



中山大學
SUN YAT-SEN UNIVERSITY

Part II [Problem]

8. Test Generation



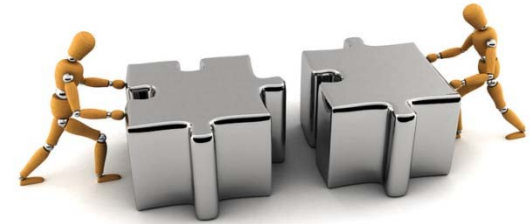
SE-307 Software Testing Techniques

<http://my.ss.sysu.edu.cn/wiki/display/SE307/Home>

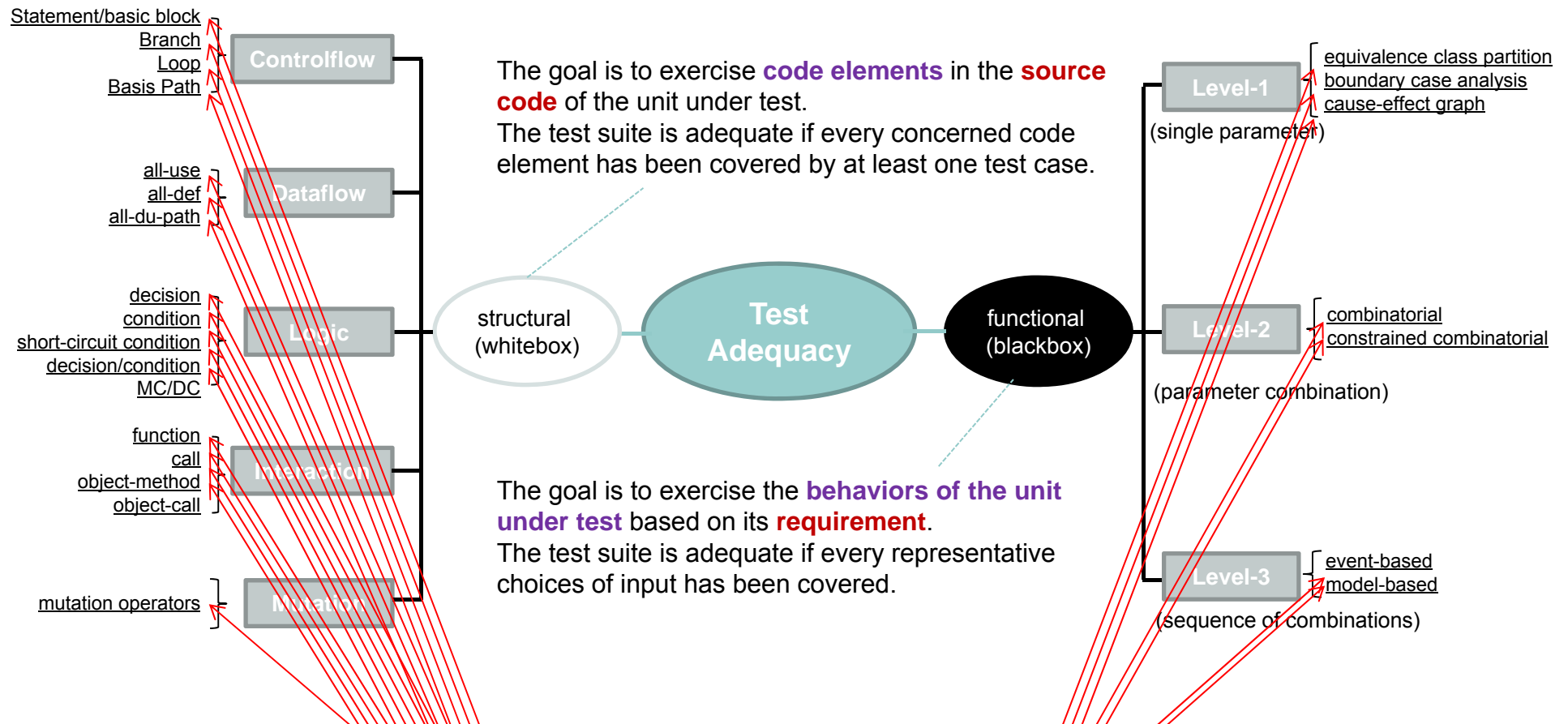
Instructor: Dr. Wang Xinming, School of Software, Sun Yat-Sen University

Review: Problems in Testing

- Fundamental problems
 - Test oracle (测试结果判定)
 - Test adequacy (测试充分性标准)
 - **Test generation (测试数据生成)**
- Important problems
 - Test automation (测试自动化)
 - Test management (测试计划和过程)



Test Generation Problem



How to generate test input data to satisfy these adequacy criteria?

- At different test levels: unit, integration, system, acceptance, ...
- In different domains: database, web-application, embedded system ...

Challenges of Test Generation: 1

- Theoretically difficult

```
void foo(string x){  
    if (MD5_Hash(x) == 0x2132411) {  
        ...  
    }  
}
```

This is actually how OS implements login!

```
void login(string username, string password){  
    if (MD5_Hash(password) == 0x2132411) {  
        //login success  
    }  
}
```

Can you generate the value of x to cover the true branch?

Challenges of Test Generation: 2

- The relation between the selection of test data and the satisfaction of test adequacy criteria is **indirect**.

C code
file as
input



What kind of C code file shall we construct to drive GCC to execute a particular line of code?

```
case MULT:
  /* If we have (mult (plus A B) C), apply the distributive
     law and then the inverse distributive law to see if
     things simplify. This occurs mostly in addresses,
     often when unrolling loops. */

  if (GET_CODE (XEXP (x, 0)) == PLUS)

    x = apply_distributive_law
      (gen_binary (PLUS, mode,
                  gen_binary (MULT, mode,
                              XEXP (XEXP (x, 0), 0),
                              XEXP (x, 1)),
                  gen_binary (MULT, mode,
                              XEXP (XEXP (x, 0), 1),
                              XEXP (x, 1))));

    if (GET_CODE (x) != MULT)
      return x;

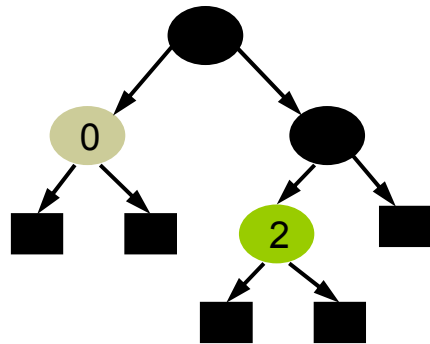
  break;
```

13 out of 1 million lines of code in GCC

Challenges of Test Generation: 3

- Test input data is subject to complex structure and constraints
- Examples:

Input: a red-black tree



Input: a XML document

```
<library>
  <book year=2001>
    <title>T1</title>
    <author>A1</author>
  </book>
  <book year=2002>
    <title>T2</title>
    <author>A2</author>
  </book>
  <book year=2003>
    <title>T3</title>
    <author>A3</author>
  </book>
</library>

/library/book[@year<2003]/title
```

Test Generation Techniques

- Whitebox (based on source code)
 - Symbolic test generation (基于符号执行)
 - Concolic test generation (基于协同执行)
- Blackbox (based on specification)
 - Random test generation (随机生成)
 - Constraint based test generation (基于约束)
 - Evolutionary test generation (遗传算法)
 - Grammar-based test generation (基于语法)

Whitebox Test Generation

Given: a code unit as the target to cover

Goal: find program input data on which the target is executed

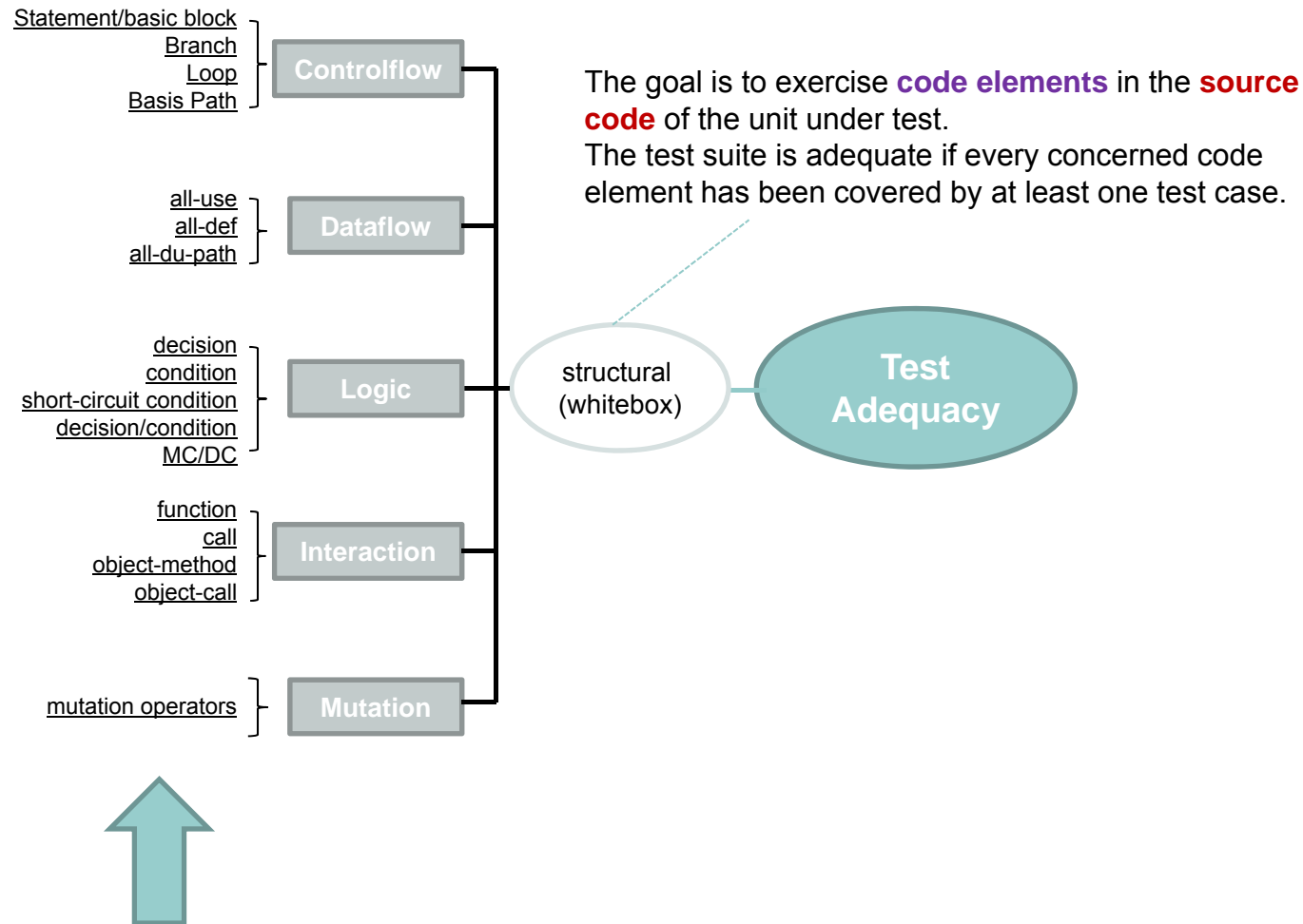
Example

```
F(int a[10], int b[10], int target) {  
    int i;  
    bool fa, fb;  
    i=1;  
    fa=false;  
    fb=false;  
    while (i < 10 {  
        if (a[i] == target) fa=true;  
        i=i + 1;  
    }  
    if (fa == true) {  
        i=1;  
        fb=true;  
        while (i < 10) {  
            if (b[i] != target) fb=false;  
            i=i+1;  
        }  
    }  
    if (fb==true) printf("message1");  
    else printf("message2");  
}
```

target statement



Fundamental Problem: Path Coverage



e.g. Condition => Path

```
1:  if(a || b && d) {    // We want condition coverage at line 1
    ...
}
```

=>

```
1:  if(a) {
2:      ...
3:  } else {
4:      if(b) {
5:          if(d) {
6:              ...
            }
        }
    }
7:
```

Cover

1 → 2 → 7 and

1 → 3 → 4 → 5 → 6 → 7 and

1 → 3 → 4 → 7 and

1 → 3 → 4 → 5 → 7

e.g. Dataflow => Path

```
1: x = 3;  
2: y = 3;  
3: if (w) {  
4:     x = y + 2;  
5: }  
6: if (z) {  
7:     y = x - 2;  
8: }  
9: n = x + y
```

We want all-use coverage for def(x) at line 1

Cover

path 1 → 2 → 3 → 6 → 7 → 9 and

path 1 → 2 → 6 → 9

e.g. Mutation => Path

```

1: x = 3;
2: y = 3;
3: if (w) {
4:     x = y + 2;
5: }
6: if (z) {
7:     y = x - 2;
8: }
9: print x;

```

Cover the paths

1 → 2 → 3 → 4 → * → 5 → 6
→ 8 → 9 or
1 → 2 → 3 → 4 → * → 5 → 6
→ 7 → 8 → 9

We want to cover the mutant that change line 4 from $x=y+2$; to $x=z+2$;

```

1: x = 3;
2: y = 3;
3: if (w) {
4:     x = z + 2;
        if(y!=z){
*           //do nothing
        }
5: }
6: if (z) {
7:     y = x - 2;
8: }
9: print x;

```

Test Generation Techniques

- Whitebox (based on source code)
 - Symbolic test generation (基于符号执行)
 - Concolic test generation (基于协同执行)
- Blackbox (based on specification)
 - Random test generation (随机数据生成)
 - Evolutionary test generation (遗传算法)
 - Grammar-based test generation (基于语法)

Basic Idea

- **Goal:** generate test data to cover a specific path
- Generate a set of **path constraints** C for a given path p such that any test data that satisfy C will cover the path p .

```
void foo(int x, int y){  
1:   int result=0;  
2:   if (x>10)  
3:       result = x;  
4:   if (y>x){  
5:       result ++;  
}
```

Path to cover: 1→2→3→4→5

Path constraints for this path: $(x>10) \ \&\& \ (y>x)$

Solution to this constraint: $(x=11, y=12)$

- Apply a *constraint solver* on C to get the actual data
 - Or you can solve the constraint by hand.

Deriving Path Constraints

Example: How to derive the path constraints for $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7$:

→	void foo (int J, int K){	• Symbolic state: $J = \alpha, K = \beta$
→	1: $J = J + 1;$	• J becomes $\alpha + 1$
→	2: if ($J > K$)	• Path constraint 1: $\alpha + 1 \leq \beta$
	3: $J = J - K;$	
	else	
→	4: $J = K - J$	• J becomes $\beta - (\alpha + 1)$
→	5: if ($J \leq -1$)	• Path constraint 2: $\beta - (\alpha + 1) > -1$
	6: $J = -J;$	
→	7: return;	
	}	
		Path constraints: $\alpha + 1 \leq \beta, \beta - (\alpha + 1) > -1$

How to Handle Language Features

- How to handle loop? e.g. `while (i<j) {...}`
 - Add the loop condition as path condition for every iteration.
- How to handle exception? e.g. `try{ a.foo();}catch (Exception e){...}`
 - Add the exception condition as the path condition.
- How to handle function call? e.g. `if (foo(x, y) < 10)`
 - Use their *symbolic output* to represent the return value.

```
int foo(int x, int y){
1:   int result=0;
2:   if (x>10)
3:       result = x;
4:   if (y>x){
5:       result ++;
6:   return result;
}
```

Symbolic output:

```
x if x>10 && y<=x
x+1 if x>10 && y>x
0 if x<=10 && y<=x
1 if x<=10 && y>x
```

Path condition for the true branch of `if(foo(x,y)<10)`:

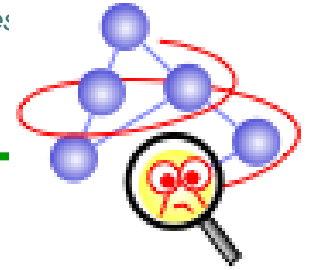
```
x>10&&y<=x&&x<10 ||
x>10&&y>x&&x+1<10 || ...
```

Generating Test Data

- Using a *constraint solver*
 - Input: constraints such as $x+1 \leq y$ and $y-(x+1) > -1$
 - Output: values that satisfy the constraints, or a conclusion that the constraints are ***unsatisfiable***.
- Example: solving linear constraints using *Gauss-Jordan elimination* (e.g. $C = \{x+y-z > 5, x-y+z < 4, y-z = 10\}$)
 - Choose a constraint c from C (e.g. $x+y-z > 5$)
 - Rewrite c into the form **$x \text{ op } \text{expr}$** (e.g. $x > 5-y+z$)
 - Eliminate x everywhere else in C by **$x \text{ op } \text{expr}$** (e.g. $4-z+y > 5-y+z$)
 - Continue until every constraint is in the form **$x \text{ op } \text{num}$** (e.g. $x > 10$)
 - Choose values that satisfy the constraints.

Generating Test Data

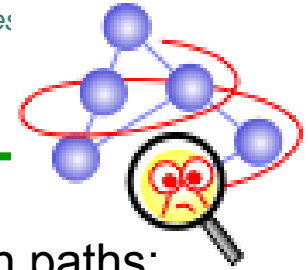
- However, constraint solving in general is a hard problem:
 - How to handle non-linear constraints? e.g. $\sin(x) > 0.5$
 - How to handle constraints about strings? e.g. `x.find("cat") == true`
 - How to handle constraints about ADTs? e.g. `stack.pop() < 10`
 - How to handle constraints about object reference or pointers?
 -
- Research is active in this field:
 - PVS: <http://pvs.csl.sri.com/>
 - JaCoP: <http://www.jacop.eu/>
 - MINION: <http://minion.sourceforge.net/>
 - ...
- But this problem has not yet been fully addressed.
 - Don't be surprised if the tool fail to handle some part of your code.



Tool 1: Java Pathfinder

<http://javapathfinder.sourceforge.net/>

- An *explicit state model checker*
 - A special JVM that runs the Java program with symbolic inputs.
 - Integrated with Eclipse.
 - Systematically explore all possible paths.
 - Path constraints and symbolic outputs are by-products.
- Have a module “symbolic” to generate one test case for every concerned path.
 - Using Choco constraint solver for linear/non-linear integer/real constraints.
 - No string, no ADT, limited support of library classes (e.g. no SWT, no networking).



Tool 1: Java Pathfinder

```
public class MyClass2 {
```

```
    ... etc ...
    private int myMethod2(int x, int y) {
        int z = x + y;
        if (z > 0) {
            z = 1;
        }
        if (x < 5) {
            z = -z;
        }
        return z;
    }
}
```

```
    ... etc ...
```

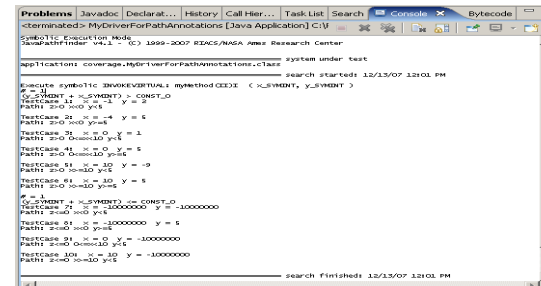
```
// The test driver
public static void main(String[] args) {
    MyClass2 mc = new MyClass2();
    int x = mc.myMethod2(1, 2);
    Debug.printPC("\nMyClass2.myMethod2 Path Condition: ");
}
}
```

Use annotation to exclude certain paths:

```
private int myMethod2(int x, int y) {
    int jpfIfCounter = 0;
    int z = x + y;
    if (z > 0) {
        jpfIfCounter++;
        z = 1;
    }
    if (x < 5) {
        jpfIfCounter++;
        Verify.ignoreIf(jpfIfCounter > 1);
        z = -z;
    }
    Verify.ignoreIf(jpfIfCounter == 0);
    return z;
}
```

Run it Java Pathfinder in Eclipse, we get:

```
Test Case 1:  y = 10000000,  x = -9999999
Test Case 2:  y = -4,        x = 5
Test Case 3:  y = -10000000, x = -10000000
Test Case 4:  y = -10000000, x = 5
```





Tool 2: PEX

<http://research.microsoft.com/en-us/projects/pex/>

- Automated unit testing for .Net
 - Come as a plugin to Microsoft Visual Studio
- Generate test input for **Parameterized Unit Test** (PUT) using symbolic execution.
 - The following PUT asserts that after adding an element to a non-null list, the element is indeed contained in the list.

```
void TestAdd(ArrayList list, object element)
{
    Assume.IsNotNull(list);
    list.Add (element);
    Assert.IsTrue(list.Contains(element));
}
```

Parameters

Precondition

Method under test

Assertion

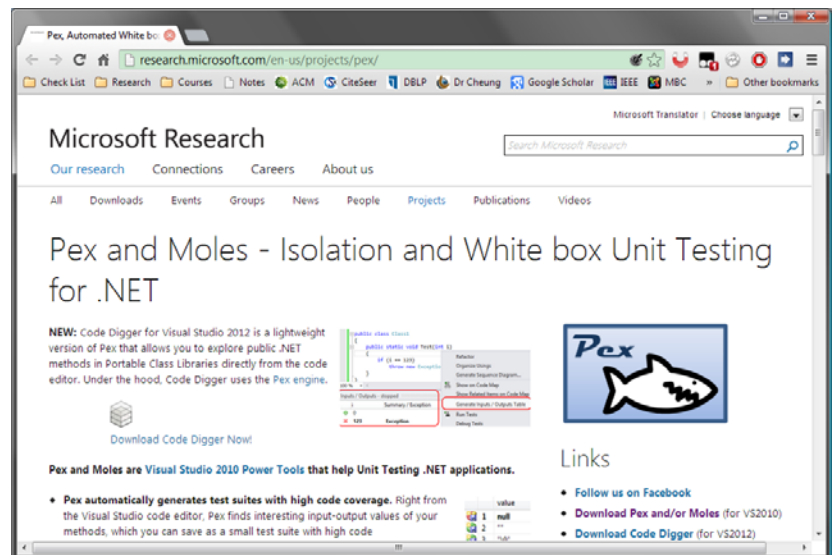
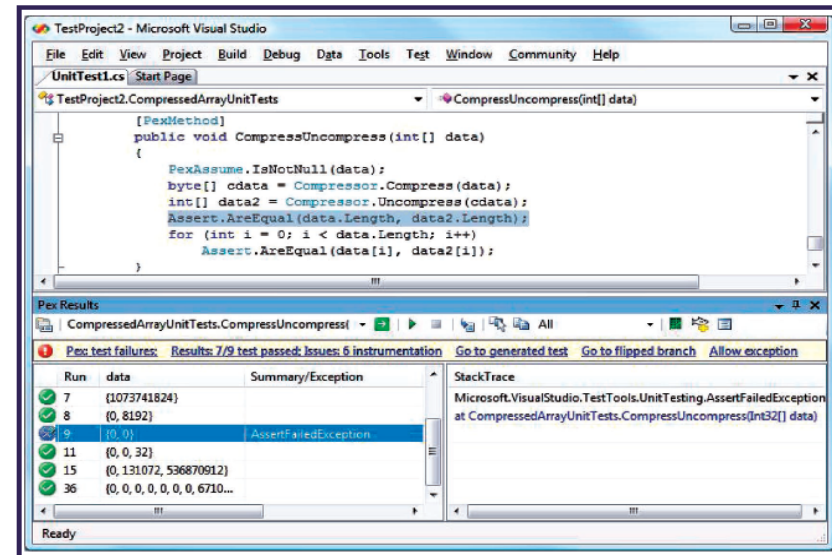
Tool 2: PEX

```
void TestAdd(ArrayList list, object element)
{
    Assume.IsNotNull(list);
    list.Add(element);
    Assert.IsTrue(list.Contains(element));
}
```

Pex will generate two concrete unit tests to achieve branch coverage:

```
void TestAdd_Generated1(){
    TestAdd(new ArrayList(0),new object());
}
```

```
void TestAdd_Generated2(){
    TestAdd(new ArrayList(1),new object());
}
```



Test Generation Techniques

- Whitebox (based on source code)
 - Symbolic test generation (基于符号执行)
 - Concolic test generation (基于协同执行)
- Blackbox (based on specification)
 - Random test generation (随机数据生成)
 - Evolutionary test generation (遗传算法)
 - Grammar-based test generation (基于语法)

Basic Idea

- Combine concrete and symbolic execution
 - **Concrete** + Symbol**ic** = **Concolic**
- In a nutshell
 - Use concrete execution over a concrete input to guide symbolic execution.
 - Concrete execution helps symbolic execution to simplify complex and unmanageable path constraints
 - By replacing symbolic values by concrete values
 - Better scalability than symbolic test generation.

Problem of Symbolic Test Generation

```
void again_test_me(int x,int y){  
S1:   z = x*x*x + 3*x*x + 9;  
S2:   if(z != y){  
S3:     printf("Good branch");  
      } else {  
S4:     printf("Bad branch");  
S5:     abort();  
      }  
}
```

- We want to cover the **bad branch**.
- Need to solve the path constraint:
 $x*x*x + 3*x*x + 9 == y$
- We have only a solver for linear constraint.
 - Fail to generate test input.



Concolic Test Generation

```

void again_test_me(int x, int y){
S1:   z = x*x*x + 3*x*x + 9;
S2:   if(z != y){
S3:     printf("Good branch");
      } else {
S4:     printf("Bad branch");
S5:     abort();
      }
}

```

- Let's first generate $x = -3$ and $y = 7$ by random test-driver.
 - Execute the program both concretely and symbolically
- After S1
 - Concrete state: $z = 9$
 - Symbolic state: $z = x_0 * x_0 * x_0 + 3 * x_0 * x_0 + 9$
- After S2
 - Concrete state:
 - go then branch as $7 \neq 9$.
 - Symbolic state:
 - Solve $x_0 * x_0 * x_0 + 3 * x_0 * x_0 + 9 == 7$ or $-3 * -3 * -3 + 3 * -3 * -3 + 9 == y_0$
 - Solution found!

A More Complete Example

```
struct cell {  
    int v;  
    cell *next;  
};
```

```
int f(int v) {  
    return 2*v + 1;  
}
```

```
int testme(cell *p, int x){  
    if (x > 0)  
        if (p != NULL)  
            if (f(x) == p->v)  
                if (p->next == p)  
                    abort();  
    return 0;  
}
```

Difficult for random test generation:

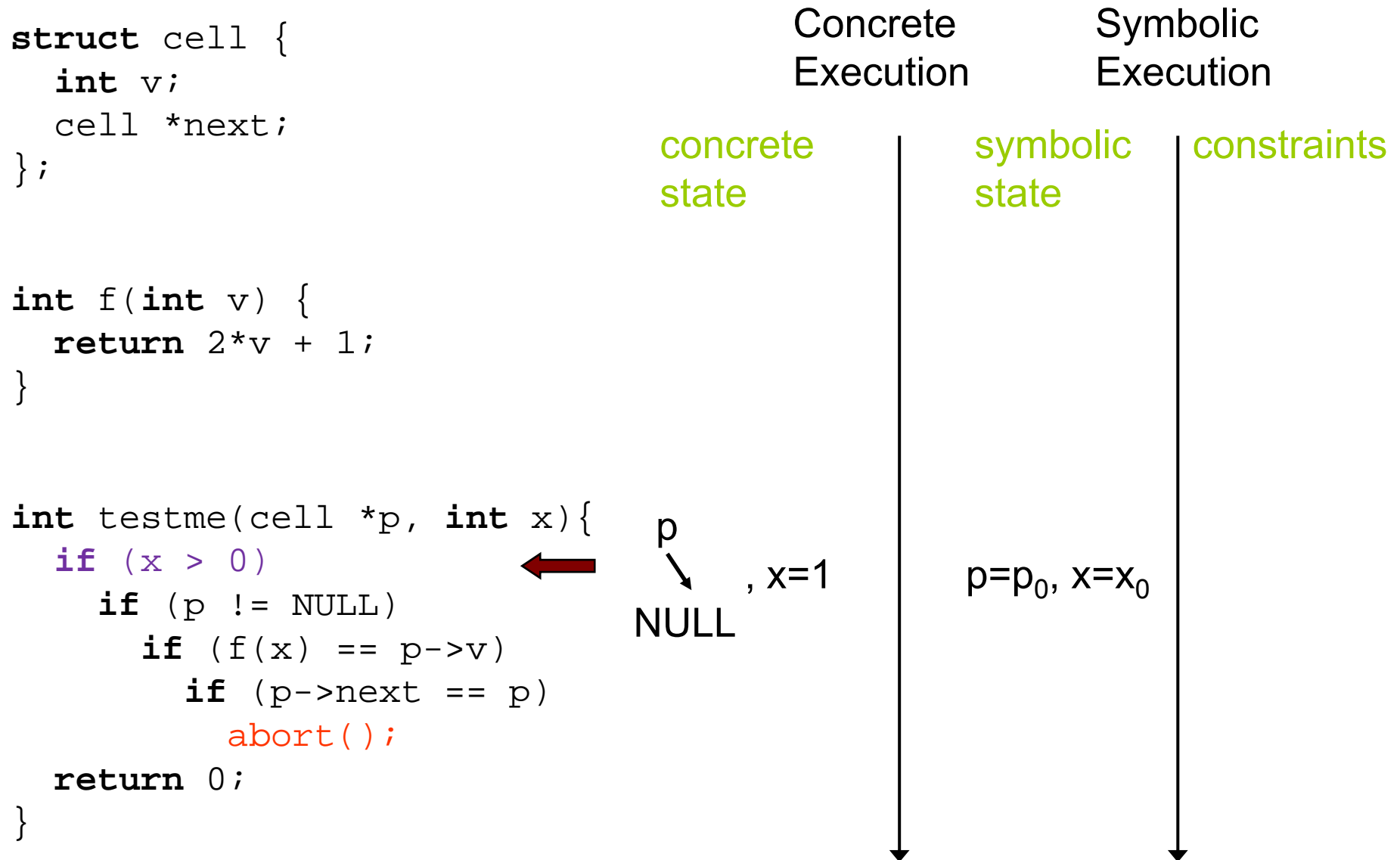
- random value for pointer p
- random value for integer x

The probability of reaching **abort()** is extremely low

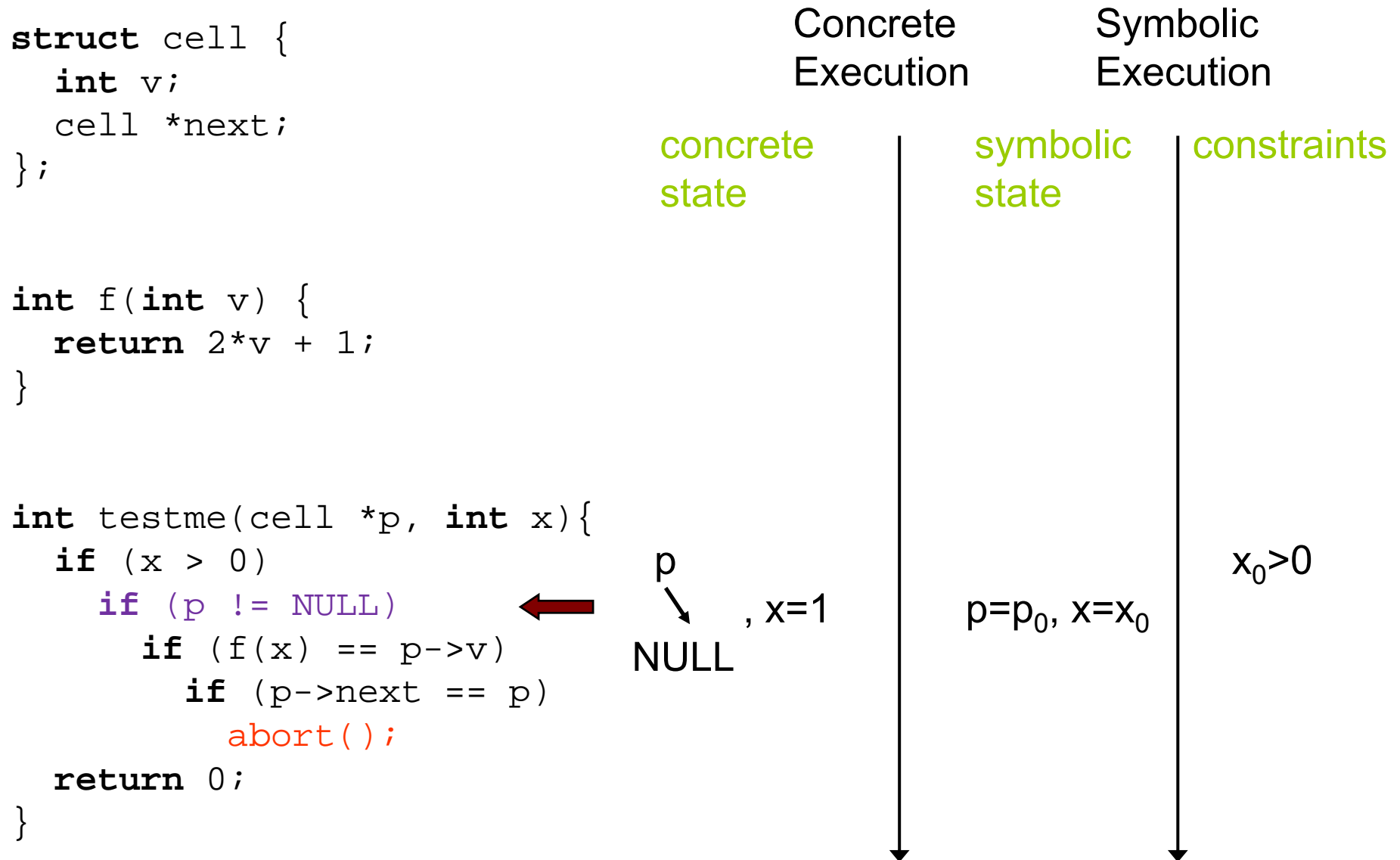
Difficult for symbol test generation as well:

- Complex path constraints

How Concolic Works



How Concolic Works



How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

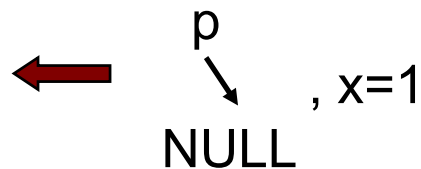
Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints



 $p \rightarrow \text{NULL}, x=1$

$p=p_0, x=x_0$

$x_0 > 0$
 $!(p_0 \neq \text{NULL})$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints

solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$

$x_0 > 0$
 $!(p_0 \neq \text{NULL})$

p
↓
NULL, $x=1$

$p=p_0, x=x_0$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

Concrete
Execution

Symbolic
Execution

concrete

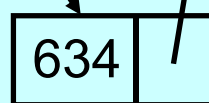
symbolic

constraints

solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$

$x_0 = 1, p_0$

NULL



$x_0 > 0$
 $!(p_0 \neq \text{NULL})$

p
↓
NULL, $x = 1$

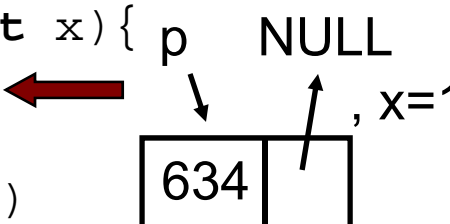
$p = p_0, x = x_0$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```



Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints

$p=p_0, x=x_0,$
 $p \rightarrow v = v_0,$
 $p \rightarrow next = n_0$

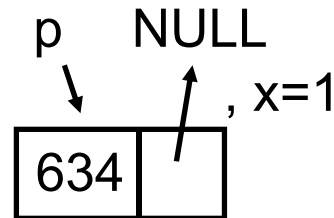
How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

concrete
state



Concrete
Execution

symbolic
state

$p=p_0, x=x_0,$
 $p \rightarrow v = v_0,$
 $p \rightarrow next = n_0$

Symbolic
Execution

constraints

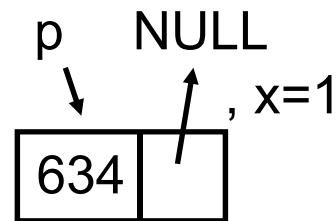
$x_0 > 0$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```



Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints

$p=p_0$, $x=x_0$,
 $p \rightarrow v = v_0$,
 $p \rightarrow \text{next} = n_0$

$x_0 > 0$
 $p_0 \neq \text{NULL}$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

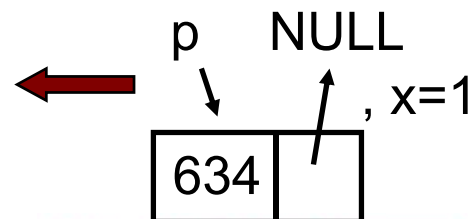
Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints



$p = p_0, x = x_0,$
 $p \rightarrow v = v_0,$
 $p \rightarrow next = n_0$

$x_0 > 0$
 $p_0 \neq \text{NULL}$
 $2x_0 + 1 \neq v_0$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

Concrete
Execution

Symbolic
Execution

concrete
state

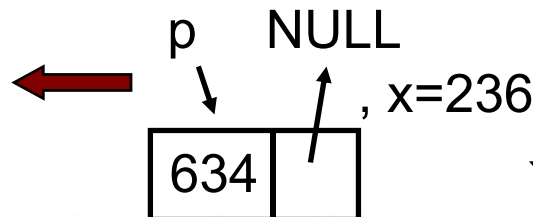
symbolic
state

constraints

solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$
and $2x_0 + 1 = v_0$

$x_0 > 0$
 $p_0 \neq \text{NULL}$
 $2x_0 + 1 \neq v_0$

$p = p_0, x = x_0,$
 $p \rightarrow v = v_0,$
 $p \rightarrow \text{next} = n_0$



How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

Concrete
Execution

Symbolic
Execution

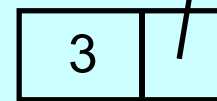
concrete

symbolic

constraints

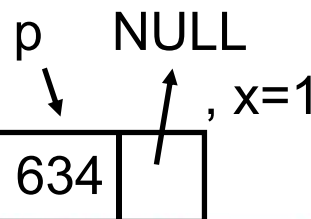
solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$
and $2x_0 + 1 = v_0$

$x_0 = 1$, p_0 NULL



$x_0 > 0$
 $p_0 \neq \text{NULL}$
 $2x_0 + 1 \neq v_0$

$p = p_0$, $x = x_0$,
 $p \rightarrow v = v_0$,
 $p \rightarrow \text{next} = n_0$

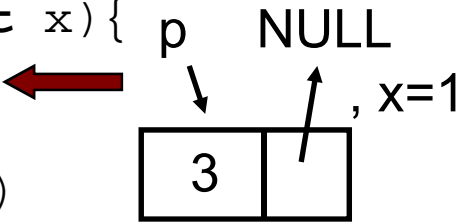


How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```



Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

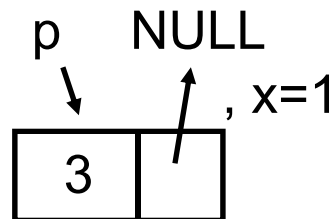
Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints



$p=p_0, x=x_0,$
 $p \rightarrow v = v_0,$
 $p \rightarrow next = n_0$

$x_0 > 0$
 $p_0 \neq \text{NULL}$
 $2x_0 + 1 = v_0$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

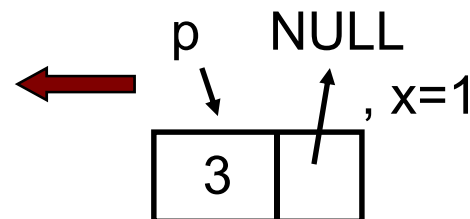
Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints



$p = p_0, x = x_0,$
 $p \rightarrow v = v_0,$
 $p \rightarrow next = n_0$

$x_0 > 0$
 $p_0 \neq \text{NULL}$
 $2x_0 + 1 = v_0$
 $n_0 \neq p_0$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

Concrete
Execution

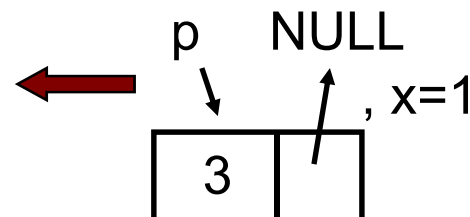
Symbolic
Execution

concrete
state

symbolic
state

constraints

solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$
and $2x_0 + 1 = v_0$ and $n_0 = p_0$



$p = p_0$, $x = x_0$,
 $p \rightarrow v = v_0$,
 $p \rightarrow \text{next} = n_0$

$x_0 > 0$
 $p_0 \neq \text{NULL}$
 $2x_0 + 1 = v_0$
 $n_0 \neq p_0$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

Concrete
Execution

Symbolic
Execution

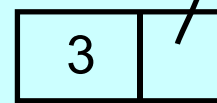
concrete
state

symbolic
state

constraints

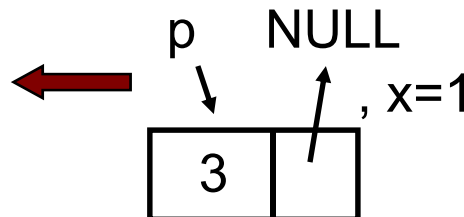
solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$
and $2x_0 + 1 = v_0$ and $n_0 = p_0$

$x_0 = 1, p_0$



$x_0 > 0$
 $p_0 \neq \text{NULL}$
 $2x_0 + 1 = v_0$
 $n_0 \neq p_0$

$p = p_0, x = x_0,$
 $p \rightarrow v = v_0,$
 $p \rightarrow \text{next} = n_0$

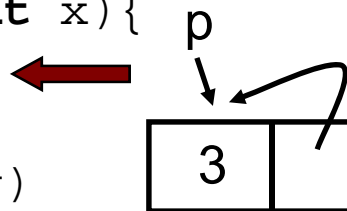


How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```



Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints

, x=1

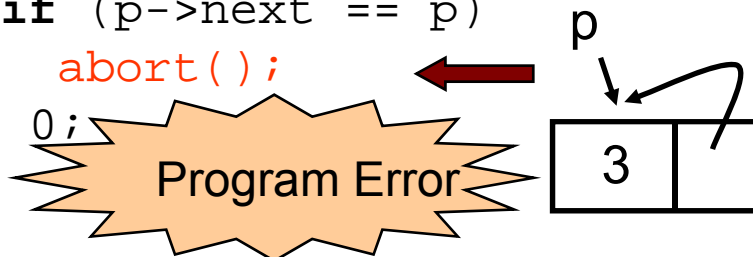
$p=p_0, x=x_0,$
 $p \rightarrow v = v_0,$
 $p \rightarrow next = n_0$

How Concolic Works

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```



Concrete
Execution

Symbolic
Execution

concrete
state

symbolic
state

constraints

, x=1

$p=p_0$, $x=x_0$,
 $p \rightarrow v = v_0$,
 $p \rightarrow next = n_0$

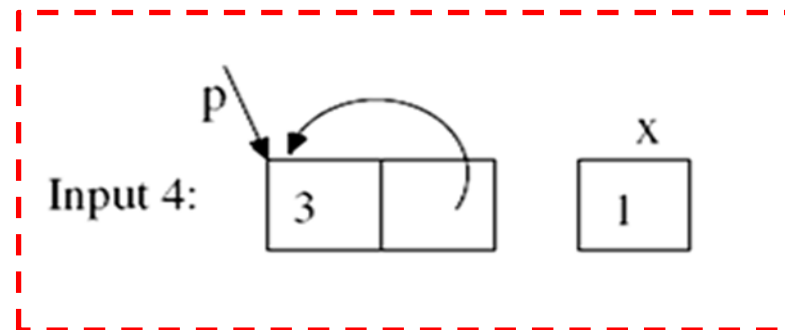
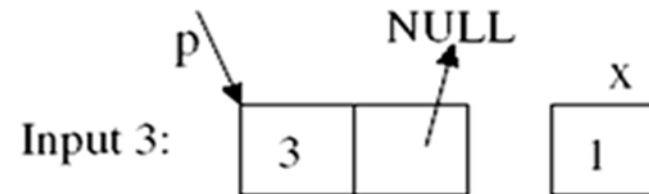
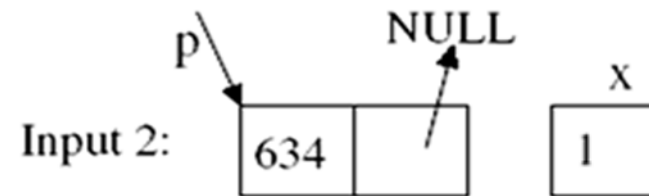
$x_0 > 0$
 $p_0 \neq \text{NULL}$
 $2x_0 + 1 = v_0$
 $n_0 = p_0$

Final Results

```
struct cell {
    int v;
    cell *next;
};
```

```
int f(int v) {
    return 2*v + 1;
}
```

```
int testme(cell *p, int x){
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```



Tool 1: CUTE and jCute

<http://osl.cs.uiuc.edu/~ksen/cute/>

- Work on C or Java programs
- Handle several non-standard data-structures
 - cu_depend (used to determine dependency during constraint solving using graph algorithm)
 - cu_linear (linear symbolic expressions)
 - cu_pointer (pointer symbolic expressions)

Tool 2: CREST

<http://code.google.com/p/crest/>

- Work on C program.
- The generated symbolic constraints are solved using Yices (<http://yices.csl.sri.com/>)
 - Only support linear, integer arithmetic.

Test Generation Techniques

- Whitebox (based on source code)
 - Symbolic test generation (基于符号执行)
 - Concolic test generation (基于协同执行)
- Blackbox (based on specification)
 - Random test generation (随机数据生成)
 - Evolutionary test generation (遗传算法)
 - Grammar-based test generation (基于语法)

Basic Idea

The principle of random testing is simple and can be described as follows:

1. For each input parameter, generate a random but legal value.
2. Apply the random values for all the parameters to the system under test
3. Check the result and then go back to step 1.

Random \neq Dumb

- Common perception: “*Random testing is, of course, the most used and least useful method*”
 - Here, random” mean “dumb”, “disorganized”, and “useless”
- Systematic random testing can be very effective
 - Theoretical work suggests that random testing is as effective as more systematic input generation techniques (Duran 1984, Hamlet 1990)
 - Fuzz testing has shown surprising results.
 - The Forrester and Miller experiment: 21% of the applications on Windows 2000 crashed when presented with random sequence of keyboard and mouse events.



Example

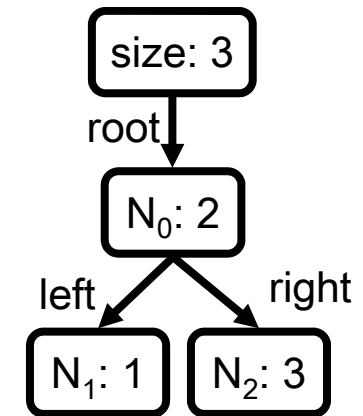
```
public BinarySearchTree {
    Node root;
    int size;
```

```
    static class Node {
        Node left, right;
        int value;
    }
```

```
    public int remove(int i) {
        .....
    }
}
```

Method under test:
b.remove();

How to randomly generate
a binary tree b?

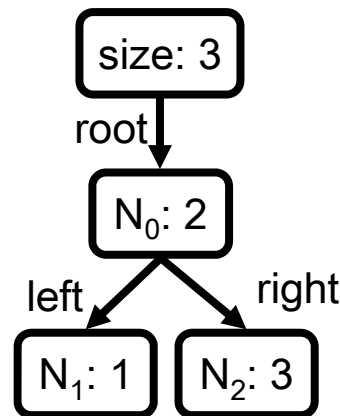


Dumb random test generation will arbitrarily set the value of root and size → most of the results are useless.

Systematic random test generation will try to generate valid binary tree only → randomize the structure of the tree in a valid way.

The key Issue

- The key issue of systematic random test generation is **how to enumerate the valid input space**.
 - This can be difficult when the input is subject to complex validity constraints.



What are the possible choices of valid binary trees?

Validity constraints:

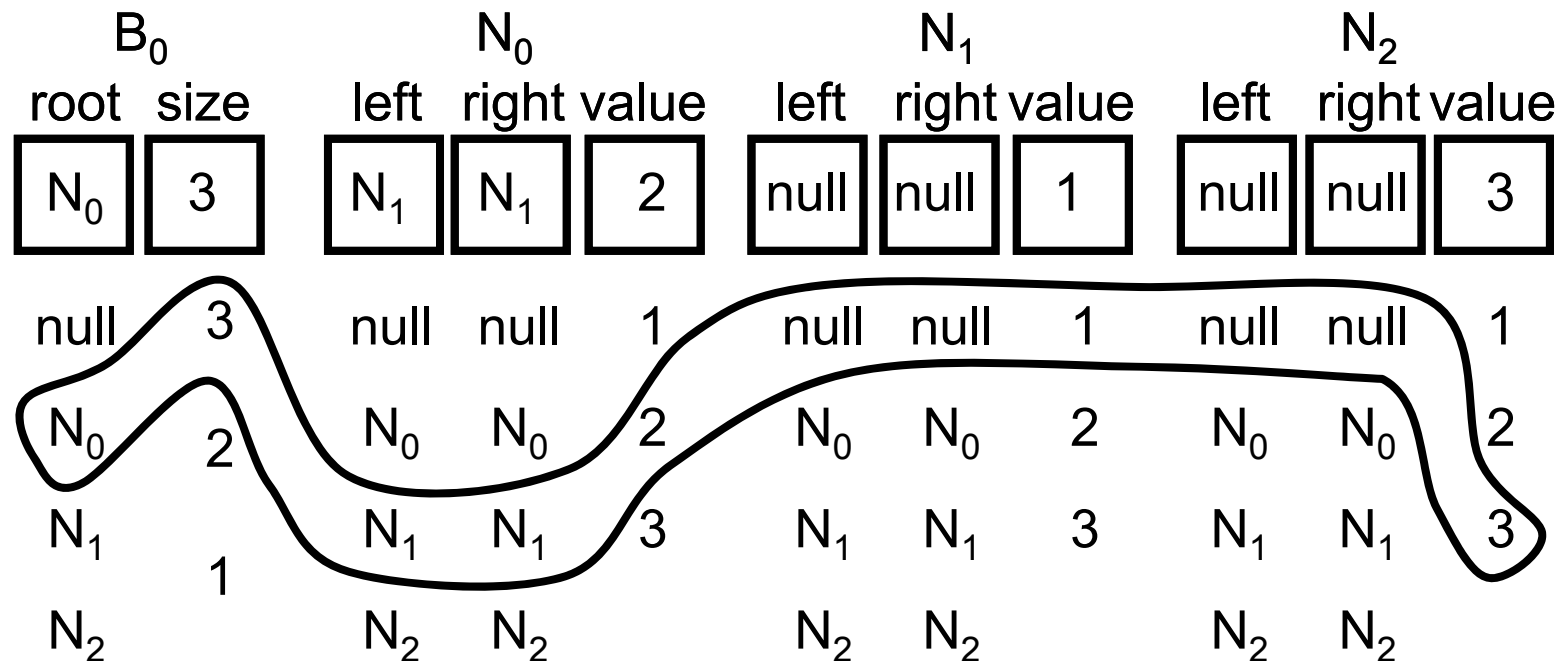
- 1) All nodes are reachable from root
- 2) No loop
- 3) Parent smaller than left child, greater or equal to right child.
- 4) size is equal to the total number of nodes.

Two Approaches to this Problem

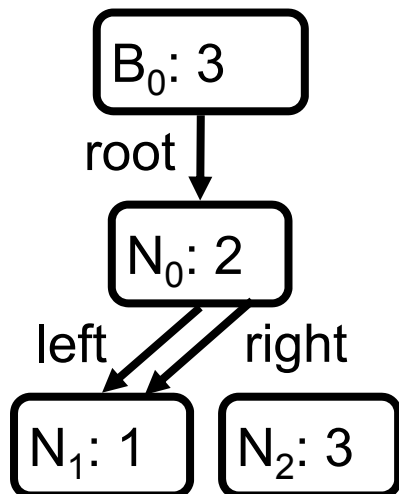
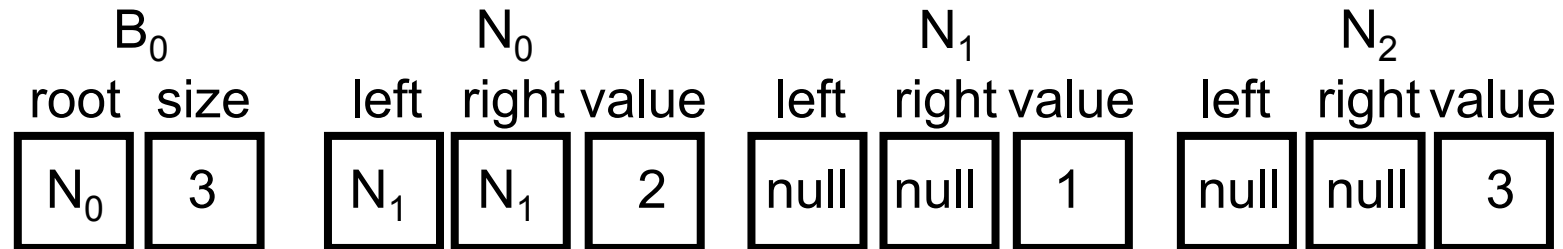
- **Approach 1:** first generate all possible input without considering validity constraints, then rule out invalid ones.
 - Example: first randomly generate all possible object graphs, then rule out those that do not satisfy the validity constraints of binary tree.
- **Approach 2:** start from a valid input, then apply all possible sequence of valid operations
 - An operation is valid if it will only transform one valid input to another valid input.
 - Example: start from an empty binary tree, then iteratively apply different sequence of the 'insert' operation.

Approach 1

- First create 1 BST object and 3 Node objects, randomly assign their fields and references to create object graphs.



Approach 1



- Step 2: Check each object graph against the validity constraints of BinaryTree. Prune those that do not satisfy the constraints.

Tool: Korat

<http://korat.sourceforge.net/>

- Rely on user-provided JML specifications to specify what are valid input object graph.
 - **Directly** generating fields and references randomly (instead of calling constructors).
 - Keep randomly generating input objects until they satisfy *the class invariant* and *precondition of the method under test*.
 - Given a predicate and a bound on the size of its inputs, Korat generates all *non-isomorphic* inputs for which the predicate returns true
 - Invokes the method-under-test on the input objects and uses method postcondition as a test oracle.

Example: Binary Tree

```
import java.util.*;
class BinaryTree {

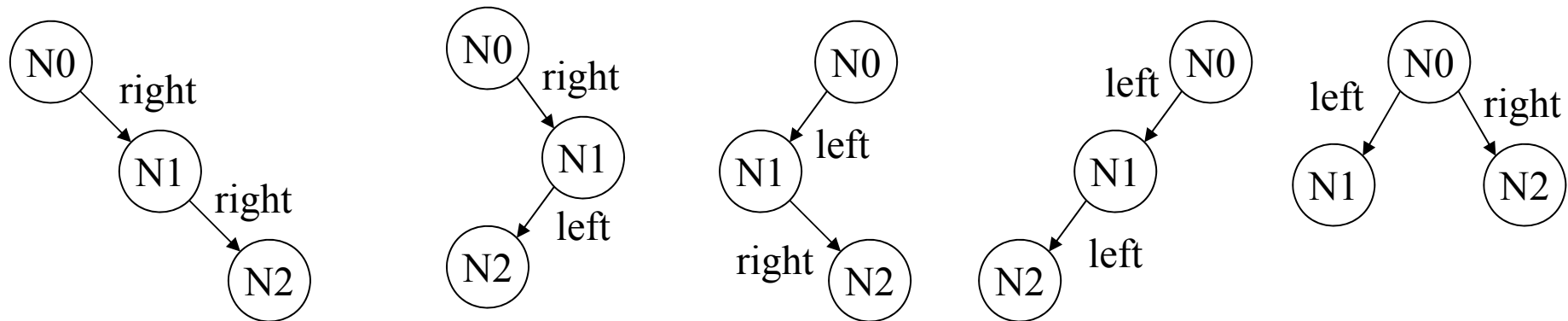
    private Node root;
    private int size;
    static class Node {
        private Node left;
        private Node right;
    }

    public boolean repOk() {
        // this method checks the class invariant:
    }
}
```

Example: Binary Tree

```
public boolean repOk() {
    // checks that empty tree has size zero
    if (root == null) return size == 0;
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node)workList.removeFirst();
        if (current.left != null) {
            // checks that tree has no cycle
            if (!visited.add(current.left)) return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            // checks that tree has no cycle
            if (!visited.add(current.right)) return false;
            workList.add(current.right);
        }
    } // checks that size is consistent
    if (visited.size() != size) return false;
    return true;
}
```

Non-isomorphic Instances



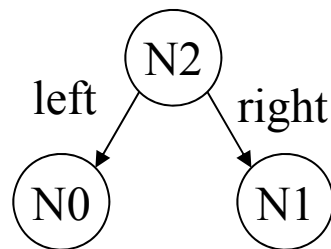
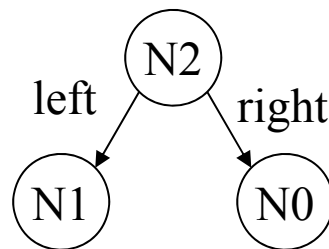
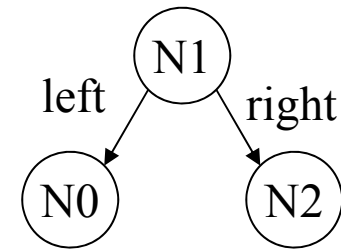
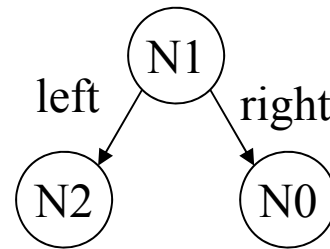
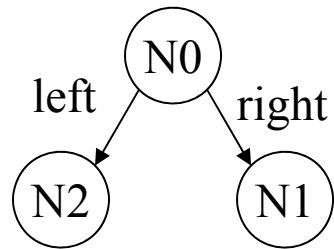
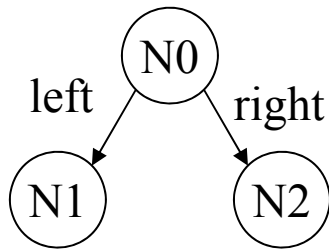
Korat automatically generates **non-isomorphic** instances within a given bound

For bound=3 (nodes), Korat generates the 5 non-isomorphic trees shown above.

Each of the above trees correspond to 6 isomorphic trees. Korat only generates one tree representing the 6 isomorphic trees.

For bound=7 (nodes), Korat generates 429 non-isomorphic trees

Isomorphic Instances



How many Instances are there?

- How many instances are there? What is the size of the state space?
- Consider the binary tree example with scope 3
 - There are three fields: root, left, right
 - Each of these fields can be assigned a node instance or null
 - There is one root field and there is one left and one right field for each node instance
- We can consider each test case for the binary tree with 3 nodes a vector with 8 values
 - The value of the root (has 4 possible values, null or a node object)
 - The value of the size (has only one possible value, which is 3)
 - For each node object (there are three of them)
 - The value of the left field (4 possible values)
 - The value of the right field (4 possible values)
- State space : $4 \times 1 \times (4 \times 4)^3$

How many Instances are there?

- Given n node instances, the state space (the set of all possible test cases) for the binary tree example is:

$$(n+1)^{2n+1}$$

- Most of these structures are not valid binary trees
 - They do not satisfy the class invariant
- Most of these structures are isomorphic
 - they are equivalent if we ignore the object identities

Isomorphism

- In Korat isomorphism is defined with respect to a root object
 - for example `this`
- Two test cases are defined to be isomorphic if the parts of their object graphs reachable from the root object are isomorphic
- The isomorphism definition partitions the state space (i.e. the input domain) to a set of isomorphism partitions
 - Korat generates only one test case for each partition class

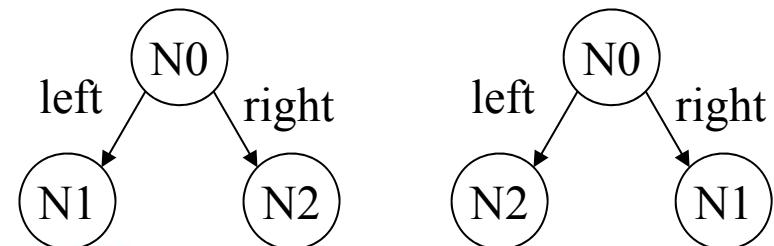
Isomorphism

- The isomorphism definition used in Korat is the following
 - O_1, O_2, \dots, O_n are objects from n classes
 - $O = O_1 \cup O_2 \cup \dots \cup O_n$
 - P : the set consisting of null and all values of primitive types that the fields of objects in O can contain
 - $r \in O$ is a special root object
 - Given a test case C , O_C is the set of objects reachable from r in C
- Two test cases C and C' are *isomorphic* iff there is a permutation π on O , mapping objects from O_C to objects from $O_{C'}$, such that:

$\forall o_1, o_2 \in O_C . \forall f \in \text{fields}(o) . \forall p \in P .$

$o_1.f == o_2 \text{ in } C \text{ iff } \pi(o_1).f == \pi(o_2) \text{ in } C' \text{ and}$

$o_1.f == p \text{ in } C \text{ iff } \pi(o_1).f == p \text{ in } C'$

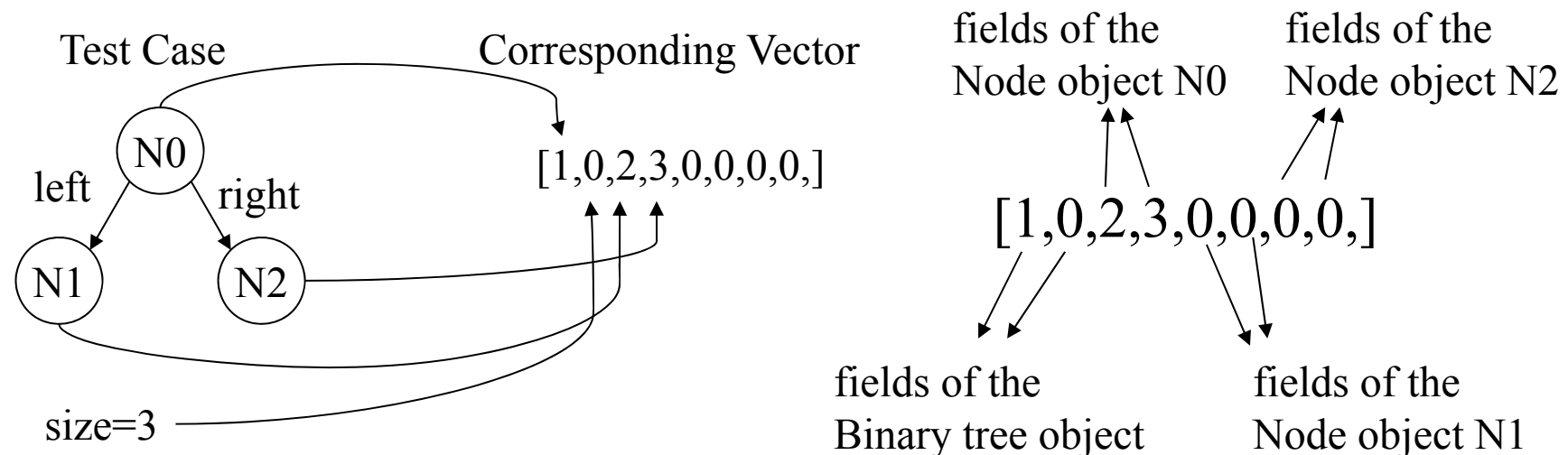


Generating Test Cases

- Korat orders all the elements in every class domain and every field domain
- Each test case is represented as a vector of indices into the corresponding field domains

For the Binary Tree example assume that:

- The class domain is ordered as $N0 < N1 < N2$
- The field domains for root, left and right are ordered as $\text{null} < N0 < N1 < N2$ (with indices 0, 1, 2, 3)
- The size domain has one element which is 3 (with index 0)



Generating Test Cases

- Search starts with a candidate vector set to all zeros.
- Korat then invokes repOk (i.e., class invariant) to check the validity of the current candidate.
 - During the execution of repOk, Korat monitors the fields that repOk accesses, it stores the indices of the fields that are accessed by the repOk (field ordering)
- Korat generates the next candidate vector by backtracking on the fields accessed by repOk.
 - First increments the field domain index for the field that is last in the field-ordering
 - If the field index exceeds the domain size, then Korat resets that index to zero and increments the domain index of the previous field in the field ordering

Using Contracts in Testing

- Korat checks the contracts written in JML on the generated instances

```
//@ public invariant repOk();           //class invariant

/*@ requires has(n)                     // precondition
   @ ensures !has(n)                    // postcondition
   @*/
public void remove(Node n) {
    ...
}
```

- Korat generates all non-isomorphic test cases within the given scope that satisfy the class invariant and the pre-condition
- The post-conditions and class invariants provide a test oracle for testing

Small Scope Hypothesis

- Success of Korat depends on the small scope hypothesis
- Small scope hypothesis
 - If there is a bug, there is typically a counter-example demonstrating the bug within a small scope.
- This is typically a valid assumption.

Approach 2

- First create a simple valid input
 - e.g. Call the default construct to create an empty binary tree.
- Next, built different valid inputs by applying different sequences of valid operations on the starting simple valid input.
 - e.g. apply a sequence of insert(int n) operations on the empty binary tree.
 - Note that if the operation has parameters then the valid values of parameters need to generated randomly as well. e.g. the parameter “n” of insert.

Tool: Randoop

<http://code.google.com/p/randoop/>

- Implement approach 2, with some improvements.
- Main improvement: use execution result to guide the generation of operation sequences.
 - Away from **redundant** or **illegal** sequences
 - Rely on user-provided JML contracts

Redundant sequence

```
Set t = new HashSet();  
s.add("hi");  
s.isEmpty();  
assertTrue(s.equals(s));
```

Illegal sequence

```
Date d = new Date(2006, 2, 14);  
d.setMonth(-1); // pre: argument >= 0  
assertTrue(d.equals(d));
```


Method Sequences

- A method sequence is a sequence of method calls:
 - `A a1 = new A();`
 - `B b3 = a1.m1(a1);`
- Each call in the sequence includes a method name and input arguments, which can be
 - Primitive values such as 0, true, null, or
 - Objects returned by previous calls
 - The receiver of the method call is treated as the first argument of the method call
- A method sequence can be written as code and executed

Generating sequences

- GenerateSequences algorithm takes a set of classes, contracts, filters and timeLimit as input
- It starts from an empty set of sequences
 - Builds sequences incrementally by extending previous sequences
- As soon as a sequence is built, it executes it to ensure that it creates a non-redundant and legal objects, as specified by filters and contracts

Algorithm

```

GenreateSequences(classes, contracts, filters, timeLimit)
errorSeqs := {}
nonErrorSeqs := {}
while timeLimit not reached do
    m(T1, T2, ..., Tk) := randomPublicMethod(classes)
    <seqs, vals> := randomSeqsAndVals(nonErrorSeqs, T1, T2, ..., Tk)
    neqSeq := extend(m, seqs, vals)
    if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
        continue
    endif
    <o, violated> := execute(newSeq, contracts)
    if violated = true then
        errorSeqs := errorSeqs ∪ {newSeq}
    else
        nonErrorSeqs := nonErrorSeqs ∪ {newSeq}
        setExtensibleFlags(neqSeq, filters, o)
    endif
endwhile
return <nonErrorSeqs, errorSeqs>

```

Errors

Some discovered errors:

- Eight of the methods in the JDK create collections that return false on `s.equals(s)`
- In Jakarta
 - a `hashCode` implementation fails to handle a valid object configuration where a field is null
 - An iterator object throws a `NullPointerException` if initialized with zero elements
- In .NET libraries
 - 155 errors are `NullReferenceExceptions` in the absence of null inputs
 - 21 are `IndexOutOfRangeException`s
 - 20 are violations of `equals`, `hashCode` and `toString` contracts

Test Generation Techniques

- Whitebox (based on source code)
 - Symbolic test generation (基于符号执行)
 - Concolic test generation (基于协同执行)
- Blackbox (based on specification)
 - Random test generation (随机数据生成)
 - Evolutionary test generation (遗传算法)
 - Grammar-based test generation (基于语法)

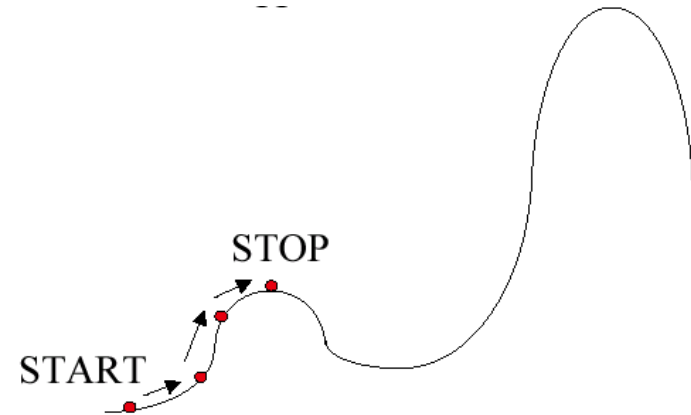
Evolutionary Test Generation

- Test generation as a search problem:
 - Given a test goal (e.g. covering a specific path), how to find the test data that satisfy this goal?
 - Problem characteristics (very hard!):
 - Large solution space
 - Discontinuous
 - Non-linearity
- Evolutionary test generation:
 - Using **genetic algorithm** to get the solution.

Test Generation as a Search Problem

- Example:

```
int foo(int x, int y) {
    ...    // k = f(x, y)
    if (k >= 100)
        XXX // branch to cover
    ...
}
```



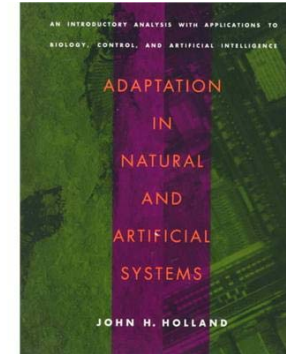
- Simple solution: Hill climbing

- Select values for x, y; if they satisfies $k \geq 100$ then stop; otherwise,
- Investigate neighboring points, such as $(x-1, y+1)$ and $(x+1, y)$
- If there is a better neighbor solution (k closer to 100), jump to it,
- Repeat steps 2-3 until current position has no better neighbors.

Problem: stop at local optimal before satisfying the coverage goal

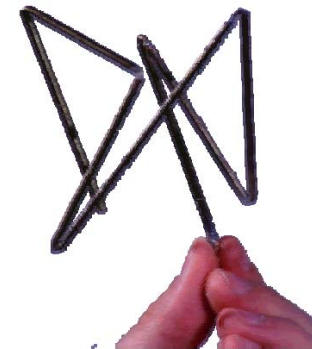
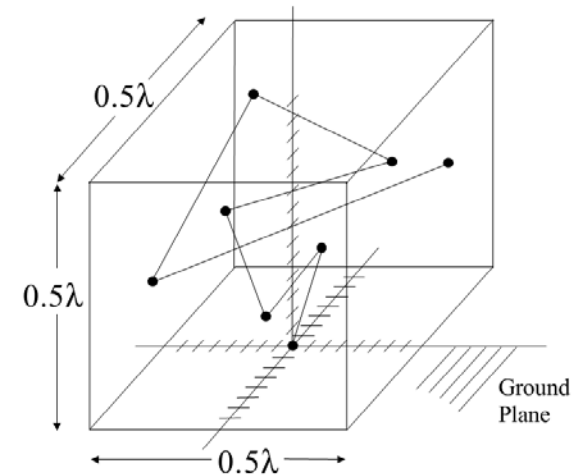
Genetic Algorithm

- Background
 - Analogy with Darwin's evolution theory
 - Developed by J. Holland in the 70s.
- Basic idea: simulate evolution
 - Maintain a population of solution candidates as individuals
 - Each individual is evaluated by its *fitness*.
 - Promising individuals are 'crossovered' or 'mutated' to get offspring individuals.
 - Repeat 1-3 until a satisfactory individual is found.



Successful Story

- NASA evolvable antenna
 - Beat the design by people with 12 years working experience.
 - “*Surprisingly good*”: Nasa intelligence report.



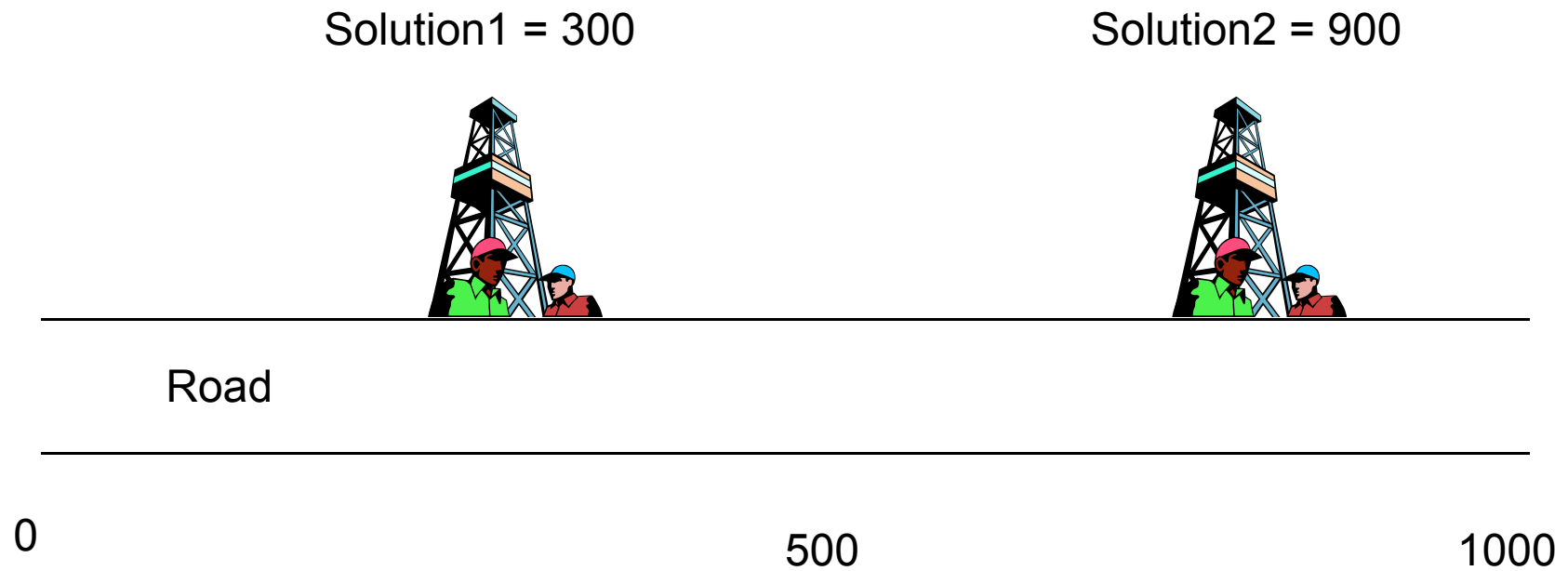
D.S. Linden. “Automated Design and Optimization of Wire Antennas using Genetic Algorithms.” Ph.D. Thesis, MIT, September 1997

Figure 7: Photograph of the Actual Crooked-Wire Genetic Antenna. From [6].

An Example - Drilling for Oil

- Imagine you had to drill for oil somewhere along a single 1km desert road
- **Problem:** choose the best place on the road that produces the most oil per day
- We could represent each solution as a position on the road
- Say, a whole number between [0..1000]

Where to drill for oil?



Digging for Oil

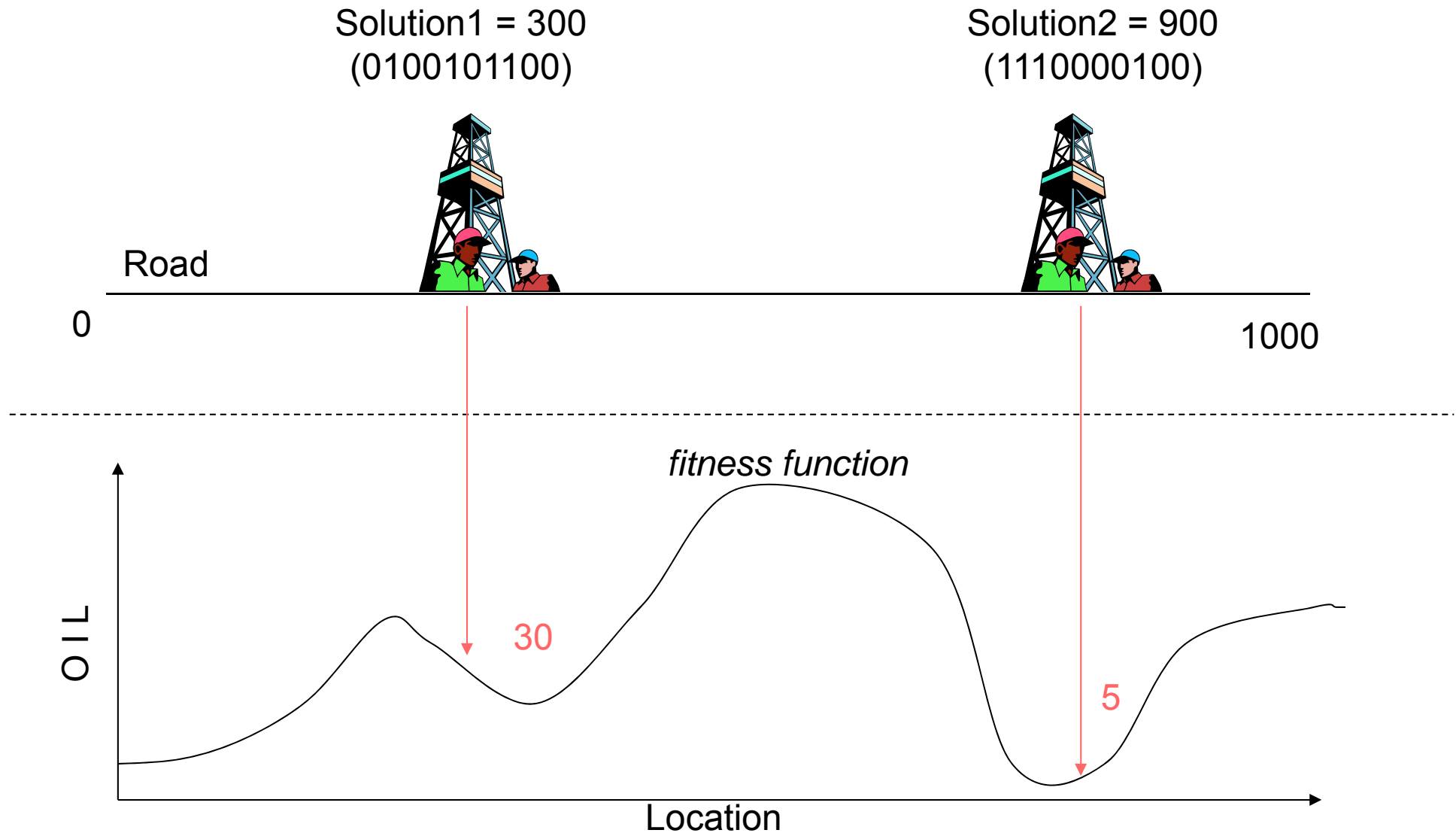
- The set of all possible solutions $[0..1000]$ is called the *search space* or *state space*
- In this case it's just one number but it could be many numbers or symbols
- Often GA's code numbers in binary producing a bitstring representing a solution
- In our example we choose 10 bits which is enough to represent $0..1000$

Convert to binary string

	512	256	128	64	32	16	8	4	2	1
900	1	1	1	0	0	0	0	1	0	0
300	0	1	0	0	1	0	1	1	0	0
1023	1	1	1	1	1	1	1	1	1	1

In GA's these encoded strings are sometimes called “*individuals*”, “*genotypes*” or “*chromosomes*” and the *bits* are sometimes called “*genes*”

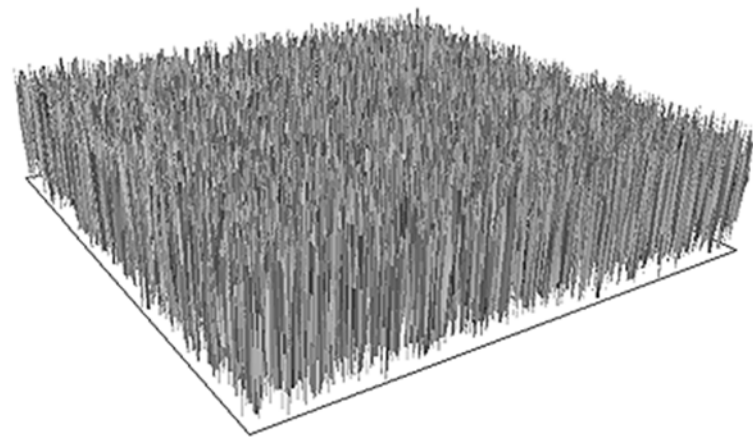
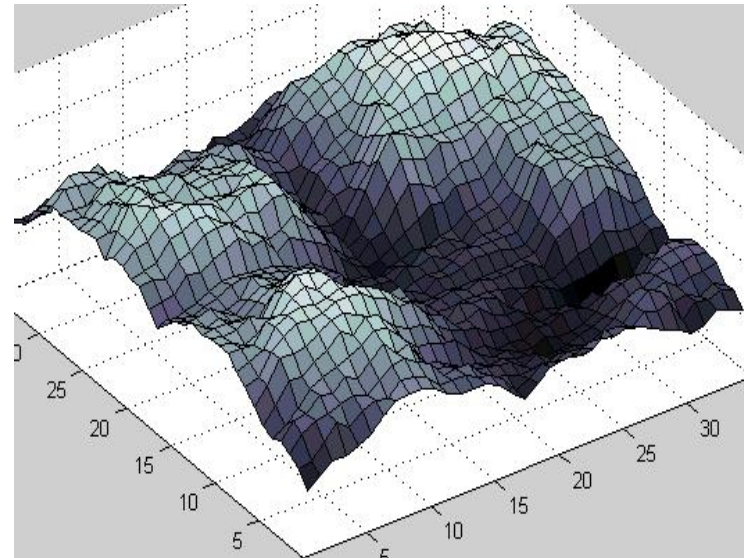
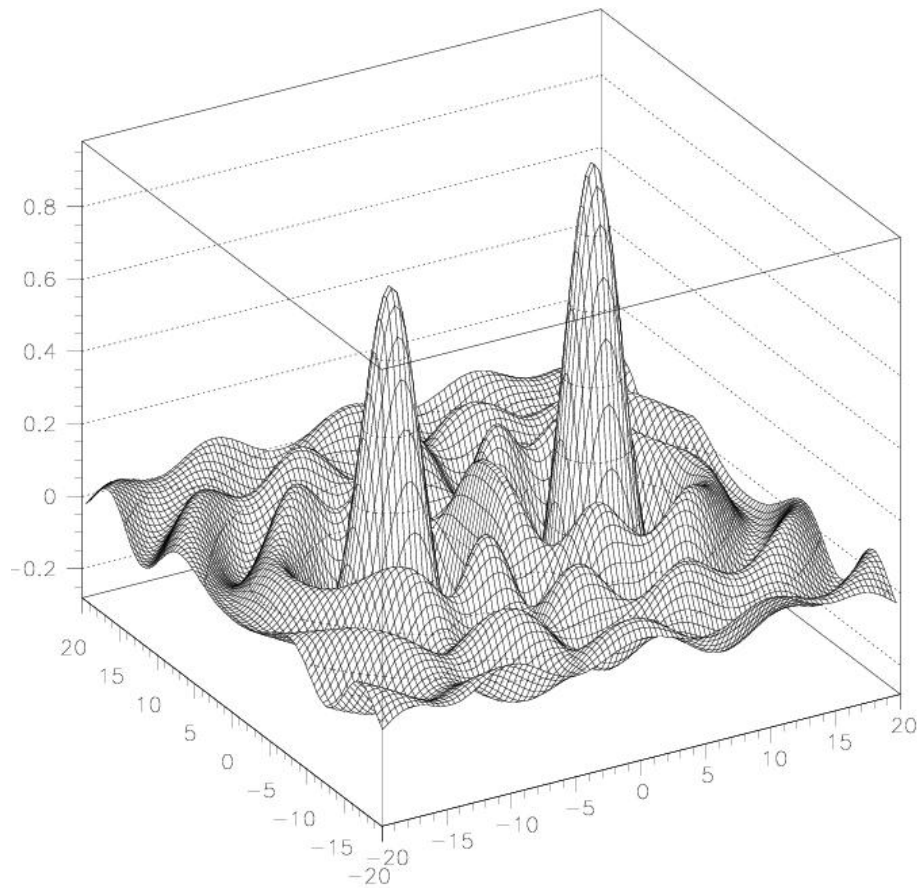
Drilling for Oil



Search Space

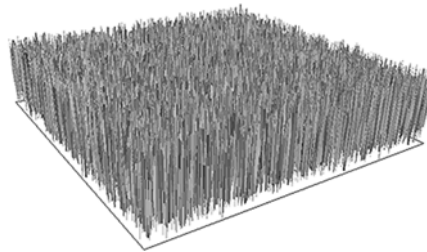
- For a simple function $f(x)$ the search space is one dimensional.
- But by encoding several values into the chromosome many dimensions can be searched e.g. two dimensions $f(x,y)$
- Search space can be visualised as a surface or *fitness landscape* in which fitness dictates height
- Each possible genotype is a point in the space
- A GA tries to move the points to better places (higher fitness) in the space

Fitness landscapes

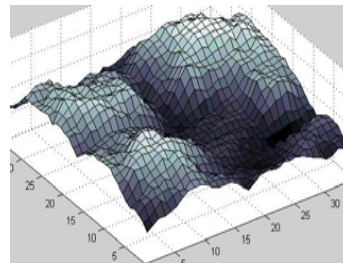


Search Space

- Obviously, the nature of the search space dictates how a GA will perform
- A completely random space would be bad for a GA



- Generally, spaces in which small improvements get closer to the global optimum are good



Back to the (GA) Algorithm

Generate a *set* of random solutions

Repeat

- Test each solution in the set (rank them)

- Remove some bad solutions from set

- Duplicate some good solutions

 - make **small changes** to some of them

Until best solution is good enough

Crossover and Mutation

- Done by selecting two parents during reproduction and combining their genes to produce offspring
 - Two high scoring “parent” bit strings (*chromosomes*) are selected and with some probability (crossover rate) combined
 - Producing two new *offspring* (bit strings)
 - Each offspring may then be changed randomly (*mutation*)

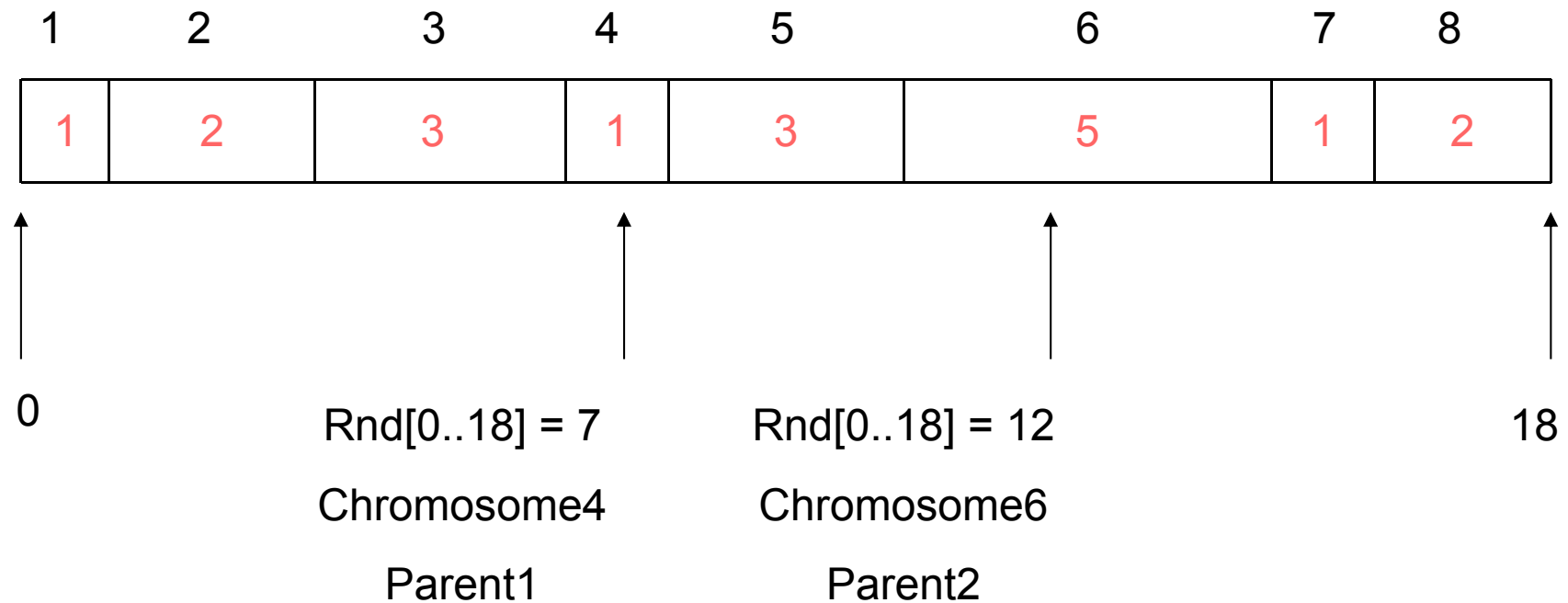
Selecting Parents

- Many schemes are possible so long as better scoring chromosomes more likely selected
- Score is often termed the *fitness*
- “Roulette Wheel” selection can be used:
 - Add up the fitness's of all chromosomes
 - Generate a random number R in that range
 - Select the first chromosome in the population that - when all previous fitness's are added - gives you at least the value R

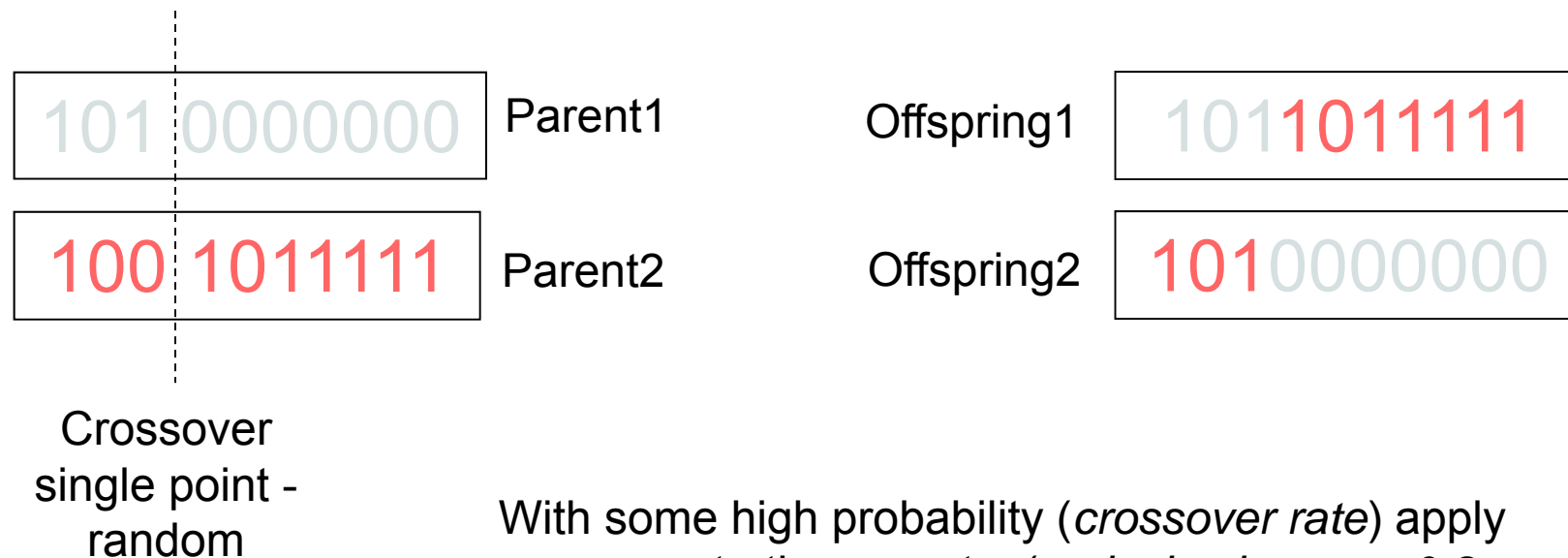
Example population

No.	Chromosome	Fitness
1	1010011010	1
2	1111100001	2
3	1011001100	3
4	1010000000	1
5	0000010000	3
6	1001011111	5
7	0101010101	1
8	1011100111	2

Roulette Wheel Selection

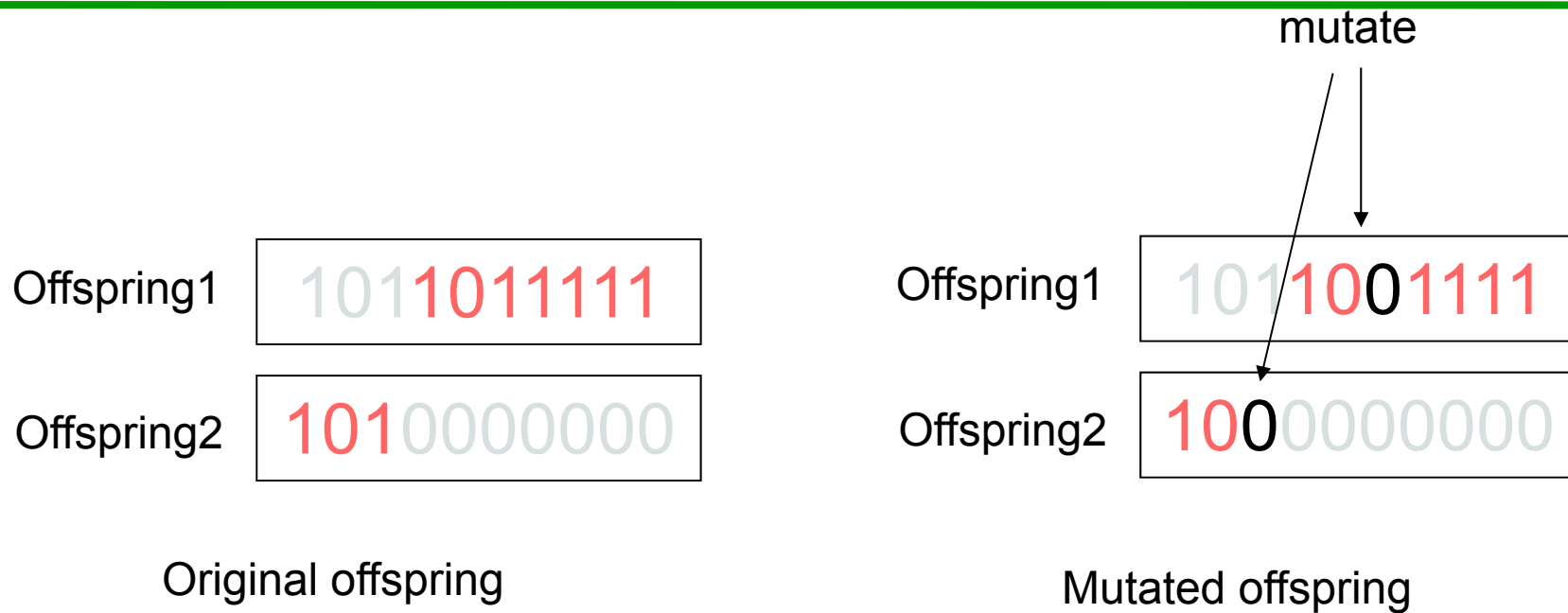


Crossover - Recombination



With some high probability (*crossover rate*) apply crossover to the parents. (*typical values are 0.8 to 0.95*)

Mutation



With some small probability (the *mutation rate*)
flip each bit in the offspring (*typical values
between 0.1 and 0.001*)

Back to the (GA) Algorithm

Generate a *population* of random chromosomes

Repeat (each generation)

- Calculate fitness of each chromosome

- Repeat

 - Use roulette selection to select pairs of parents

 - Generate offspring with crossover and mutation

- Until a new population has been produced

Until best solution is good enough

Many Variants of GA

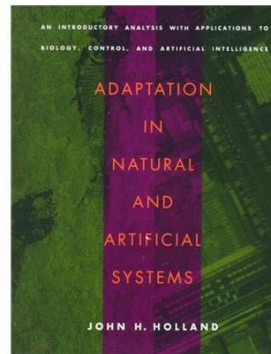
- Different kinds of selection (not roulette)
 - Tournament
 - Elitism, etc.
- Different recombination
 - Multi-point crossover
 - 3 way crossover etc.
- Different kinds of encoding other than bitstring
 - Integer values
 - Ordered set of symbols
- Different kinds of mutation

Many parameters to set

- Any GA implementation needs to decide on a number of parameters: Population size (N), mutation rate (m), crossover rate (c)
- Often these have to be “tuned” based on results obtained - no general theory to deduce good values
- Typical values might be: $N = 50$, $m = 0.05$, $c = 0.9$

Why does crossover work?

- A lot of theory about this and some controversy
- Holland introduced “Schema” theory
- The idea is that crossover preserves “good bits” from different parents, combining them to produce better solutions
- A good encoding scheme would therefore try to preserve “good bits” during crossover and mutation

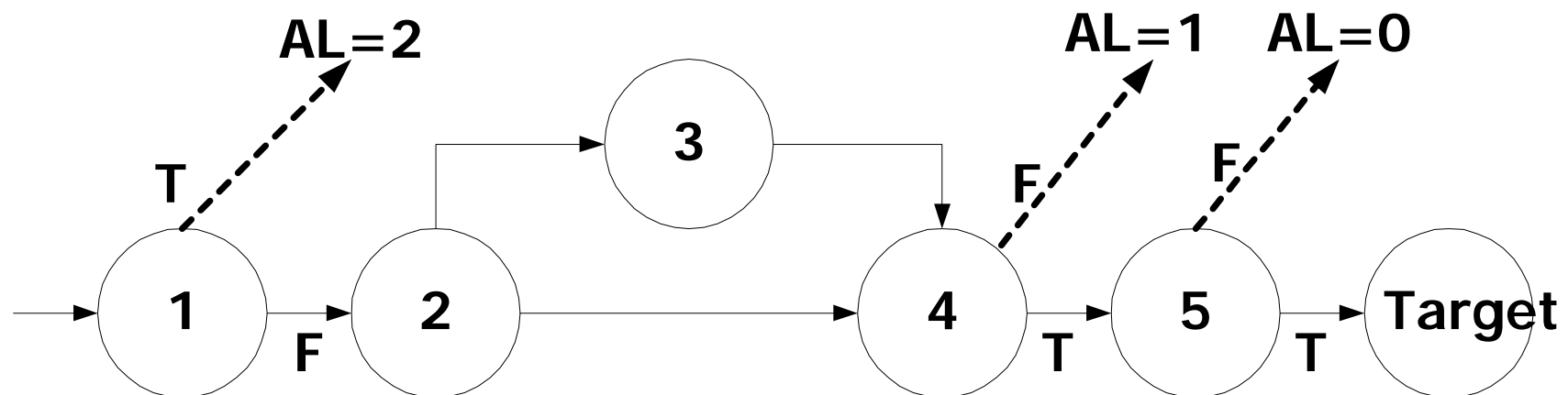


Test Generation with GA

- Testing a function with numeric parameters
 - Input variables (x_1, x_2, \dots, x_k)
 - Program Domain (D_1, D_2, \dots, D_k)
 - Individuals decode input of the program
- **Goal:** to find input data that satisfies coverage criteria (e.g. statement)

Fitness Function: $AL + D$

- AL: Approximation level
 - Critical branch: branch missing the Target
 - $AL = (\text{Number of critical branches between target and diverging point}) - 1$

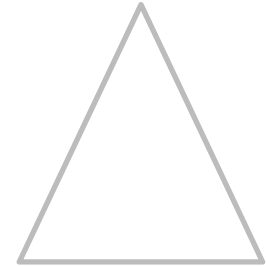


Fitness Function: $AL + D$

- D: branch distance
 - If $(x==y)$ then ...
 - if $x!=y$ then $D=abs(x-y)$ else $D=0$
 - if $(x<y)$ then ...
 - If $x>=y$ then $D=x-y$ else $D=0$
 - If (flag) then ...
 - If $flag==false$ then $D=1$ else $D=0$
- D normalize to $[0,1]$

Case Study: Triangle

- Given a small program that tell the types of triangle :
 - Scalene triangle: no two sides are equal;
 - Isosceles triangle: two equal sides;
 - Equilateral triangle: three sides of equal length;
 - Not a triangle.
- The program input : (a, b, c)
 - Representing the lengths of the sides of a triangle.
 - Limit their range as [0, 16].
- Goal: full branch coverage
 - Sub-goals: scalene, isosceles, equilateral, not-triangle



Code

Program triangle1 Fortran-like version

Dim a, b, c match As INTEGER

Input (a, b, c)

match = 0

If a = b '(1)

Then match = match + 1 '(2)

EndIf

If a = c '(3)

Then match = match + 2 '(4)

EndIf

If b = c '(5)

Then match = match + 3 '(6)

EndIf

If match = 0 '(7)

Then If (a+b) <= c '(8)

Then Output ("Not A Triangle") '(12.1)

Else If (b+c) <= a '(9)

Then Output ("Not A Triangle") '(12.2)

Else If (a+c) <= b '(10)

Then Output ("Not A Triangle") '(12.3)

Else Output ("Scalene") '(11)

EndIf

EndIf

EndIf

Else If match=1 '(13)

Then If (a+c) <= b '(14)

Then Output ("Not A Triangle") '(12.4)

Else Output ("Isosceles") '(15.1)

EndIf

Else If match=2 '(16)

Then If (a+c) <= b

Then Output ("Not A Triangle") '(12.5)

Else Output ("Isosceles") '(15.2)

EndIf

Else If match = 3 '(18)

Then If (b+c) <= a '(19)

Then Output ("Not A Triangle") '(12.6)

Else Output ("Isosceles") '(15.3)

EndIf

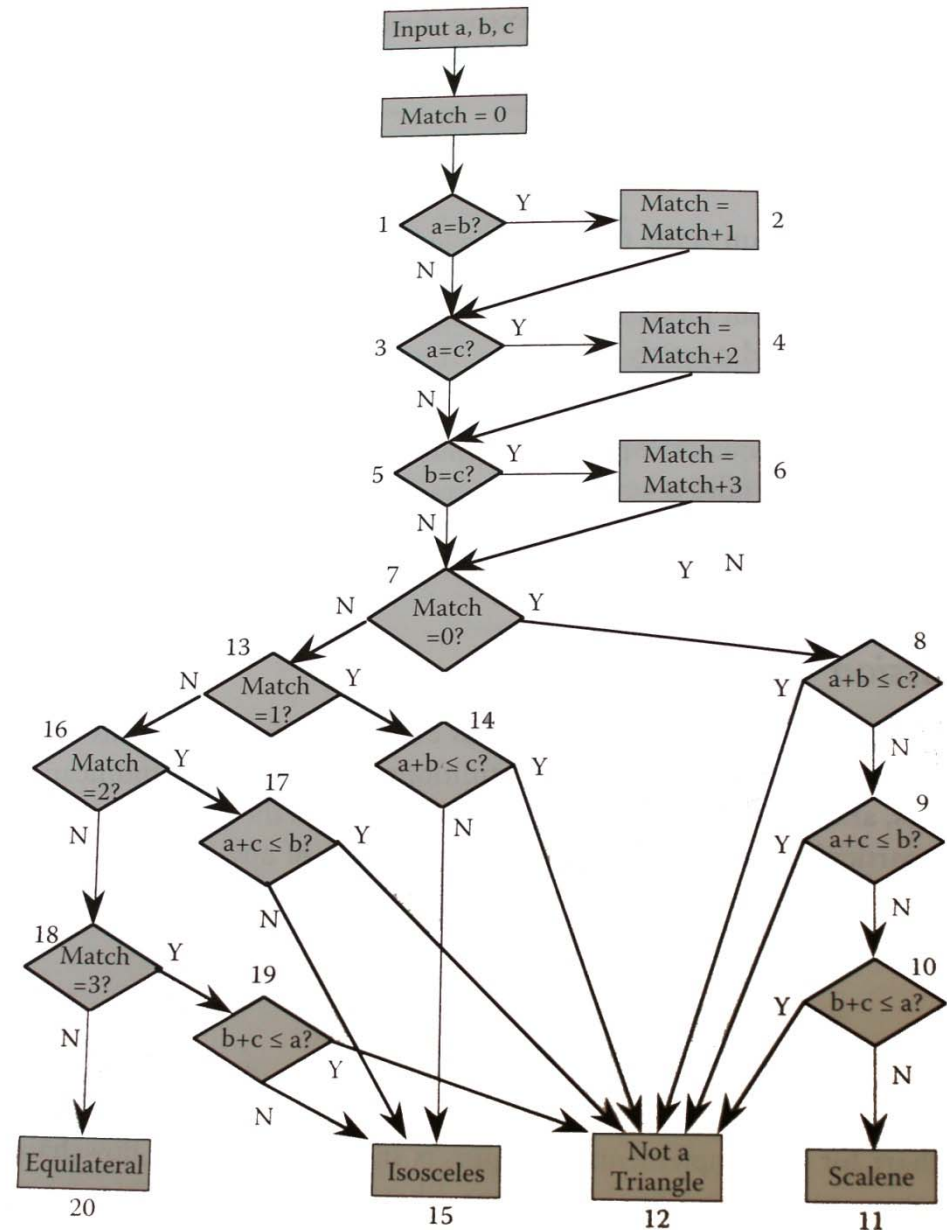
Else Output ("Equilateral") '(20)

EndIf

EndIf

EndIf

EndIf



Coding scheme

- Program domain:
 - (unsigned int, unsigned int, unsigned int)
- Encode each individual as a 0-1 string

5	7	9
0101	0111	1001
a	b	c

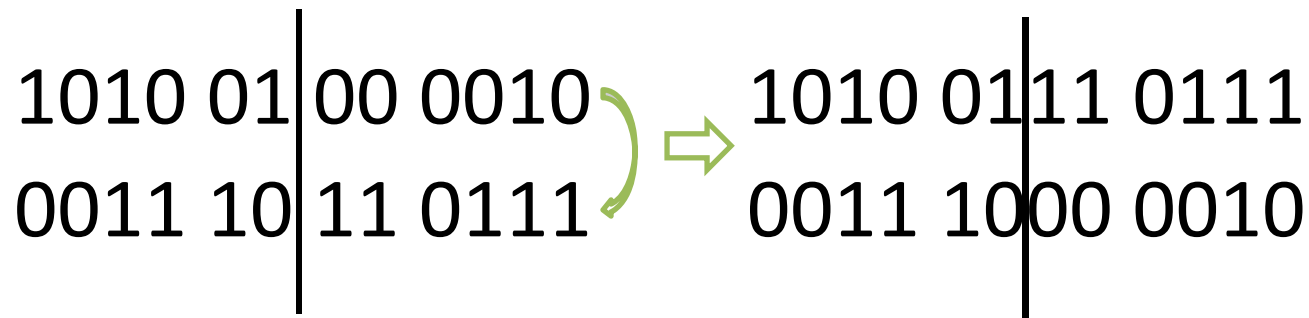
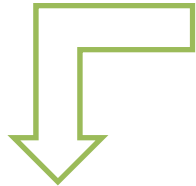
- Mutation of individuals
 - Toggling a random bit: 0 to 1, 1 to 0

Crossover Between Individual

Population: $\langle 9, 13, 5 \rangle$ Scalene triangle

$\langle 10, 4, 2 \rangle$ Not a triangle

$\langle 3, 11, 7 \rangle$ Not a triangle



Result: $\langle 10, 7, 7 \rangle$ equilateral triangle

How does this work?

- Promising individuals contain good ‘genes’ that contribute to the final solution.
 - Cross over enables them to group together.
 - Mutation enables them to evolve.
- Of course, EA is no magic. Some important factors decide the result:
 - The coding scheme of individuals.
 - The fitness function.
 - Selection method: e.g. elitist, roulette-wheel, steady-state, hierarchical, ...
 - Crossover/mutation operators

Tool 1: ECJ

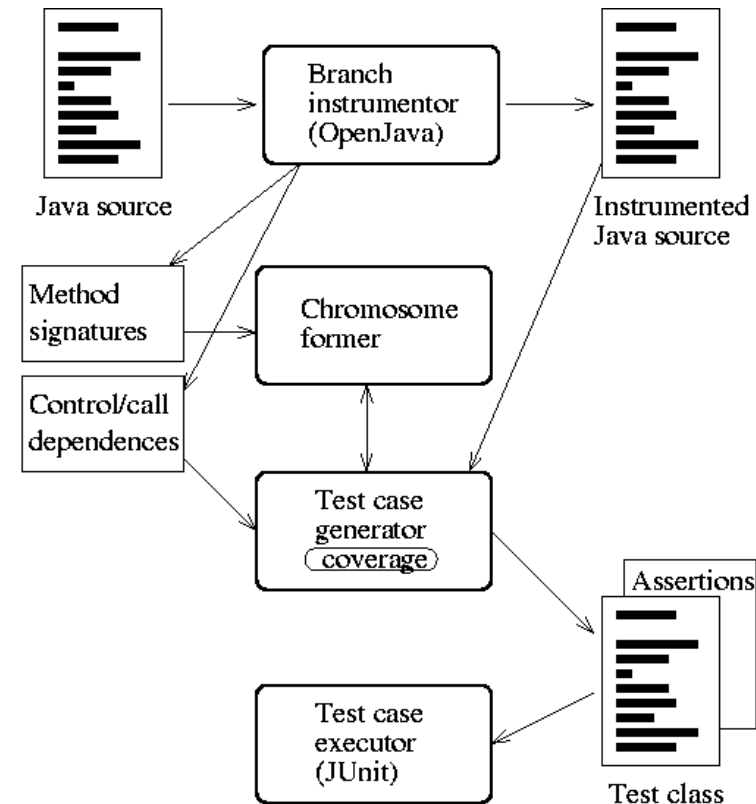
<http://cs.gmu.edu/~eclab/projects/ecj/>

- A Java-based Evolutionary Computation Research System
 - Support both *genetic algorithm* and *genetic programming*.
 - Define your own finiteness function, individual, crossover/mutation operators with Java.
 - The tool manages the evolution process with different configurable strategies.
 - Note: ECJ is also the acronym of Court of Justice of the European Union, don't Google it.

Tool 2: Etoc

<http://star.fbk.eu/etoc/>

- Open source tool for Java unit testing with GA
 - Branch coverage
 - Generate JUnit test cases
 - Developers define assertion as test oracle
 - Everything fixed



Test Generation Techniques

- Whitebox (based on source code)
 - Symbolic test generation (基于符号执行)
 - Concolic test generation (基于协同执行)
- Blackbox (based on specification)
 - Random test generation (随机数据生成)
 - Evolutionary test generation (遗传算法)
 - Grammar-based test generation (基于语法)

The Idea

- Use context grammar to describe the structure of valid input
 - Then use this grammar to generate valid input.
- Context free grammars are good at:
 - Representing inputs of varying and unbounded size
 - With recursive structure
 - And boundary conditions
- Examples:
 - Complex textual inputs
 - Trees (search trees, parse trees, ...)
 - Note XML and HTML are trees in textual form
 - Program structures
 - Which are also tree structures in textual format!

BNF

- **BNF is a notation for context-free grammars**
- BNF stands for either Backus-Naur Form or Backus Normal Form
- There are many dialects of BNF in use, but the differences are almost always minor

Basic BNF

- $\langle \rangle$ indicate a *nonterminal* that needs to be further expanded, e.g. $\langle \text{variable} \rangle$
- Symbols not enclosed in $\langle \rangle$ are *terminals*; they represent themselves, e.g. `if`, `while`, `(`
- The symbol $::=$ means *is defined as*
- The symbol $|$ means *or*; it separates alternatives, e.g. $\langle \text{addop} \rangle ::= + \mid -$
- This is *all there is* to “plain” BNF; but we will discuss *extended* BNF (EBNF) later in this lecture

Recursion

- $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{integer} \rangle \langle \text{digit} \rangle$
or
 $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle$
- Recursion is all that is needed (at least, in a formal sense)
- "Extended BNF" allows repetition as well as recursion
- Repetition is usually better when using BNF to construct a compiler

BNF Examples

- $\langle \text{digit} \rangle ::=$
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- $\langle \text{if statement} \rangle ::=$
if ($\langle \text{condition} \rangle$) $\langle \text{statement} \rangle$
| if ($\langle \text{condition} \rangle$) $\langle \text{statement} \rangle$
else $\langle \text{statement} \rangle$
- $\langle \text{unsigned integer} \rangle ::=$
 $\langle \text{digit} \rangle$ | $\langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$
- $\langle \text{integer} \rangle ::=$
 $\langle \text{unsigned integer} \rangle$
| + $\langle \text{unsigned integer} \rangle$
| - $\langle \text{unsigned integer} \rangle$

Extended BNF

- The following are pretty standard:
 - [] enclose an optional part of the rule
 - Example:
`<if statement> ::=`
`if (<condition>) <statement> [else <statement>]`
 - { } mean the enclosed can be repeated any number of times (including zero)
 - Example:
`<parameter list> ::= ()`
`| ({ <parameter> , } <parameter>)`

Case Study 1: the TSL Example

- The check-configuration function checks the validity of configuration for a laptop product. (e.g. Lenovo T series)
- The parameters of check-configuration are:
 - Model
 - Set of components

Grammar

<Model>	::= <modelNumber> <compSequence> <optCompSequence>
<compSequence>	::= <Component> <compSequence> empty
<optCompSequence>	::= <OptionalComponent> <optCompSequence> empty
<Component>	::= <ComponentType> <ComponentValue>
<OptionalComponent>	::= <ComponentType>
<modelNumber>	::= string
<ComponentType>	::= string
<ComponentValue>	::= string

Case Study 2

- (1) Input the customer number.
- (2) Check the customer number against the Customer Master File.
If the customer number does not exist, reject the transaction. If the customer number exists and its status is de-activated, go to step (3). If the customer number exists and its status is activated, go to step (4).
- (3) Check whether the transaction is specially approved by the management. If yes, accept the transaction without further checking. If not, reject the transaction.
- (4) Check the existence of a credit limit. If yes, go to step (5). Otherwise reject the transaction.
- (5) If a credit limit exists, confirm whether it is suspended. If so, reject the transaction. Otherwise go to step (6).
- (6) If the credit limit is not suspended, compare the credit limit and the invoice amount. If the credit limit is not less than the invoice amount, accept the transaction. Otherwise reject the transaction.

Translating Grammar

- (1) Input the customer number.
- (2) Check the customer number against the Customer Master File.
If the customer number does not exist, reject the transaction. If the customer number exists and its status is de-activated, go to step (3).
If the customer number exists and its status is activated, go to step (4).

<Input> ::= ID_not_exist | ID_exist deactivated Step3 | ID_exist activated Step4

Translating Grammar

- (1) Input the customer number.
- (2) Check the customer number against the Customer Master File.
If the customer number does not exist, reject the transaction. If the customer number exists and its status is de-activated, go to step (3). If the customer number exists and its status is activated, go to step (4).
- (3) Check whether the transaction is specially approved by the management. If yes, accept the transaction without further checking. If not, reject the transaction.

<Step3> ::= specially_approve | not_specially_approve

Translating Grammar

- (1) Input the customer number.
- (2) Check the customer number against the Customer Master File.
If the customer number does not exist, reject the transaction. If the customer number exists and its status is de-activated, go to step (3). If the customer number exists and its status is activated, go to step (4).
- (3) Check whether the transaction is specially approved by the management. If yes, accept the transaction without further checking. If not, reject the transaction.
- (4) Check the existence of a credit limit. If yes, go to step (5). Otherwise reject the transaction.

<Step4> ::= credit_limit_exists Step5 | credit_limit_not_exists

Translating Grammar

- (1) Input the customer number.
- (2) Check the customer number against the Customer Master File.
If the customer number does not exist, reject the transaction. If the customer number exists and its status is de-activated, go to step (3). If the customer number exists and its status is activated, go to step (4).
- (3) Check whether the transaction is specially approved by the management. If yes, accept the transaction without further checking. If not, reject the transaction.
- (4) Check the existence of a credit limit. If yes, go to step (5). Otherwise reject the transaction.
- (5) If a credit limit exists, confirm whether it is suspended. If so, reject the transaction. Otherwise go to step (6).

<Step5> ::= credit_limit_suspended | credit_limit_not_suspended Step6

Translating Grammar

- (1) Input the customer number.
- (2) Check the customer number against the Customer Master File.
If the customer number does not exist, reject the transaction. If the customer number exists and its status is de-activated, go to step (3). If the customer number exists and its status is activated, go to step (4).
- (3) Check whether the transaction is specially approved by the management. If yes, accept the transaction without further checking. If not, reject the transaction.
- (4) Check the existence of a credit limit. If yes, go to step (5). Otherwise reject the transaction.
- (5) If a credit limit exists, confirm whether it is suspended. If so, reject the transaction. Otherwise go to step (6).
- (6) If the credit limit is not suspended, compare the credit limit and the invoice amount. If the credit limit is not less than the invoice amount, accept the transaction. Otherwise reject the transaction.

<Step6> ::= within_credit_limit | exceed_credit_limit

Grammar

<Input> ::= ID_not_exist | ID_exist deactivated Step3 | ID_exist activated Step4

<Step3> ::= specially_approve | not_specially_approve

<Step4> ::= credit_limit_exists Step5 | credit_limit_not_exists

<Step5> ::= credit_limit_suspended | credit_limit_not_suspended Step6

<Step6> ::= within_credit_limit | exceed_credit_limit

Case Study 3: STS

```

RECORDS ::= {PITCH_RECORD | TEAM_RECORD}*
PITCH_RECORD ::= '<begin' 'pitch' '>' {PITCH_ATTRIBUTES}* '<\end' 'pitch' '>'
PITCH_ATTRIBUTES ::= PITCH_LENGTH | PITCH_WIDTH
PITCH_LENGTH ::= '<begin' 'length' '>' NUMBER '<\end' 'length' '>'
PITCH_WIDTH ::= '<begin' 'width' '>' NUMBER '<\end' 'width' '>'
TEAM_RECORD ::= '<begin' 'team' '>' {TEAM_ATTRIBUTES}* '<\end' 'team' '>'
TEAM_ATTRIBUTES ::= TEAM_NAME | TEAM_NUMBER_OF_PLAYERS | TEAM_STRATEGY
TEAM_NAME ::= WORD {' ' WORD}
TEAM_NUMBER_OF_PLAYERS ::= NUMBER
TEAM_STRATEGY ::= 'random' | 'custom' TEAM_REGIONS
TEAM_REGIONS ::= {'<region> POINT POINT '<\region>'}*
POINT ::= '(' NUMBER ',' NUMBER ')'
NUMBER ::= DIGIT NUMBER | DIGIT
DIGIT ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
WORD ::= LETTER WORD | LETTER
LETTER ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'

```

Generating Input from BNF

$$S ::= \langle \text{Brace} \rangle \mid \langle \text{Curly} \rangle \mid \varepsilon$$

$$\text{Brace} ::= (\langle S \rangle) \langle S \rangle$$

$$\text{Curly} ::= \{ \langle S \rangle \} \langle S \rangle$$

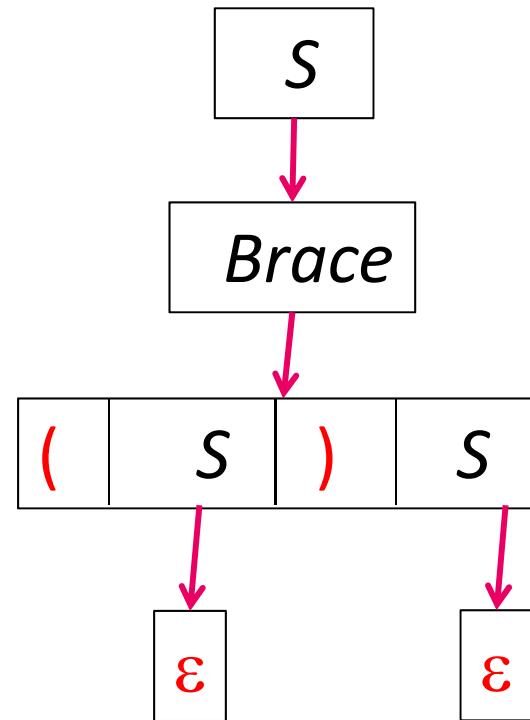
A **derivation** is a series of expansion of the grammar that result in a sequence of terminal symbols. It follows that the sequence is a valid sentence of the grammar. We can use this to generate valid sentences. Example :

$$\begin{aligned} S &\rightarrow \text{Brace} \\ &\rightarrow (S) S \\ &\rightarrow (\varepsilon) S \\ &\rightarrow (\varepsilon) \varepsilon \end{aligned}$$

Derivation Tree

$S ::= \langle \text{Brace} \rangle$
 $S ::= \langle \text{Curly} \rangle$
 $S ::= \epsilon$
 $\text{Brace} ::= (\langle S \rangle) \langle S \rangle$
 $\text{Curly} ::= \{ \langle S \rangle \} \langle S \rangle$

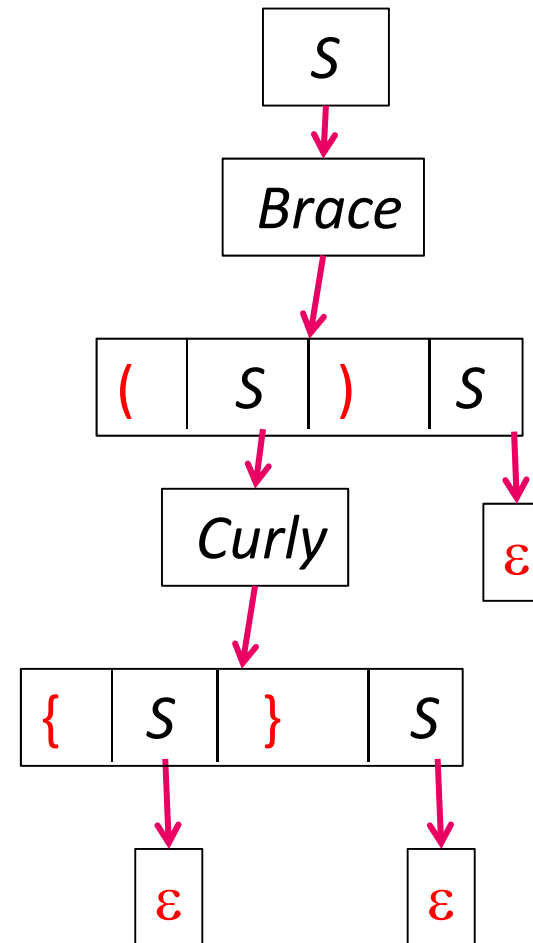
$S \rightarrow \text{Brace}$
 $\rightarrow (S) S$
 $\rightarrow (\epsilon) S$
 $\rightarrow (\epsilon) \epsilon$



A derivation can also be described by a derivation tree such as above. Given such a tree, you can reconstruct what the derived sentence is.

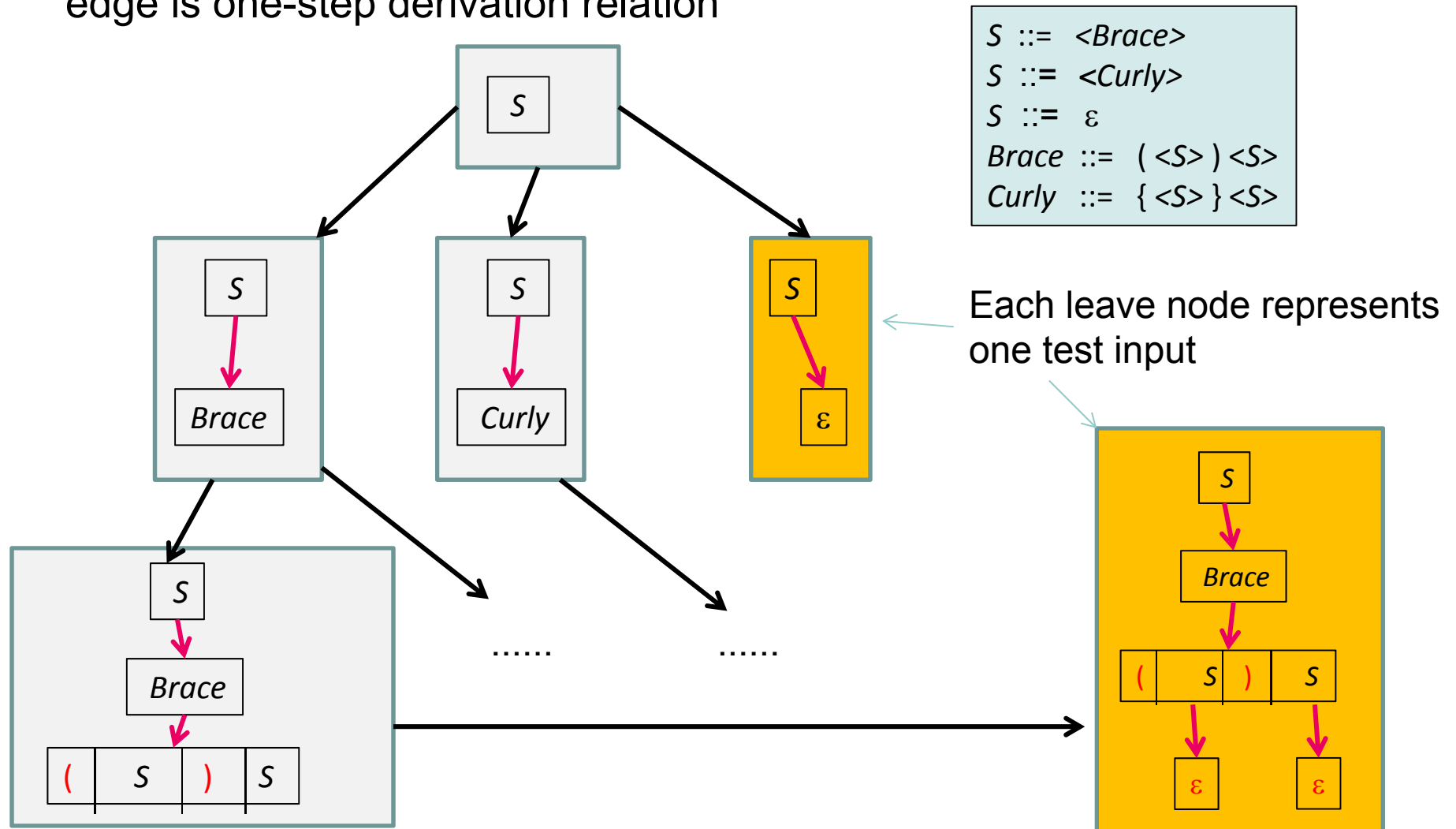
One More Example

$S ::= \langle \text{Brace} \rangle$
 $S ::= \langle \text{Curly} \rangle$
 $S ::= \varepsilon$
 $\text{Brace} ::= (\langle S \rangle) \langle S \rangle$
 $\text{Curly} ::= \{ \langle S \rangle \} \langle S \rangle$



Basic Idea of Test Generation

- Depth-first traverse of a graph in which node is derivation tree, and edge is one-step derivation relation



Thank you!

