TESTING
- TEST-SUITE ESTIMATION

CS3213 FSE

Prof. Abhik Roychoudhury
National University of Singapore

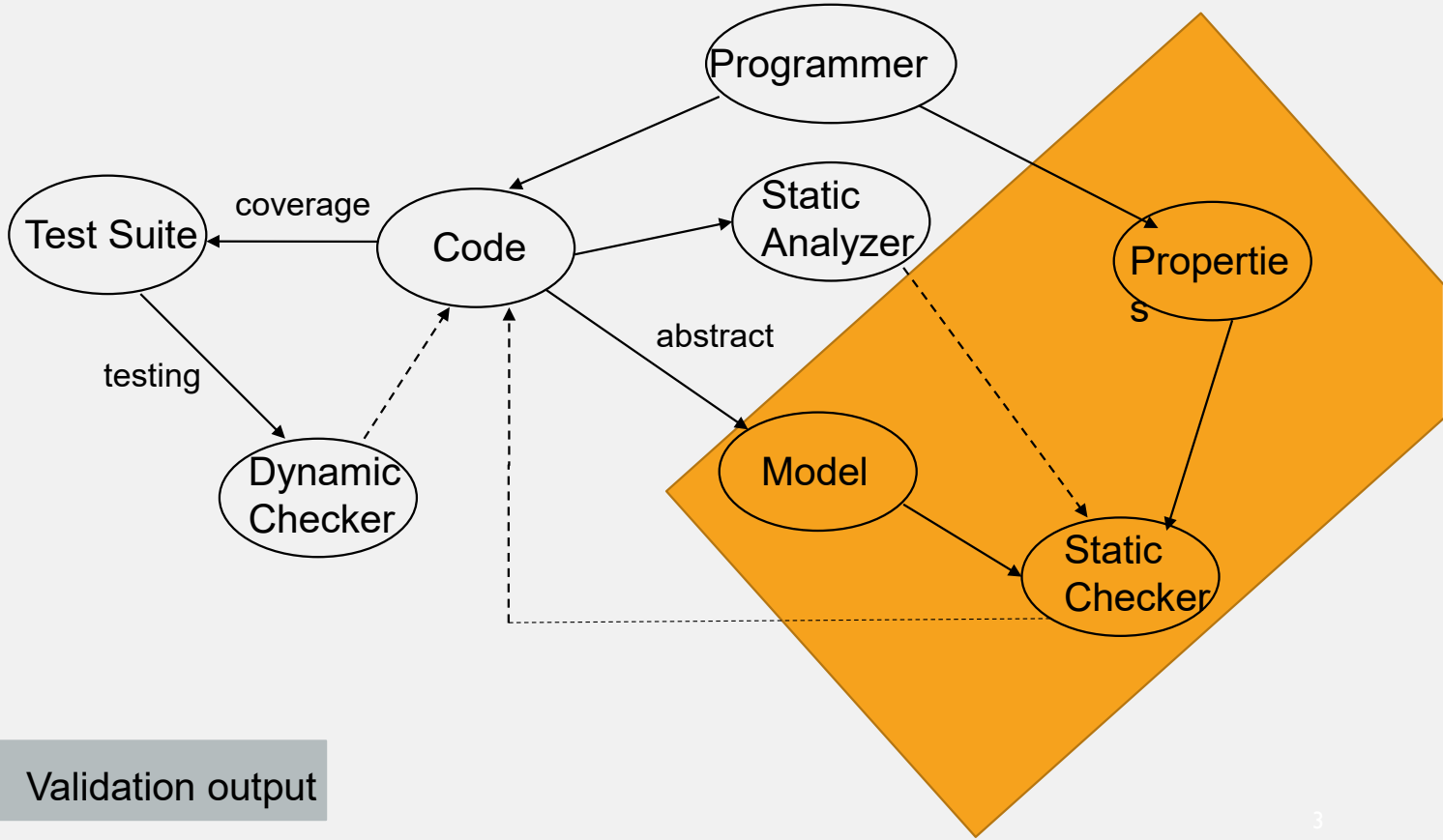
Ack: Materials from Prof. Mauro Pezze

WHAT WE DID EARLIER

- System Requirements: Use-cases, Scenarios, Sequence Diagrams
- System structure: Class diagrams
- Discussion on semantics
- System behavior: State diagrams

- Today
  - **Testing**

NO MODEL MAY BE AVAILABLE.



PROGRAMMING

Creativity

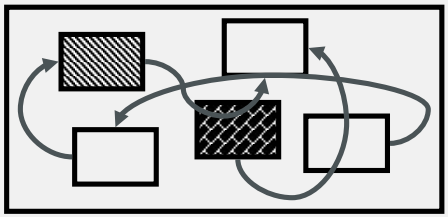
+

Precision

APPROACHES TO TESTING

- Black Box/Functional/Requirements based – treat requirements as rule

- White Box/Structural/Implementation based - *today*



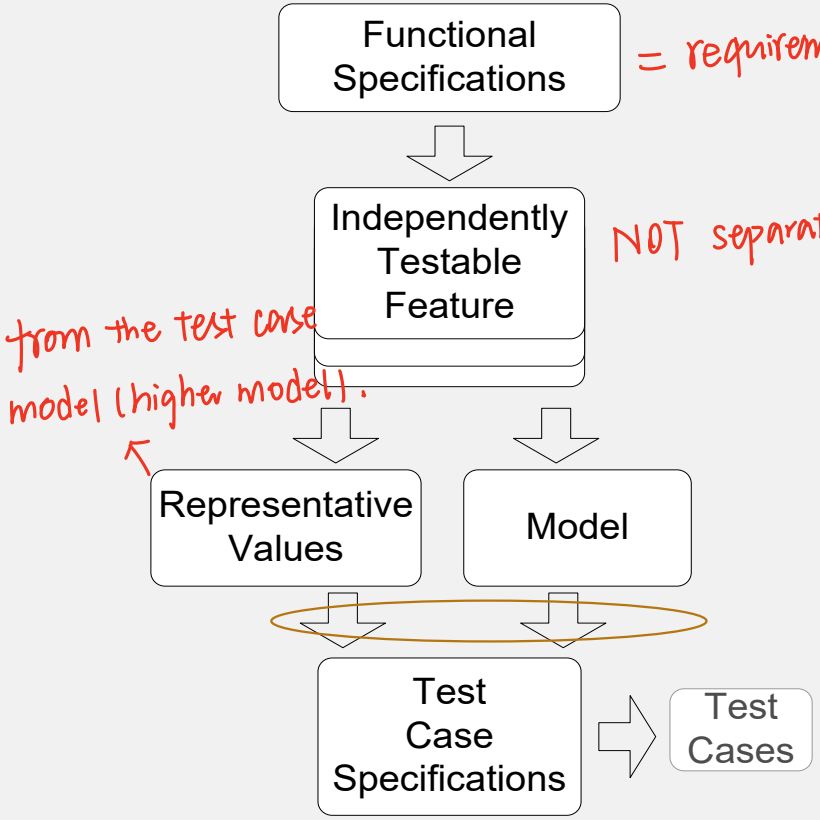
FUNCTIONAL TESTING

Black-box testing

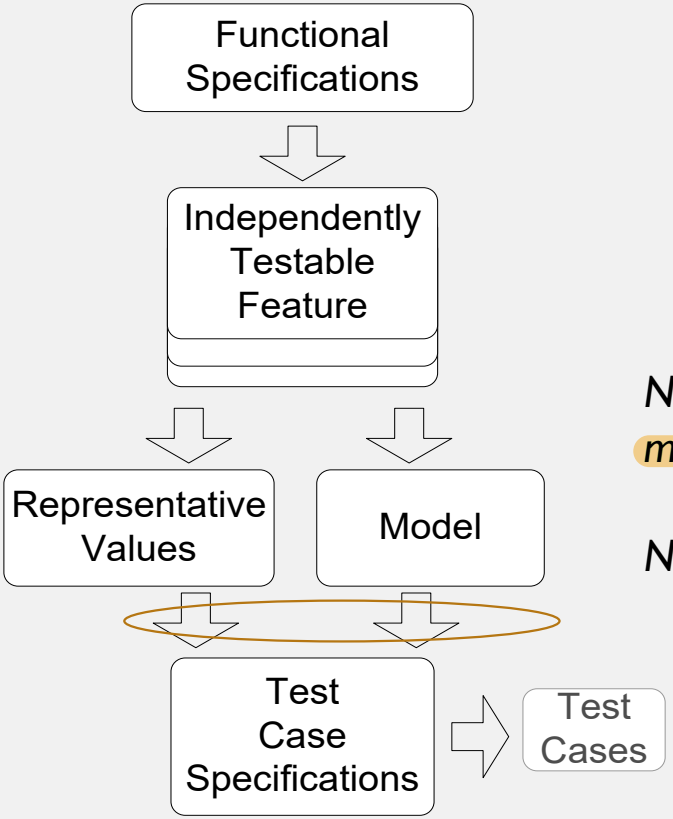
Reading: Software Testing and Analysis
Pezze and Young, Chapter 11

Requirements driven testing

FUNCTIONAL TESTING



WHY COMBINATORIAL TESTING



All feasible combinations.

Need to consider combinations of values / models from different testable features.

Need to deal with combinatorial explosion.

BASIC IDEAS

- Generate combinations automatically
- - and test them automatically too.

- Eliminate infeasible combinations – do not even generate them!
- - helps contain combinatorial explosion.
- - still exhaustively covering all feasible combinations may be difficult.

- Pair-wise testing
- - move away from exhaustively covering all feasible combinations.
- - cover combinations *systematically*, but not exhaustively.

KEY APPROACHES

- Category-partition testing
  - separate (manual) identification of values that characterize the input space from (automatic) generation of combinations for test cases
- Pair-wise testing
  - systematically test interactions among attributes of the program input space with a relatively small number of test cases

AN INFORMAL SPECIFICATION

- Check Configuration
- **Check the validity of a computer configuration from a sales web-site**
- The parameters of check-configuration are:
  - Model
  - Set of components

different combinations of components

INFORMAL SPEC PARAMETER *MODEL*

- Model
- A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs
- *Example:*
- *The required "slots" of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer*

INFORMAL SPEC
PARAMETER *SET OF COMPONENTS*

- A set of *(slot, component)* pairs, corresponding to the required and optional slots of the model. A *component* is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value *empty* is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.
- *Example:*
- *The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, empty is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of hard disk.*

CATEGORY PARTITION (MANUAL STEPS)

1. Decompose the specification into independently testable features
   - for each feature identify
     - parameters
     - environment elements
   - for each parameter and environment element identify elementary characteristics (categories)
2. Identify relevant values
   - for each characteristic (category) identify (classes of) values
     - normal values
     - boundary values
     - special values
     - error values
3. Introduce constraints reduce num of test cases.
4. Enumerate tests

STEP 1: IDENTIFY INDEPENDENTLY TESTABLE UNITS

Parameter *Model*

- Model number
- Number of required slots for selected model (#SMRS)
- Number of optional slots for selected model (#SMOS)

Categories

Parameter *Components*

for each slot:

- Correspondence of selection with model slots
- Number of required components with selection ≠ empty
- Required component selection
- Number of optional components with selection ≠ empty
- Optional component selection

Categories

Environment element: *Product database*

- Number of models in database (#DBM)
- Number of components in database (#DBC)

Categories

STEP 1: IDENTIFY INDEPENDENTLY TESTABLE UNITS

- Choosing categories
  - no hard-and-fast rules for choosing categories
  - not a trivial task!
- Categories reflect test designer's judgment
  - regarding which classes of values may be treated differently by an implementation
- Choosing categories well requires experience and knowledge
  - of the application domain and product architecture. The test designer must look under the surface of the specification and identify hidden characteristics
- For our example: Direct sales web-site of Chipmunk Computers
  - Configuration == Set of selected options for a computer say by customer
  - *Check configuration* is a testable feature – checks if a chosen configuration is valid
    - e.g. Digital LCD monitor with analog video-card is an invalid configuration!

STEP 2: IDENTIFY RELEVANT VALUES

introducing constraints = less test cases

- Identify (list) representative classes of values for each of the categories
  - Ignore interactions among values for different categories (considered in the next step)
- Representative values may be identified by applying
  - Boundary value testing
    - select extreme values within a class
    - select values outside but as close as possible to the class
    - select interior (non-extreme) values of the class
  - Erroneous condition testing
    - select values outside the normal domain of the program

STEP 2: IDENTIFY RELEVANT VALUES: MODEL

Model number

- Malformed
- Not in database
- Valid

Number of required slots for selected model (#SMRS)

- 0
- 1
- Many

Number of optional slots for selected model (#SMOS)

- 0
- 1
- Many

STEP 2: IDENTIFY RELEVANT VALUES: COMPONENT

Correspondence of selection with model slots

- Omitted slots
- Extra slots
- Mismatched slots
- Complete correspondence

eg. Choose x as memory.
Choose y as processor.

Number of required components with non empty selection

- 0
- < number required slots
- = number required slots

Required component selection

- Some defaults
- All valid
- ≥ 1 incompatible with slots
- ≥ 1 incompatible with another selection
- ≥ 1 incompatible with model
- ≥ 1 not in database

Number of optional components with non empty selection

- 0
- < #SMOS
- = #SMOS

Optional component selection

- Some defaults
- All valid
- ≥ 1 incompatible with slots
- ≥ 1 incompatible with another selection
- ≥ 1 incompatible with model
- ≥ 1 not in database

STEP 2: IDENTIFY RELEVANT VALUES: DATABASE

Number of models in database (#DBM)

- 0
- 1
- Many

Number of components in database (#DBC)

- 0
- 1
- Many

Note 0 and 1 are unusual (special) values. They might cause unanticipated behavior alone or in combination with particular values of other parameters.

STEP 3: INTRODUCE CONSTRAINTS

- A combination of values for each category corresponds to a test case specification
  - in the example we have $3^7 * 6^2 * 4 = 314{,}928$ test cases
  - most of which are impossible!
    - example
      zero slots and *at least one incompatible slot*
- Introduce constraints to
  - rule out impossible combinations
  - reduce the size of the test suite if too large

STEP 3: ERROR CONSTRAINT

[error] indicates a value class that

- corresponds to a erroneous values
- need be tried only once

Example

Model number: Malformed and Not in database

error value classes

- No need to test all possible combinations of errors
- One test is enough (we assume that handling an error case bypasses other program logic)

STEP 3: *ERROR* CONSTRAINT

Model number

- Malformed [error]
- Not in database [error]
- Valid

Correspondence of selection with model slots

- Omitted slots [error]
- Extra slots [error]
- Mismatched slots [error]
- Complete correspondence

Number of required comp. with non empty selection

- 0 [error]
- < number of required slots [error]

Required comp. selection

- ≥ 1 not in database [error]

Number of models in database (#DBM)

- 0 [error]

Number of components in database (#DBC)

- 0 [error]

Error constraints reduce test suite from 314,928 to 2,711 test cases

STEP 3: *PROPERTY* CONSTRAINTS

constraint [property] [if-property] rule out invalid combinations of values

[property] groups values of a single parameter to identify subsets of values with common properties

[if-property] bounds the choices of values for a category that can be combined with a particular value selected for a different category

Example

combine

Number of required comp. with non empty selection = number of required slots [if RSMANY]

only with

Number of required slots for selected model (#SMRS) = Many

EXAMPLE - STEP 3: PROPERTY CONSTRAINTS

Number of required slots for selected model (#SMRS)

1	[property RSNE]
Many	[property RSNE] **[property RSMANY]**

Number of optional slots for selected model (#SMOS)

1	[property OSNE]
Many	[property OSNE] [property OSMANY]

value class "many" is only possible if the property is "OSMANY".

Number of required comp. with non empty selection

0	[if RSNE] [error]
< number required slots	[if RSNE] [error]
= number required slots	**[if RSMANY]**

Number of optional comp. with non empty selection

< number required slots	[if OSNE]
= number required slots	[if OSMANY]

→ "<" only if property is "OSNE".

from 2,711 to 908 test cases

STEP 3 (CONT): *SINGLE* CONSTRAINTS

[single] indicates a value class that test designers choose to test only once to reduce the number of test cases

Example

value some default for required component selection and optional component selection may be tested only once despite not being an erroneous condition

note -

single and error have the same effect but differ in rationale. Keeping them distinct is important for documentation and regression testing

STEP 3: *SINGLE* CONSTRAINTS

Number of required slots for selected model (#SMRS)

0	[single]
1	[property RSNE] [single]

Number of optional slots for selected model (#SMOS)

0	[single]
1	[single] [property OSNE]

Required component selection

Some default	[single]

Optional component selection

Some default	[single]

Number of models in database (#DBM)

1	[single]

Number of components in database (#DBC)

1	[single]

from 908 to 69 test cases

CHECK CONFIGURATION – SUMMARY

Parameter Model

- Model number
  - Malformed [error]
  - Not in database [error]
  - Valid
- Number of required slots for selected model (#SMRS)
  - 0 [single]
  - 1 [property RSNE] [single]
  - Many [property RSNE] [property RSMANY]
- Number of optional slots for selected model (#SMOS)
  - 0 [single]
  - 1 [property OSNE] [single]
  - Many [property OSNE] [property OSMANY]

Environment Product data base

- Number of models in database (#DBM)
  - 0 [error]
  - 1 [single]
  - Many
- Number of components in database (#DBC)
  - 0 [error]
  - 1 [single]
  - Many

Parameter Component

- Correspondence of selection with model slots
  - Omitted slots [error]
  - Extra slots [error]
  - Mismatched slots [error]
  - Complete correspondence
- # of required components (selection ≠ empty)
  - 0 [if RSNE] [error]
  - < number required slots [if RSNE] [error]
  - = number required slots [if RSMANY]
- Required component selection
  - Some defaults [single]
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database [error]
- # of optional components (selection ≠ empty)
  - 0
  - < #SMOS [if OSNE]
  - = #SMOS [if OSMANY]
- Optional component selection
  - Some defaults [single]
  - All valid
  - ≥ 1 incompatible with slots, ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model, ≥ 1 not in database [error]

NEXT …

- Category partition testing gave us
  - Systematic approach: Identify characteristics and values (the creative step), generate combinations (the mechanical step)
- But ...
  - Test suite size grows very rapidly with number of categories. Can we use a non-exhaustive approach?
- Pairwise (and n-way) combinatorial testing do
  - Combine values systematically but not exhaustively
  - Rationale: Most unplanned interactions are among just two or a few parameters or parameter characteristics

PAIRWISE COMBINATORIAL TESTING

- Category partition works well when intuitive constraints reduce the number of combinations to a small amount of test cases
  - Without many constraints, the number of combinations may be unmanageable ☒☒☒
- Pair-wise combination (instead of exhaustive)
  - Generate combinations that efficiently cover all pairs (triples,…) of classes.
  - Rationale: most failures are triggered by single values or combinations of a few values. Covering pairs (triples,…) reduces the number of test cases, but reveals most faults.

not all possible combinations.

EXAMPLE: DISPLAY CONTROL

No constraints reduce the total number of combinations

☒☒☒432 (3x4x3x4x3) test cases

if we consider all combinations

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

PAIRWISE COMBINATIONS: 17 TEST CASES

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

ILLUSTRATING THE APPROACH

Display	Screen Size
Full graphics	Hand-held
Full graphics	Laptop
Full graphics	Full-size
Text only	Hand-held
Text only	Laptop
Text only	Full-size
Limited bandwidth	Hand-held
Limited bandwidth	Laptop
Limited bandwidth	Full-size

2 categories at a time.
All possible pairs of these 2 categories.
But not correct test cases yet (the rest of categories are not here)

ILLUSTRATING THE APPROACH

Display	Screen Size	Fonts
Full graphics	Hand-held	Minimal
Full graphics	Laptop	Standard
Full graphics	Full-size	Document loaded
Text only	Hand-held	Standard
Text only	Laptop	Document loaded
Text only	Full-size	Minimal
Limited bandwidth	Hand-held	Document loaded
Limited bandwidth	Laptop	Minimal
Limited bandwidth	Full-size	Standard

9 tests instead of $3 \times 3 \times 3 = 27$ test cases.

Same number of tests (9), fonts are assigned carefully to the earlier table, s.t.

--- Covers all pairs of font × screen size

--- Covers all pairs of display × font

pair wise coverage with as few tests as possible.

eg. full × minimal only seen once.

ADDING CONSTRAINTS

- Simple constraints

  example: color monochrome not compatible with screen laptop and full size

  can be handled by considering the case in separate tables

EXAMPLE: MONOCHROME ONLY WITH HAND-HELD

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	
limited-bandwidth	Spanish	Document-loaded	16-bit	
	Portuguese		True-color	

Take care of the constraint

OMIT(*, *, *, Monochrome, laptop)

during test generation.

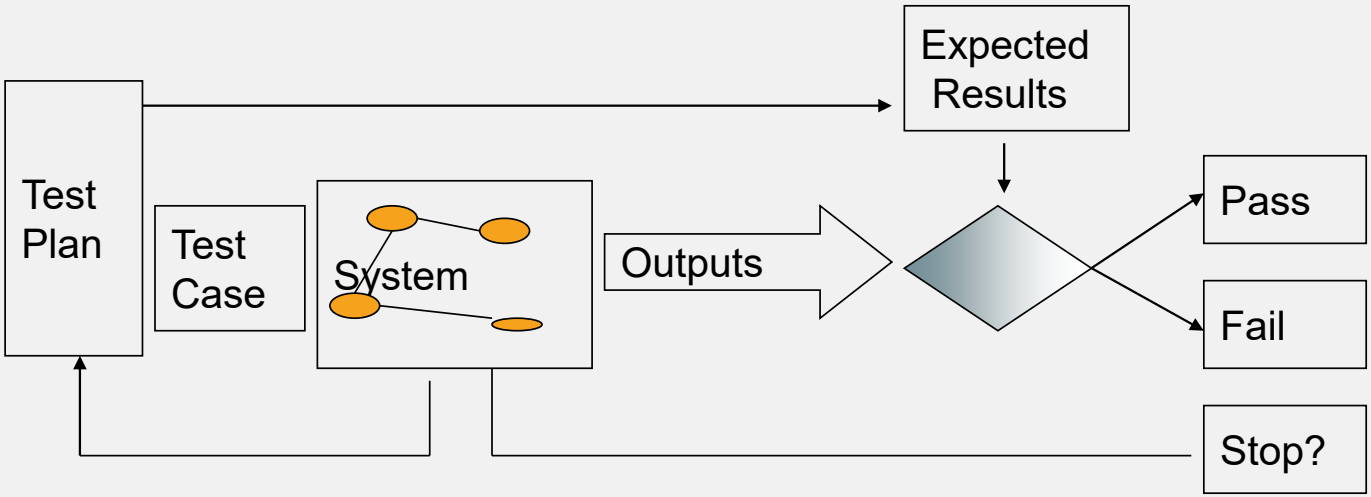
When doing pair-wise assignment, must follow constraints, not generate this pair.

WHITE-BOX TESTING

Testing that takes into account the internal mechanism of a system or component.

— IEEE

- aka Structural Testing, Glass Box Testing

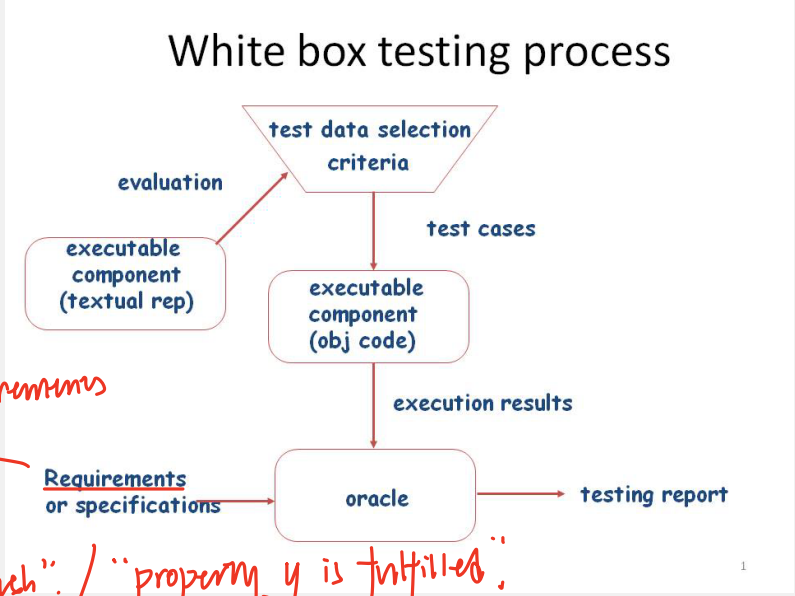


STRUCTURAL TESTING

Structural Coverage based on control-flow criteria

WHITE BOX/STRUCTURAL TEST DATA SELECTION

- **Coverage based**
  - *Control- flow and data-flow criteria.*
- Fault-based
  - e.g., mutation testing



white-box testing still needs requirements

i. for each input, what's expected output.

ii. expected output can be "not crash". / "property y is fulfilled".

LEARNING OBJECTIVES

- Understand rationale for structural testing
  - How structural (code-based or glass-box) testing complements functional (black-box) testing
- Recognize and distinguish basic terms
  - Adequacy, coverage
- Recognize and distinguish characteristics of common structural criteria
- Understand practical uses and limitations of structural testing

WHY STRUCTURAL (CODE-BASED) TESTING?

- One way of answering the question "What is *missing* in our test suite?"
  - If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed
  - But what's a "part"?
    - Typically, a control flow element or combination:
    - Statements (or CFG nodes), Branches (or CFG edges)
    - Fragments and combinations: Conditions, paths
- Complements functional testing: Another way to recognize cases that are treated differently
  - Recall fundamental rationale: Prefer test cases that are treated *differently* over cases treated the same

NO GUARANTEES

- Executing all control flow elements does not guarantee finding all faults
  - Execution of a faulty statement may not always result in a failure
    - The state may not be corrupted when the statement is executed with some data values
    - Corrupt state may not propagate through execution to eventually lead to failure
- What is the value of structural coverage?
  - Increases confidence in thoroughness of testing
    - Removes some obvious *inadequacies*

EXAMPLE- ERRORS GETTING MASKED

```
1  int x; /* Input variable */
2  int y;
3  int o; /* Output variable */
4
5  input(x);
6
7  if (x > 0) {
8      y = 3; //change: y = 2;
9      if (x - y > 0)
10         o = y;
11     else
12         o = 0;
13 } else
14     o = -1;
15
16 if (x > 20)
17     o = 10;
18
19 output(o);
```

X > 2 : determines direction of execution.

Questions for the class

When will the effects of the change be seen?

When will the effects of the change be masked?

STRUCTURAL TESTING *COMPLEMENTS* FUNCTIONAL TESTING

- Control flow testing includes cases that may not be identified from specifications alone
  - Typical case: implementation of a single item of the specification by multiple parts of the program
  - Example: hash table collision (invisible in interface spec)
- Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria
  - Typical case: missing path faults

STRUCTURAL TEST DATA

black-box testing.

- **Create functional test suite first,** then measure structural coverage to identify see what is missing

- Question to be discussed later:
  - ***Can structural test generation be automated?***

- Questions discussed now:
  - ***Various coverage criteria***

STATEMENT TESTING

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\text{ executed statements}}{\#\text{ statements}}$$

- Rationale: a fault in a statement can only be revealed by executing the faulty statement

STATEMENTS OR BLOCKS?

- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to *basic block* coverage or *node coverage*
  - Difference in granularity, not in concept

不改变 flow 的最大单元.

米：注意可能有 edge return back to this node.

node 里不能有 initialization 的.

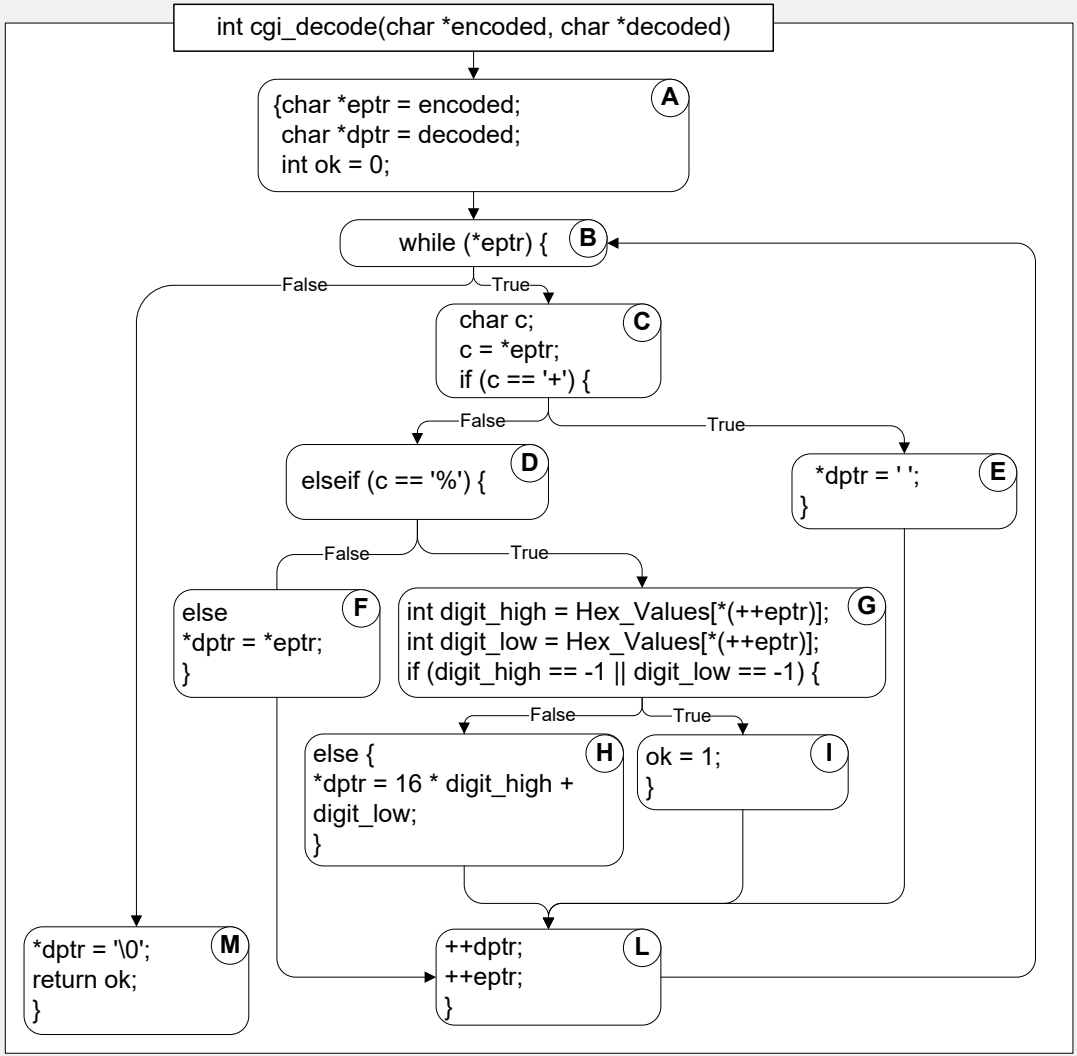
不然每次 return back 都重新 initialize 一遍.

EXAMPLE

T_0 =
{“test”,
“test+case%1Dadequacy”}
17/18 = 94% Stmt Cov.

T_1 =
{“adequate+test%0Dexecuti
on%7U”}
18/18 = 100% Stmt Cov.

T_2 =
{“%3D”, “%A”, “a+b”,
“test”}
18/18 = 100% Stmt Cov.



COVERAGE IS NOT SIZE

- Coverage does not depend on the number of test cases
  - $T_0, T_1 : T_1 >_{coverage} T_0$ $\quad T_1 <_{cardinality} T_0$
  - $T_1, T_2 : T_2 =_{coverage} T_1$ $\quad T_2 >_{cardinality} T_1$
- Minimizing test suite size is seldom the goal
  - small test cases make failure diagnosis easier
  - a failing test case in T_2 gives more information for fault localization than a failing test case in T_1

T_2 cardinality $> T_1$.

IS IT ENOUGH?

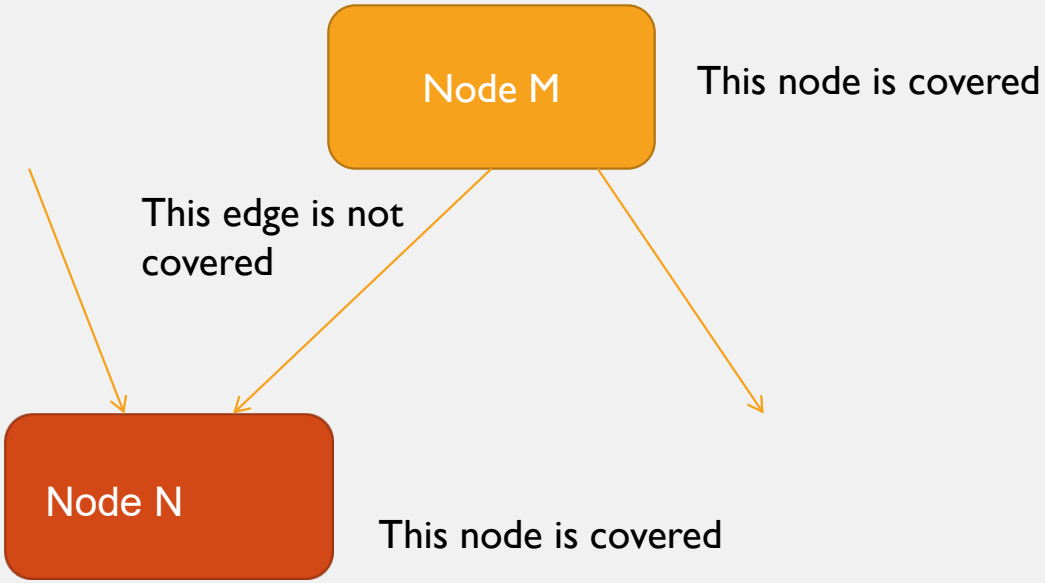
- Why statement coverage may not be adequate?
- Complete statement coverage may not imply executing all branches in a program.

- **Construct an example program now in class to show it.**

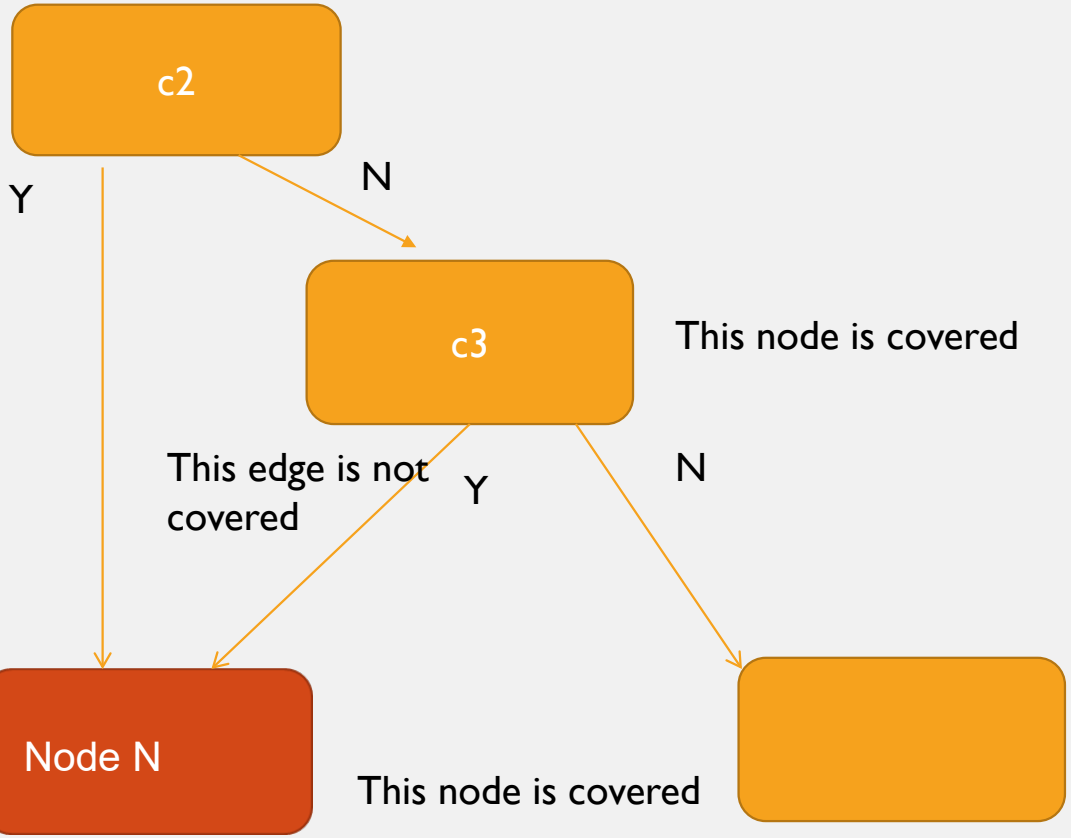
```
input in;
out = 0;
if (in > 0){
    out = 1;
}
return out;
```

Consider T= {<in=1, out=1>}

NODE COVERAGE != EDGE COVERAGE



Exercise: Try to construct a program construct whose control flow graph exhibits the above pattern. Do it now!!


c2
Y
N
c3
This node is covered
This edge is not covered
Y
N
Node N
This node is covered

BRANCH TESTING

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

T_3 = {"", "+%0D+%4J"}

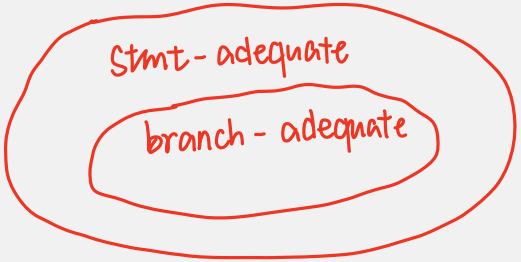
100% Stmt Cov. 88% Branch Cov. (7/8 branches)

T_2 = {"%3D", "%A", "a+b", "test"}

100% Stmt Cov. 100% Branch Cov. (8/8 branches)

STATEMENTS VS BRANCHES

- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)



"ALL BRANCHES" CAN STILL MISS CONDITIONS

branch is covered.
but there still fault in condition.

- Sample fault: missing operator

```
digit_high == 1 || digit_low == -1
```

- Branch adequacy criterion can be satisfied by varying only digit_low
  - The faulty sub-expression might never determine the result
  - We might never really test the faulty condition, even though we tested both outcomes of the branch

EXAMPLE

- Condition h == 1 || l == -1

- Suppose it is buggy
  - Should be h == -1 || l == -1
  - Achieve branch coverage
    - < h == 0, l == 0>
    - < h == 0, l == -1>

Cannot know "h==-1" is error.

  - Do not vary the faulty condition at all, and the variables involved!!

BASIC CONDITION TESTING

- Adequacy criterion:
  - each basic condition must be executed at least once to true, and …
  - at least once to false.
- Coverage:

$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$

BASIC CONDITIONS VS BRANCHES

- Basic condition adequacy criterion can be satisfied without satisfying branch coverage

Construct an example program to show this claim.

Branch and basic condition are not comparable

(neither implies the other)

test1: T||F → T

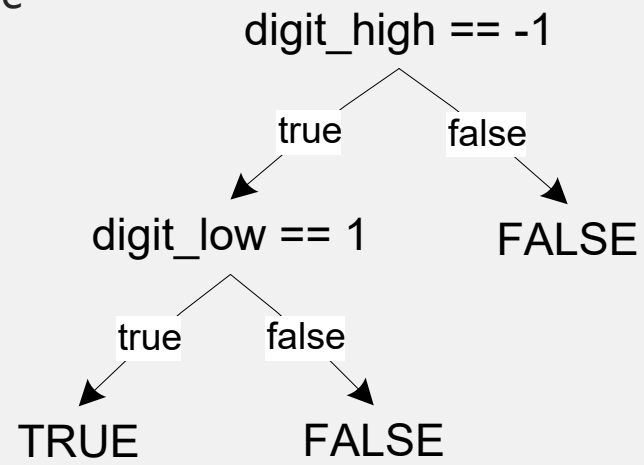
test2: F||T → T.

① basic condition adequacy: Yes.
- each basic condition is tested.

② branch coverage: No.
- never test else branch.

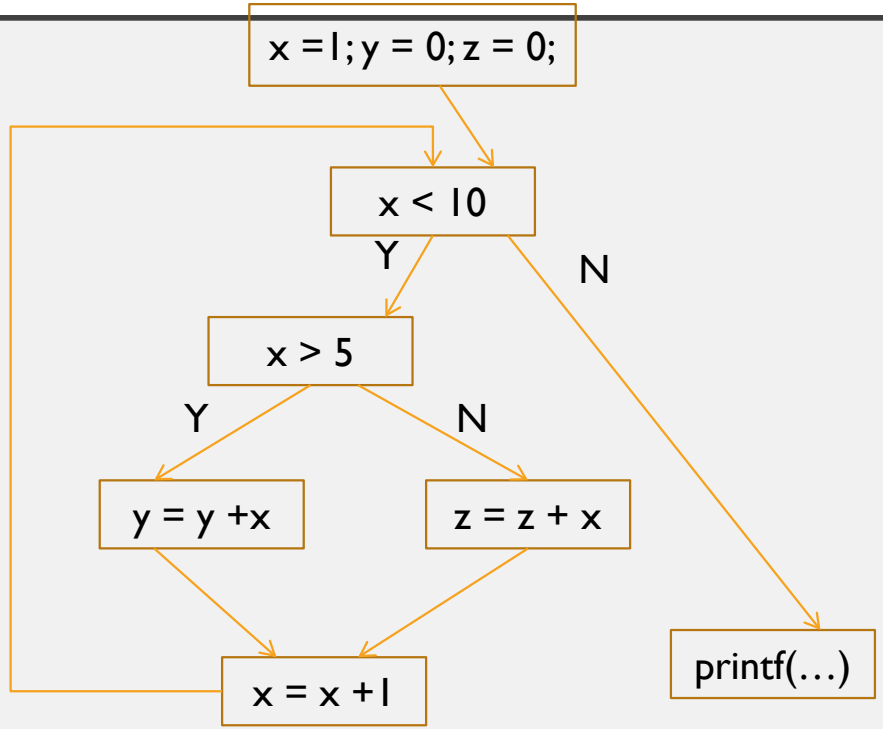
COVERING BRANCHES AND CONDITIONS

- Branch and condition adequacy:
  - cover all conditions and all decisions
- Compound condition adequacy:
  - Cover all possible evaluations of compound conditions
  - Cover all branches of a decision tree



BASICS - CONTROL FLOW GRAPH

```
x = 1; y = 0; z = 0;
while (x < 10){
    if (x > 5)
            y = y + x;
    else   z = z + x;
    x = x + 1;
}
printf(….);
```



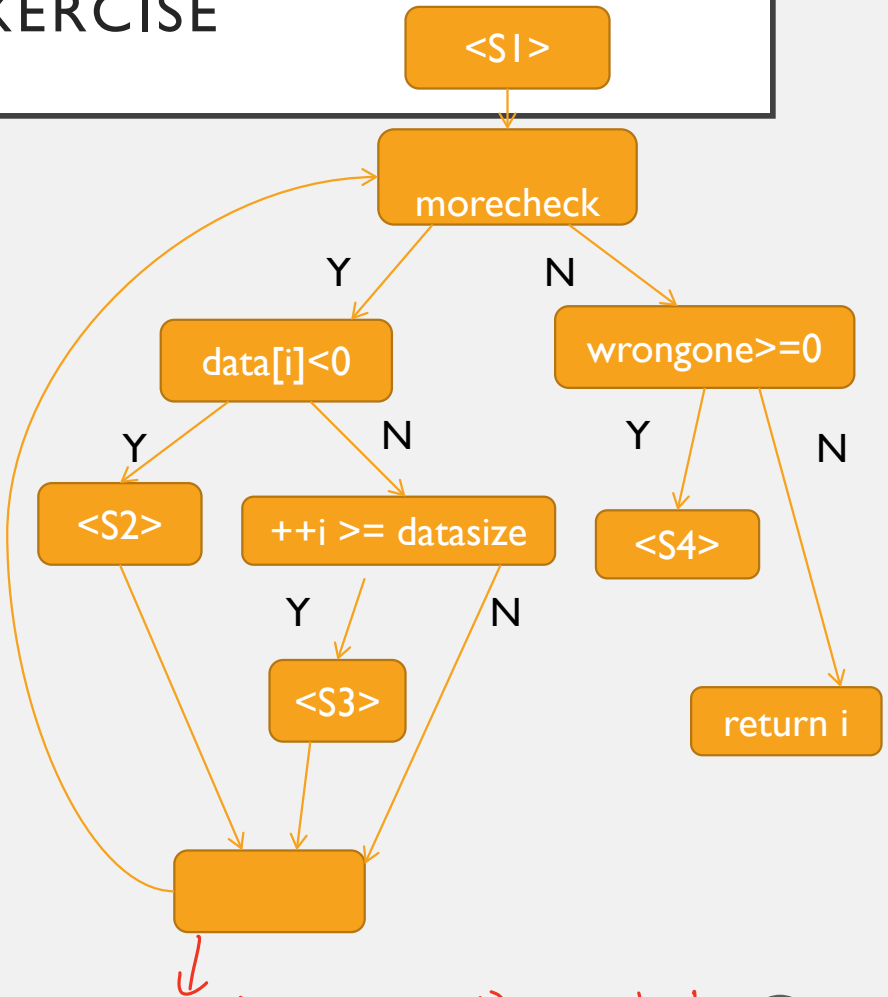
Nodes of the graph, basic blocks, are maximal code fragments executed without control transfer. The edges denote control transfer.

EXERCISE ON CFG

```
procedure Check_data()
{       int  i = 0, morecheck = 1, wrongone = -1, datasize = 10;
  L:    while (morecheck)
  LB:   {
              if  (data[i] < 0)
  A:              {  wrongone = i;   morecheck = 0; }
              else
  B:                  if  (++i >= datasize) morecheck = 0;
            }
            if (wrongone >= 0)
   C:               { handle_exception(wrongone); return 0; }
   C':    else  return i;
}
```

EXERCISE

```
procedure Check_data()
{        …. <S1>
   L:    while (morecheck)
   LB:   {
                if  (data[i] < 0)
   A:               { …. <S2> }
                else
   B:           if  (++i >= datasize)
                              … <S3>;
            }
            if (wrongone >= 0)
    C:              { … <S4> }
    C':     else  return i;
}
```



也可以直接 <S2>,<S3> 接 morecheck

EXERCISE

Consider the following program fragment. Draw its control flow graph.

```
sum = 0; i =0;
while (i <= 999){   if  (i % 2 == 0)  sum = sum + i; i=i+1;}
return sum;
```

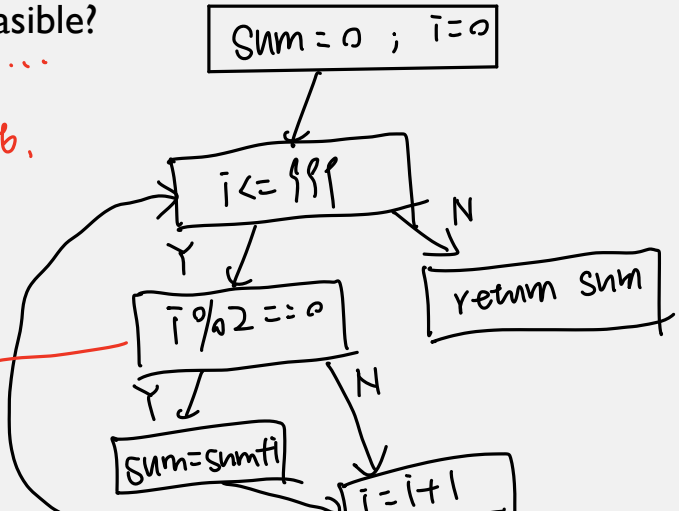
How many paths are there in the control flow graph? How many of these are feasible, and how many are infeasible?

Total path = 2×2×2×...
每个 iteration 有 2种可能.

(path是从头到尾).

Feasible path

只有1个 feasible path:



COMPOUND CONDITIONS: EXPONENTIAL COMPLEXITY

`(((a || b) && c) || d) && e`

Test	a	b	c	d	e
(1)	T	—	T	—	T
(2)	F	T	T	—	T
(3)	T	—	F	T	T
(4)	F	T	F	T	T
(5)	F	F	—	T	T
(6)	T	—	T	—	F
(7)	F	T	T	—	F
(8)	T	—	F	T	F
(9)	F	T	F	T	F
(10)	F	F	—	T	F
(11)	T	—	F	F	—
(12)	F	T	F	F	—
(13)	F	F	—	F	—

MODIFIED CONDITION/DECISION (MC/DC)

- Motivation: Effectively test important combinations of conditions, without exponential blowup in test suite size
  - "Important" combinations means: Each basic condition shown to independently affect the outcome of each decision
- Requires:
  - For each basic condition C, two test cases, (1个true, 1个false).
  - values of all *evaluated* conditions except C are the same
  - compound condition as a whole evaluates to *true* for one and *false* for the other

MC/DC: LINEAR COMPLEXITY

- N+1 test cases for N basic conditions

```
(((a || b) && c) || d) && e
```

6 test for 5 basic conditions.

Test	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

- Underlined values independently affect the output of the decision
- Required by the RTCA/DO-178B standard

To test only a: (1). (13). → b, c, d, e are compliant in (1). (13).

c: (1). (11).

COMMENTS ON MC/DC

- MC/DC is
  - basic condition coverage (C)
  - branch coverage (DC) Decision coverage.
  - plus one additional condition (M): Independent. every condition must *independently affect* the decision's output
- It is subsumed by compound conditions and subsumes all other criteria discussed so far
  - stronger than statement and branch coverage
- A good balance of thoroughness and test size (and therefore widely used)

MC/DC – INDUSTRY STANDARD

basic condition coverage

- "*Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible outcomes.*"

branch coverage.

PATH ADEQUACY

Not practical

- Decision and condition adequacy criteria consider individual program decisions
- Path testing focuses consider combinations of decisions along paths
- Adequacy criterion: each path must be executed at least once
- Coverage:

infeasible paths

$$\frac{\text{\# executed paths}}{\text{\# paths}}$$

PRACTICAL PATH COVERAGE CRITERIA

- The number of paths in a program with loops is unbounded
  - the simple criterion is usually impossible to satisfy
- For a feasible criterion: Partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
  - the number of traversals of loops
  - the length of the paths to be traversed
  - the dependencies among selected paths

SUMMARY

- We defined a number of adequacy criteria
  - Test-suite estimation, NOT test-suite construction
- Full coverage is usually unattainable
  - Remember that attainability is an undecidable problem!
- …and when attainable, "test generation" is usually hard
  - How do I find program inputs allowing to cover something buried deeply in the CFG?
  - Automated support (e.g., **symbolic execution**) may be necessary
- Rather than requiring full adequacy, the "degree of adequacy" of a test suite is estimated by coverage measures
  - May drive test improvement

DATA FLOW TESTING

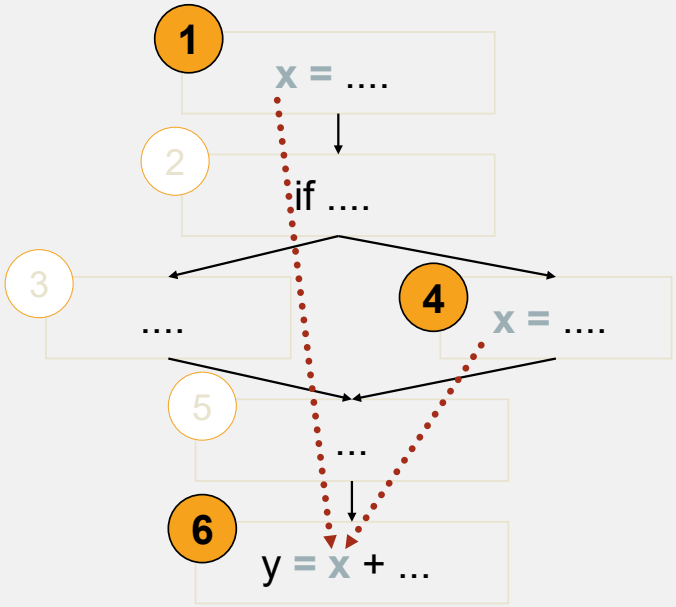
White-box testing

Coverage based on data-flow criteria

MOTIVATION

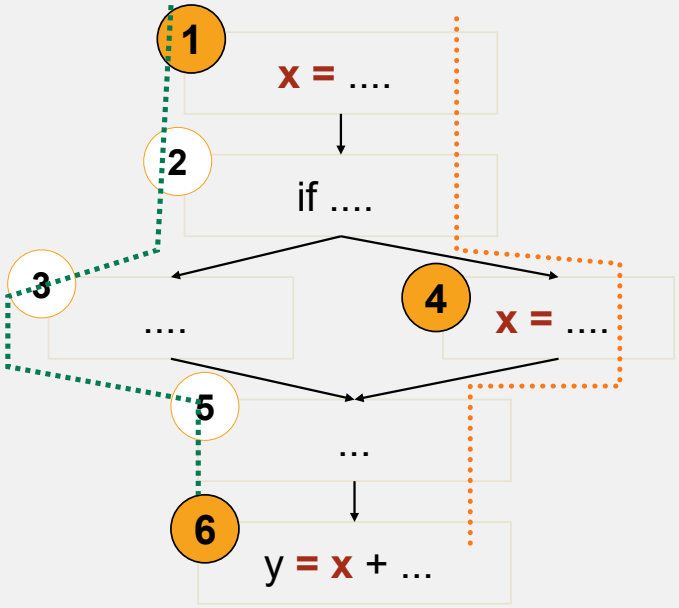
- Middle ground in structural testing
  - Node and edge coverage don't test interactions
  - Path-based criteria require impractical number of test cases
    - And only a few paths uncover additional faults, anyway
  - Need to distinguish "important" paths
- Intuition: Statements interact through *data flow*
  - Value computed in one statement, used in another
  - Bad value computation revealed only when it is used

RECAP: REACHING DEF.



- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are *def-use (DU) pairs*
  - defs at 1,4 (define, use).
  - use at 6

DEFINITION-CLEAR PATH



- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is "killed" (reassigned) at node 4
- (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

DU pair :
path from D to U is a definition-clear path.

There is a test that executes write at line 1 and line 6.

ADEQUACY CRITERIA

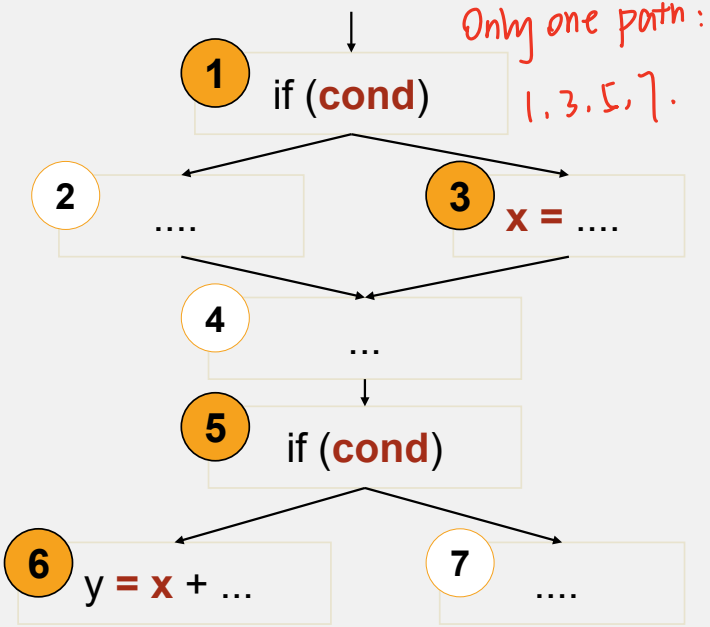
- **All DU pairs**: Each DU pair is exercised by at least one test case

Corresponding coverage fractions can also be defined

ALIASING

- x[i] = ... ; ... ; y = x[j]
  - DU pair (only) if i==j
- p = &x ; ... ; *p = 99 ; ... ; q = x
  - *p is an alias of x
- m.putFoo(...); ... ; y=n.getFoo(...);
  - Are m and n the same object?
  - Do m and n share a "foo" field?
- Problem of *aliases*: Which references are (always or sometimes) the same?

INFEASIBILITY



- Suppose *cond* has not changed between 1 and 5
  - Or the conditions could be different, but the first implies the second
- Then (3,6) is not a (feasible) DU pair
  - But it is difficult or impossible to determine which pairs are infeasible
- Infeasible test obligations are a problem
  - No test case can cover them

INFEASIBILITY

- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant
  - Combinations of elements matter!
  - Impossible to (infallibly) distinguish feasible from infeasible paths. More paths = more work to check manually.
- In practice, reasonable coverage is (often, not always) achievable

SUMMARY

- Data flow testing attempts to distinguish "important" paths: Interactions between statements
  - Intermediate between simple statement and branch coverage and more expensive path-based structural testing
- Cover Def-Use (DU) pairs: From computation of value to its use
  - Intuition: Bad computed value is revealed only when it is used
  - Levels: All DU pairs, all DU paths, all defs (some use)
- Limits: Aliases, infeasible paths
  - Worst case is bad (undecidable properties, exponential blowup of paths), so pragmatic compromises are required

MUTATION TESTING

Abhik Roychoudhury

National University of Singapore

TEST-SUITE ESTIMATION

- Change the program slightly
  - One line change to introduce an error.
  - Called a Mutant program.
- Check if your test suite can "detect" the error
  - At least one test fails.
- Decide if your test suite is "adequate"

INADEQUATE TEST-SUITES

- Suppose, no test can kill a given mutant.
- Why could this happen?
  - Test suite does not check all behaviors?
  - The mutant is semantically equivalent to the original program?
    - Program equivalence checking – undecidable.

EXAMPLE - MUTANTS

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index;
4.   address = base + offset;
5.   output address, sentinel
```

```
Input: a, index
1.   base = a - 1;
2.   sentinel = base;
3.   offset = index;
4.   address = base + offset;
5.   output address, sentinel
```

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index - 1;
4.   address = base + offset;
5.   output address, sentinel
```

WHY MUTATE?

- Develop program P
- Come up with test suite T based on use-cases and your own intuition
- Test P against T, fix all failing tests.
- P now passes against T
  - Take it for code review in your company.
  - A comment from a colleague
    - In line 75 in file xyz, shouldn't we have

```
sentinel = base+1
```

HOW TO COUNTER SUCH COMMENTS?

- Depend on your reputation
  - I have been coding for 25 years – I know what I did, program passed all tests !
- Connect it back to requirements –
  - may be hard to do, as all program variables do not correspond to quantities mentioned in requirements.
- **Submit the results from *Mutation Testing***

MUTATION TESTING

- Develop program P and test-suite T.
- Generate all mutants of P automatically
  - As per the given mutation operators of P, decided by the programming language.
- How many of the mutants are killed by T
  - Mutation score = (# of killed mutants) / (Total # of mutants)

MUTATION SCORE

Mutation score = (# of killed mutants) / (Total # of mutants)

Can modify it to

$$\text{Mutation score} = \frac{\#\ \text{of killed mutants}}{\text{Total \# of mutants - \# of equivalent mutants}}$$

# of equivalent mutants cannot be found exactly – undecidable.

Can replace it with # of equivalent mutants found (using some heuristics, which must be incomplete).

```
public class Add {
        public static int sum (int a, int b){
          return a+b;
        }
        public static double sum (double a, double
        b){
           return a+b;
        }
        public static long sum (long a, long b){
            return a+b;
        }
}
```

TC1:

Add o = new Add();
print(o.sum(1,2));
print(o.sum(1.0,2.0));

MutationScore(*TC1*) = *?*

```
public class Add {
        public static int sum(int a, int b) {
        return ++a + b;}
        public static double sum(double a, double b) {
        return a + b;}
        public static long sum(long a, long b) {
        return a + b;}
}
```

```
public class Add {
        public static int sum(int a, int b) {
        return  a + b;}
        public static double sum(double a, double b) {
        return a + b;}
        public static long sum(long a, long b) {
        return --a + b;}
}
```

```
public class Add {
        public static int sum(int a, int b) {
        return a + b;}
        public static double sum(double a, double b) {
        return a - b;}
        public static long sum(long a, long b) {
        return a + b;}
}
```

LARGE NUMBER OF MUTANTS!

```
int triangle(int a, int b, int c) {
  if (a <= 0 || b <= 0 || c <= 0) {
      return 4; // invalid
  }
  if (! (a + b > c && a + c > b && b + c > a)) {
      return 4; // invalid
  }
  if (a == b && b == c) {
      return 1; // equilateral
  }
  if (a == b || b == c || a == c) {
      return 2; // isosceles
  }
  return 3;  // scalene
}
```

a + b > c 42 mutants

a - b > c	a * b > c	a / b > c
a % b > c	a > c	b > c
abs(a) - b > c	a - abs(b) > c	a - b > abs(c)
abs(a - b) > c	0 - b > c	a - 0 > c
a - b >= c	a - b < c	a - b <= c
a - b = c	a - b != c	b - b > c
a - a > c	c - b > c	a - c > c
a - b > a	a - b > b	a - b > c
++a - b > c	a - ++b > c	a - b > ++c
--a - b > c	a - --b > c	a - b > --c
++(a - b) > c	--(a - b) > c	-a - b > c
a - -b > c	a - b > -c	(a - b) > c
a - b > 0	-abs(a) - b > c	a - -abs(b) > c
a - b > -abs(c)	-abs(a - b) > c	0 > c

WEAK MUTATION

- Problem: There are lots of mutants. Running each test case to completion on every mutant is expensive
  - Number of mutants grows with the square of program size
- Approach:
  - Execute meta-mutant (with many seeded faults) together with original program
  - Mark a seeded fault as "killed" as soon as a difference in intermediate state is found
    - Without waiting for program completion
    - Re-start with new mutant selection after each "kill"

USING COVERAGE INFORMATION

- Select only test cases which cover the changed code.
- For a test to kill a mutant
  - It should execute the changed code (E)
  - Infect the program state (I, typically achieved)
  - Propagate the infection to program output (P)
- Without execution of changed code, no difference in behavior can be observed!

USING COVERAGE INFORMATION

```
int triangle(int a, int b, int c){

    if (a <= 0 || b <=0 || c <= 0){
        return 4;   // not  a triangle
    }
    if (!(a+b >c && a +c > b && b + c >a)){
        return 4;   // not a triangle
    }
    if  (a == b && b == c){
        return 1;  // equilateral
    }
    if (a == b || b == c || a == c){
        return 2;  // isosceles
    }
    return 3;   // scalene
}
```

(0,0,0)
(1,1,3)
(2,2,2)
(2, 2,3)
(2,3, 4)
(0,1,1)
(4,3,2)
(1,1,1)
(2,3,2)

Only these tests execute mutants in this line

MUTATION TESTING ASSUMPTIONS

- **Competent programmer hypothesis:**
  - Programs are nearly correct
    - Real faults are small variations from the correct program
    - => Mutants are reasonable models of real buggy programs
- **Coupling effect hypothesis:**
  - Tests that find simple faults also find more complex faults
    - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too