SOFT 3416
Software Verification and Validation
Week XII

Ahmet Feyzi Ateş

Işık University Faculty of Engineering and Natural Sciences
Department of Computer Science Engineering
2023-2024 Spring Semester

Integration Tetsing Example: NextDate function from Calendar

- ***Problem Statement:*** NextDate is a function of three variables: month, date, and year. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions.
  - c1. $1 \le \text{month} \le 12$
  - c2. $1 \le \text{day} \le 31$
  - c3. $1842 \le \text{year} \le 2042$ (the year range staring in 1842 and ending in 2042 is arbitrary)
- If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value;
  - for example, “Value of month not in the range 1...12.”
- Because numerous invalid day–month–year combinations exist (such as June 31 of any year.), NextDate collapses these into one message:
  - “Invalid Input Date.”

Example: NextDate from Calendar

A main program with a functional decomposition into several procedures and functions.

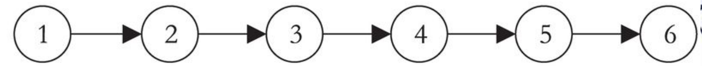
```
1  Main integrationNextDate

     Type  Date
         Month As Integer
         Day As Integer
         Year As Integer
     EndType

     Dim today As Date
     Dim tomorrow As Date

2    GetDate(today)                          'msg1
3    PrintDate(today)                        'msg2
4    tomorrow = IncrementDate(today)         'msg3
5    PrintDate(tomorrow)                     'msg4
6  End Main
```

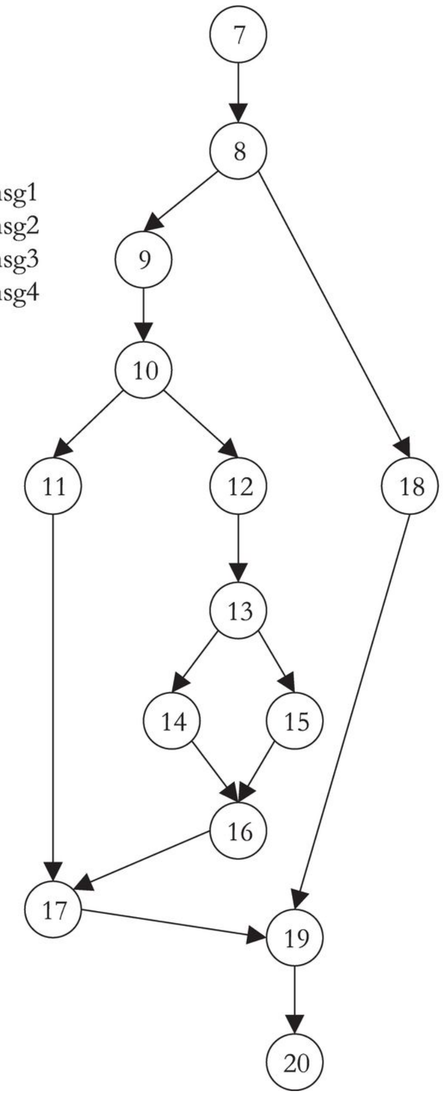
Main V(G) = 1



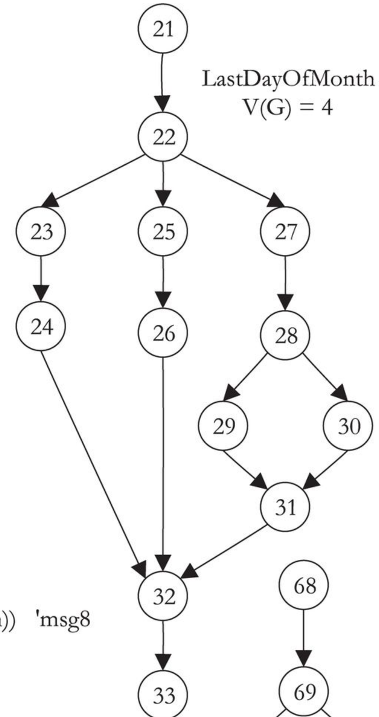
isLeap V(G) = 4

the source code, the program graphs, and the cyclomatic complexity of the units in the procedural version of *NextDate()* is given.

```
7  Function isLeap(year)  Boolean
8    If (year divisible by 4)
9      Then
10         If (year is NOT divisible by 100)
11           Then isLeap = True
12           Else
13             If (year is divisible by 400)
14               Then isLeap = True
15               Else isLeap = False
16             EndIf
17         EndIf
18       Else isLeap = False
19     EndIf
20 End (Function isLeap)
```

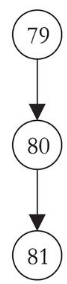


```
21 Function lastDayOfMonth(month, year)  Integer
22   Case month Of
23     Case 1: 1, 3, 5, 7, 8, 10, 12
24       lastDayOfMonth = 31
25     Case 2: 4, 6, 9, 11
26       lastDayOfMonth = 30
27     Case 3: 2
28       If (isLeap(year))                        'msg5
29         Then lastDayOfMonth = 29
30         Else lastDayOfMonth = 28
31       EndIf
32   EndCase
33 End (Function lastDayOfMonth)
```



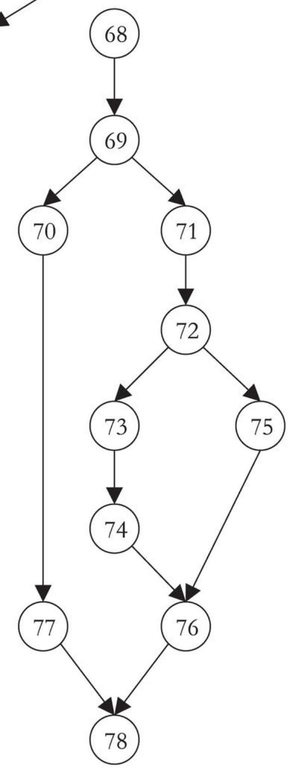
LastDayOfMonth
V(G) = 4

```
68 Function IncrementDate(aDate)  Date
69   If (aDate.Day < lastDayOfMonth(aDate.Month))   'msg8
70     Then aDate.Day = aDate.Day + 1
71     Else aDate.Day = 1
72       If (aDate.Month = 12)
73         Then aDate.Month = 1
74           aDate.Year = aDate.Year + 1
75         Else aDate.Month = aDate.Month + 1
76       EndIf
77   EndIf
78 End (IncrementDate)
```



PrintDate V(G) = 1

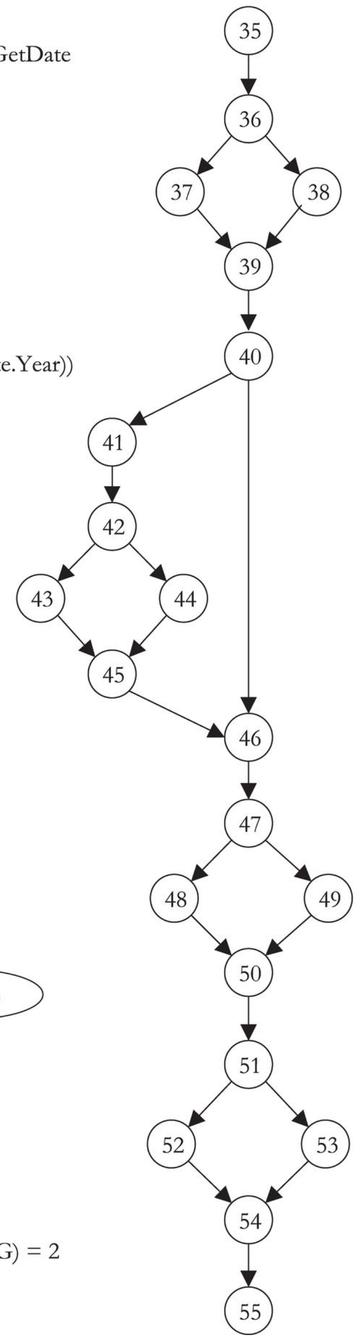
```
79 Procedure PrintDate(aDate)
80   Output( "Day is ", aDate.Month, "/", aDate.Day, "/",
aDate.Year)
81 End (PrintDate)
```



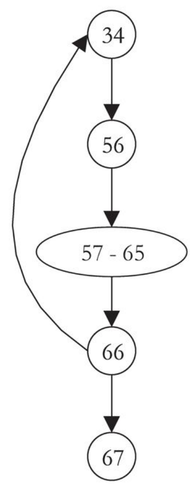
IncrementDate V(G) = 3

```
34 Function GetDate(aDate)  Date
   dim aDate As Date
35  Function ValidDate(aDate)  Boolean  'within scope of GetDate
   dim aDate As Date
   dim dayOK, monthOK, yearOK As Boolean
36    If ((aDate.Month > 0) AND (aDate.Month <=12)
          'added decisional complexity = +1
37      Then monthOK = True
38      Else monthOK = False
39    EndIf
40    If (monthOK)
41      Then
42        If ((aDate.Day > 0) AND               'msg6
        (aDate.Day <= lastDayOfMonth(aDate.Month, aDate.Year))
          'added decisional complexity = +1
43          Then dayOK = True
44          Else dayOK = False
45        EndIf
46      EndIf
47   If ((aDate.Year > 1811) AND (aDate.Year <= 2012)
          'added decisional complexity = +1
48      Then yearOK = True
49      Else yearOK = False
50      EndIf
51   If (monthOK AND dayOK AND yearOK)
          'added decisional complexity = +2
52      Then ValidDate = True
53      Else ValidDate = False
54    EndIf
55    End (Function ValidDate)
```

ValidDate V(G) = 6

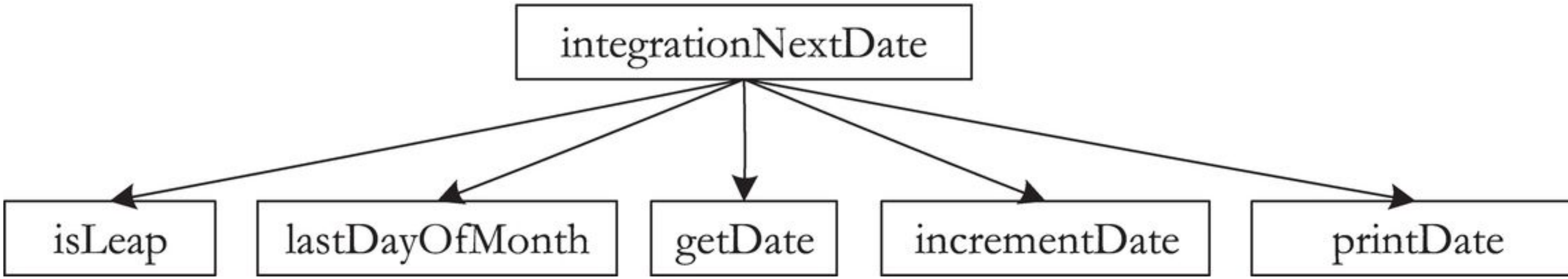



```
 ' GetDate body begins here
56   Do
57    Output("enter a month")
58    Input(aDate.Month)
59    Output("enter a day")
60    Input(aDate.Day)
61    Output("enter a year")
62    Input(aDate.Year)
63    GetDate.Month = aDate.Month
64    GetDate.Day = aDate.Day
65    GetDate.Year = aDate.Year
66   Until (ValidDate(aDate))              'msg7
67   End (Function GetDate)
```



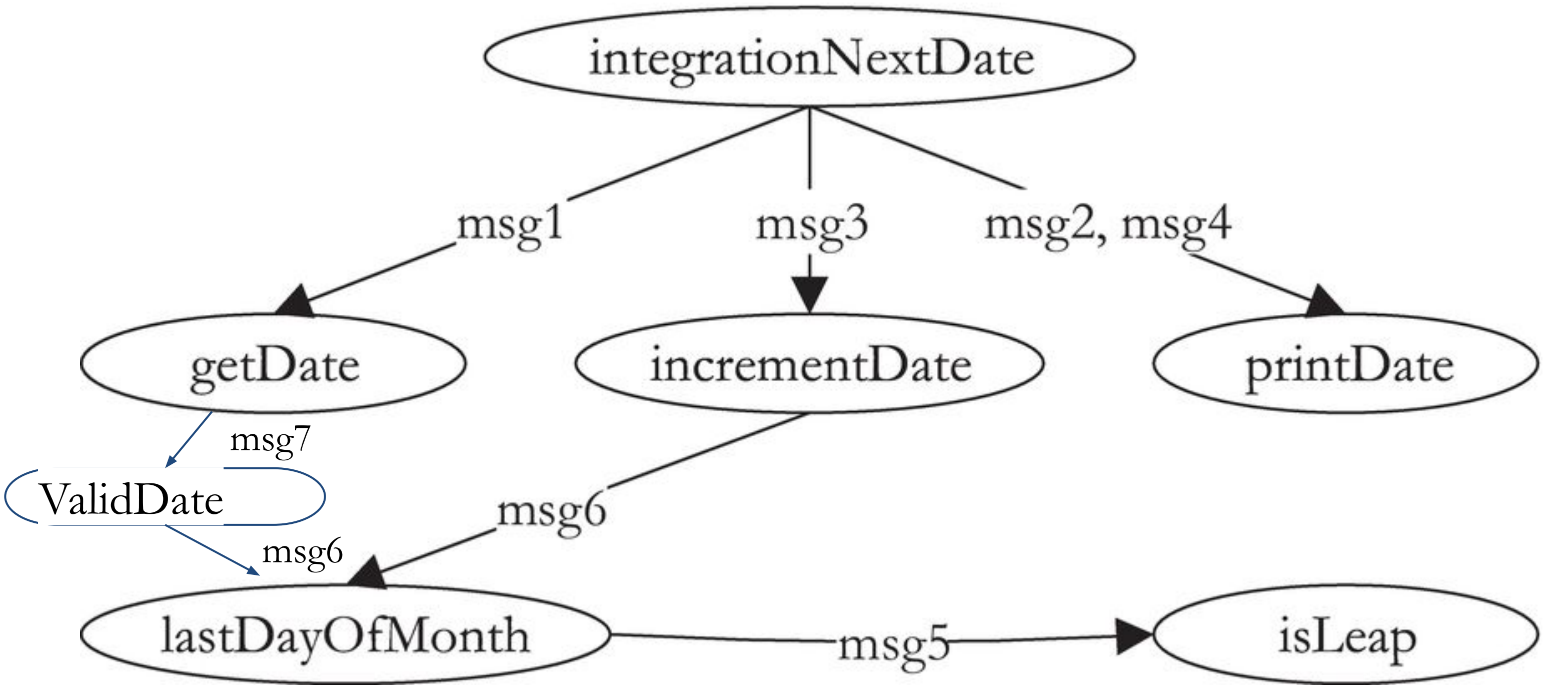
GetDate V(G) = 2

Decomposition Based Integration of NextDate



- Integration based on the decomposition in given above is problematic,
- The *isLeap* and *lastDayOfMonth* functions are never directly called by the Main program, so these integration sessions would be empty.
- The pairs involving *integrationNextDate* and *GetDate*, *IncrementDate*, and *PrintDate* are all useful (but short) sessions.

Call Graph Based Integration



- An improvement over that for the decomposition-based pairwise integration. There are no empty integration sessions because edges refer to actual unit references.
- Sandwich integration is appropriate because this example is so small.
- Neighborhood integration based on the call graph would likely proceed with the neighborhood of *lastDayOfMonth*, neighborhood of *getDate*, followed by the neighborhood of *incrementDate*.

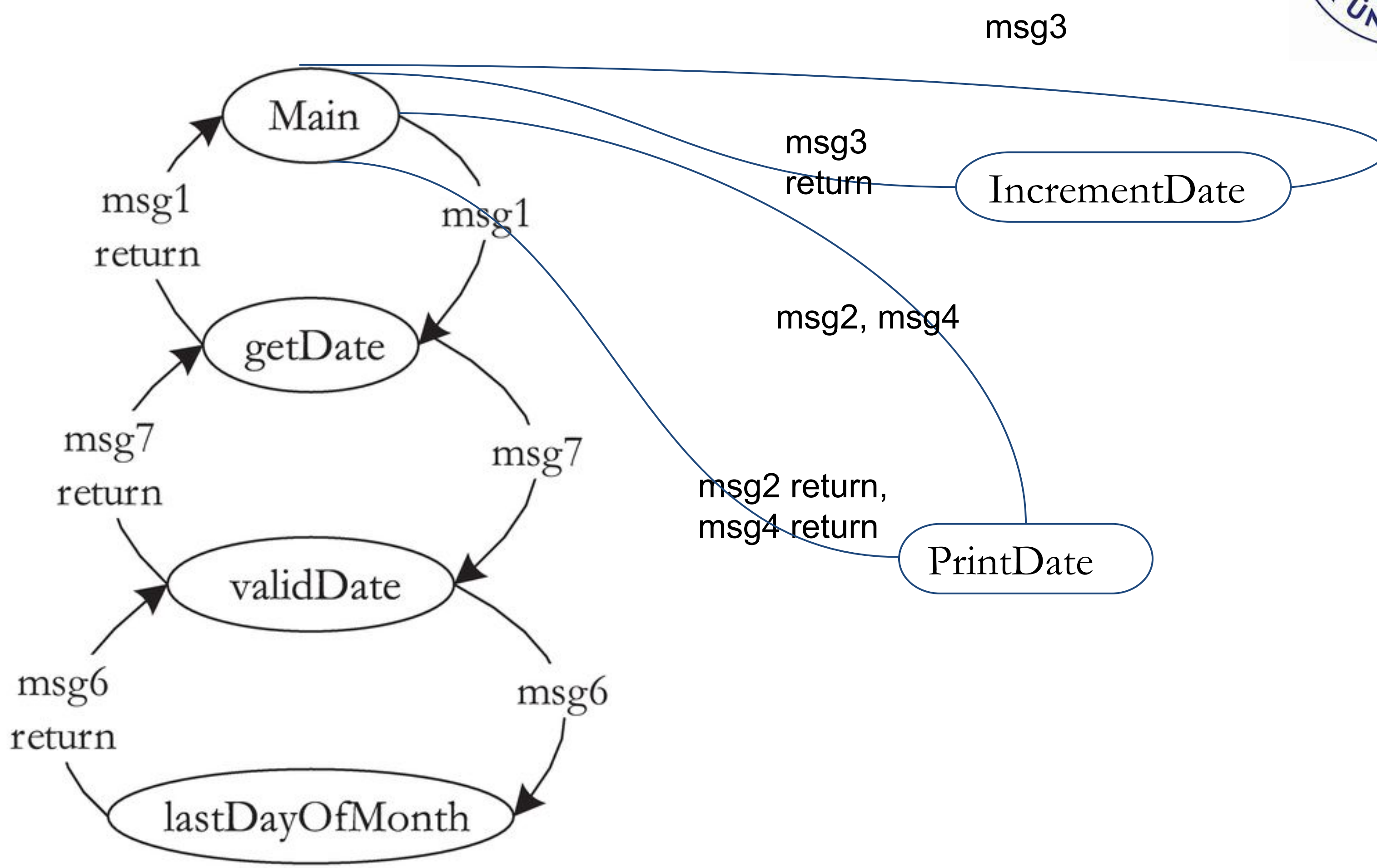
Integration Based on MM-Paths

- For traditional (procedural) software, MM-Paths will always begin (and end) in the main program.
- The depth of an MM-Path is determined by message quiescence which occurs occurs when a unit that sends no messages is reached;
  - In a sense, this could be taken as a "midpoint" of an MM-Path—the remaining execution consists of message returns

The four MM-Paths for May 27, 2020 :

Main (1, 2, 3)
    msg1
    GetDate (34, 56, 57-65, 66)
    msg 7
    validDate(35, 36, 37, 39, 40, 41, 42)
    msg 6
    LastDayOfMonth(22, 23, 24, 25, 33)
    msg 6 return
    validDate(42, 43, 45, 46, 47, 48, 50, 51, 52, 54, 55)
    msg7 return
    GetDate (66, 67)
    msg1 return
Main(3, 4)
    msg2
    PrintDate(58, 59, 60)
    msg2 return
Main(4, 5)
    msg3
    IncrementDate(68, 69)
    msg 8
    LastDayOfMonth(22, 23, 24, 25, 33)
    msg 8 return
    IncrementDate(70, 72, 75, 76, 77, 78)
    msg 3 return
Main(5, 6)
    msg4
    PrintDate(58, 59, 60)
    msg4 return
Main (6, 7)

MM-Path Graph of NextDate (for May 27, 2020)



Coverage Metrics for MM-Paths

- Given a set of MM-Paths:
  - MMP_0: Every message sent
    - Design test cases so that every message is sent at least once in the MM-Path Graph
  - MMP_1: Correct response received for every message sent.
    - Design test cases so that every message is sent at least once in the MM-Path Graph, and a correct response is received for every sent message.
  - MMP_2: Every unit execution path is traversed
    - Design test cases so that every execution path in the MM-Path Graph is traversed.

Comparison of Integration Testing Strategies

- Table below summarizes the comparisons between various integration testing approaches.
- The significant improvement of MM-Paths as a basis for integration testing is due to their exact representation of dynamic software behavior.

Strategy Basis	Ability to test Interfaces	Ability to test interactions (co-functionality)	fault isolation resolution
Functional Decomposition based	acceptable but can be deceptive	limited to pairs of units	good, down to faulty unit
Call Graph based	acceptable	limited to pairs of units	good, down to faulty unit
Path based	acceptable	complete	excellent, down to faulty unit execution path

Combinatorial Testing

- Objectives:
  - Rationale behind applying combinatorial techniques in testing.
  - Learn how to apply some representative combinatorial approaches

Combinatorial Explosion

- The behavior of a software application may be affected by many factors, e.g., input parameters, environment configurations, and state variables.
- Techniques like *equivalence partitioning*, and *boundary-value-analysis* can be used to identify the possible values of individual factors.
- It is usually impractical to test all possible combinations of values of all those factors.
- Assume that an application has 10 parameters each of which can take 5 values.
  - How many possible combinations are possible?
    - **Answer** : 5^{10}

Combinatorial Design

- Instead of testing all possible combinations, a subset of combinations is generated to satisfy some well-defined combination strategies.
- A key observation is that not every factor contributes to every fault, and it is often the case that a fault is caused by interactions among a few factors.
- Combinatorial design can dramatically reduce the number of combinations to be covered, but remains very effective in terms of fault detection.

Fault Model

- A t-way interaction fault is a fault that is triggered by a certain combination of t input values.
- A simple fault is a t-way fault where t=1; a pairwise fault is a t-way fault where t=2.
- In practice, a majority of software faults consist of simple and pairwise faults.

Example - Pairwise Faults

```
begin
    int x, y, z;
    input (x, y, z);
    if (x == x1 and y == y2)
        output (f(x, y, z));
    else if (x == x2 and y == y1)
        output (g(x, y));
    else                        // should have one more else if clause for x = x1 and y = y1
        output (f(x, y, z) + g(x, y))
end
```

Test Case Inputs		*Expected Output*
x = x1 and y = y1	=>	f(x, y, z) – g(x, y);
x = x2 and y = y2	=>	f(x, y, z) + g(x, y)

Example 3-way Fault

```
// assume x, y ∈ {-1, 0, 1}, and z ∈ {0, 1}
begin
    int x, y, z, p;
    input (x, y, z);
    p = (x + y) * z          // should be p = (x – y) * z
    if (p >= 0)
        output (f(x, y, z));
    else
        output (g(x, y));
end
```

Any test case which contains *y=0* or *z=0* would not be able to reveal the bug that is present in the code.

All Combinations Coverage

- Every possible combination of values of the parameters must be covered.
- For example, if we have three parameters:
  - P1 = {A, B}
  - P2 = {1, 2, 3}
  - P3 = {x, y}
- Then all combinations coverage requires 12 test cases:
  - {(A, 1, x), (A, 1, y), (A, 2, x), (A, 2, y), (A, 3, x), (A, 3, y), (B, 1, x), (B, 1, y), (B, 2, x), (B, 2, y), (B, 3, x), (B, 3, y)}

Each Choice Coverage

- Each parameter value must be covered in at least one test case.
- For the previous example, a test suite that satisfies each choice coverage is the following:
  - {(A, 1, x), (B, 2, y), (A, 3, x)}

Pairwise Coverage

- Given ***any*** two parameter, every combination of values of these two parameters are covered in at least one test case.
- A pairwise test suite of the previous example is the following:

P1	P2	P3
A	1	x
A	2	x
A	3	x
A	-	y
B	1	y
B	2	y
B	3	y
B	-	x

T-Wise Coverage

- Given any t parameters, every combination of values of these t parameters must be covered in at least one test case.
- For example, a 3-wise coverage requires every triple be covered in at least one test case.
- Note that all combinations, each choice, and pairwise coverage can be considered to be a special case of t-wise coverage.

Base Choice Coverage

- For each parameter, one of the possible values is designated as a base choice of that parameter.
- A base test is formed by using the base choice for each parameter.
- Subsequent tests are chosen by holding all base choices constant, except for one, which is replaced using a non-base choice of the corresponding parameter.
- A base choice test suite of the previous example is the following (the first line represents the base test case):

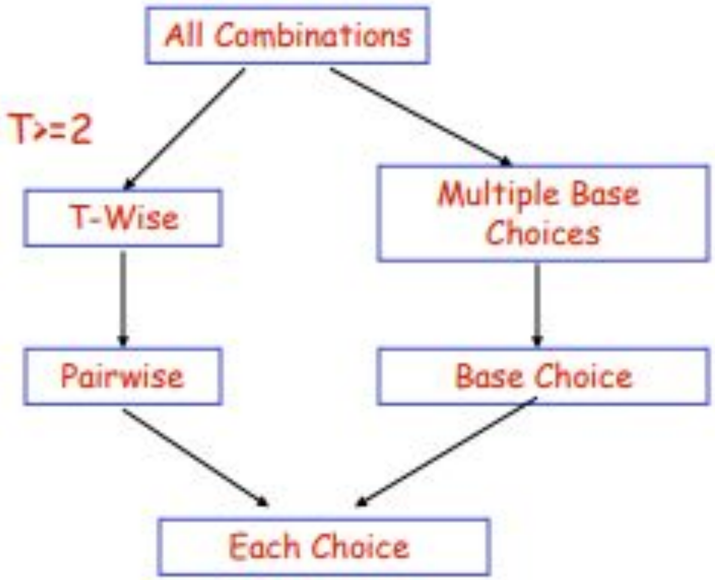
P1	P2	P3
A	1	x
B	1	x
A	2	x
A	3	x
A	1	y

Multiple Base Choice Coverage

- At least one, and possible more base choices are designated for each parameter.
- The notions of a base test and subsequent tests are defined the same as base choice coverage.

Subsumption Relation

Following picture shows the subsumption relation among several coverage criteria based on the input test space they cover.



Pairwise (2-way) Coverage

- Many faults are caused by the interactions between two parameters.
  - 92% statement coverage, 85% branch coverage
- Not practical to cover all parameter interactions
  - Consider a system with n parameters, each with m values.
  - There must be $\mathbf{m}^n$ interactions to be covered.
- A trade-off must be made between test effort and fault detection.
  - For a system with 20 parameters each with 15 values;
    - Pairwise testing only requires less than 412 tests,
    - Whereas exhaustive (all combinations coverage) testing requires 15^{20} tests.

Example

- Consider a system with the following parameters and values:
  - parameter A has values A1 and A2
  - parameter B has values B1 and B2, and
  - parameter C has values C1, C2, and C3

sample pair-wise tests

A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
A2	B1	C2
A1	B2	C3

minimal

A	B	C
A1	B1	C1
A1	B2	C1
A2	B1	C2
A2	B2	C3
A2	B1	C1
A1	B2	C2
A1	B1	C3

A	B	C
A1	B1	C1
A1	B2	C1
A2	B1	C2
A2	B2	C2
A2	B1	C1
A1	B1	C2
A1	B1	C3
A2	B2	C3

The IPO Strategy

- First generate a pairwise test set for the first two parameters, then for the first three parameters, and so on.
- A pairwise test set for the first *n* parameters is built by extending the pairwise test set for the first *n-1* parameters in two steps.
  - 1st step; Horizontal growth : Extend each existing test case by adding one value of the new parameter.
  - 2nd step; Vertical growth: Adds new tests, if necessary.

Algorithm IPO_H (T, p_i)

Assume that the domain of p_i contains values v_1, v_2, …, and v_q;

π = { pairs between values of p_i and values of p_1, p_2, …, and p_{i-1}}

if ($|T| <= q$)

for $1 <= j <= |T|$, **extend** the j^{th} test in T by adding value v_j and

remove from π pairs covered by the extended test

else

for $1 <= j <= q$, **extend** the j^{th} test in T by adding value v_j and

remove from π pairs covered by the extended test;

for $q < j <= |T|$, **extend** the j^{th} test in T by adding one value of p_i

such that the resulting test covers the most number of pairs in π, and **remove** from π pairs covered by the extended test

Algorithm IPO_V(T, π)

let T' be an empty set;

for each pair in π

assume that the pair contains value w of p_k, $1 \le k < i$, and value u of p_i ;

if (T' contains a test with "-" as the value of p_k and u as the value of p_i)

modify this test by replacing the "-" with w

else

add a new test to T' that has w as the value of p_k, u as the value of p_i , and "-" as the value of every other parameter;

T = T U T'

Exercise

Show how to apply the IPO strategy to construct the pairwise test set for the example system given above.

Summary

- Combinatorial testing makes an excellent tradeoff between test effort and test effectiveness.
- Pairwise testing can often reduce the number of dramatically, but it can still detect faults effectively.
- The IPO strategy constructs a pairwise test set incrementally, one parameter at a time.
- In practice, some combinations may be invalid from the domain semantics, and must be excluded, e.g., by means of constraint processing.