Software Testing Techniques

Prof. Lionel Briand

Simula Research Laboratory

Oslo, Norway

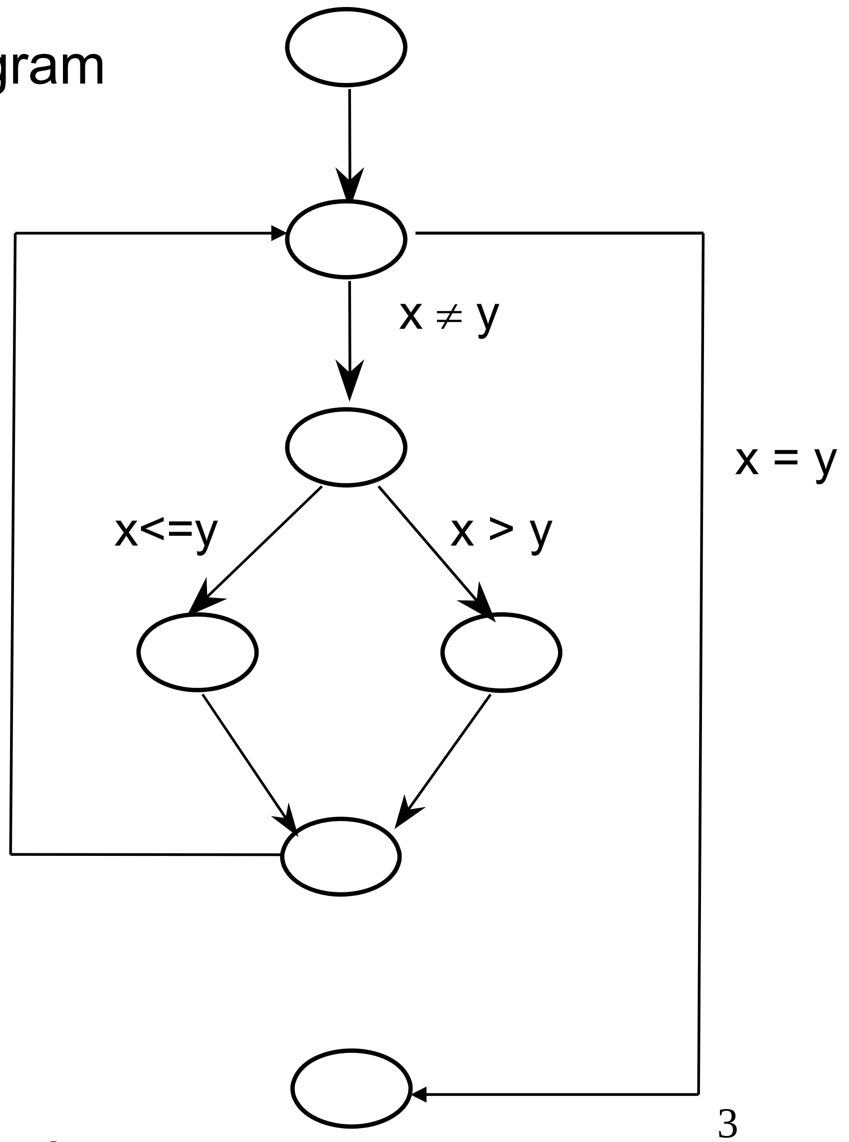
briand@simula.no

White-Box Testing: Control Flow Analysis

Control Flow Coverage (CFG) - Example

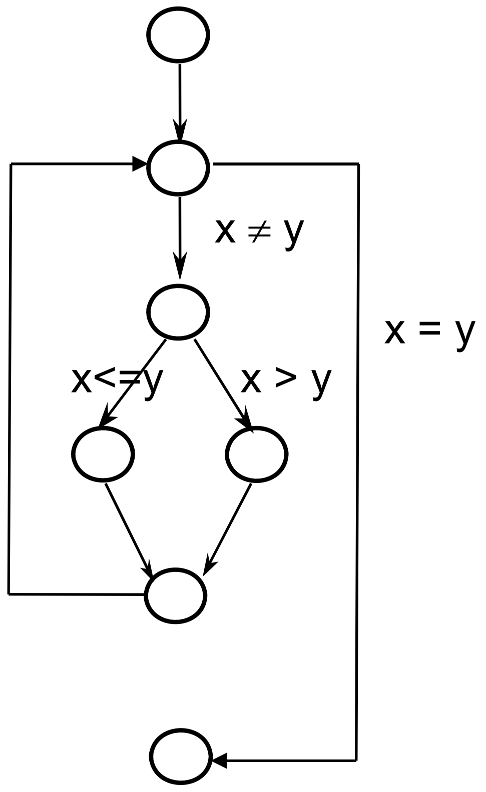
Greatest common divisor (GCD) program

```
read(x);
read(y);
while x ≠ y loop
   if x>y then
        x := x - y;
   else
        y := y - x;
   end if;
end loop;
gcd := x;
```

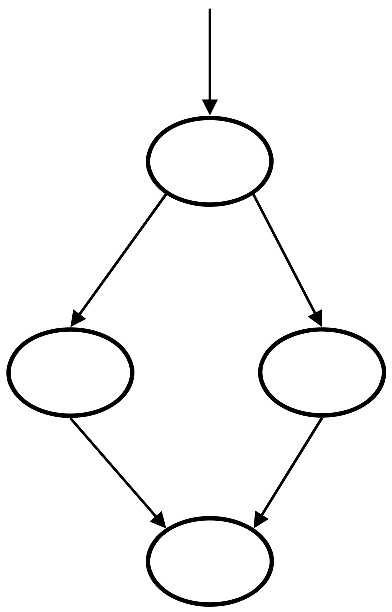


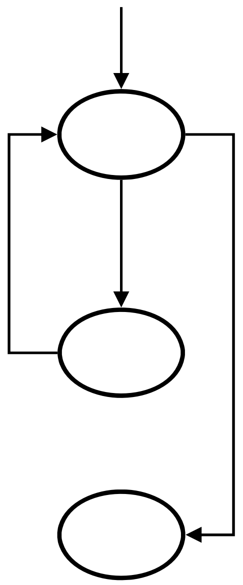
Control Flow Coverage (CFG) - Definitions

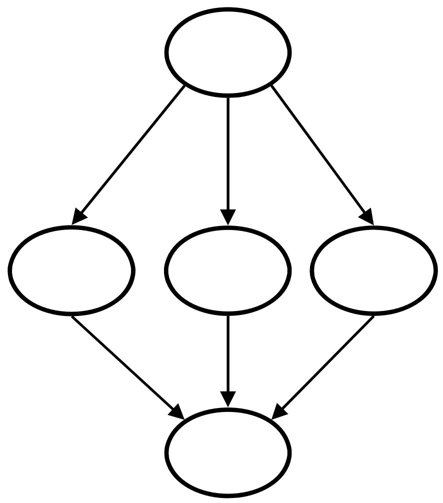
- Directed graph
- Nodes are blocks of sequential statements
- Edges are transfers of control
- Edges may be labeled with predicate representing the condition of control transfer
- There are several conventions for flow graph models with subtle differences (e.g., hierarchical CFGs, concurrent CFGs)



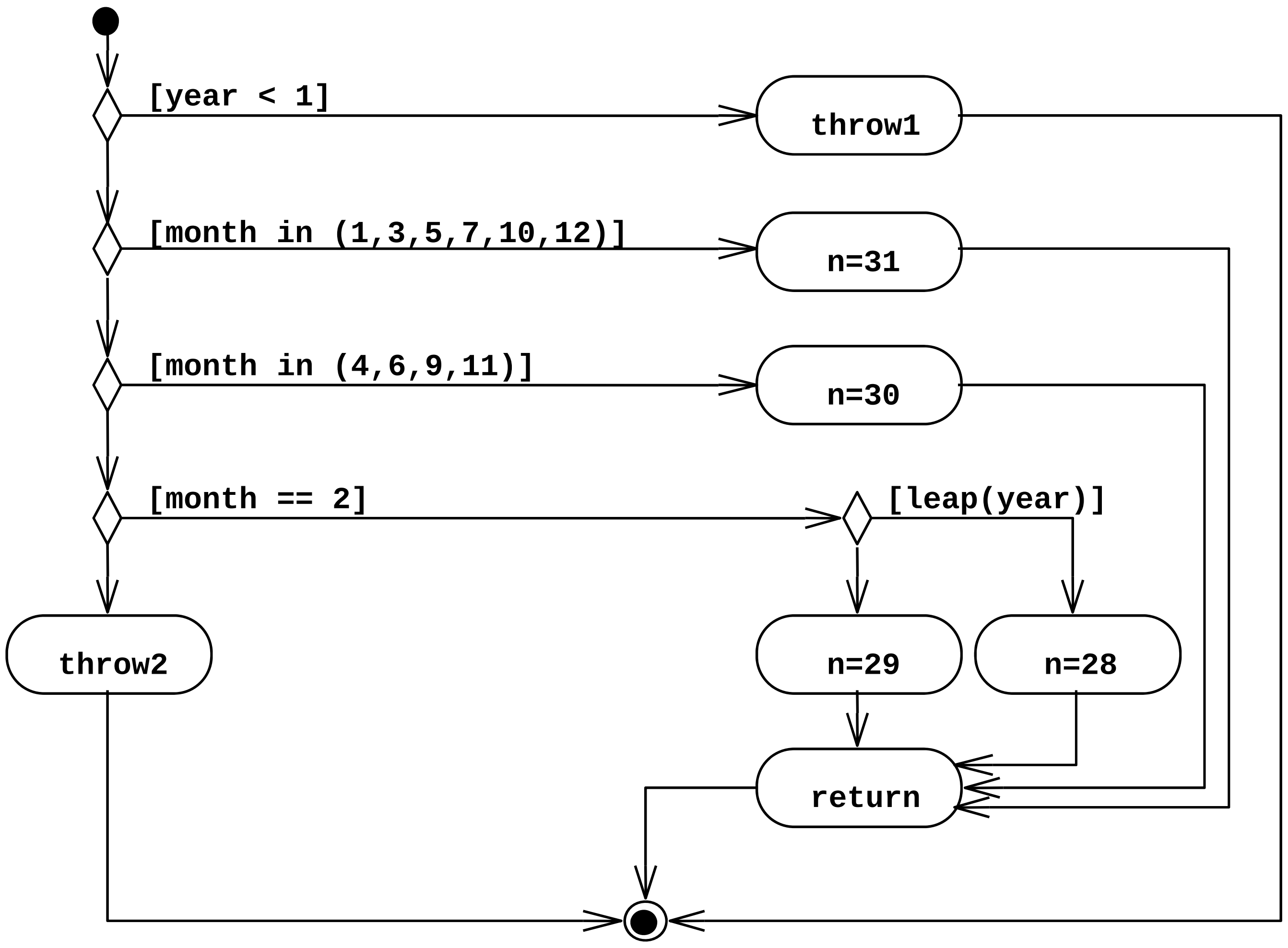
Basics of CFG: Blocks


If-Then-Else


While loop

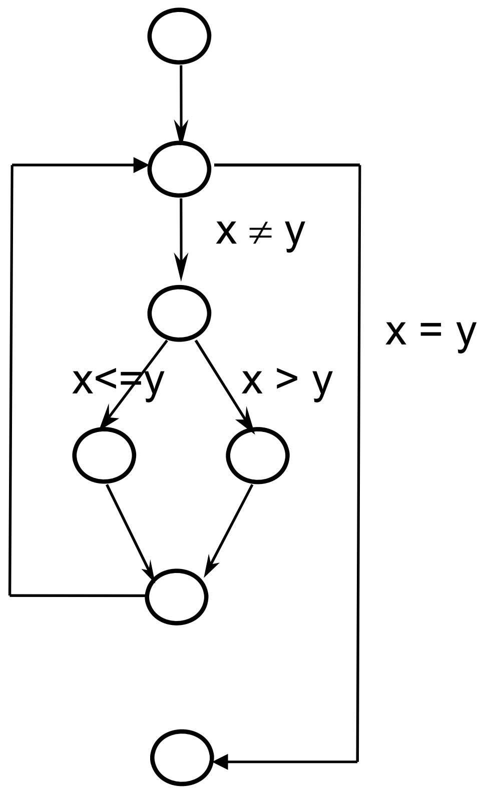

Switch

UML Activity Diagrams: Can also be used to model CFGs



Testing Coverage of Control flow

- As a testing strategy, we may want to ensure that testing exercises control flow:
  - Statement/Node Coverage
  - Edge Coverage
  - Condition Coverage
  - Path Coverage
- Discussed in detail next…



Adequacy Criterion

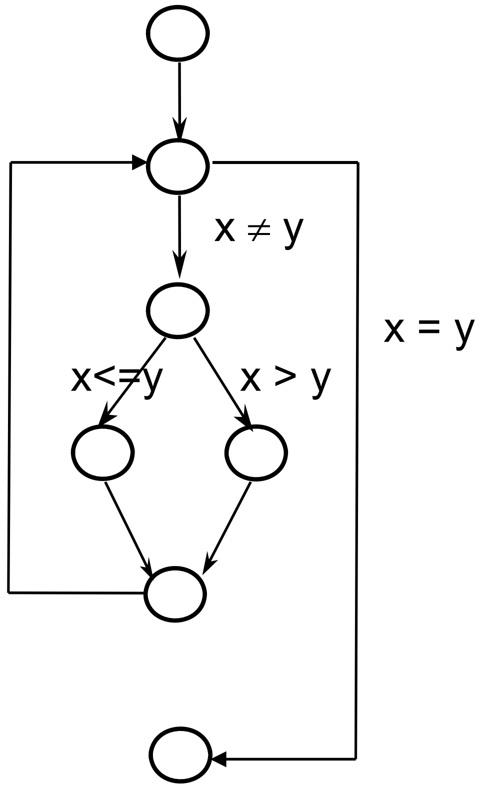
- **Way to define a test objective**
- **Given a criterion C for a model M**
  - The *coverage ratio* of a test set T is the proportion of the elements in M defined by C covered by the test set.
  - A test set T is said to be adequate for C, or simply C-adequate, when the coverage ratio achieves 100% for criterion C.
- **Example:**
  - M is the control flow graph of a function
  - C is the set of all the edges in the graph

Types of control flow coverage

- Statement/Node Coverage
- Edge Coverage
- Condition Coverage
- Path Coverage

Statement/Node Coverage

- Hypothesis: Faults cannot be discovered if the parts containing them are not executed
- Statement coverage criteria: Equivalent to covering all nodes in CFG
- Executing a statement is a weak guarantee of correctness, but easy to achieve
- In general, several inputs execute the same statements
- An important question in practice is: how can we minimize (the number of) test cases so we can achieve a given statement coverage ratio (e.g., 90%)?



Statement/Node Coverage: Incompleteness

- Statement coverage may lead to incompleteness
- An example:

```
if x < 0 then
     x := -x;
else
     null;
end if
z := x;
```

A negative x would result in the coverage of all statements.

But not exercising x >= 0 would not cover all cases (*implicit* code in italic and gray).

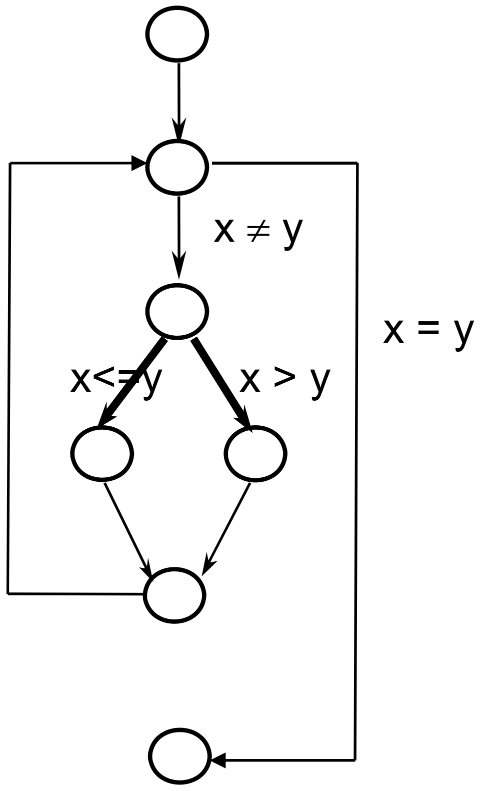
However, doing nothing for the case x >= 0 may turn out to be wrong and need to be tested.

Types of control flow coverage

- Statement/Node Coverage
- Edge Coverage
- Condition Coverage
- Path Coverage

Edge Coverage

- Based on the program structure, the control flow graph (CFG)
- Edge coverage criterion: Select a test set T such that, by executing P for each test case t in T, each edge of P's control flow graph is traversed at least once
- Exercise all conditions that govern control flow of the program with true and false values



Code Example: Searching for an element in a table

Should have been $\leq$

```
counter:= 0;
found := false;
if number_of_items ≠ 0 then
   counter :=1;
   while (not found) and counter $\boxed{<}$ number_of_items loop
        if table(counter) = desired_element then
                found := true;
        end if;
        counter := counter + 1;
   end loop;
end if;
if found then write (“the desired element exists in the table”);
else write (“the desired element does not exists in the table”);
end if;
```

Test Set

- We choose a test set with two test cases:
  - One table with 0 items and,
  - A table with 3 items, the second element being the desired one
  - Size of test set in this case, |T| = 2
- For the second test case, the “while” loop body is executed twice, once executing the “if-then” branch.
- The edge coverage criterion is fulfilled and the error is not discovered by the test set

```
...
while (not found) and counter < number_of_items loop
...
```

- The reason for the above problem?
  - Not all possible values of the *constituents* of the condition in the while loop have been exercised: **`counter < number_of_items`** never set to false
  - Fault is detected in case when trying to set both to false (item in last position)

Types of control flow coverage

- Statement/Node Coverage
- Edge Coverage
- Condition Coverage
- Path Coverage

Condition Coverage

- We need to further strengthen the edge coverage (recall the possible limitation)
- Condition Coverage (CC) Criterion: Select a test set T such that, by executing P for each element in T, each edge of P's control flow graph is traversed, and all possible values of the constituents of *compound conditions* (defined below) are exercised at least once
- Compound conditions: C1 and C2 or C3 … where Ci's are relational expressions or Boolean variables (atomic conditions)
- Another version: Modified Condition Coverage (MCC) Criterion: Only combinations of values such that every Ci drives the overall condition truth value twice (true and false).
- Examples are next…

Condition Coverage: Uncover <u>hidden</u> edges

- Two equivalent programs
  - though you would write the left one

```
if c1 and c2 then
   st;
else
   sf;
end if;
```

```
if c1 then
   if c2 then
        st;
   else
        sf;
   end if;
else
   sf;
end if;
```

- Edge coverage
  - would not compulsorily cover the “hidden” edges in the right one, (e.g., the 1st else) This is where C2 = false
  - Example: C2 = false might not be covered
- Condition coverage would cover C2 = false

Another Example

- The international standard DO-178B for Airborne Systems Certification (since 1992) requires testing the airborne software systems with modified condition coverage.
- Example : A ∧ (B ∨ C), e.g., in a *while* loop, let's look at its truth table…

	ABC	Results	Corresponding negate Case
1	TTT	T	A (5)
2	TTF	T	A (6), B (4)
3	TFT	T	A (7), C (4)
4	TFF	F	B (2), C (3)
5	FTT	F	A (1)
6	FTF	F	A (2)
7	FFT	F	A (3)
8	FFF	F	-

Deriving a modified condition criterion (MCC) Test suite: Take a pair for each constituent:

- A: (1,5), or (2,6), or (3,7)
- B: (2,4)
- C: (3,4)

Two minimal sets to cover the MCC:

- (2,3,4,6) or (2,3,4,7)

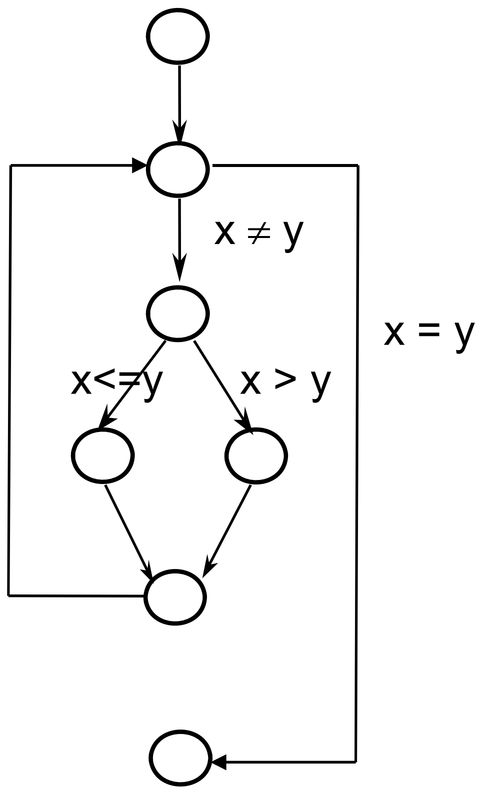
That is 4 test cases instead of 8 (1…8) for all possible combinations.

Types of Control Flow Coverage

- Statement/Node Coverage
- Edge Coverage
- Condition Coverage
- Path Coverage

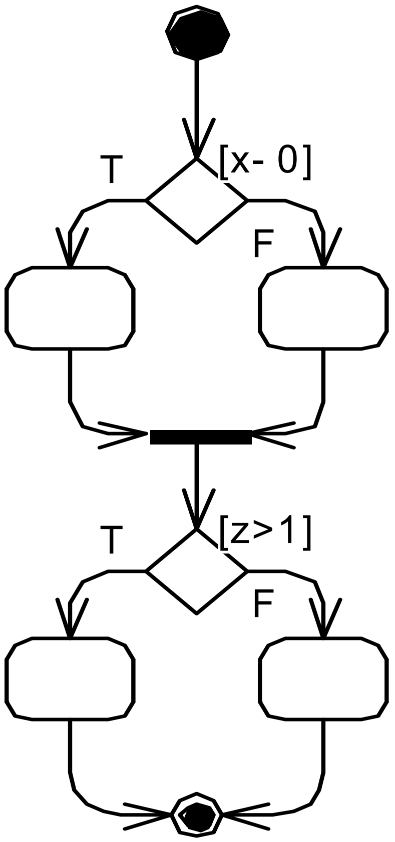
Path Coverage

- Path Coverage Criterion: Select a test set T such that, by executing P for each test case t in T, all paths leading from the initial to the final node of P's control flow graph are traversed
- In practice, however, the number of paths is too large, if not infinite (e.g., when we have loops)
- Some paths are infeasible (e.g., not practical given the system's business logic)
- Sometime, it is important to determine "critical paths", leading to more system load, security intrusions, etc.



Path Coverage - Example

```
if x ≠ 0 then
      y := 5;
else
      z := z – x;
end if;
if z > 1 then
      z := z / x;
else
      z := 0;
end if;
```



Let us compare how the following two test sets *cover* this CFG:

T1 (test set) =

{TC11:<x=0, z =1>,

TC12:<x =1, z=3>}

T2 =

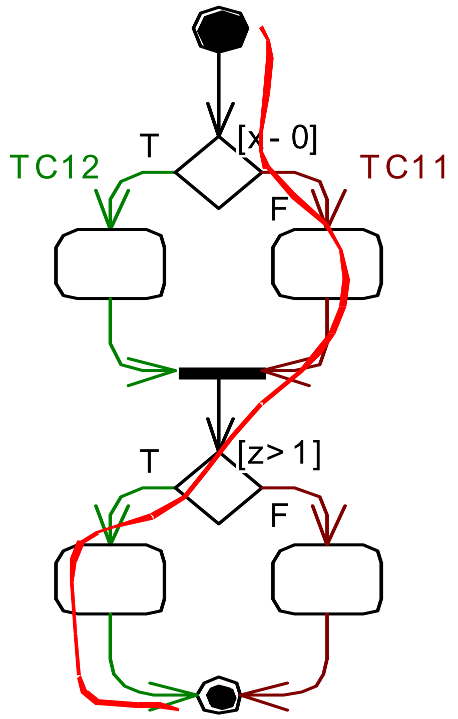
{TC21:<x=0, z =3>,

TC22:<x =1, z=1>}

Path Coverage – Example

T1's coverage

```
if x ≠ 0 then
        y := 5;
else
        z := z – x;
end if;
if z > 1 then
        z := z / x;
else
        z := 0;
end if;
```



T1's coverage:

T1 (test set) =
{TC11:<x=0, z =1>,
TC12:<x =1, z=3>}

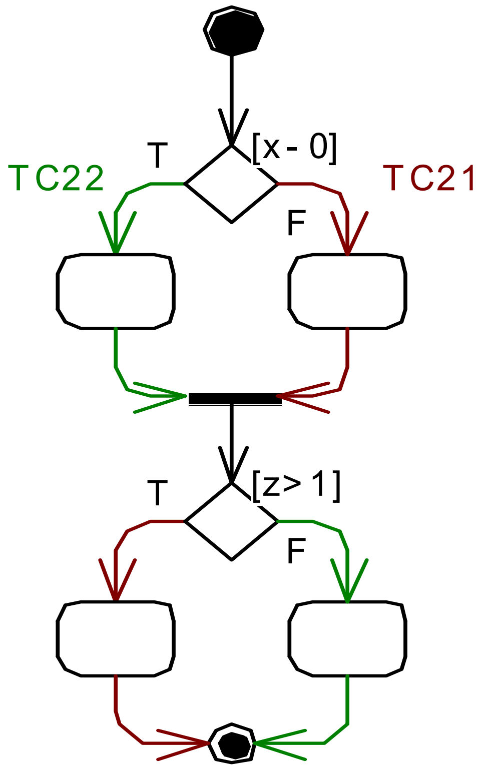
T1 executes all edges but…!

Do you see any testing issue (hidden paths which can be sources of failure)?

T1 executes all edges and all conditions but does not test risk of division by 0. (See the red "path")

Path Coverage – Example
T2's coverage

```
if x ≠ 0 then
      y := 5;
else
      z := z – x;
end if;
if z > 1 then
      z := z / x;
else
      z := 0;
end if;
```

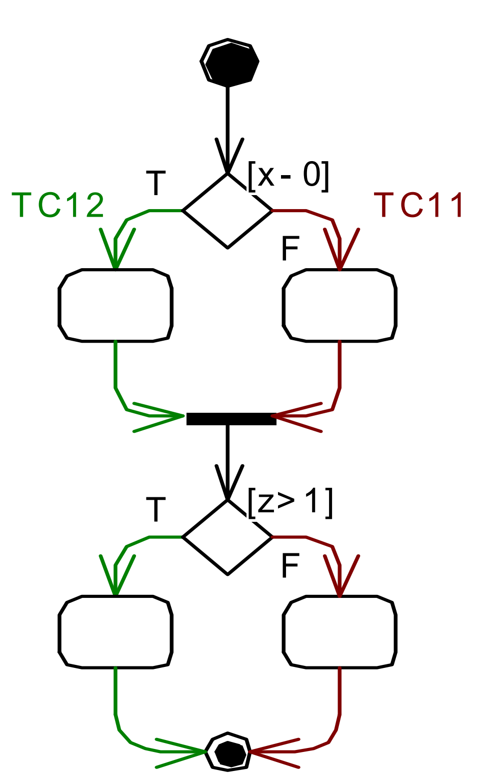


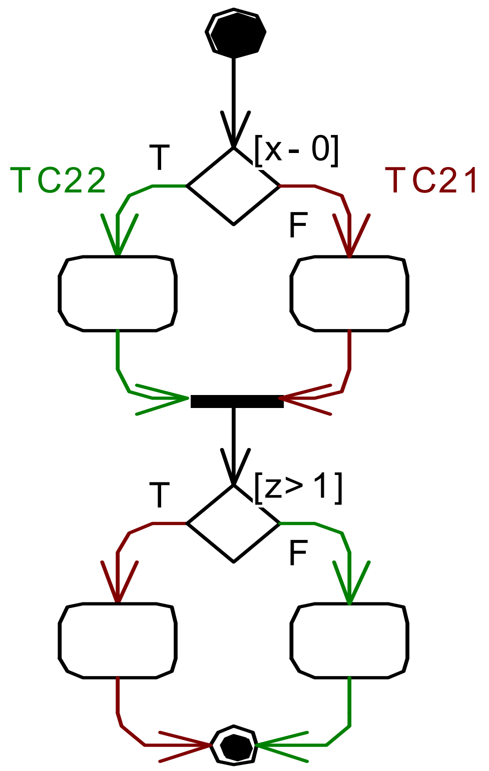
T2's coverage:

T2 =
{TC21:<x=0, z =3>,
TC22:<x =1, z=1>}

T2 would find the problem (testing the risk of division by 0) by exercising the remaining possible flows of control through the program fragment.

Path Coverage - Example





T1 (test set) = {TC11:<x=0, z =1>, TC12:<x =1, z=3>}

T1 executes all edges but do not show risk of division by 0

T2 = {TC21:<x=0, z =3>, TC22:<x =1, z=1>}

T2 would find the problem by exercising the remaining possible flows of control through the program fragment

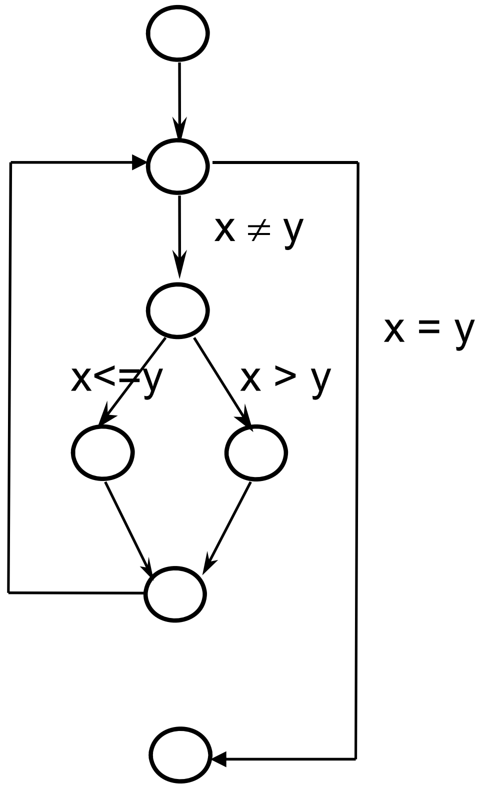
Observation:

T1 ∪ T2 -> all paths covered

Testing all four paths is required here for find the fault

Path Coverage: Issue

- Path Coverage Criterion: Select a test set T such that, by executing P for each test case t in T, all paths leading from the initial to the final node of P's control flow graph are traversed
- In practice, however, the number of paths can be too large, if not infinite (e.g., when we have loops)
- Some paths are infeasible (e.g., contradicting conditions, not feasible given the system's business logic).
- Sometime, it is important to determine "critical paths", leading to more system load, security intrusions, etc.



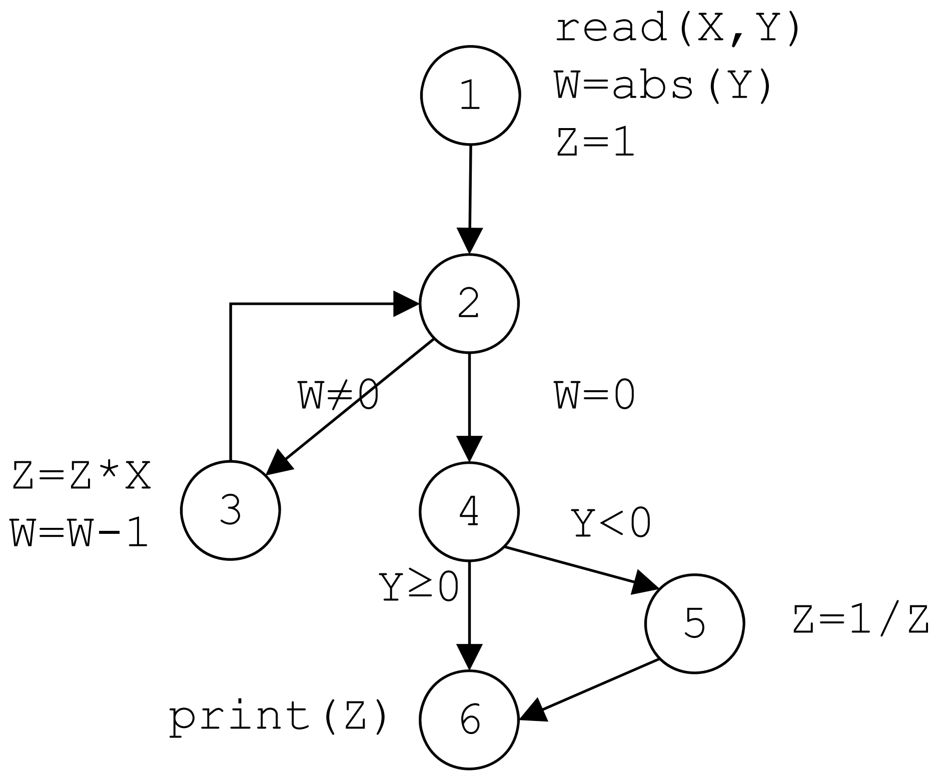
Path Coverage - Dealing with Loops

- In practice, however, the number of paths can be too large, if not infinite (e.g., when we have loops) → The size of test suites must be minimized
- A heuristic to tackle this problem: Look for conditions that execute loops
  - Zero times
  - A average number of times (statistical criterion)
  - A maximum number of times
- For example, in the array search algorithm (Slide 14)
  - Skipping the loop (the table is empty)
  - Executing the loop once or twice and then finding the element
  - Searching the entire table without finding the desired element

Path Coverage – Dealing with Loops
Example: Power Function

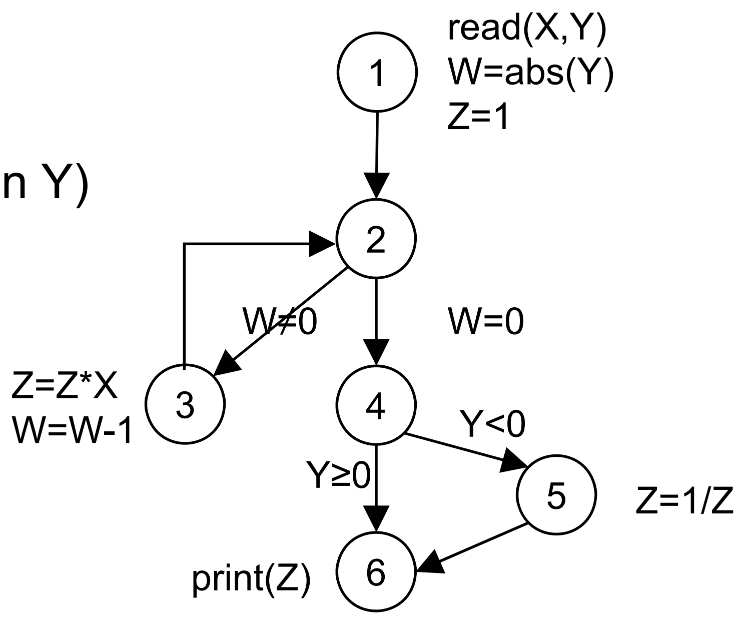
Program computing Z=X^Y

```
BEGIN
  read (X, Y) ;
  W = abs(Y) ;
  Z = 1 ;
  WHILE (W <> 0) DO
    Z = Z * X ;
    W = W - 1 ;
  END
  IF (Y < 0) THEN
    Z = 1 / Z ;
  END
  print (Z) ;
END
```



Path Coverage – comparison with “all edges” and “all statements”

- All paths
  - Infeasible path
    - ☞ 1 → 2 → 4 → 5 → 6, Why infeasible?
    - ☞ How Y and W relate (W = |Y|).
  - Potentially large number of paths (depends on Y)
    - ☞ As many ways to iterate 2 → (3 → 2)* as values of Abs(Y)
- All edges / branches
  - Two test cases are enough
    - ☞ Y<0 : 1 → 2 → (3 → 2)+ → 4 → 5 → 6
    - ☞ Y>0 : 1 → 2 → (3 → 2)* → 4 → 6
- All statements
  - One test case is enough
    - ☞ Y<0 : 1 → 2 → (3 → 2)+ → 4 → 5 → 6



Hierarchy of Coverage Criteria

The subsumption relation between criteria associated with the same model:

- Given a model M, and two criteria C1 and C2 for that model:

  C1 subsumes C2 if any C1-adequate test set is also C2-adequate.

- Example: All edges subsumes all nodes in CFG.
- If C1 subsumes C2, we assume:
  - Satisfying C1 is more expensive (e.g., # of test cases) than satisfying C2
  - C1 allows the detection of more faults than C2

Control Flow Coverage: Deriving Input Values

- Not all statements are usually reachable in real-world programs
- It is not always possible to decide automatically if a statement is reachable and the percentage of reachable statements
- When one does not reach a 100% coverage, it is therefore difficult to determine the reason
- Tools are needed to support this activity – and there exist good tools, e.g., Rose real-time
- But the generation of input values from coverage criteria cannot be fully automated
- Control-flow testing is, in general, more applicable to testing in the small

Generating CF-based Tests

- To find test inputs that will execute an arbitrary statement Q within a program source, the tester must work backward from Q through the program's flow of control to an input statement
- For simple programs, this amounts to solving a set of simultaneous inequalities on the input variables of the program, each inequality describing the branch through one conditional
- Conditionals may be expressed in local variable values derived from the inputs and those must figure in the inequalities as well

Example

```
int z;
scanf("%d%d", &x, &y);
if (x > 3) {
  z = x+y;
  y+= x;
  if (2*z == y) {
    /* statement to be covered */
…
```

Inequalities:

- $x > 3$
- $2(x+y) = x+y$

  $\Leftrightarrow x = -y$

1 Solution:

X = 4

Y= -4

Problems

- The presence of loops and recursion in the code makes it impossible to write and (automatically) solve the inequalities in general
- Each pass through a loop may alter the values of variables that figure in a following conditional and the number of passes cannot be determined by static analysis
- Coverage may be 100% and the tester may yet miss some functionalities (omission faults)

Black-Box Testing: Deriving Test Specifications from System Specifications

Basic Principles

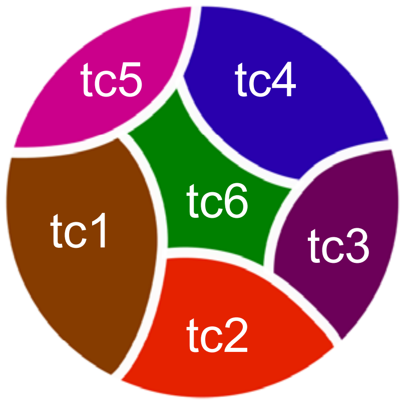
- Based on the definition of a program’s specification, as opposed to its structure
- The notion of complete coverage can also be applied to functional (Black-box) testing
- Rigorous specifications have another benefit, they help functional testing, e.g., categorize inputs, derive expected outputs
- In other words, they help test case generation and test oracles

Equivalence Class Testing

- Motivation: we would like to have a sense of complete testing and we would hope to avoid redundancy
- Equivalence classes: partitions of the input set in which input data have the same effect on the program (e.g., the result in the same output)
- Entire input set is covered: completeness

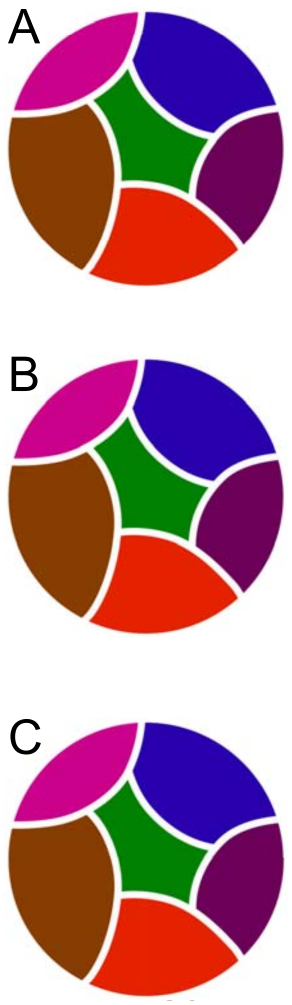
- Disjoint classes: avoid redundancy
- Test cases: one element of each equivalence class
- But equivalence classes have to be chosen wisely …
- Guessing the likely system behavior might be needed

A SUT's input set



Weak/Strong ECT

- For an example SUT, suppose there are three input variables from three domains: A, B, C
- $A = A_1 \cup A_2 \cup A_3 \cup \ldots \cup A_m$ where $a_i \in A_i$
- $B = B_1 \cup B_2 \cup B_3 \cup \ldots \cup B_n$ where $b_i \in B_i$
- $C = C_1 \cup C_2 \cup C_3 \cup \ldots \cup C_o$ where $c_i \in C_i$
- Weak Equivalence Class Testing: Choosing one variable value form each equivalence class (one a_i, b_i, and c_i) such that all classes are covered.
- Strong Equivalence Class Testing: Is based on the Cartesian product of the partition subsets ($A \times B \times C$), i.e., testing all interactions of all equivalence classes
- Examples next…



Weak ECT Test Cases

- For an example SUT, suppose there are three input variables from three domains: A, B, C
- $A = A_1 \cup A_2 \cup A_3$ where $a_i \in A_i$
- $B = B_1 \cup B_2 \cup B_3 \cup B_4$ where $b_i \in B_i$
- $C = C_1 \cup C_2$ where $c_i \in C_i$
- Number of WETCs needed=Max size w.r.t. the number of equivalence classes of {A, B, C}
- 4 WETCs are enough.

Test Case	A	B	C
WETC1	a1	b1	c1
WETC2	a2	b2	c2
WETC3	a3	b3	c1
WETC4	a1	b4	c2

Strong ECT (WECT) Test Cases

- |A| (number of equivalence classes) = 3
- |B| = 4
- |C| = 2

Test Case	A	B	C
SETC1	a1	b1	c1
SETC2	a1	b1	c2
SETC3	a1	b2	c1
SETC4	a1	b2	c2
SETC5	a1	b3	c1
SETC6	a1	b3	c2
SETC7	a1	b4	c1
SETC8	a1	b4	c2
SETC9	a2	b1	c1
SETC10	a2	b1	c2
SETC11	a2	b2	c1
SETC12	a2	b2	c2
SETC13	a2	b3	c1
SETC14	a2	b3	c2
SETC15	a2	b4	c1
SETC16	a2	b4	c2
SETC17	a3	b1	c1
SETC18	a3	b1	c2
SETC19	a3	b2	c1
SETC20	a3	b2	c2
SETC21	a3	b3	c1
SETC22	a3	b3	c2
SETC23	a3	b4	c1
SETC24	a3	b4	c2

Equivalence Class Testing

`NextDate` Example

- **`NextDate`** is a function with three variables: **`month`**, **`day`**, **`year`**. It returns the date of the day after the input date. Limitation: years 1812-2012
- Treatment Summary: if it is not the last day of the month, the next date function will simply increment the day value. At the end of a month, the next day is 1 and the month is incremented. At the end of the year, both the day and the month are reset to 1, and the year incremented. Finally, the problem of leap year makes determining the last day of a month interesting.
- From Wikipedia (http://en.wikipedia.org/wiki/Leap_years): The Gregorian calendar adds a 29th day to February in all years evenly divisible by 4, except for centennial years (those ending in -00) which are not evenly divisible by 400. Thus 1600, 2000 and 2400 are leap years but 1700, 1800, 1900, 2100, 2200 and 2300 are not.
- The year 1900 falls in the 1812-2012 period

`NextDate` Equivalence Classes

- M1 = {month | month has 30 days}
- M2 = {month | month has 31 days}
- M3 = {February} (the impact of leap years)

- D1 = {day | 1<= day <= 28}
- D2 = {29}
- D3 = {30}
- D4 = {31}

- Y1 = {1900}
- Y2 = {year | 1812 <= year <= 2012 AND (year != 1900) AND (year mod 4 = 0)}
- Y3 = {year | (1812 <= year <= 2012 AND year mod 4 != 0)}

`NextDate`
Weak Equivalence Class Testing (WECT)

- Number of WECT test cases=maximum partition size (D)=4

Test Case ID	Month	Day	Year	Output
WETC1	6	14	1900	6/15/1900
WETC2	7	29	1912	7/30/1912
WETC3	2	30	1913	Invalid Input date
WETC4	6	31	1900	Invalid Input date

NextDate Strong Equivalence Class Testing (SECT)

- Number of SECT test cases= partition size (D) x partition size (M) x partition size (Y) = 3x4x3=36 test cases

Test Case ID	Month	Day	Year	Expected Output
SE1	6	14	1900	6/15/1900
SE2	6	14	1912	6/15/1912
SE3	6	14	1913	6/15/1913
SE4	6	29	1900	6/30/1900
SE5	6	29	1912	6/30/1912
SE6	6	29	1913	6/30/1913
SE7	6	30	1900	7/1/1900
SE8	6	30	1912	7/1/1912
SE9	6	30	1913	7/1/1913
SE10	6	31	1900	ERROR
SE11	6	31	1912	ERROR
SE12	6	31	1913	ERROR
SE13	7	14	1900	7/15/1900
SE14	7	14	1912	7/15/1912
SE15	7	14	1913	7/15/1913
SE16	7	29	1900	7/30/1900
SE17	7	29	1912	7/30/1912
SE18	7	29	1913	7/30/1913
SE19	7	30	1900	7/31/1900
SE20	7	30	1912	7/31/1912
SE21	7	30	1913	7/31/1913
SE22	7	31	1900	8/1/1900
SE23	7	31	1912	8/1/1912
SE24	7	31	1913	8/1/1913
SE25	2	14	1900	2/15/1900
SE26	2	14	1912	2/15/1912
SE27	2	14	1913	2/15/1913
SE28	2	29	1900	ERROR
SE29	2	29	1912	3/1/1912
SE30	2	29	1913	ERROR
SE31	2	30	1900	ERROR
SE32	2	30	1912	ERROR
SE33	2	30	1913	ERROR
SE34	2	31	1900	ERROR
SE35	2	31	1912	ERROR
SE36	2	31	1913	ERROR

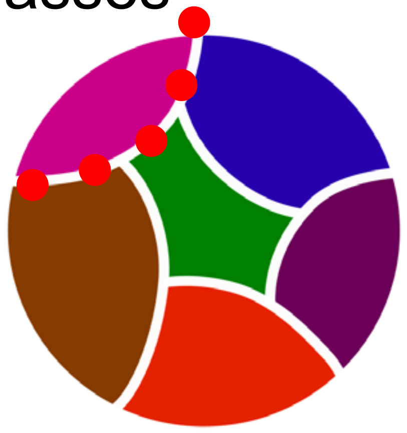
Equivalence Class Testing

`NextDate` Example - Discussion

- If error conditions are a high priority, we should extend strong equivalence class testing to include invalid classes
- Equivalence Class Testing is appropriate when input data defined in terms of ranges and sets of discrete values
- SECT makes the assumption that the variables are independent – dependencies will generate “error” test cases
- Possibly too many of them …
- We will discuss the “category-partition” test techniques next to address this issue

Boundary-Value Analysis Motivations

- In equivalence class testing, we partition input domains into equivalence classes, on the assumption that the behavior of the program is “similar” for all input values of a equivalence class
- But the above assumption may not be true in all cases as…
- Some typical programming errors happen to be at the boundary between different equivalence classes
- This is what boundary value testing focuses on
- Simpler but complementary to equivalence class testing



Boundary-Value Analysis: Motivations

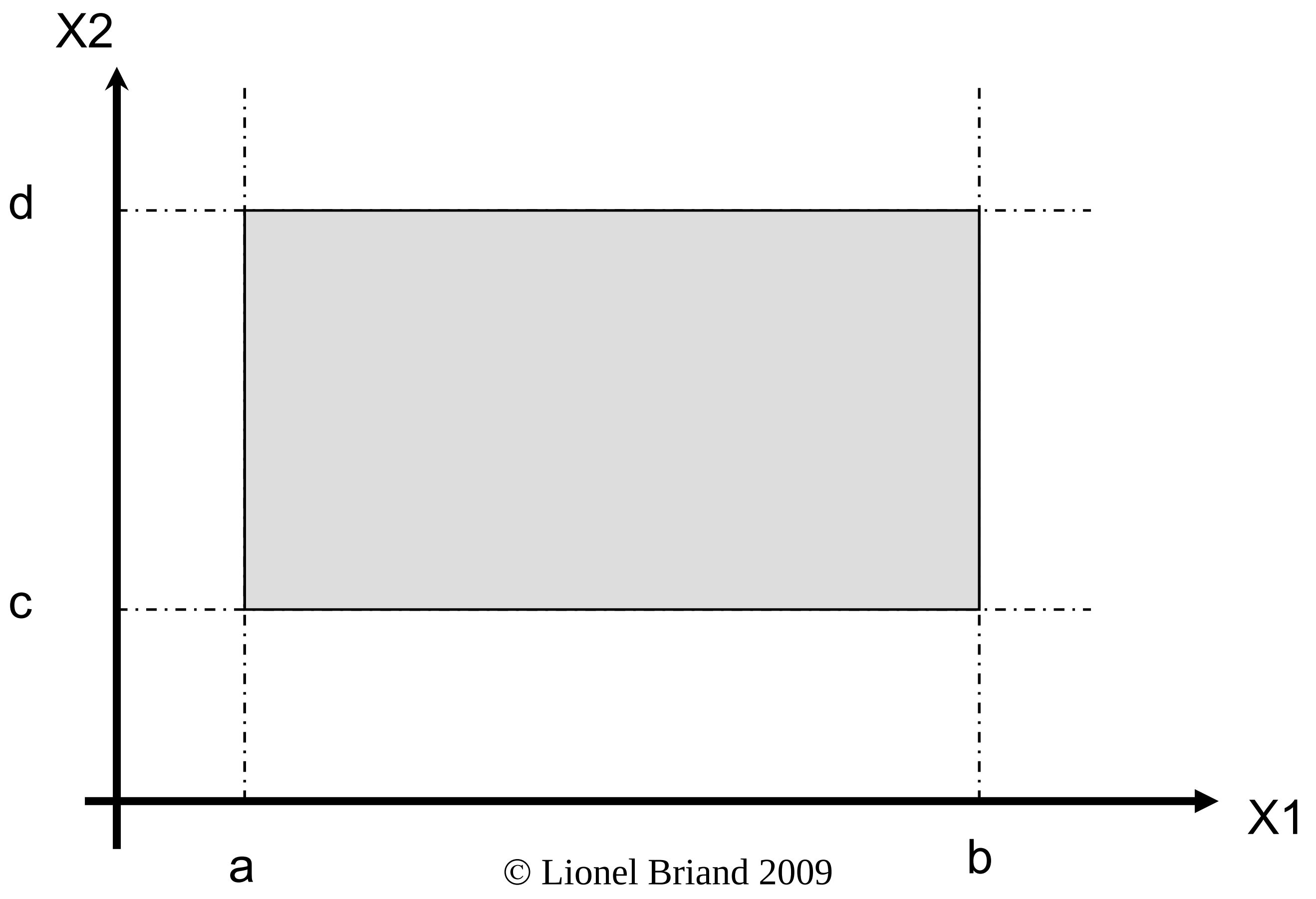
- Assume a function F, with two variables x1 and x2
- (Possibly unstated) boundaries: a <= x1 <= b, c <= x2 <= d
- In some programming languages, strong typing allows the specification of such intervals
  - Strong typing enforces checks during conversions between type domains to ensure that the value will make sense in the new domain.
- In boundary-value analysis (testing) the focus is on the boundary of the input space for identifying test cases
- The rationale is that errors tend to occur near extreme values of input variables

Boundary-Value Analysis: Basic Ideas

- Setting values for input variable at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum.
- Convention for the above notions:
  - min, min+, nom, max-, max
  - Assuming x1 is an integer: a, a+1, *, b-1, b
- A usual strategy for all input variables: Holding the values of all but one variable at their nominal values, letting one variable assume its extreme value
  - a, c +2<= x2 <= d-2

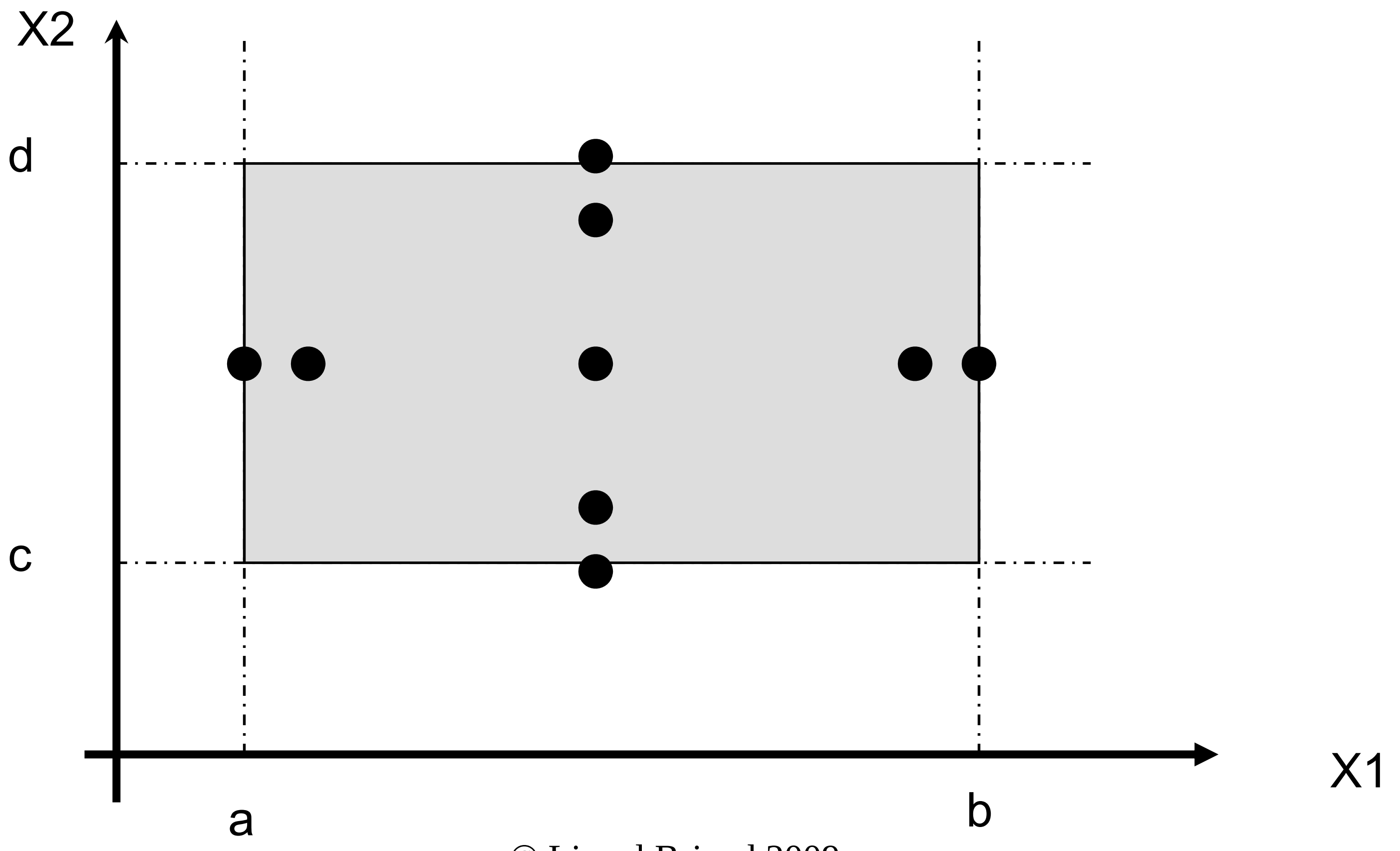
BVA of Input Domain of Function F

- Assume a function F, with two variables x1 and x2
- (Possibly unstated) boundaries: a <= x1 <= b, c <= x2 <= d



BVA: Test Cases

- Test set ={ <$x1_{nom}$, $x2_{min}$>, <$x1_{nom}$, $x2_{min+}$>, <$x1_{nom}$, $x2_{nom}$>, <$x1_{nom}$, $x2_{max-}$>, <$x1_{nom}$, $x2_{max}$>, <$x1_{min}$, $x2_{nom}$,>, <$x1_{min+}$, $x2_{nom}$,>, <$x1_{max-}$, $x2_{nom}$>, <$x1_{max}$, $x2_{nom}$>}. Number of test cases=9

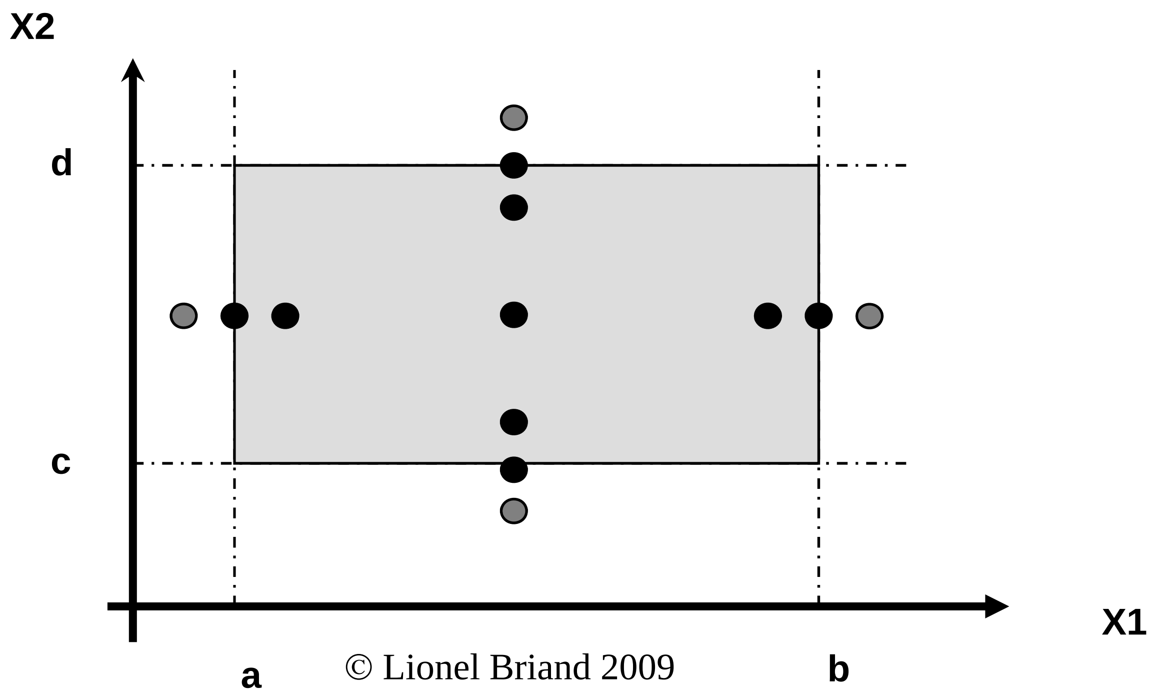


BVA: General Case and Limitations

- A function with n variables will require 4n + 1 test cases
- Works well with variables that represent bounded physical quantities
- No consideration of the nature of the function and the meaning of variables
- An elementary technique that is a extendable to robustness testing

BVA: Robustness Testing

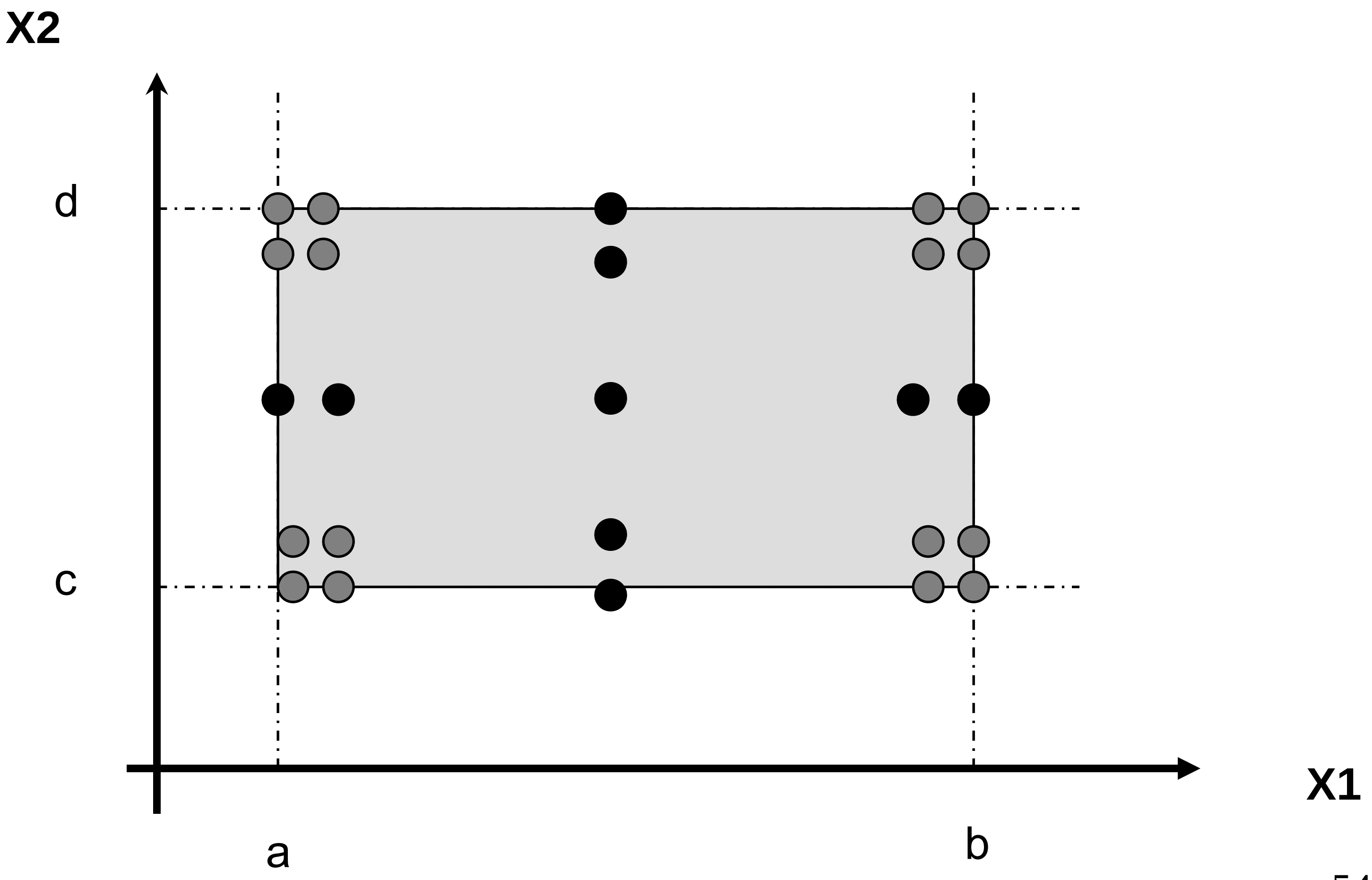
- In Robustness Testing, we also look at the behavior of the system when the variable extremes are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min-)
- This is an extension of boundary-value testing.



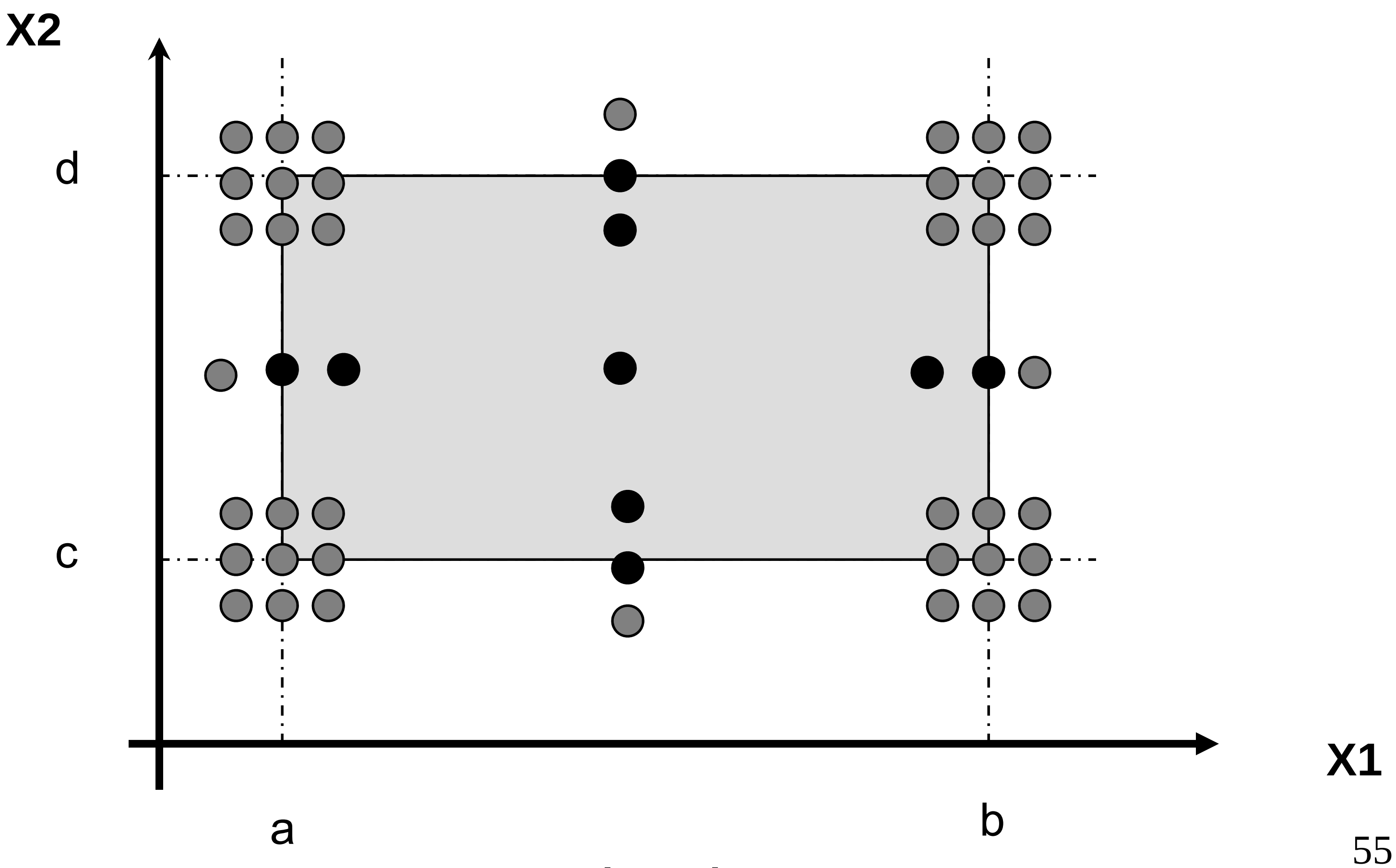
BVA: Worst Case Testing (WCT)

- Boundary-value analysis makes the common assumption that failures, most of the time, originate from one fault
- What happens when more than one variable has an extreme value?
- The idea comes from electronics in circuit analysis
- Cartesian product of {min, min+, nom, max-, max}
- Clearly more thorough than boundary-value analysis, but much more effort: 5^n test cases (n: number of variables)
- WCT is a good strategy when physical variables have numerous interactions, and where failure is costly

BVA: Worst Case Testing for 2 variables



BVA: Robust Worst Case Testing for 2 variables



Category-Partition Testing Steps

- The system is divided into individual “functions” that can be independently tested (e.g., use cases)
- The method identifies the *parameters* of each “function” and, for each parameter, identifies distinct *categories*
- *Categories* are major properties or characteristics for each parameter
- Besides parameters, *environment characteristics,* under which the function operates (characteristics of the system state), can also be considered
- The *categories* are further subdivided into *choices* in the same way as equivalence partitioning is applied (possible “values”)
- Encompasses Equivalence Class Partitioning

Category-Partition Testing: A Small Example

- Function: Sorting an array
- Characteristics (Categories):
  - Length of array (Len)
  - Type of elements
  - Max value
  - Min value
  - Position of the max value (Max pos)
  - Position of the min value
- Choices for Max pos: {1, 2..Len-1, Len}

Category-Partition Testing Steps (II)

- The *constraints* operating between choices are then identified, i.e., how the occurrence of one choice can affect the existence of another
    - E.g., in the array sorting example, if Len = 0, then the rest does not matter
- *Test frames* (or test specifications) are then generated which are defined as the allowable combinations of choices in the categories
- Test frames are then converted into *test data*
- Examples next…

Category-Partition Testing: An Example

- Specification: The program prompts the user for a positive integer in the range 1 to 20 and then for a string of characters of that length.
- The program then prompts for a character and returns either the position in the string at which the character was first found or a message indicating that the character was not present in the string.
- The user has the option to search for more characters.

Parameters and Categories

- Three parameters: integer x (length), the string a, and the character c
- For the length x, an interesting category is whether it is “in-range” according to the specification (1-20)
- For string a, an interesting category is its length
- For character c, an interesting category is the location of c in string a
- Choosing categories is based on understanding the specifications and the behavior of the software under test - This is not a mechanical task

Choices

- For Integer x:
  - out-of-range: 0, 21
  - in-range: 1, 2-19, 20
- For String a:
  - minimal, maximal, intermediate length
  - 1, 2-19, 20
- For Character c:
  - first, middle, last, does not occur
- Note: should have more than one choice in each category
- Combine boundary analysis, robustness and equivalence class partitioning

Test Specifications with Constraints

x (length):

Choices

1)	0	[error]	
2)	1	[property stringOk, MinLength]	independent
3)	2-19	[property stringOk, MidLength]	
4)	20	[property stringOk, MaxLength]	
5)	21	[error]	

a (string):

Choices

1)	Of length 1	[if stringOk and MinLength]	dependent
2)	Of length 2-19	[if stringOk and MidLength]	
3)	Of length 20	[if stringOk and MaxLength]	

c (character):

Choices

1)	At first position in string	[if stringOk]
2)	At last position in string	[if stringOk and not MinLength]
3)	In middle of string	[if stringOk and not MinLength]
4)	Not in string	[if stringOk]

Specifying constraints among choices

- *Properties,* and *Selectors* are associated with choices

3 properties

Category A

ChoiceA1 [property X, Y, Z]

ChoiceA2

Category B

A selector

ChoiceB1

ChoiceB2 [if X and Z]

- E.g., in the above abstract example, ChoiceA2 and ChoiceB2 would not be combined into a test frame

Special Constraints

- **[Error]**
  - It is assumed that if the parameter or environment variable has this particular choice, any call of the function using that choice will result in the same error.
  - A choice marked with [Error] is not combined with choices in the other categories to create *test frames*.
  - During the test, the tester can set the test's other parameters and environment conditions at will.
- **[Single]**
  - This notation is intended to describe special, unusual, or redundant conditions that do not have to be combined with all possible choices, e.g., 1900 in NextDate().
  - A judgment by the tester that the marked choice can be adequately tested with only one test case

Test Frames and Test Cases

12 test cases. The number depends on the inter-choice constraint dependencies

x (length):

1)	0	[error (out of range)]
2)	1	[property stringOk, MinLength]
3)	2-19	[property stringOk, MidLength]
4)	20	[property stringOk, MaxLength]
5)	21	[error (out of range)]

a (string):

1)	Of length 1	[if stringOk and MinLength]
2)	Of length 2-19	[if stringOk and MidLength]
3)	Of length 20	[if stringOk and MaxLength]

c (character):

1)	At first position in string	[if stringOk]
2)	At last position in string	[if stringOk and not MinLength]
3)	In middle of string	[if stringOk and not MinLength]
4)	Not in string	[if stringOk]

x1	**x = 0**
x2a1c1	**x = 1, a = ‘A’, c = ‘A’**
x2a1c4	**x = 1, a = ‘A’, c = ‘B’**
x3a2c1	**x = 7, a = ‘ABCDEFG’, c = ‘A’**
x3a2c2	**x = 7, a = ‘ABCDEFG’, c = ‘G’**
x3a2c3	**x = 7, a = ‘ABCDEFG’, c = ‘D’**
x3a2c4	**x = 7, a = ‘ABCDEFG’, c = ‘X’**
x4a3c1	**x = 20, a = ‘ABCDEFGHIJKLMNOPQRST’, c = ‘A’**
x4a3c2	**x = 20, a = ‘ABCDEFGHIJKLMNOPQRST’, c = ‘T’**
x4a3c3	**x = 20, a = ‘ABCDEFGHIJKLMNOPQRST’, c = ‘J’**
x4a3c4	**x = 20, a = ‘ABCDEFGHIJKLMNOPQRST’, c = ‘X’**
x5	**x = 21**

Criteria Using Choices

- All Combinations (AC): This is what was shown in the previous example, what is typically done when using category-partition. One value for every choice of every parameter must be used with one value of every (possible) choice of every other category.
- Each choice (EC): This is a weaker criterion. One value from each choice for each category must be used at least in one test case.
- Base Choice (BC): Compromise. A base choice is chosen for each category, and a first base test is formed by using the base choice for each category. Subsequent tests are chosen by holding all but one base choice constant and forming choice combinations by covering all non-base choices of the selected category. This procedure is repeated for each category.
- The base choice can be the simplest, smallest, first in some ordering, or most likely from an end-user point of view, e.g., in the previous example, character c occurs in the middle of the string, length x is within 2-19.
  - Base choices only: x = 7, a = ‘ABCDEFG’, c = ‘D’
  - Two base choices, one non-base choice: x = 7, a = ‘ABCDEFG’, c = ‘G’

Category Partition: Conclusions

- Identifying parameters and environments conditions, and categories, heavily relies on the experience of the tester
- Makes testing decisions explicit (e.g., constraints), open for review
- Combine boundary-value analysis, robustness testing, and equivalence-class partitioning
- Once specifying categories, choices, and constraints is completed, the technique is straightforward and can be automated (e.g., Siemens tool)
- The criteria for test-case reduction makes it useful for practical testing

Combining WB and BB Testing

Brian Marick recommends the following approach (for large scale testing):

1. Use black-box testing to generate functional tests from requirements and design to try every function.
2. Check the structural coverage after the functional tests are all verified to be successful.
3. Where the structural coverage (e.g., edge) is imperfect, generate functional tests (not structural) that induce the additional coverage.

This works because form (structure) should follow function!

- Uncovered code must have some purpose, and that purpose has not been invoked, so some function is untested