                                               Α
                                                  A<B
                                                                        A+1
                                                                             A<B
                  A B A
                           A<B
                                                  A%5≠0
                                                                             A%5≠0
                  4 6 4
4 6
                                                                             A+1<B
                                            6
                                               4
                                                                 4 6
                                                                        5
```



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



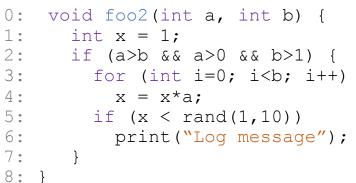
- Summary of test cases we found:
 - { a=3, b=1 }: no loop
 - { a=1, b=3 }: loop 2 times without executing location 3
 - { a=4, b=7 }: loop 2 times executing location 3 (2nd iteration only)



• Consider the following function $f \circ \circ 2$, written in a C-like language:

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

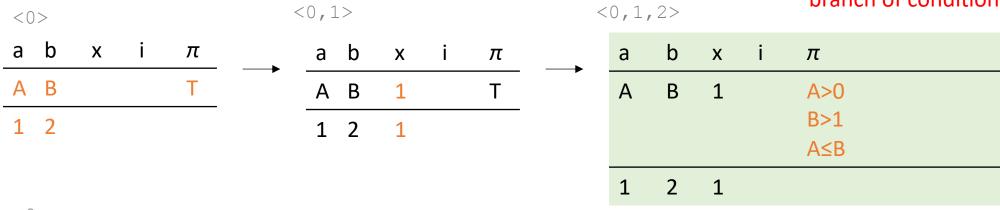
- Run concolic execution to explore all possible branches
- Write the pseudocode of a possible generational fuzzer that should test $f \circ 2$.
- Given the fuzzer defined in the previous point
 - What is the chance of executing a path that includes location 6?
 - How much time do you have to wait on 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





 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



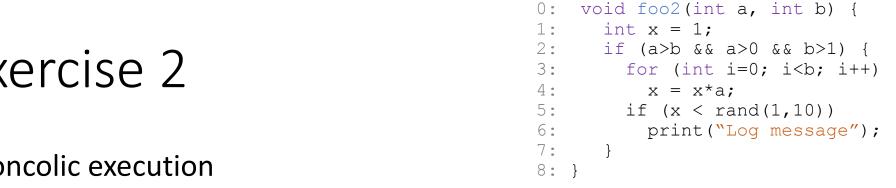
a b x i π A B T

4 2

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

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

Concolic execution





Test case $\{a = 4, b = 2\}$ executes the T

```
branch of condition 2
                         <0,1>
                                                     <0,1,2>
< 0>
                                                           b
a b
                              b
                                  Χ
       Χ
                π
                           a
                                           π
                                                               Χ
                                                                        π
                                                                        A>0, B>1, A>B
A B
                           A B
                                                           В
                             2
4 2
                                                           2
                           4
                                                       4
                                            <0,1,2,3,4>
<0,1,2,3>
                                                   b
 a
                   π
                                                               π
          Χ
                                              a
                                                       Χ
 Α
                   B>1, A>B
                                                                B>1, A>B
                                                       Α
                   <del>0<B</del>
                                                       4
                                                            0
     2
 4
         1
```



Concolic execution

1:

3:

4: 5:

7:

8: }

void foo2(int a, int b) {

x = x*a;

if (a>b && a>0 && b>1) {

if (x < rand(1,10))

for (int i=0; i<b; i++)

print("Log message");

int x = 1;

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

Concolic execution



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 \land B=2 \land ¬(A*A \ge rand(1,10))

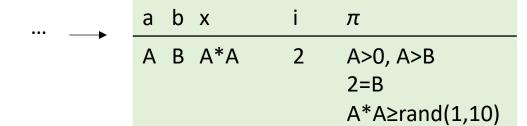
Symbolic-to-concrete step:

A>B
$$\land$$
 B=2 \land ¬(A*A \ge 10) =>
A>B \land B=2 \land A*A<10

Concolic execution

4 2 16

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



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: }</pre>
```



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 \land B=2 \land ¬(A*A \ge rand(1,10))

Symbolic-to-concrete step:

 $A>B \land B=2 \land \neg (A*A \ge 10)$

Solve:

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



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: }</pre>
```

```
{a = 3, b = 2}
                                                           Solve: A>B \land B=2 \land A*A<10
                               <0,1>
                                                         <0,1,2>
< 0 >
                            a b
                                   Χ
                                             π
                                                                  Χ
a b
       x i
                                                                           π
                 π
                            A B
                                                                           A>0, B>1, A>B
                                                              В
                            3 2
                                                              2
3 2
                                              <0,1,2,3,4>
<0,1,2,3>
                                                     b
                                                               i \pi
  a
          Χ
                   π
                                                a
                                                         X
                                                        Α
  Α
                    B>1, A>B
                                                     В
                                                               0 B>1, A>B
               0
                    <del>0<B</del>
                                                3
                                                     2
                                                         3
                                                               0
```



Concolic execution

3 2 3

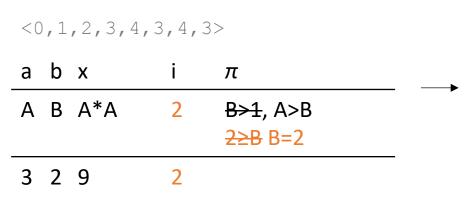
1:

3:

4:

7:

8: }



<0,1,2,3,4,3,4,3,5,6>						
а	b	X	i	π		
Α	В	A*A	2	B>0, A>B B=2 A*A <rand(1,10)< td=""></rand(1,10)<>		
4	2	9	2			

void foo2(int a, int b) {

x = x*a;

if (a>b && a>0 && b>1) {

if (x < rand(1,10))

for (int i=0; i<b; i++)

print("Log message");

int x = 1;

Assume we draw again 10.

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



We can define the following generational fuzzer for the function foo?

```
0: int[] fuzzer() {
   int b = rand();
   int a = rand();
   return [a,b];
4: }
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 \land B=2 \land A*A<rand(1,10)
 - Just one possibility: {a=3, b=2}



- To execute location 6, we need to generate inputs that satisfy the following condition: A>B \land B=2 \land 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 = $P(a=3 \land b=2 \land 9 < rand(1,10)) = (1/4.3*10^9)*(1/4.3*10^9)*(1/10) = 5.4*10^{-21}$



- 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



- 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> \rightarrow <0,0>, <0,1>, <0,2>, <1,0>, <1,2>, <2,0>, <2,1>, <2,2>

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



- 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) {
   int x = 1;
   if (a>b && a>0 && b>1) {
      for (int i=0; i<b; i++)
        x = x*a;
   if (x < rand(1,10))
      print("Log message");
   }
8: }</pre>
```



- 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: }
```



- 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)



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

```
0: int dist2(int a, int b) {
1:    return norm(dist_subb1(b,a));
1: }
```



- 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))
    ...
</pre>
```

- Assuming we can observe the execution of $f \circ \circ 2$ 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



- 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



- 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



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

```
0: string fuzzer() {
1:    out = "";
2:    for (int i=0; i<10, i++)
3:        out += chr(rand(32, 64));
4:    return out;
5: }</pre>
```

- 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



Test generation procedure that uses the previous generational fuzzer

```
0: list test_procedure(int budget) {
1:    archive = list();
2:    for (int i=0; i<budget, i++) {
3:        input = fuzzer();
4:        command = "curl -X GET http://my.url/api/v1/events?date=" + input;
5:        result = system(command);
6:        if (result.status_code >= 500)
7:        archive.append(input);
8:    }
9:    return archive;
10: }
Oracle: check occurrence of
server errors 5xx
```



- The Testing budget should be defined according to the probability of generating syntactically valid inputs through the fuzzer
- We have to generate
 - 2 digits \rightarrow (10/32)*(10/32)
 - "-" → 1/32
 - 2 digits \rightarrow (10/32)*(10/32)
 - "-" → 1/32
 - 4 digits \rightarrow (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 = $1.1*10^7*10 = 1.1*10^8$



 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, \overline{l}en(s))
   val = ord(s[pos])
                                                  ord(c) = int encoding of char c
     if (val > start && val < end)</pre>
     val += rand choice([-1,1])
      else if (val == start)
     val += 1
7: else if (val == end)
    val += -1
3:
      return s[:pos] + char(val) + s[pos+1:]
10: }
       Substring from 0 to pos-1 Substring from pos+1 to the end
```



Mutational fuzzer: semantic mutator (consider the meaning of the input and modify by 1

month at most)

```
string semantic mutator(string s) {
1: d = s[:2]; m = s[3:5]; y = s[6:];
2: d += rand choice([-1,1]) * rand(0, 31)
3: if (d > 3\overline{1}) {
4: d -= 31;
5: m += 1;
6: } else if (d < 0) {
7: d = 31-d;
8: m -= 1;
10: if (m > 12) {
11: m = 1;
12: y += 1;
13: } else if (m < 1) {
   m = 12;
14:
   y -= 1;
15:
16:
   return d + "-" + m + "-" y;
17:
18: }
```



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

```
0: string bar(string s) {
    r = "";
2:    i = 0;
3:    while (i < length(s)) {
        c = s[i];
        if (c != '+')
            r += c;
        else if (i+1 < length(s))
            r += decode(s[i+1])
        i += 1;
10: return r;
11: }</pre>
```

- 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



• 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)

```
0: string bar(string s) {
1:    r = "";
2:    i = 0;
3:    while (i < length(s)) {
        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: }</pre>
```



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)) {
        c = s[i];
5:        if (c != '+')
            r += c;
7:        else if (i+1 < length(s))
8:        r += decode(s[i+1])
9:        i += 1;
10:        return r;
11: }</pre>
```



- 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)) {
        c = s[i];
5:        if (c != '+')
            r += c;
7:        else if (i+1 < length(s))
8:        r += decode(s[i+1])
9:        i += 1;
10:        return r;
11: }</pre>
```



Code instrumentation

```
0: 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;
11: }
```



Code instrumentation

```
0: boolean eval_1(int[] dist, int i, int a, int b) {
1:    if (a < b)
2:        dist[i] = 0;
3:        dist[i] = a-b+1;
4:        return dist[i] == 0;
5: }

0: boolean eval_2(int[] dist, int i, int a, int b) {
1:        if (a != b)
2:            dist[i] = 0;
3:            dist[i] = 1;
4:            return dist[i] == 0;
5: }</pre>
```



Implementation of the fitness function

```
0: double fitness(string s) {
1:    int[] dist = [0,0,0]
2:    instrumented_bar(s, dist)
4:    return norm(dist[0]) + norm(dist[1]) + norm(dist[2]);
5: }
```