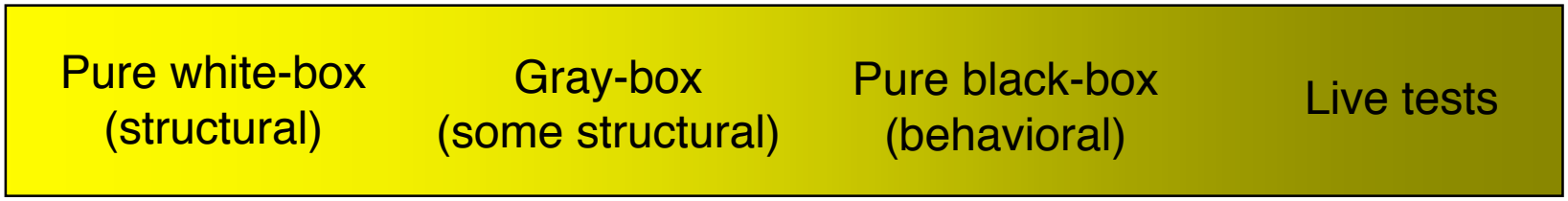
Empirical Aspects of Testing



Development activity

Testing activity

Technical support,
beta testing

Black-Box Testing Tools

- **Plan & Checklists**: tests to perform
  - With each test, pre-condition, post-condition
- **Test matrix/spreadsheet**: listing tests against **use cases** and potential error areas
- **Logging capability**
  - Note pad
  - Keystroke recorder (for playback)
  - Event logger
  - Screen recorder
  - Video camera
- **Tracking Database**

Testing Matrix

Tested by					
	Test 1	Test 2	Test 3	Test 4	Test 5
Use Case 1	x				
Use Case 2		x	x		
Use Case 3		x		x	x

Test Result Categorization

- Area of defect
- Severity level of defect
- Follow up:
  - Responsibility
  - Time to fix
  - Lines of code affected

10 Sample Severity Levels, with examples (Boris Beizer)

- **Mild**: misspellings in output
- **Moderate**: misleading or redundant behavior
- **Annoying**: truncated names, etc.
- **Disturbing**: some transactions not processed
- **Serious**: lost transaction
- **Very serious**: incorrect output
- **Extreme**: frequent "very serious" errors
- **Intolerable**: data corrupted
- **Catastrophic**: shutdown
- **Infectious**: shutdown spreading to others

4 Possible Reaction Levels to failed tests

- **Defer**: fix as time permits
- **Schedule**: fix by some future date
- **Required**: fix before acceptance
- **Immediate**: fix before testing is continued

Testing Forms and Tracking

- Discrepancy report form
- Error investigation form
- Error reporting/tracking system/database

Discrepancy Report Form

(Sherry Pfleeger)

DISCREPANCY REPORT FORM

DRF Number:_____ Tester name:_____
Date: _____ Time: _____
Test Number: _____
Script step executed when failure occurred: _____
Description of failure: _____
_____
_____
_____
Activities before occurrence of failure:
_____
_____
Expected results:
_____
_____
Requirements affected:
_____
_____
Effect of failure on test:
_____
_____
Effect of failure on system:
_____
_____
Severity level:
(LOW) 1 2 3 4 5 (HIGH)

FAULT REPORT S.P0204.6.10.3016

ORIGINATOR: Joe Bloggs

BRIEF TITLE: Exception 1 in dps_c.c line 620 raised by NAS

FULL DESCRIPTION Started NAS endurance and allowed it to run for a few minutes. Disabled the active NAS link (emulator switched to standby link), then re-enabled the disabled link and CDIS exceptioned as above.
(I think the re-enabling is a red herring.)

ASSIGNED FOR EVALUATION TO: **DATE:**

CATEGORISATION: **0 (1) 2 3 Design Spec Docn**
SEND COPIES FOR INFORMATION TO:
EVALUATOR: **DATE:** 8/7/92

CONFIGURATION ID	ASSIGNED TO	PART
dpo_s.c		

COMMENTS: dpo_s.c appears to try to use an invalid CID, instead of rejecting the message. AWJ

ITEMS CHANGED

CONFIGURATION ID	IMPLEMENTOR/DATE	REVIEWER/DATE	BUILD/ISSUE NUM	INTEGRATOR/DATE
dpo_s.c v.10	AWJ 8/7/92	MAR 8/7/92	6.120	RA 8-7-92

COMMENTS:

CLOSED

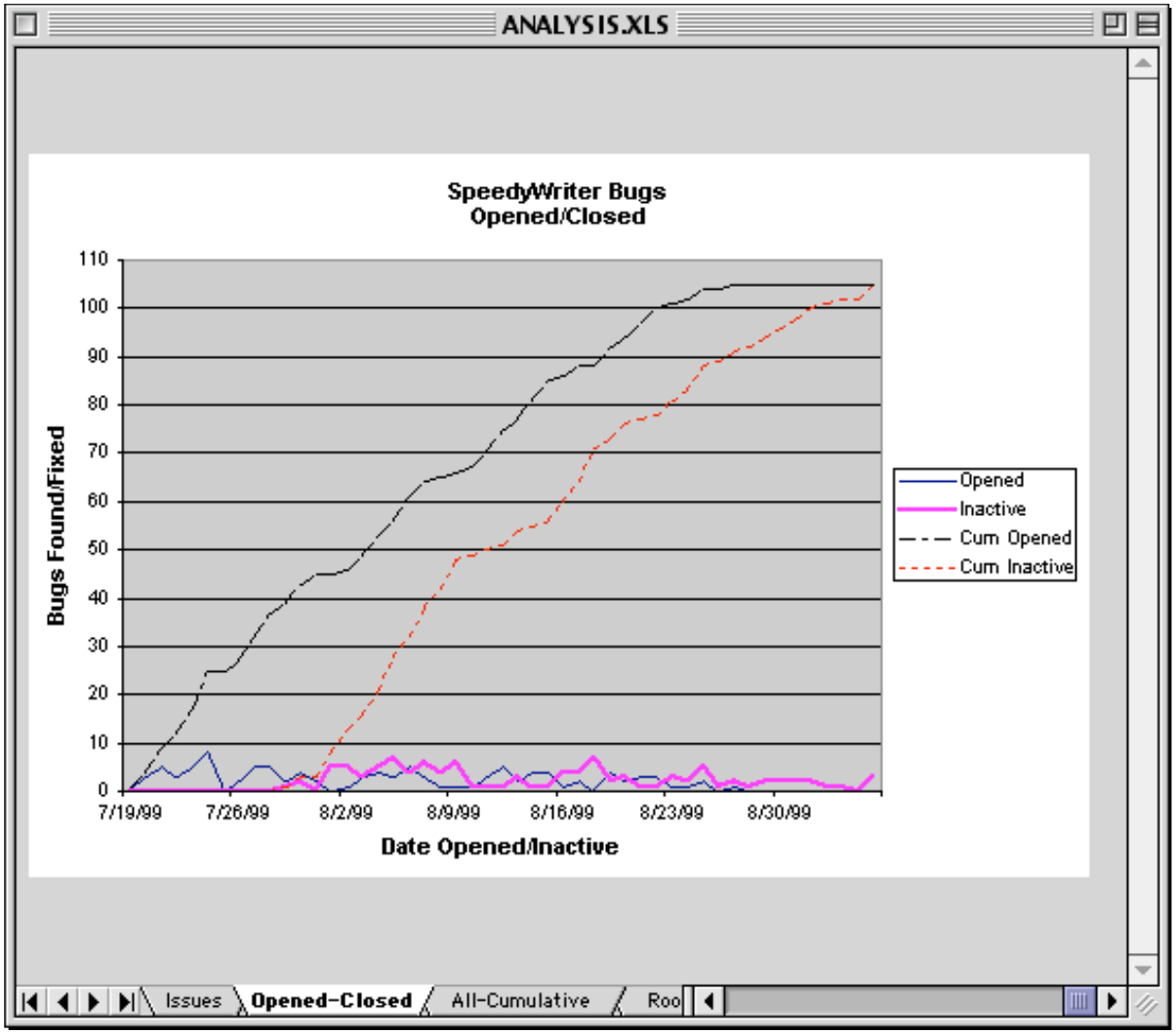
FAULT CONTROLLER: **DATE: 9/7/92**

Testing Issue Spreadsheet

ANALYSIS.XLS

	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	Date Open	Sev	Pric	Risk	Owner	Estimated	Su	Ste	Isc	St	Subsystem	Configuration	Close Date	Resolution	Root Cause	
2	7/21/99	1	1	1	Muhammad Zam	8/12/99	du	du	du	du	Edit Engine	A.X.0.0.001	8/3/99		Functional	
3	7/21/99	2	2	4	Chuck Chavall	7/28/99	du	du	du	du	User Interface	A.X.0.0.001	7/30/99		System	
4	7/24/99	3	3	9	Jenny Chung	8/12/99	du	du	du	du	Edit Engine	A.Y.0.0.001	8/1/99		Process	
5	7/23/99	3	3	9	Muhammad Zam	7/24/99	du	du	du	du	Edit Engine	A.Z.0.0.001	8/2/99		Data	
6	7/24/99	1	4	4	Chuck Chavall	7/31/99	du	du	du	du	User Interface	B.Y.0.0.001	8/1/99		Functional	
7	7/24/99	2	5	10	Jenny Chung	8/17/99	du	du	du	du	Tools	C.X.0.0.001	8/7/99		System	
8	7/23/99	3	3	9	Larry Kanemetsu	8/4/99	du	du	du	du	Edit Engine	C.Y.0.0.001	8/4/99		Process	
9	7/20/99	4	1	4	Frank Currant	8/14/99	du	du	du	du	User Interface	C.Z.0.0.001	8/5/99		Data	
10	7/22/99	5	1	5	Chuck Chavall	8/18/99	du	du	du	du	Edit Engine	D.Z.0.0.001	8/5/99		Functional	
11	7/22/99	1	1	1	Jenny Chung	7/28/99	du	du	du	du	User Interface	A.X.0.0.001	8/4/99		Documentation	
12	7/23/99	2	2	4	Muhammad Zam	8/17/99	du	du	du	du	User Interface	A.X.0.0.001	8/7/99		Functional	
13	7/23/99	1	3	3	Chuck Chavall	7/26/99	du	du	du	du	Edit Engine	A.X.0.0.001	8/3/99		System	
14	7/21/99	2	1	2	Jenny Chung	8/13/99	du	du	du	du	Other	A.X.0.0.001	8/5/99		Process	
15	7/23/99	1	2	2	Larry Kanemetsu	7/27/99	du	du	du	du	Unknown	A.X.0.0.001	8/7/99		Data	
16	7/24/99	2	3	6	Frank Currant	7/25/99	du	du	du	du	N/A	A.Y.0.0.001	8/7/99		System	
17	7/24/99	3	4	12	Chuck Chavall	8/13/99	du	du	du	du	Edit Engine	A.Z.0.0.001	8/2/99		Functional	
18	7/24/99	4	1	4	Jenny Chung	8/19/99	du	du	du	du	User Interface	B.Y.0.0.001	8/1/99		Functional	
19	7/21/99	1	2	2	Muhammad Zam	7/27/99	du	du	du	du	Tools	C.X.0.0.001	7/30/99		System	
20	7/19/99	1	3	3	Chuck Chavall	7/27/99	du	du	du	du	File	C.Y.0.0.001	8/1/99		Process	
21	7/22/99	2	4	8	Jenny Chung	8/18/99	du	du	du	du	User Interface	C.Z.0.0.001	8/1/99		Data	
22	7/21/99	3	5	15	Larry Kanemetsu	7/22/99	du	du	du	du	Edit Engine	D.Z.0.0.001	7/29/99		Process	
23	7/20/99	4	3	12	Muhammad Zam	8/13/99	du	du	du	du	User Interface	C.Y.0.0.001	8/4/99		Documentation	
24	7/20/99	1	2	2	Chuck Chavall	7/30/99	du	du	du	du	Edit Engine	C.Z.0.0.001	8/2/99		Functional	
25	7/24/99	4	3	12	Jenny Chung	8/12/99	du	du	du	du	User Interface	D.Z.0.0.001	8/5/99		System	
26	7/24/99	5	4	20	Larry Kanemetsu	8/8/99	du	du	du	du	User Interface	A.X.0.0.001	8/2/99		Process	
27	7/28/99	1	5	5	Frank Currant	8/22/99	du	du	du	du	Edit Engine	A.X.0.0.001	8/7/99		Data	
28	7/29/99	2	3	6	Chuck Chavall	8/20/99	du	du	du	du	Other	A.X.0.0.001	8/11/99		Code	
29	7/31/99	1	3	3	Jenny Chung	8/5/99	du	du	du	du	Unknown	A.X.0.0.001	8/9/99		Functional	
30	7/28/99	2	3	6	Larry Kanemetsu	8/17/99	du	du	du	du	Edit Engine	B.Y.0.0.001	8/8/99		System	
31	7/27/99	1	1	1	Muhammad Zam	8/1/99	du	du	du	du	User Interface	B.Y.0.0.001	8/10/99		Process	
32	7/30/99	2	2	4	Chuck Chavall	8/20/99	du	du	du	du	Tools	C.X.0.0.001	8/6/99		Data	
33	7/27/99	3	3	9	Jenny Chung	8/3/99	du	du	du	du	File	C.Y.0.0.001	8/5/99		Code	
34	7/26/99	4	4	16	Muhammad Zam	8/14/99	du	du	du	du	Install/Config	C.Z.0.0.001	8/2/99		System	
35	7/27/99	1	5	5	Chuck Chavall	8/12/99	du	du	du	du	Edit Engine	D.Z.0.0.001	8/4/99		Functional	

Open/Closed Bug Tracking



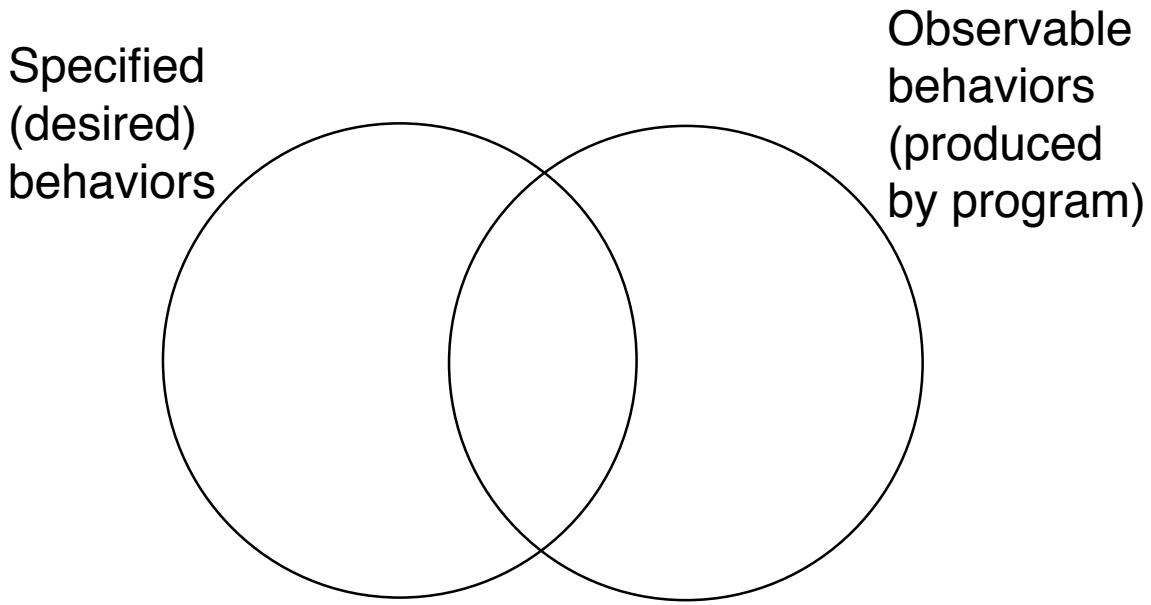
General Testing Approach

- Create lists of **potential** problems and categorize them by area.
- Design **repeatable** tests, for proof of problems and problem resolutions.

Broad Categories for Errors

- Errors in interpreting requirements
- Errors in translating requirements into design, i.e. in programming
- Errors in implementing design
- Errors in the testing process itself

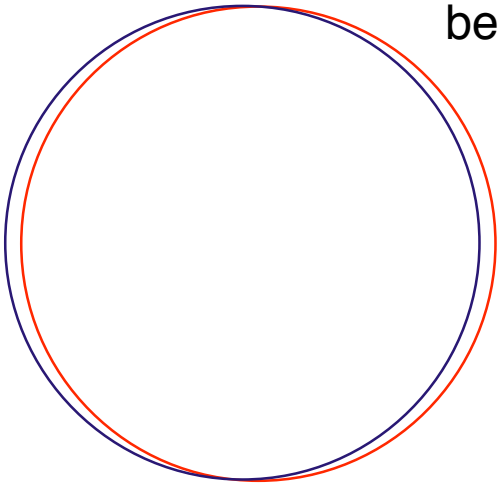
Program Behavior Views
(sets of behaviors taken over all inputs)



The Ideal

Specified
(desired)
behaviors

Observable
behaviors

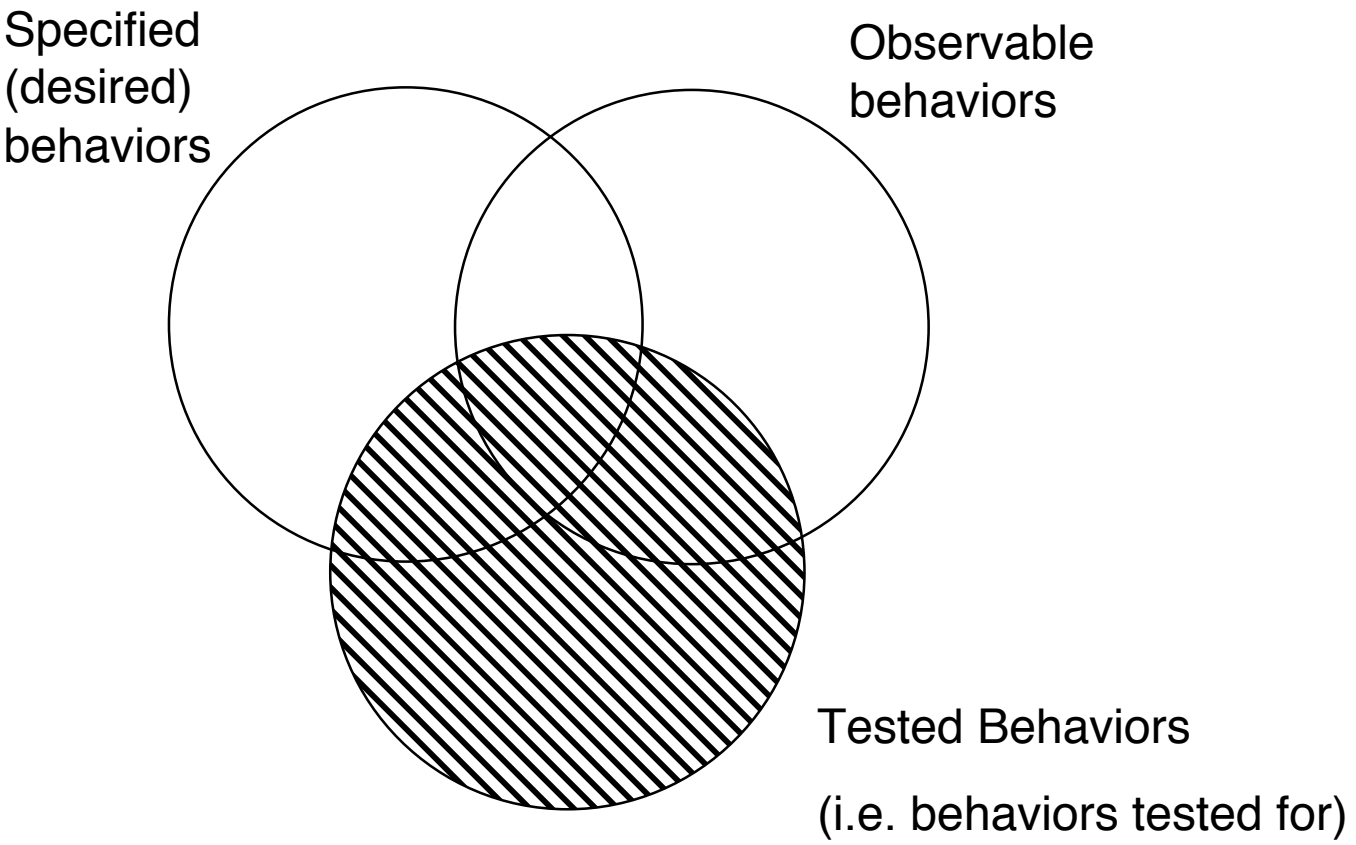


(coinciding)

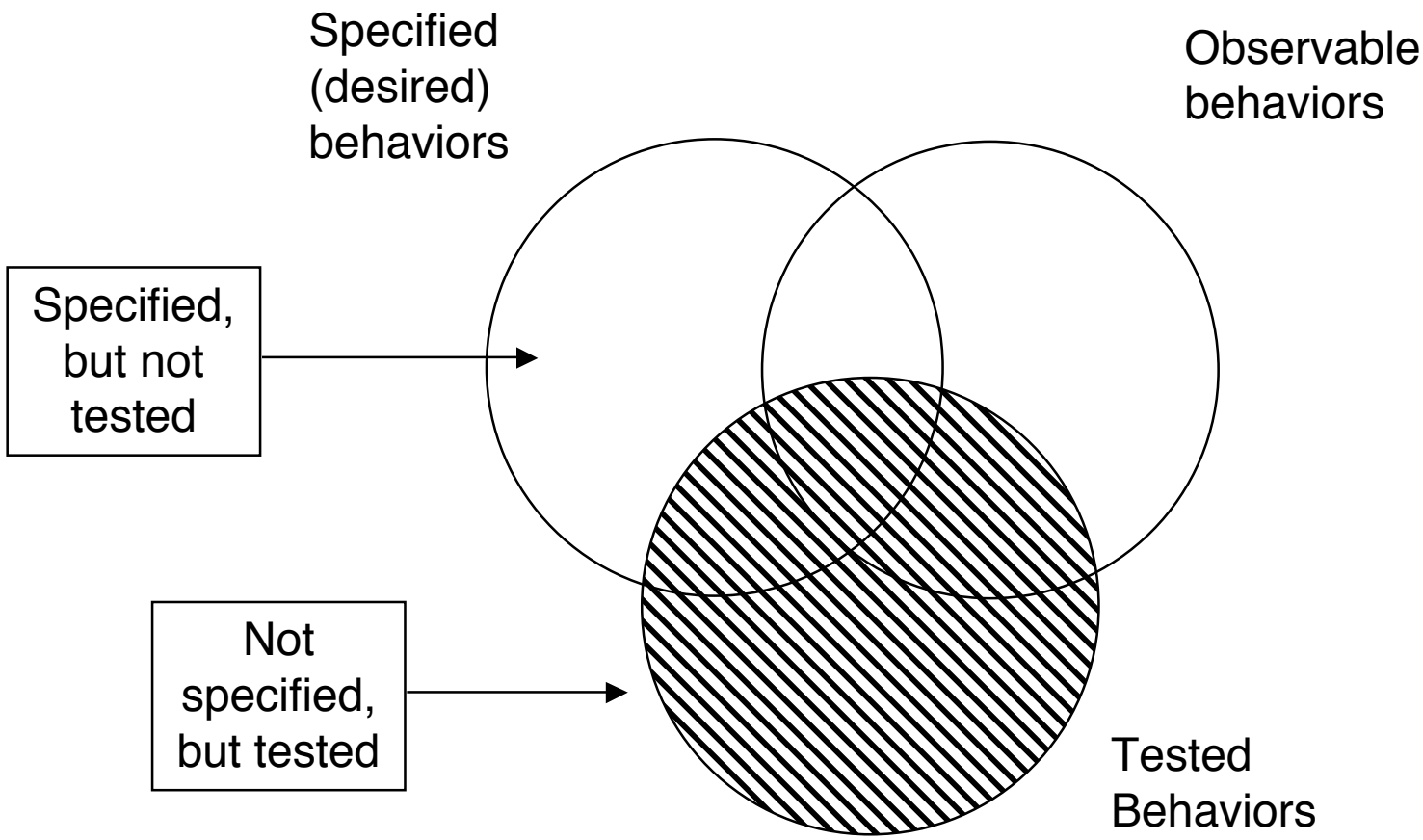
The ideal might not be fully realizable because

- Some aspects of a specification may be left arbitrary, unspecified,
- meaning either:
  - The specification is to be regarded as incomplete, or
  - *Any* behavior ***consistent with*** the specification will be accepted.

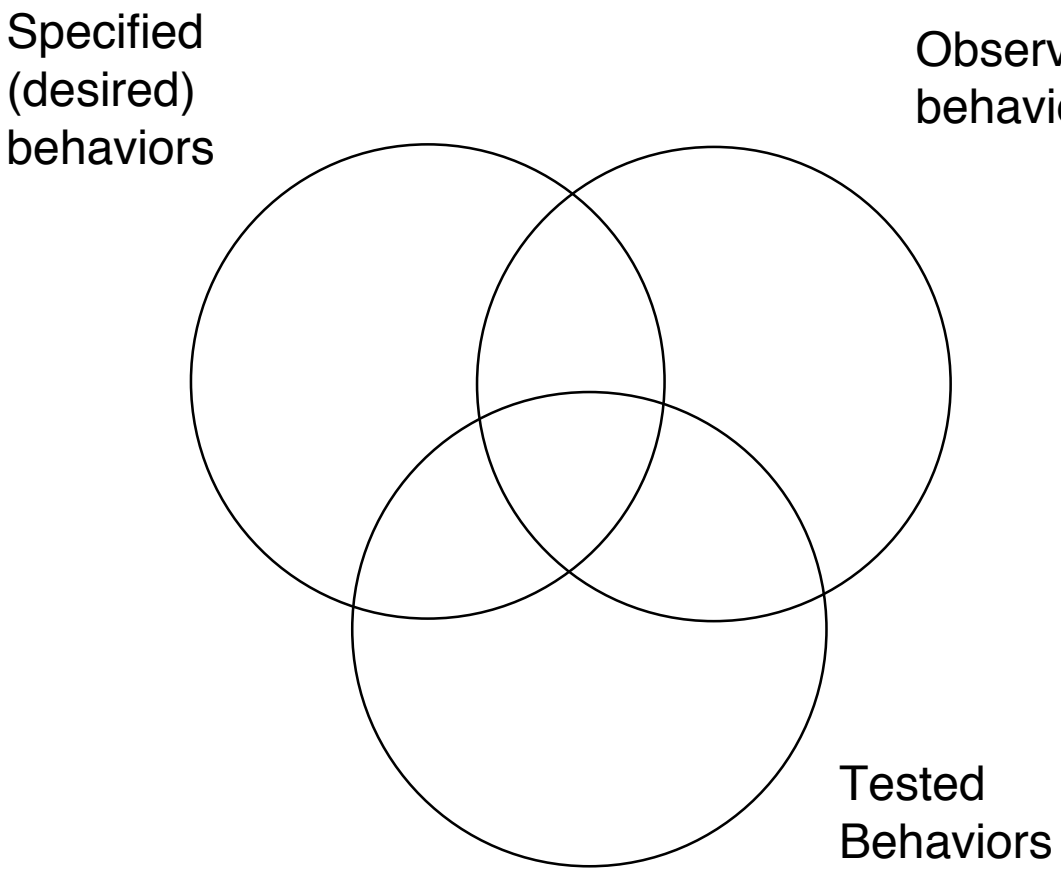
Testing asks questions:
Can a behavior occur?



Testing


Specified (desired) behaviors
Observable behaviors
Specified, but not tested
Not specified, but tested
Tested Behaviors

Which regions of the diagram indicate detected errors?

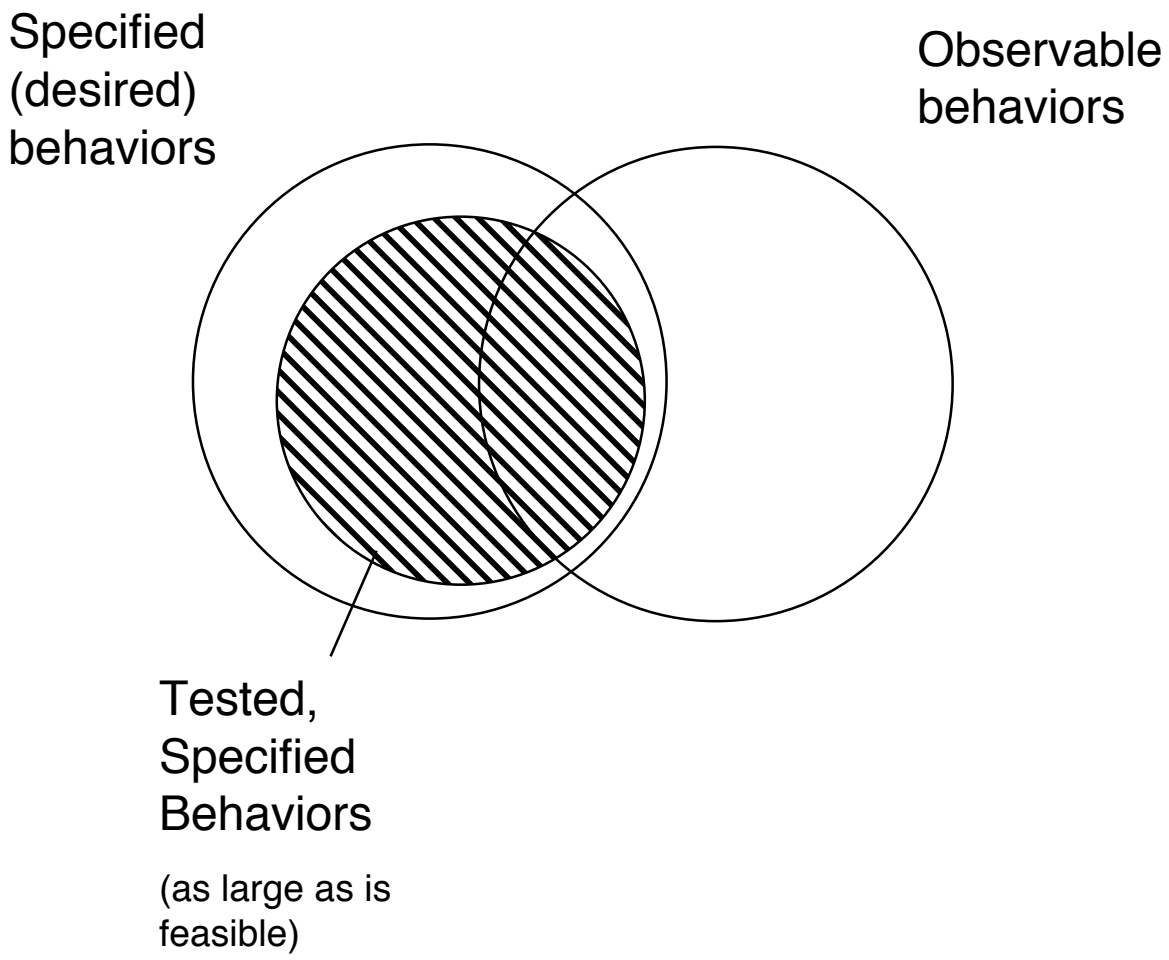


Black- vs. White Box
(recap)

- **Black-box** focuses on the **specification**: What is in the **spec** that the program doesn't do?
- **White-box** focuses on the **program**: What does the **program** do that is not in the spec?
- Normally don't rely on one or the other exclusively.

Black-Box Testing

Good Black-Box Test Plan



Testing Example (3)

- A third reputable programmer has produced a self-contained program "triangle" that reads triples of numbers at a time and classifies them as to whether they are the lengths of the sides of some triangle, and if so, what kind.
- Negative side-length counts as a side with the length as absolute value of the specified length.
- The sides are in a specified range between 1E-150 and 1E150.

Black-Box Techniques

- Recall that "black box" means we do not get to see the code; we only have access to an installation of the product.
- Also called "functional testing" (vs. "structural testing", which would be "white box")
- Driven by requirements, use cases

Black-Box Techniques

- **Comparison Testing**:

  Test product side-by-side with a **"gold standard"**,

  a program believed to be correctly operating (such as an earlier version having most, if not all, of the features)

Black-Box Techniques

- **Garbage-In Test**: See if unusual input characters, click sequences, etc. can force the system into inconsistent states.
  - **Open-Book Test**
- **Test Monkey**: Provide large sequence of random inputs, with hopes of eliciting a crash of the software.
- **Data-Quantity Stress Test**: See if unusually large amounts of data cause nominal values to be exceeded, revealing untested overflow conditions, etc.
- **Soak Test**: Test the stability and "up-time" of integrated applications by running the program for a long time under varying conditions.

Quantitative Black-Box Testing

Equivalence Partitioning

- Partition the input space into classes, such that one may reasonably assume that:
  - The program behaves analogously for inputs in the same class.
  - A single test with a representative value from a class is sufficient to test for the entire class.
  - If representative detects fault then other class members will detect the same fault.

Black-Box Techniques

- **Equivalence Partitioning**:
  - Use a small number of test **equivalence classes**, rather than a large number of individual test data points.
  - The actual tests are representatives of the equivalence classes.
  - Partitioning based on their relative likelihood of exposing logic errors in the code.
  - Example (application-dependent): Partition a number space into:
    - less than 0
    - equal to 0
    - greater than 0, less than 100
    - greater than or equal to 100

Black-Box Techniques

- Equivalence Partitioning Examples (cont'd):
  - Partition a **two-dimensional** number space into (x, y) where:
    - x < y
    - x == y
    - x > y
  - Why these?

  - Partition a String space into
    - length = 0
    - length = 1
    - length = 2
    - length = 3
    - length between 4 and 1023
    - length greater than 1023

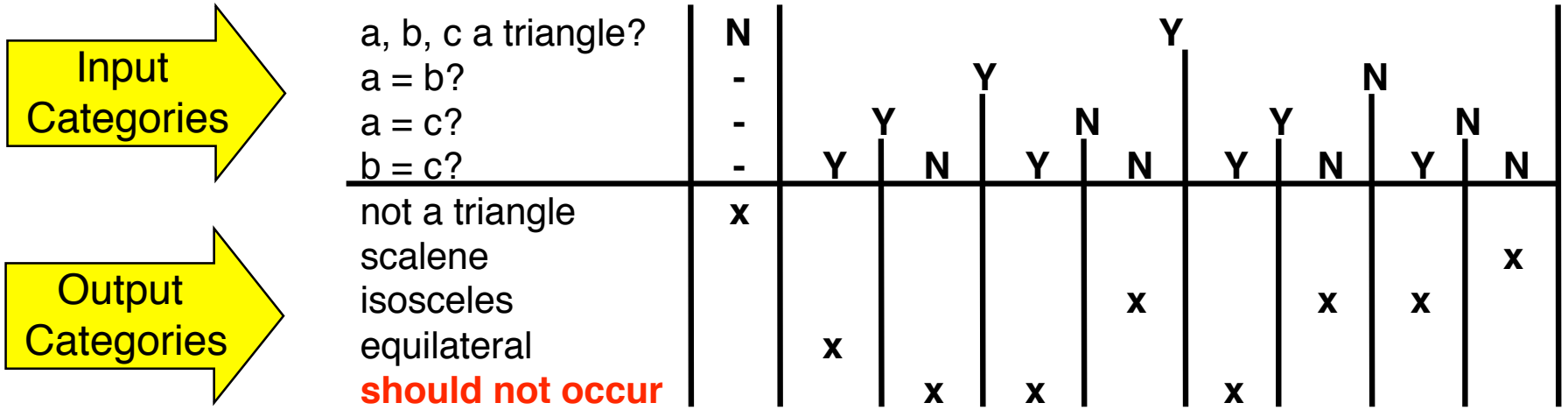
Black-Box Techniques

- How would you equivalence-partition the triangle program input space?
- What is the rationale?

Decision Table

A "declarative" means to categorize input

- Partitions input space (triples of numbers) into equivalence classes.
- All inputs in a cell should have the same anticipated equivalence class.



Decision Table Variant

- More condensed, based on logical equivalences
- Eliminate or reduce "should not occur" entries

Input Categories

Output Categories

a, b, c a triangle?	N	Y					
a = b?	-	Y		N			
a = c?	-	Y	N	Y	N		
b = c?	-	Y	N	N	Y	N	
not a triangle	x						
scalene						x	
isosceles			x	x	x		
equilateral		x					

Black-Box Techniques

- **Boundary-value testing**: Pick test cases on, and near to, "natural" boundaries in data space, so as to test whether the program performs the correct classification of borderline cases.

Black-Box Techniques

- **Logarithmic testing**: Repeatedly split the data space into two, testing sample points in the upper and lower halves of the split, then repeat on the lower half only.

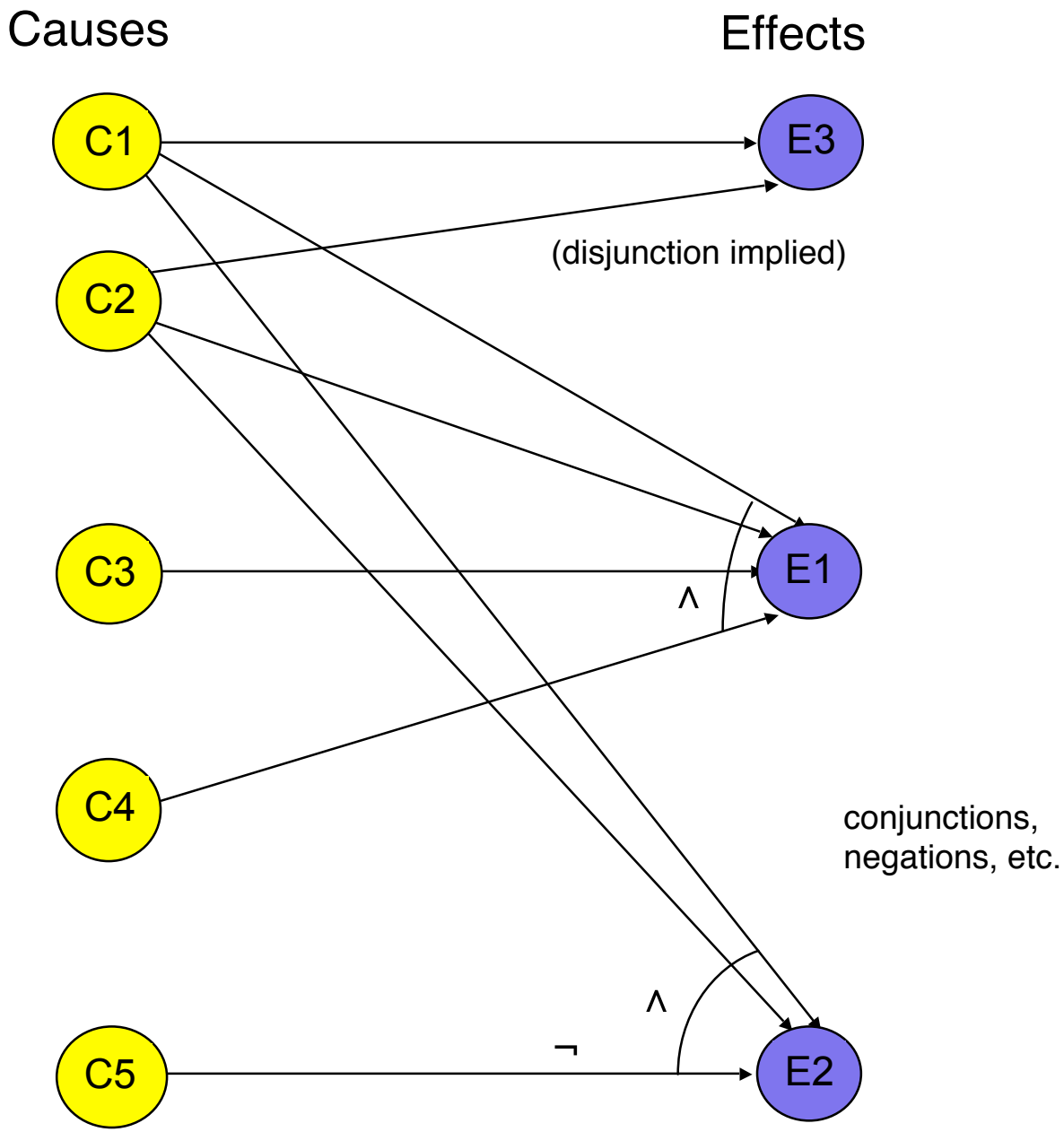
Black-Box Techniques

- **Cause-Effect Graphing**: Examine requirements specification for logical **chains of conditions**; develop test cases that check whether these chains are actually observed.
  - Example: "If the clipboard is empty, then the paste menu option should *not* be selectable."
  - Therefore: Develop a test in which the clipboard should be empty, and check that the menu option is not selectable.

Black-Box Techniques

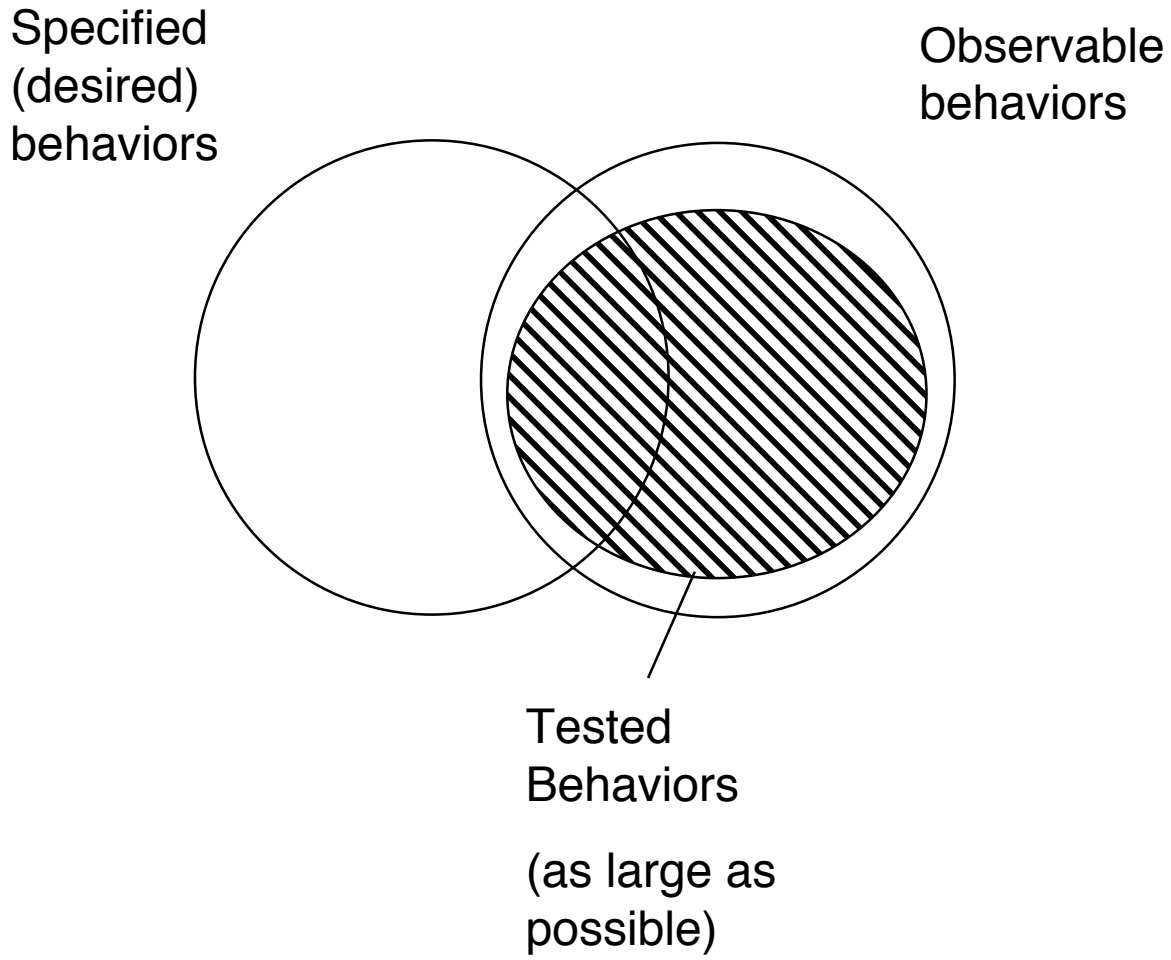
- **Cause-Effect Graphing**, possible relationships:
  - A condition *implies* an action
  - A condition *precludes* an action
  - Two actions are *mutually* exclusive
  - A *combination* (conjunction) of two conditions implies an action
  - etc.

Example Cause-Effect Graph



White-Box Testing

Good White-Box Test Plan



Approaches

- Qualitative/Empirical
- Structural: Things to make code more testable
- Quantitative: Mathematical analysis

Code Inspection and Walkthrough

- Used for:
  - Identifying errors
  - Identifying test cases
  - Helping yourself and others to understand how the program works.

Categories of Programming Errors

- Logic errors
- Pointer errors
- Numeric accuracy/precision/roundoff errors
- Input/output representation errors
- Data structure usage errors
- Memory errors
- User-interface errors (windows, etc.)
- Environment errors, such as misuse of file-system, devices, etc.
- Timing errors, such as in a real-time system

Exercise

- Each team take one category of error.
- List as many specific sub-categories of this error as you can.
- Are there testing approaches specific to this sub-category of error?
- **Example**: Logic Errors:

Logic Error Categories (1)

- Numeric and character boundary
  - Off-by-1
- Array-out-of-bound
  - Insufficient space allocated
  - Input or output buffer overflow
- Inequality comparison (used > instead of >= or <)
- Used ++ instead of --.
- Used pre-incrementation rather than post.

Logic Error Categories (2)

- Negation error
- Loop continuation criterion
  - Infinite loops
- Flag not cleared when used
- Flag, count, or sum not initialized
- Routine not reinitialized before subsequent use
- Dynamic type-casting exception

White-Box in Conjunction with Verification Techniques

- Verification is based on techniques for reasoning about programs.
- These techniques can also be used to simplify the number of white-box test cases.

Using Live Assertions

- Example from Triangles program:

```
order(a, b);
order(b, c);

assert( a <= b && b <= c );
```

- The program will now tell us when the assumption is wrong.
- It will provide an indication of what has to be rethought.

Verification + W.B. Testing

- Could use verification to help fix the problem.
- Having *verified* the assertion

```
assert( a <= b && b <= c );
```

  will testing be simplified?

- How?

Some Ideas for More Effective Testability

Design for Test (DFT)

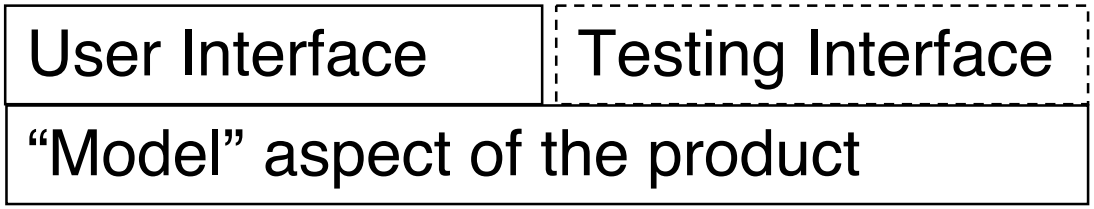
- **Instrument** your code as you build it; this could help with unit tests
  - Code-in traces, explanations, indications, ...
  - Being able to turn instrumentation on or off can help understand whether sub-systems are working correctly.

Example: C++ Instrumentation

```
class Trace
{
static int currentLevel;

static void trace(string message, int level)
     {
     if( level >= currentLevel )
             cerr << message << endl;
     }

static void setLevel(int level)
     {
     currentLevel = level;
     }
};
```

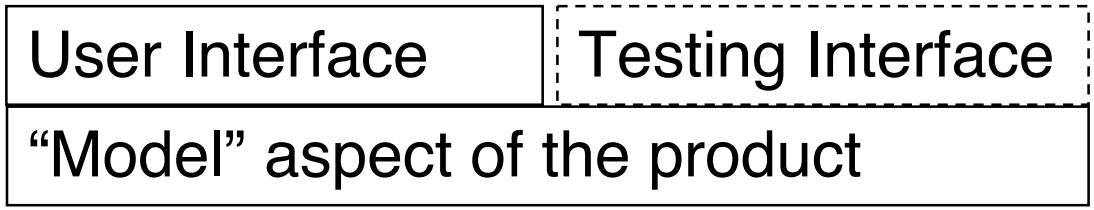
Design for Test
(used extensively in hardware domain)

- Build **testing interfaces** into the code. These are interfaces that can be seen by the developer but not the user.



- These interfaces allow greater automation in the testing process, since they can be driven by a testing program more readily than one requiring a user interface (such as a GUI or CLI).

Testing Interface



Build Self-Test Routines into the Code

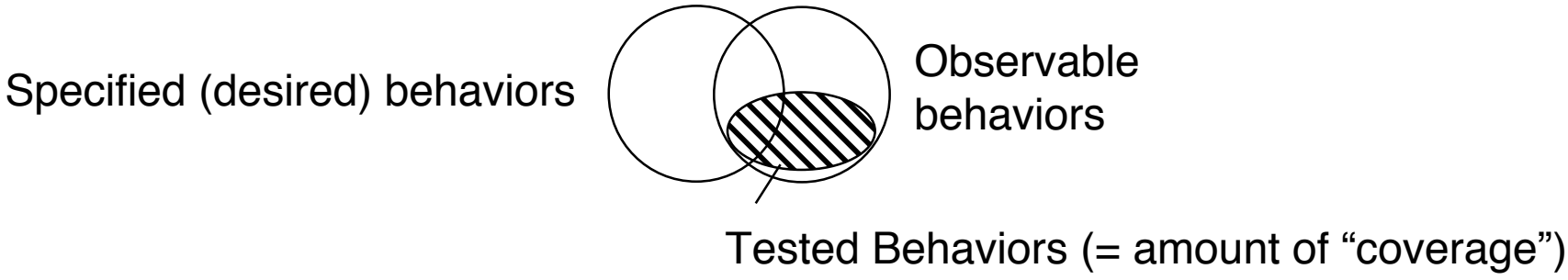
- **Self-tests** can employ data generators that will stress test the code.

Quantitative Aspects of Whitebox Testing

Coverage Analysis

White-Box Coverage

- **"Coverage" notion**:
  - View the program as a directed graph (flowchart or data-flow diagram)
  - Develop (external or internal) tests that "cover", i.e. exercise program to various degrees.



Pairwise Testing
(aka "All Pairs" Testing)

[Applies to BB and WB.]

Combinatorial Explosion

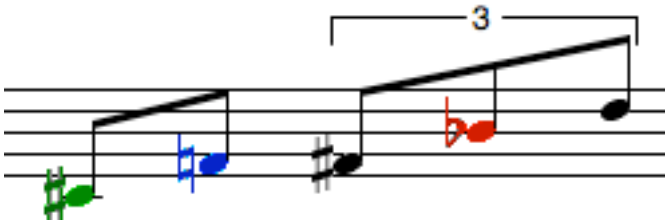
- If there are N different attributes, with value set sizes S_1, S_2, …, S_N, then the number of combinations is the product:

$$S_1 S_2 \ldots S_N$$

- Assuming $S_i \geq 2$ for each i, we have a number of combinations exponential in N.

Example from Impro-Visor

- Note sequence display:
  - Note value (1, 2, 4, 8, 16, 32)
  - Dotted or not
  - #, b, natural, no accidental
  - Singlets, Triplets, 5-Tuplets
  - Time Signature (4 vs. 3)
  - Coloration or not
  - Clef (bass, treble, grand)
  - Ledger lines or not
  - Beamed or not
  - Tied or not



- Combinations = 6 x 2 x 4 x 3 x 2 x 2 x 3 x 2 x 2 x 2 =

Example: Web media serving

- 5 different browsers {IE, Firefox, Netscape, Opera, Safari} (but some have multiple versions
- 2 different media players {MediaPlayer, Real}
- 3 different webservers {IIS, Apache, Weblogic}
- 3 different operating systems {Widows, Apple, Linux}
- At least 5 x 2 x 3 x 3 = 80 combinations
  (not considering multiple release versions)

Pairwise Testing

- Pairwise testing is a **heuristic** for achieving good test coverage *without* trying all combinations:

- Select a set of combinations such that:
  - For any pair of attributes, there is a combination that includes that pair.

Example

- Combinations = {1, 2} x {3, 4}
- Every combination is needed

1	3
1	4
2	3
2	4

Example

- Combinations = {1, 2} x {3, 4} x {5, 6}
- 4 of 8 possible suffice for pairwise testing

1	3	5
1	4	6
2	3	6
2	4	5

Does this alternate work?

- Combinations = {1, 2} x {3, 4} x {5, 6}

1	3	5
1	4	5
2	3	6
2	4	6

Is this heuristic effective?

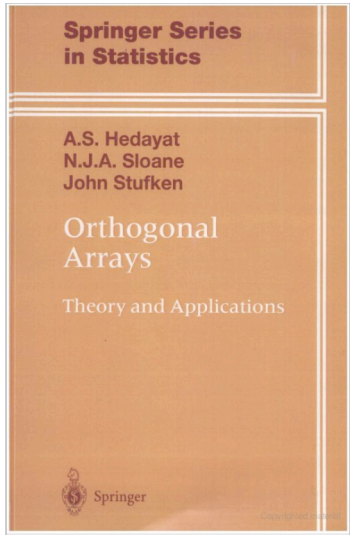
- **Empirical examples from: http://www.pairwise.org/results.asp**

> [...] a set of 29 pair-wise AETG tests gave 90% block coverage for the UNIX sort command. We also compared pair-wise testing with random input testing and found that pair-wise testing gave better coverage.
>
> [D. M. Cohen et al., 1997]

> Our initial trial of this was on a subset Nortel's internal e-mail system where we able cover 97% of branches with less than 100 valid and invalid testcases, as opposed to 27 trillion exhaustive testcases.
>
> [K. Burr and W. Young, 1998]

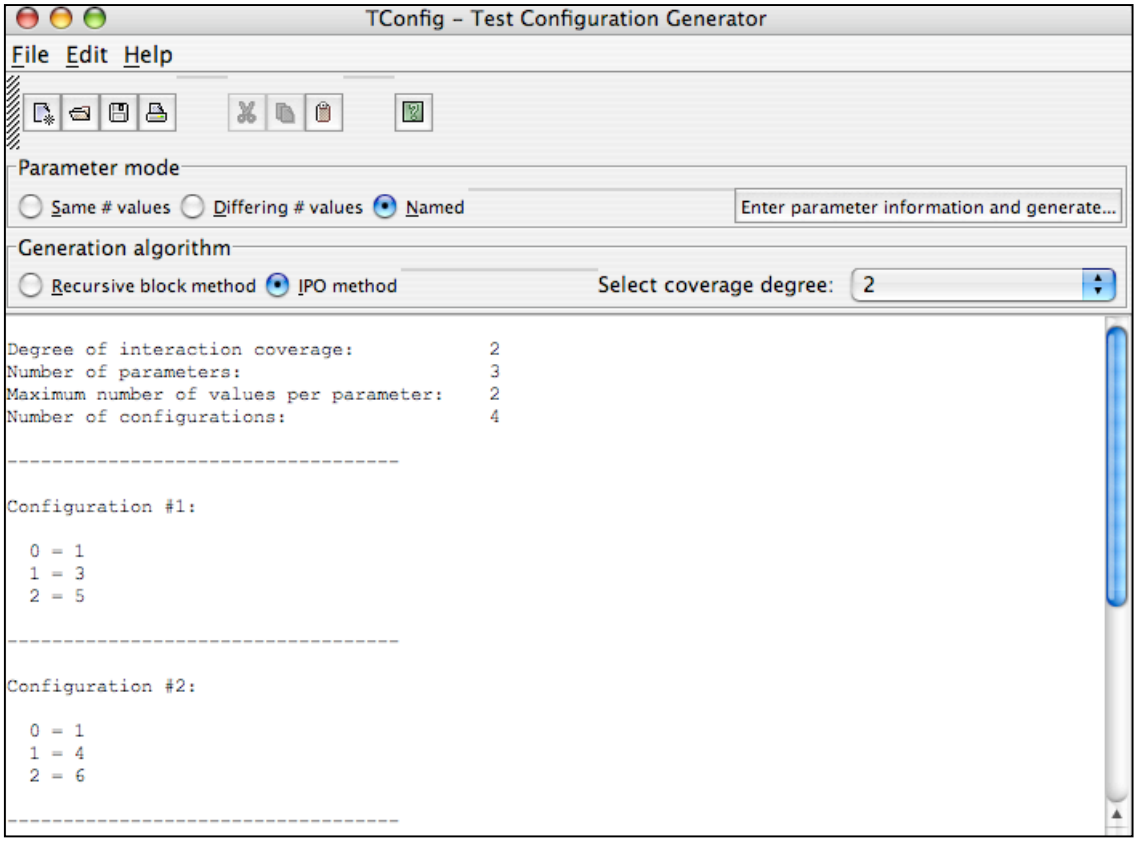
Checking for all-pairs coverage

- It is easy to check whether a proposed set of combinations covers all pairwise combinations.
- Generating such a set, while keeping the number of combinations to a minimum, is another matter.
- General area is "Orthogonal Arrays" (distinct from orthogonal matrices).



See also: http://www.research.att.com/~njas/oadir/

TConfig tool: Alan Williams (http://www.site.uottawa.ca/~awilliam/)



Concept extends to N-tuple-wise

More than 70% of bugs were detected with two or fewer conditions (75% for browser and 70% for server) and approximately 90% of the bugs reported were detected with three or fewer conditions (95% for browser and 89% for server). [...] It is interesting that a small number of conditions (n<=6) are sufficient to detect all reported errors for the browser and server software.

[R. Kuhn and M. J. Reilly, 2002]

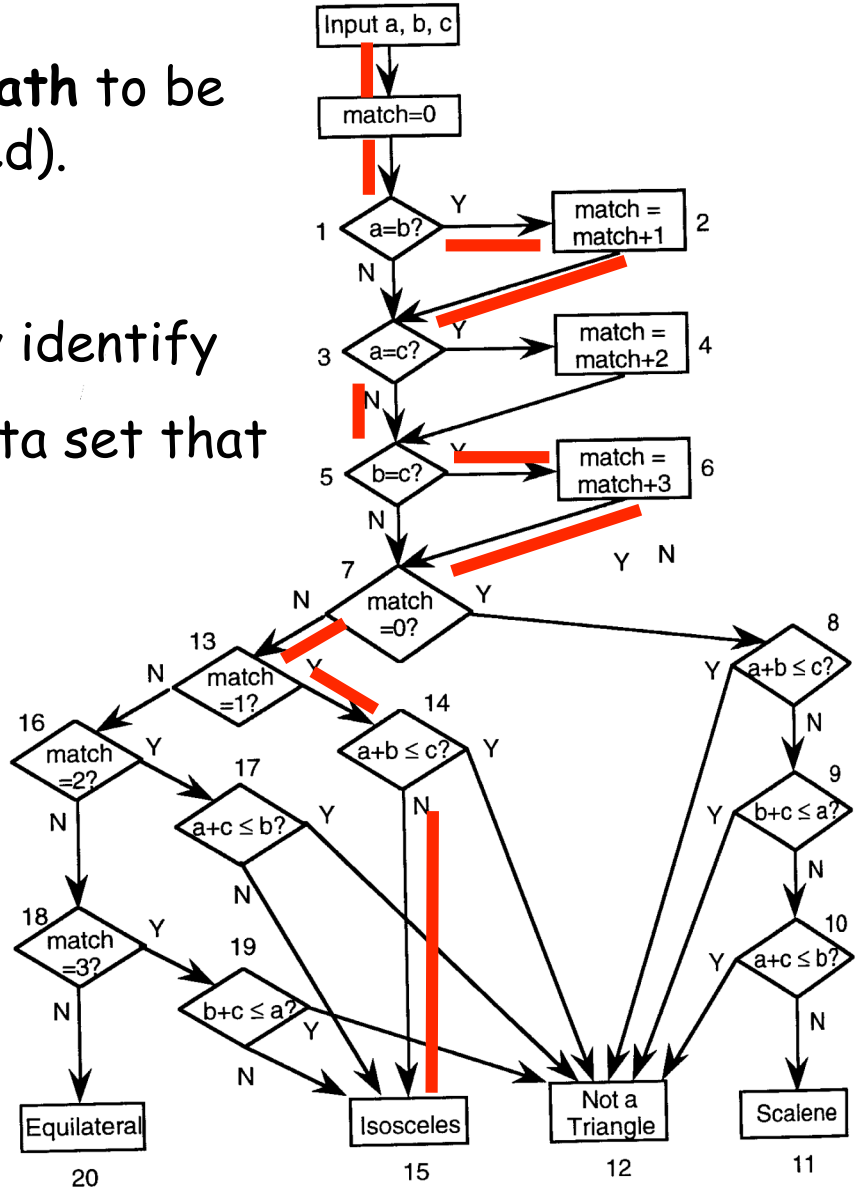
Minimal Coverage

- Tests should exercise, at least once, every:
  - variable
  - assignment box
  - decision box
  - simple path through flow chart

Testing Based on Paths

Example of **a single path** to be covered by test (in red).

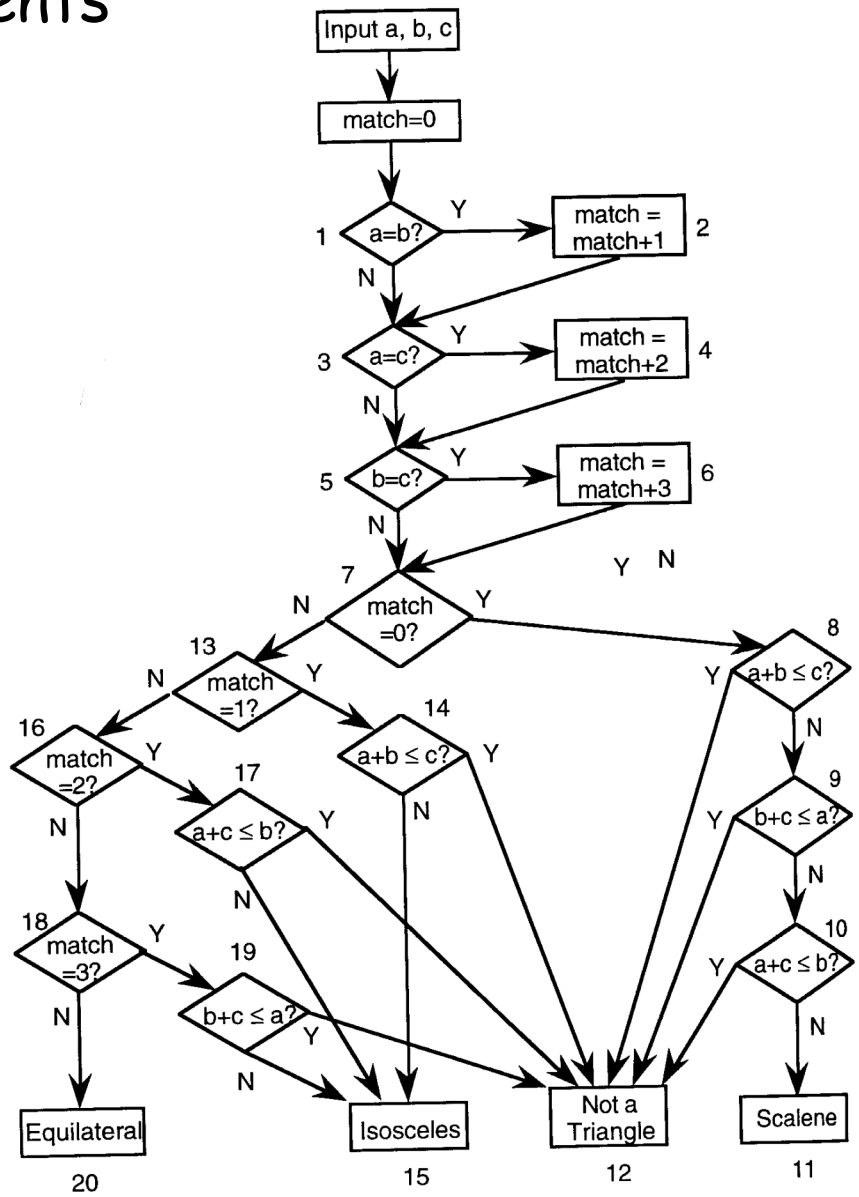
Insufficient to simply identify path; need to have data set that will exercise it.



Coverage "Basis"

- Infeasible to test ***all*** paths, etc.
- Instead, identify a "**basis**" from which all paths can be constructed.
- Make sure every element of basis is covered by ***some*** test
- **Example**: Basis could be set of all edges; Compute a set of tests that covers all.

Estimate the number of elements in a basis set of paths that will cover all edges.



D-D Path Nomenclature (Ed Miller 1977)

- D-D = "Decision to Decision"
- Path between two decisions, that contains no decision itself
- *Dependence* among D-D paths, e.g. a variable defined in one path is referenced in another.

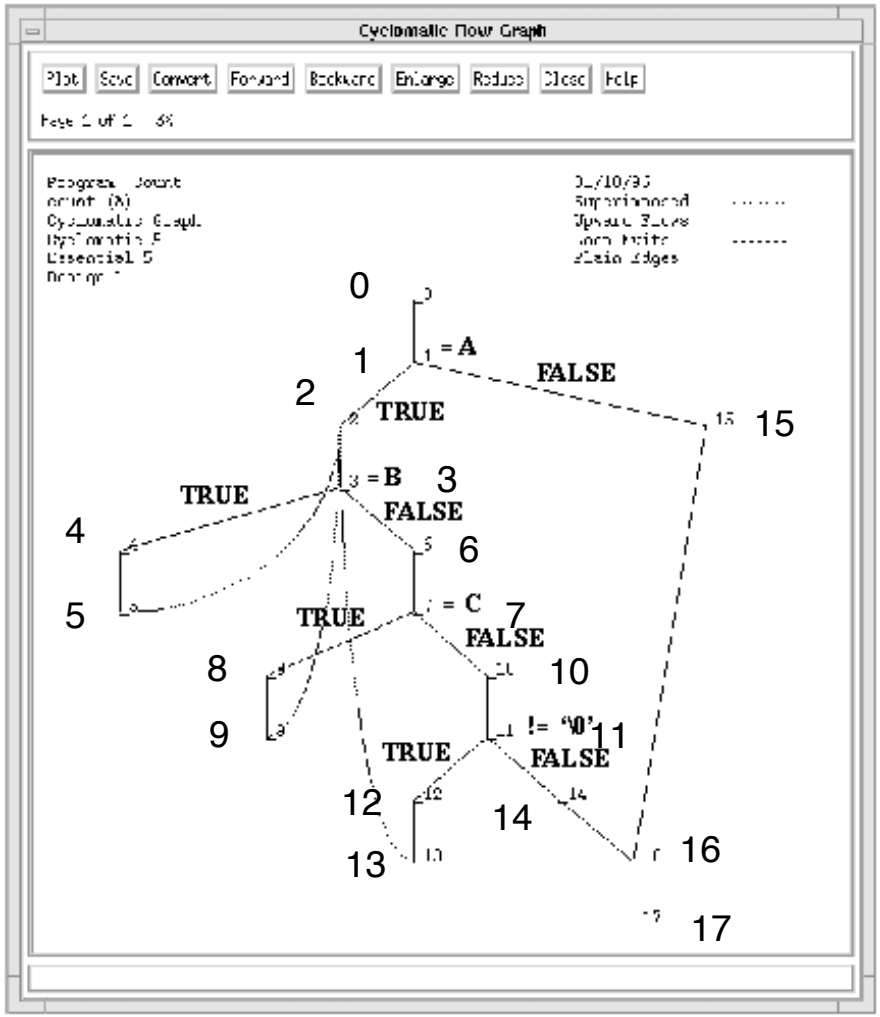
Classification of Structural Test Coverage

- C_0 Every statement tested
- C_1 Every D-D path tested
- C_1p Every **p**redicate & outcome tested
- C_d C_1 + every *inter-dependent pair* of D-D paths tested
- C_i^k Every path that contains up to k repetitions of a loop (e.g. k = 2) tested
- C_∞ Every path tested

"Baseline"-based method for constructing a basis

- Pick a single linear path through the program, the "baseline".
- Pick the next path by taking decisions alternate to the baseline.
- Repeat picking other alternates to the alternates, etc.
- Eventually a basis number of tests will be reached.

Baseline Testing Method (in McCabe's Cyclomatic Tool)



Test Path 1 (baseline): 0 1 2 3 4 5 2 3 6 7 10 11 14 16 17

11(1): string[index]=='A' ==> TRUE

13(3): string[index]=='B' ==> TRUE

13(3): string[index]=='B' ==> FALSE

18(7): string[index]=='C' ==> FALSE

25(11): string[index]!='\0' ==> FALSE

Test Path 2: 0 1 15 16 17

11(1): string[index]=='A' ==> FALSE

Test Path 3: 0 1 2 3 6 7 10 11 14 16 17

11(1): string[index]=='A' ==> TRUE

13(3): string[index]=='B' ==> FALSE

18(7): string[index]=='C' ==> FALSE

25(11): string[index]!='\0' ==> FALSE

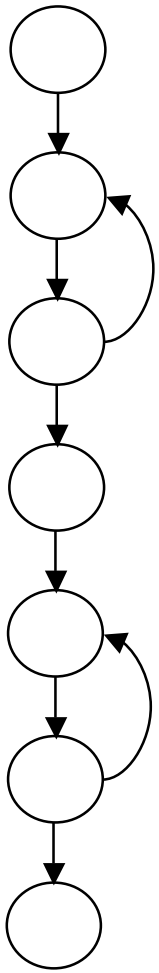
Test Path 4: 0 1 2 3 4 5 2 3 6 7 8 9 2 3 6 7 10 11 14 16 17

Complexity-Based Testing

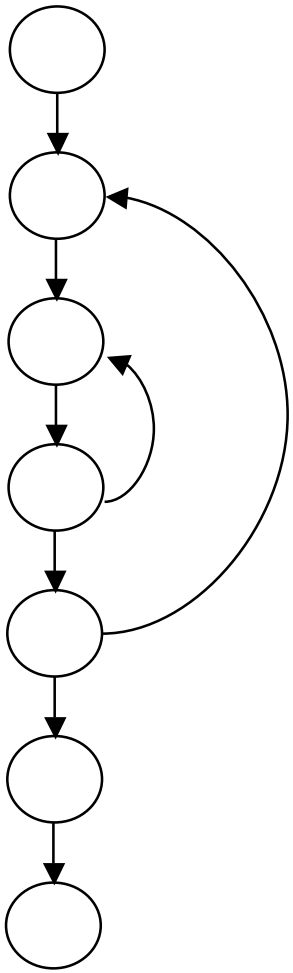
- Construct the flowchart for the program.
- Calculate the graph complexity *C*.
- **Find *C* independent paths** and corresponding test data for each.
- Execute program on test data.
- Check results with what is expected.

Classifying Loop Complexity (Informal)

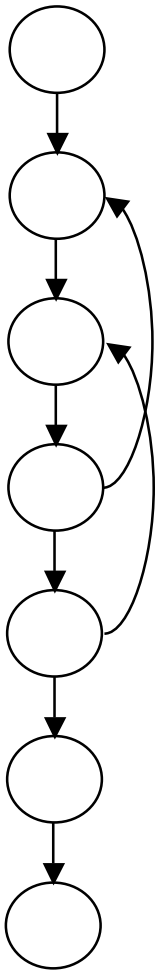
Concatenated



Nested



Intersecting



Program Graph Complexity Measures

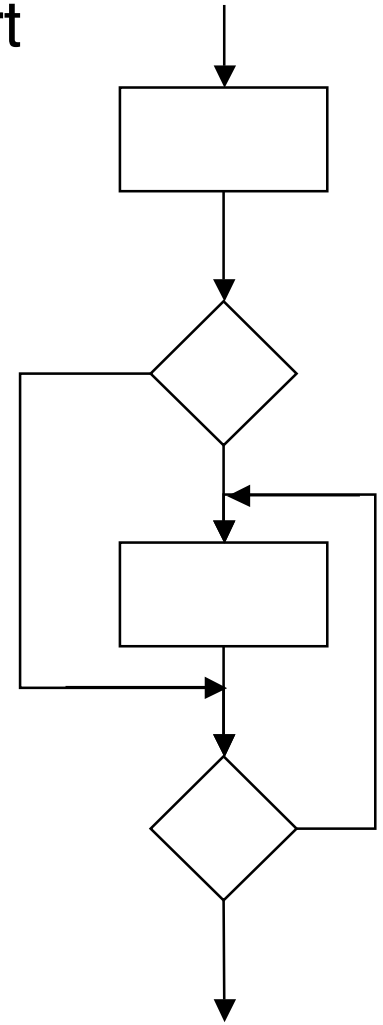
- Used to determine ***how much*** testing required for a given sub-system
- Examples
  - McCabe Cyclomatic complexity
  - Halstead software metric

McCabe Cyclomatic complexity for a program graph

- IEEE Trans. Softw. Engrg., Dec. 1976
- Popular, but theoretically questionable
- $\nu(G) = e - n + p$
  - e is the number of **edges** (arcs), which represent data processing **boxes**
  - n is the number of **nodes**, which represent **points** where edges are connected
  - p is the number of **separate parts** (connected sub-graphs)

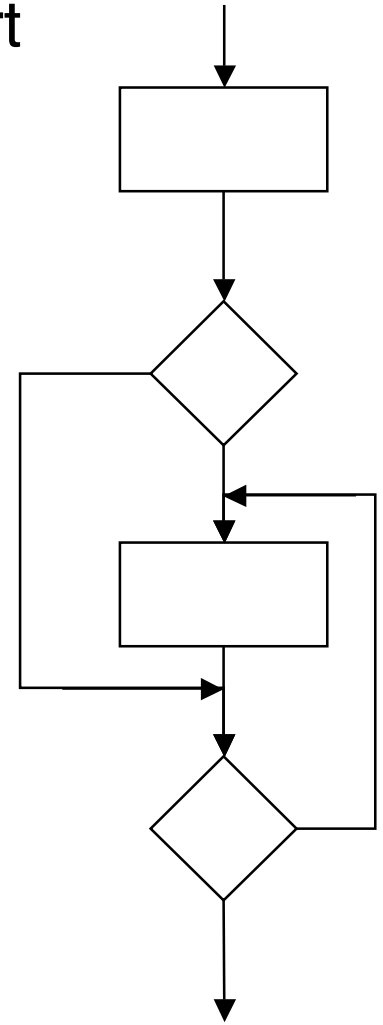
Example

Flowchart

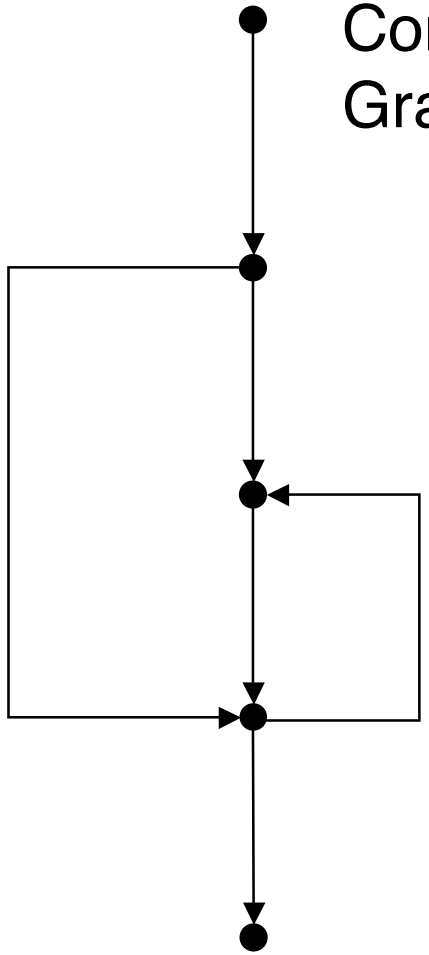


Example

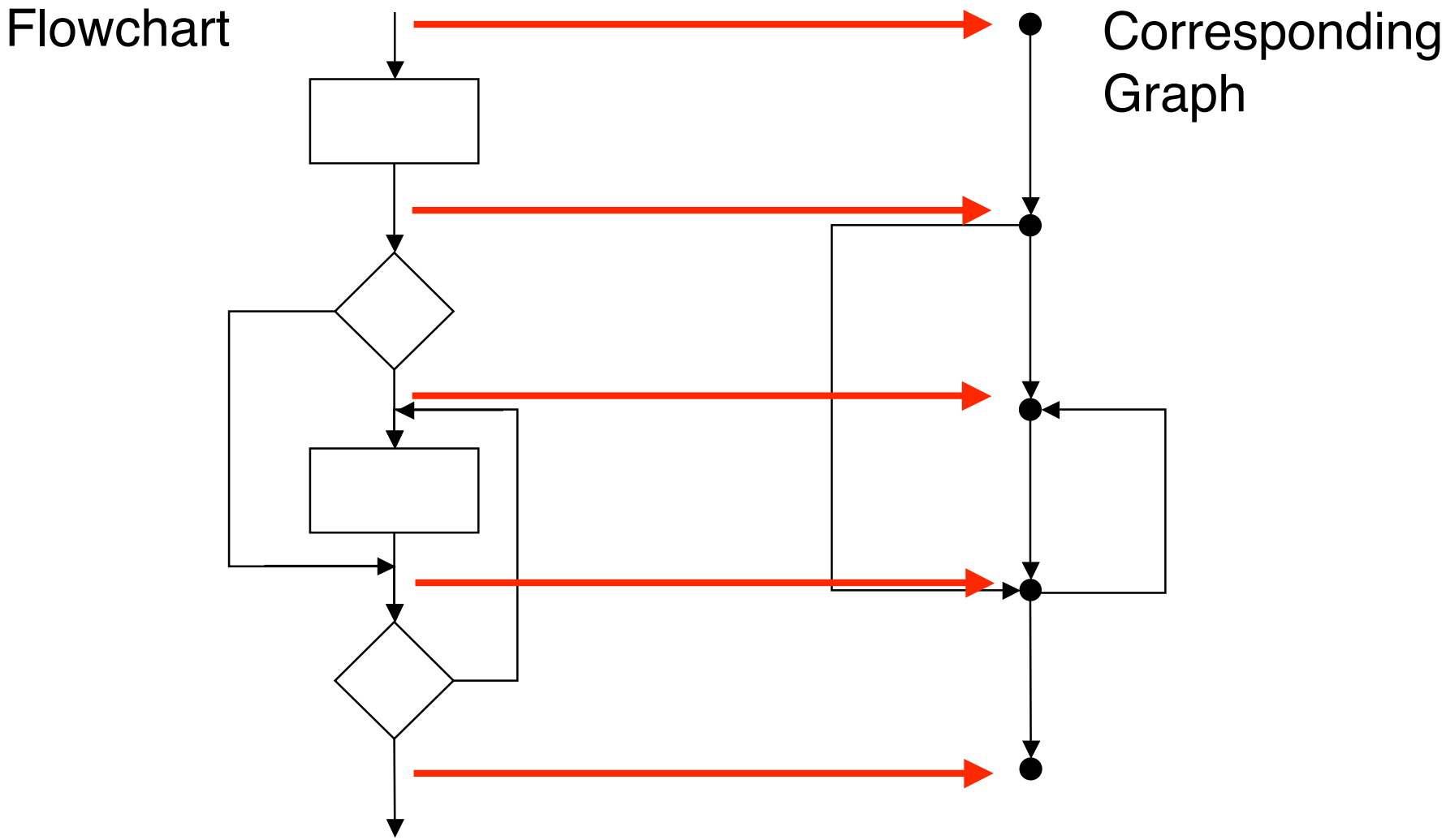
Flowchart



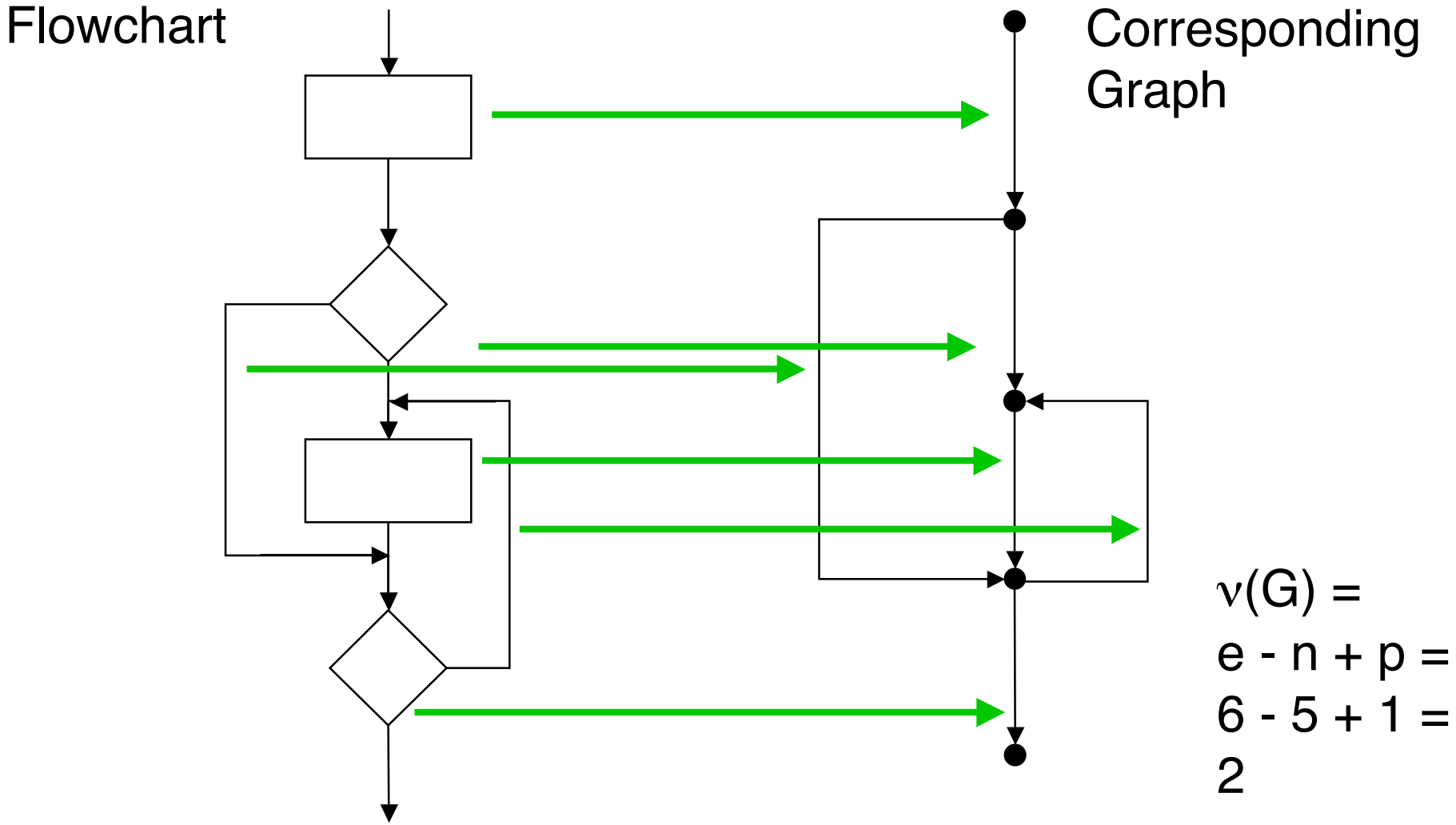
Corresponding Graph



Node Correspondence



Edge Correspondence



McCabe's measure drawn from Cyclomatic Number of a Graph

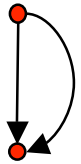
- Defined by Berge (*Theory of graphs and its applications*, 1962), motivated by Euler
- Originally defined for ***undirected*** graphs
- $\nu(G)$ = edges - nodes + separate parts
- **Property**: *Connecting* two nodes increases ν value by 1 if there was a path between the two nodes; otherwise it leaves ν value the same.

Illustration of Property

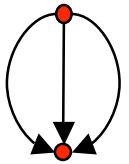
Connecting two nodes increases ν value by 1 if there was a path between the two nodes; otherwise leaves ν value the same.

ν= 0 - 2 + 2 = 0

ν= 1 - 2 + 1 = 0

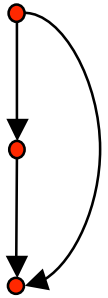


ν= 2 - 2 + 1 = 1

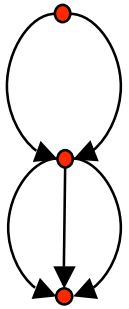


ν= 3 - 2 + 1 = 2

ν= 2 - 3 + 1 = 0



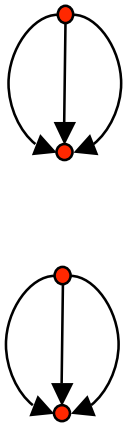
ν= 3 - 3 + 1 = 1

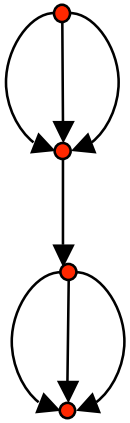


ν= 5 - 3 + 1 = 3

Illustration of Property

Connecting two nodes increases ν value by 1 if there was a path between the two nodes; otherwise leaves ν value the same.





$\nu = 6 - 4 + 2 = 4$

$\nu = 7 - 4 + 1 = 4$

Further Properties of the Cyclomatic *Number* of a Graph

- $\nu(G) = 0$ iff G contains no ***undirected*** cycles (i.e. is "straight-line").

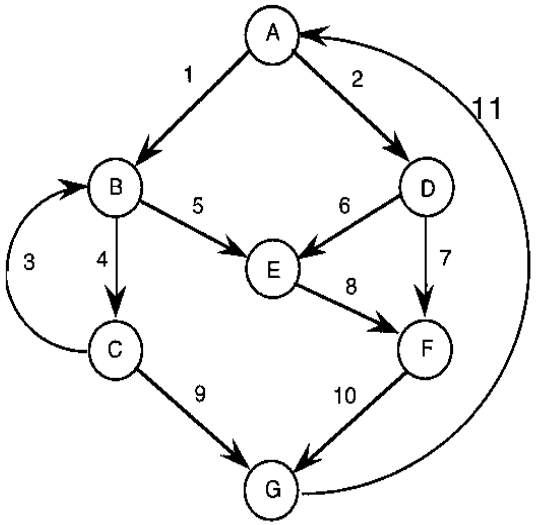
- If G is strongly connected (has only one component), then $\nu(G)$ is the maximum number of linearly- independent undirected cycles.

  [This can be used to compute size of a basis.]

Linear Independent Circuits?

- Think of the edges of a graph as components of a **vector**.
- A circuit is abstracted by the number of times each edge is traversed in the (undirected) circuit.
- One circuit can be constructed from others by *adding* (mod 2) the corresponding vectors.
- (But not all vectors so-constructed correspond to executable paths, or even circuits themselves.)

Adding Circuits



c1 = 1-4-9-11 = (1 0 0 1 0 0 0 0 1 0 1)

c2 = 2-7-10-11 = (0 1 0 0 0 0 1 0 0 1 1)

c3 = 1-4-9-10-7-2 = (1 1 0 1 0 0 1 0 1 1 0)

$\nu(G) = e - n + p$

$= 11 - 7 + 1$

$= \mathbf{5}$

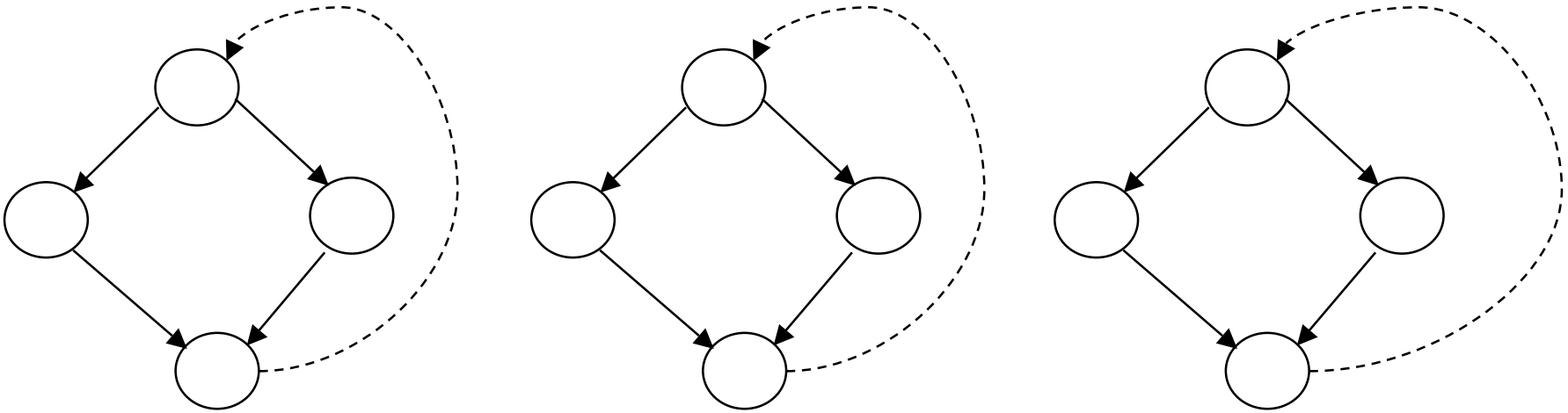
c3 = c1 + c2
c1 and c2 are linearly independent, etc.

5 circuits form a **basis**

McCabe Complexities of Typical Program Graphs

- McCabe: If a separate part of a graph is not strongly connected, add a **phantom arc** from finish to start.
- Might think of this phantom arc as representing the **repeated use** of the program graph.
- *Alternatively*, use the *modified* formula on graph without adding arc:
  $\mu(G) = e - n + 2p$
- Above, 2p accounts for one *phantom* arc in each of p separate parts.

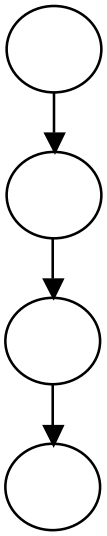
Adding one phantom edge per separate part



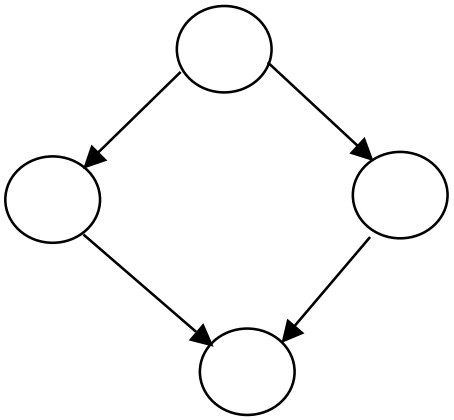
$\mu(G) = e - n + \mathbf{p}$ if phantom edges *counted* in e

$\mu(G) = e - n + \mathbf{2p}$ if phantom edges *not counted* in e

Complexities of Typical Program Graphs



$\mu(G) = e - n + 2p$
$= 3 - 4 + 2$
$= 1$

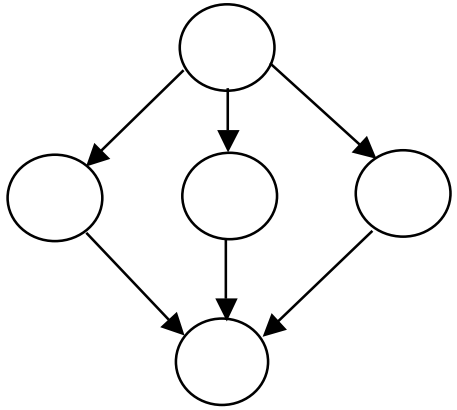


$\mu(G) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

Straight-line programs have $\mu(G) = 1$.

Two-way branch programs have $\mu(G) = 2$.

Complexities of Typical Program Graphs



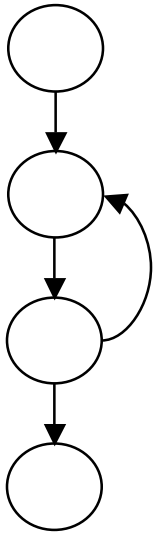
$\mu(G) = e - n + 2p$
$= 6 - 5 + 2$
$= 3$

Adding a branch increases e-n by 1, so by induction:

Simple k-way branch programs have $\mu(G) = k$.

Three-way branch programs have $\mu(G) = 3$.

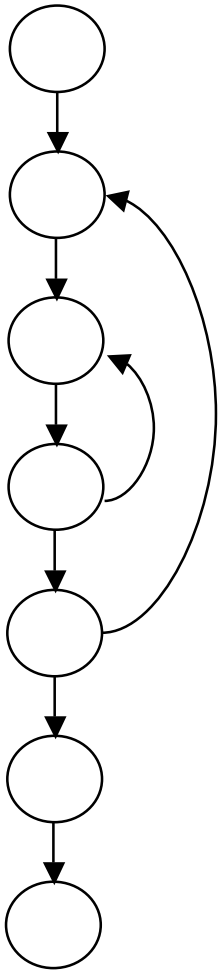
Complexities of Typical Program Graphs



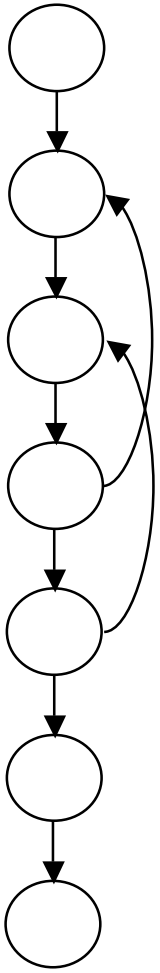
$\mu(G) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

Simple ‘while’ programs have $\mu(G) = 2$.

Nested



Intersecting

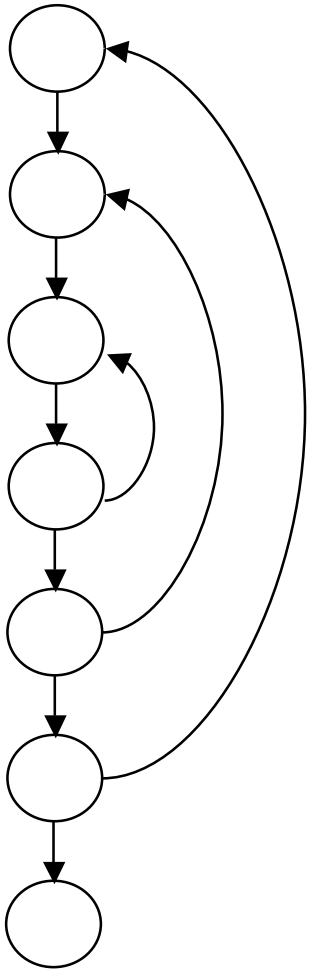


These have the same complexity. This may be misleading.

$\mu(G) = e - n + 2p$
$= 8 - 7 + 2$
$= 3$

Complexities of Typical Program Graphs

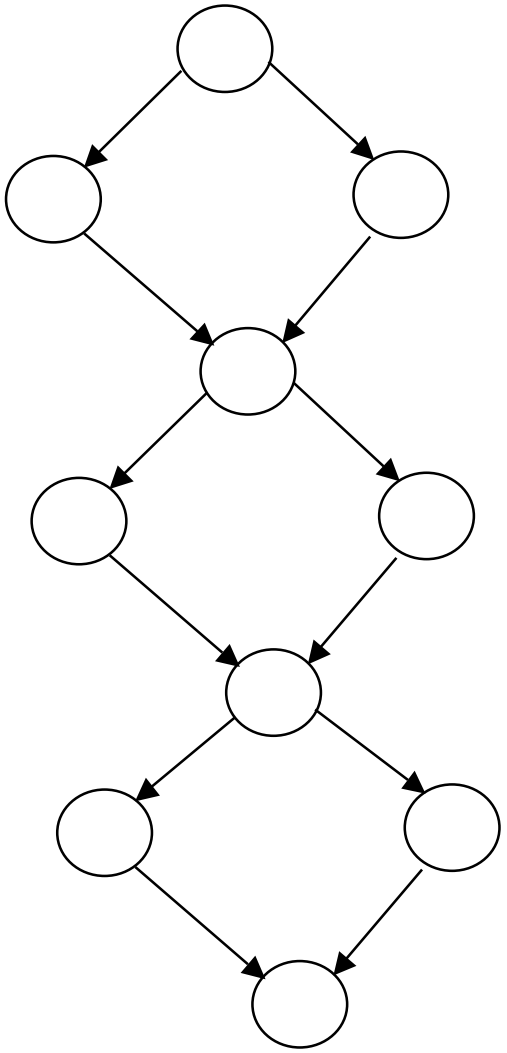
Triply-Nested



$$\mu(G) = e - n + 2p$$
$$= 9 - 7 + 2$$
$$= 4$$

Simple k-tuply nested programs have $\mu(G) = k+1$.

Complexities of Typical Program Graphs



$$\mu(G) = e - n + 2p$$
$$= 12 \;\; 10 + 2$$
$$= 4$$

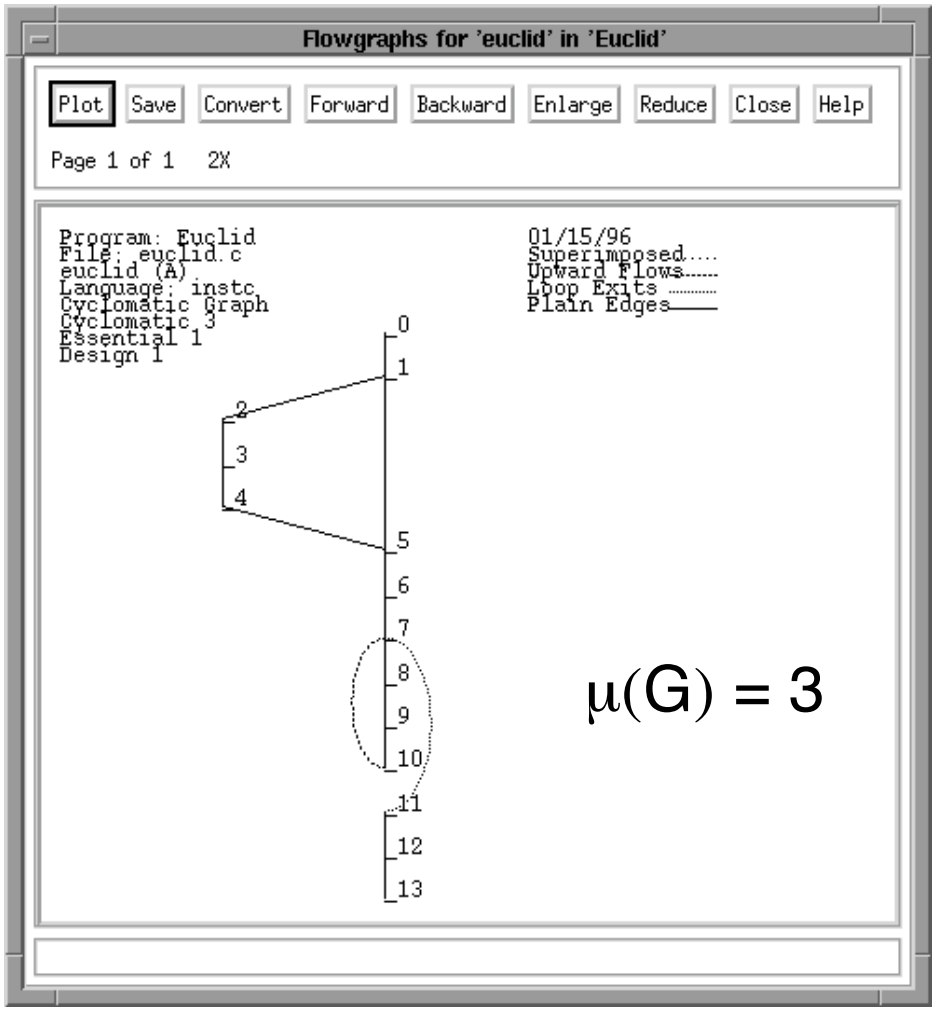
k-section “lattices”
have $\mu(G) = k+1$.

Somehow this seems to hide
the possible “combinatorial explosion.”

Uses of complexity $\mu(G)$

- $\mu(G)$ can be used to indicate the **minimum number of test cases required.**
- Keep $\mu(G)$ small (e.g. < 10) for "understandable" programs.
- Certain constructs (e.g. switch) may be exempted from the count in some organizations.

Cyclomatic Tools

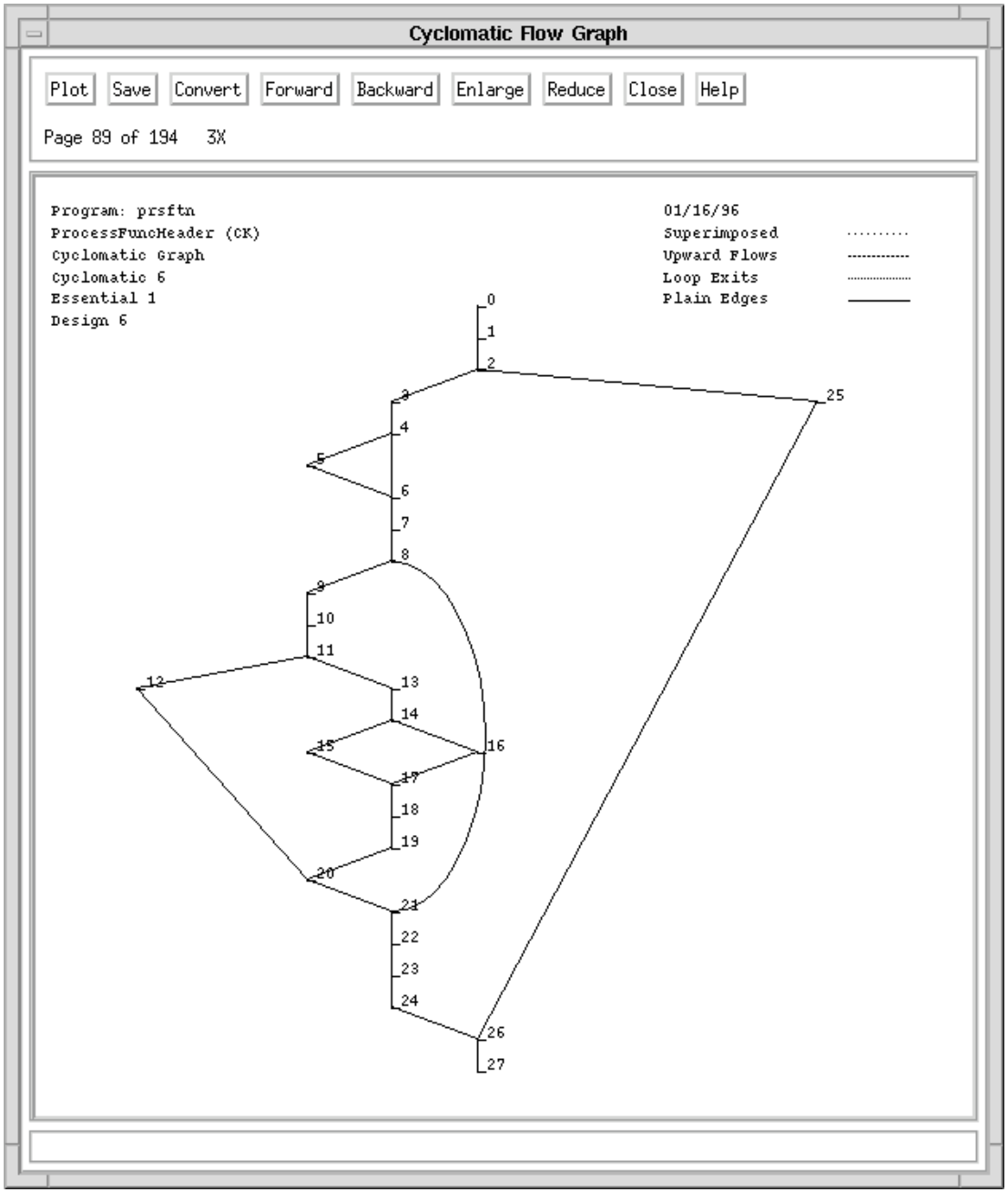


$\mu(G) = 3$

```
Module: euclid
Basis Test Paths: 3 Paths

Test Path B1: 0 1 5 6 7 11 12 13

        8(    1): n>m ==> FALSE

        14(    7): r!=0 ==> FALSE

Test Path B2: 0 1 2 3 4 5 6 7 11 12 13

        8(    1): n>m ==> TRUE

        14(    7): r!=0 ==> FALSE

Test Path B3: 0 1 5 6 7 8 9 10 7 11 12 13

        8(    1): n>m ==> FALSE

        14(    7): r!=0 ==> TRUE

        14(    7): r!=0 ==> FALSE
```

Source: Watson & McCabe, 1996

$\mu(G) = 6$

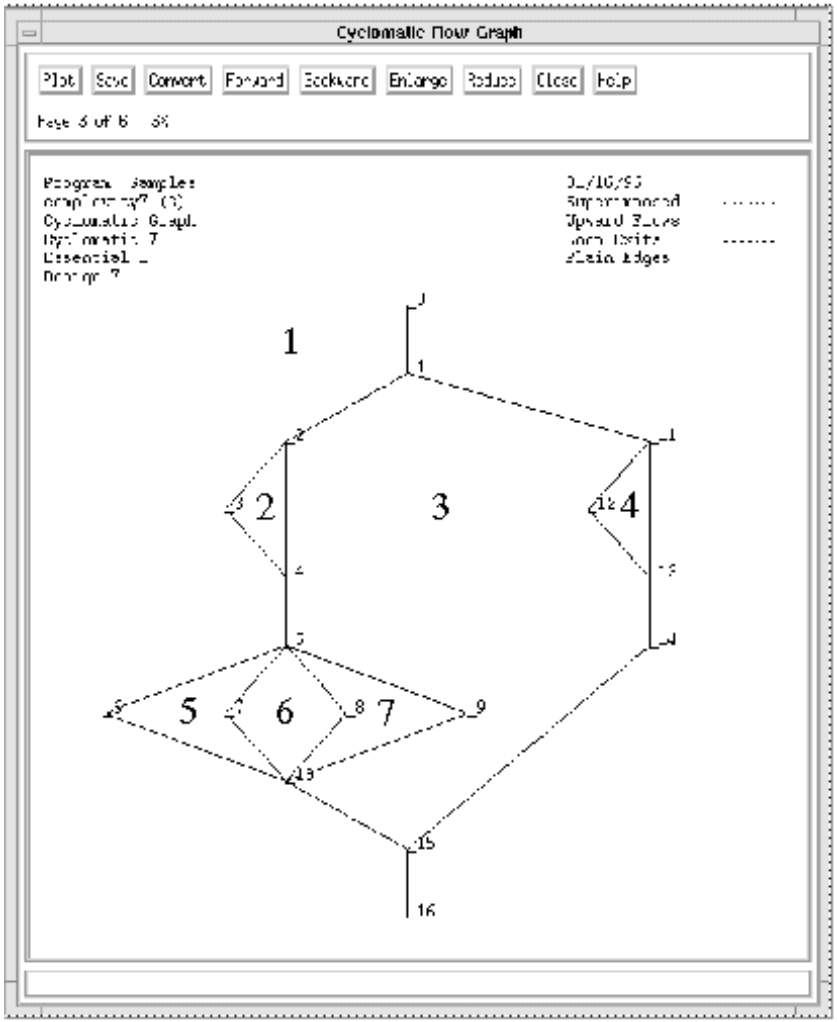

Cyclomatic Flow Graph
Plot
Save
Convert
Forward
Backward
Enlarge
Reduce
Close
Help
Page 89 of 194 3X
Program: prsftn
ProcessFuncHeader (CK)
Cyclomatic Graph
Cyclomatic 6
Essential 1
Design 6
01/16/96
Superimposed
Upward Flows
Loop Exits
Plain Edges
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Simplified Complexity Measures

- Counting binary predicate tests only:
  $\eta(G)$ = predicates + 1

- Counting ***regions*** = e - n + 2p
  (from Euler's formula: N-E+F=2, F = faces or regions)

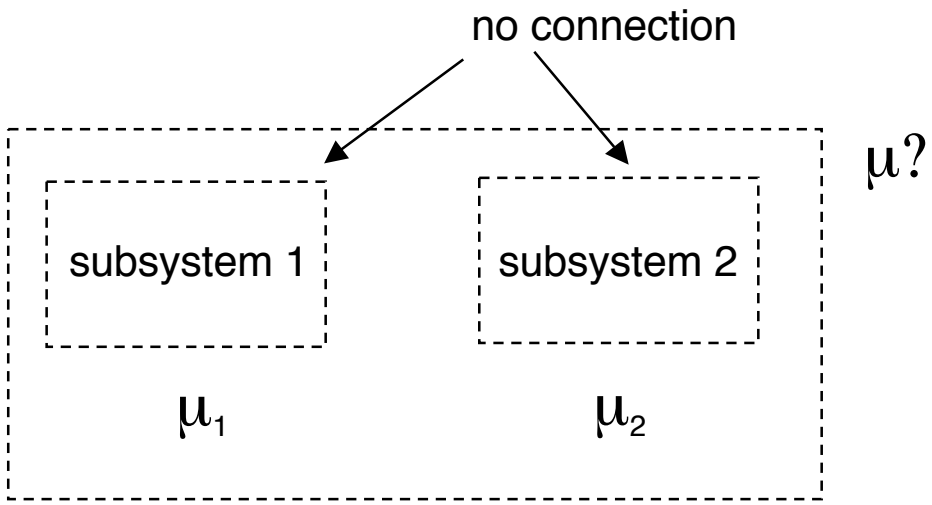
Counting Regions



regions numbered

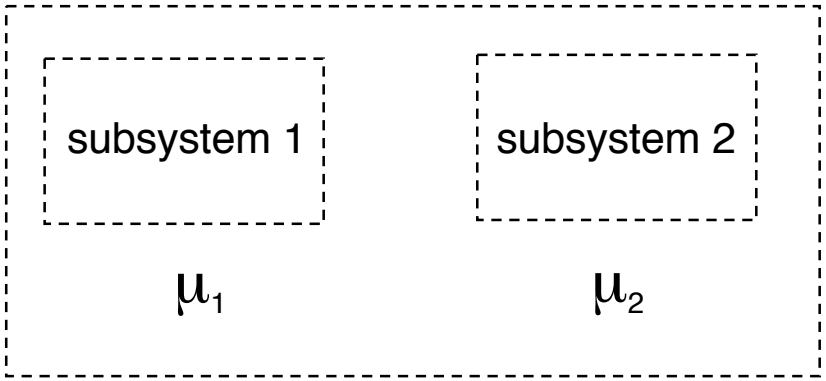
Use of μ(G) in **Integration Testing**

- Composition of modules' complexity: what is μ of overall system in terms of individual μ 's?



Use of μ(G) in Integration Testing

- Composition of modules' complexity: what is μ of overall system in terms of individual μ's?



μ?

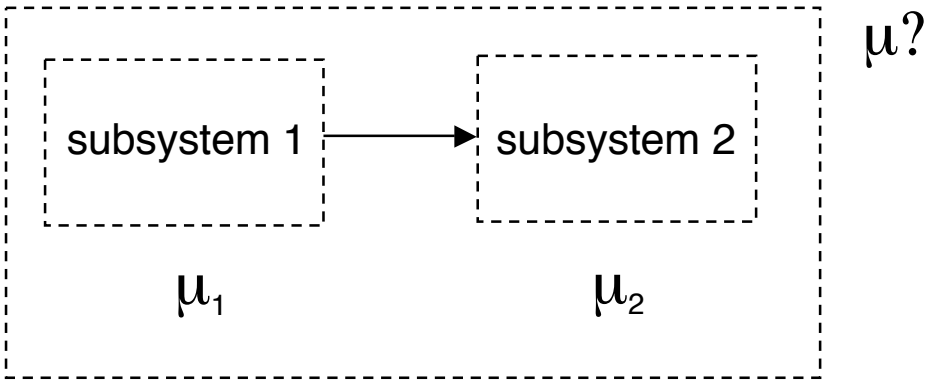
$e = e_1 + e_2$
$n = n_1 + n_2$
$p = p_1 + p_2$

$\mu = \mu_1 + \mu_2$

$\mu_1(G) = e_1 - n_1 + 2p_1$ $\mu_2(G) = e_2 - n_2 + 2p_2$

Use of $\mu(G)$ in Integration Testing

- What if connected by one edge?



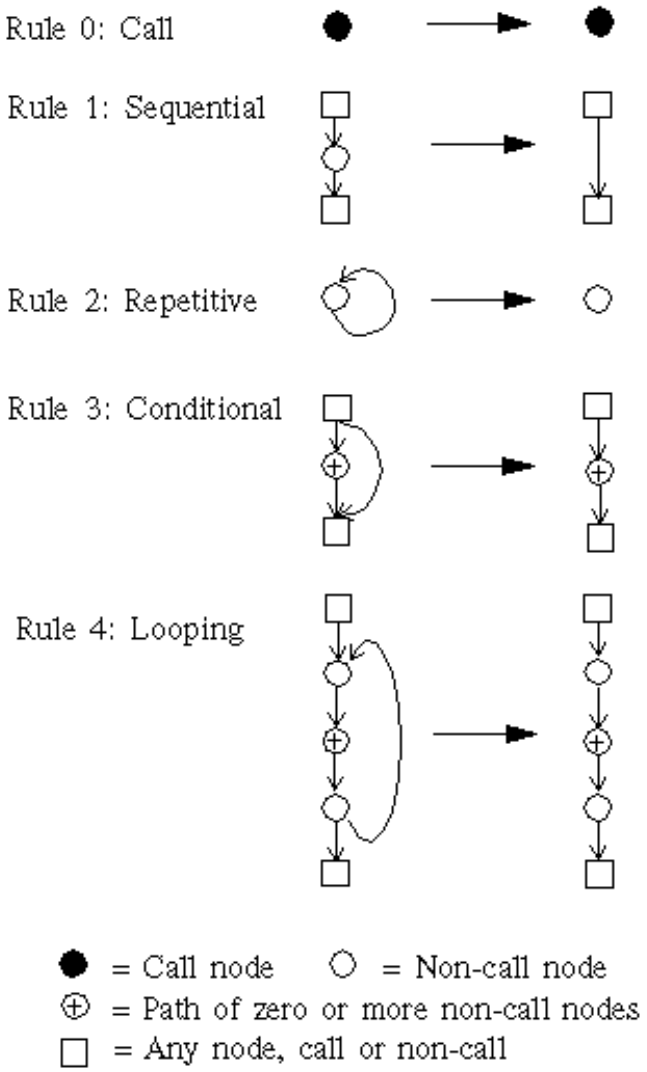
$\mu = \mu_1 + \mu_2 + 1 - 1$

one fewer separate part

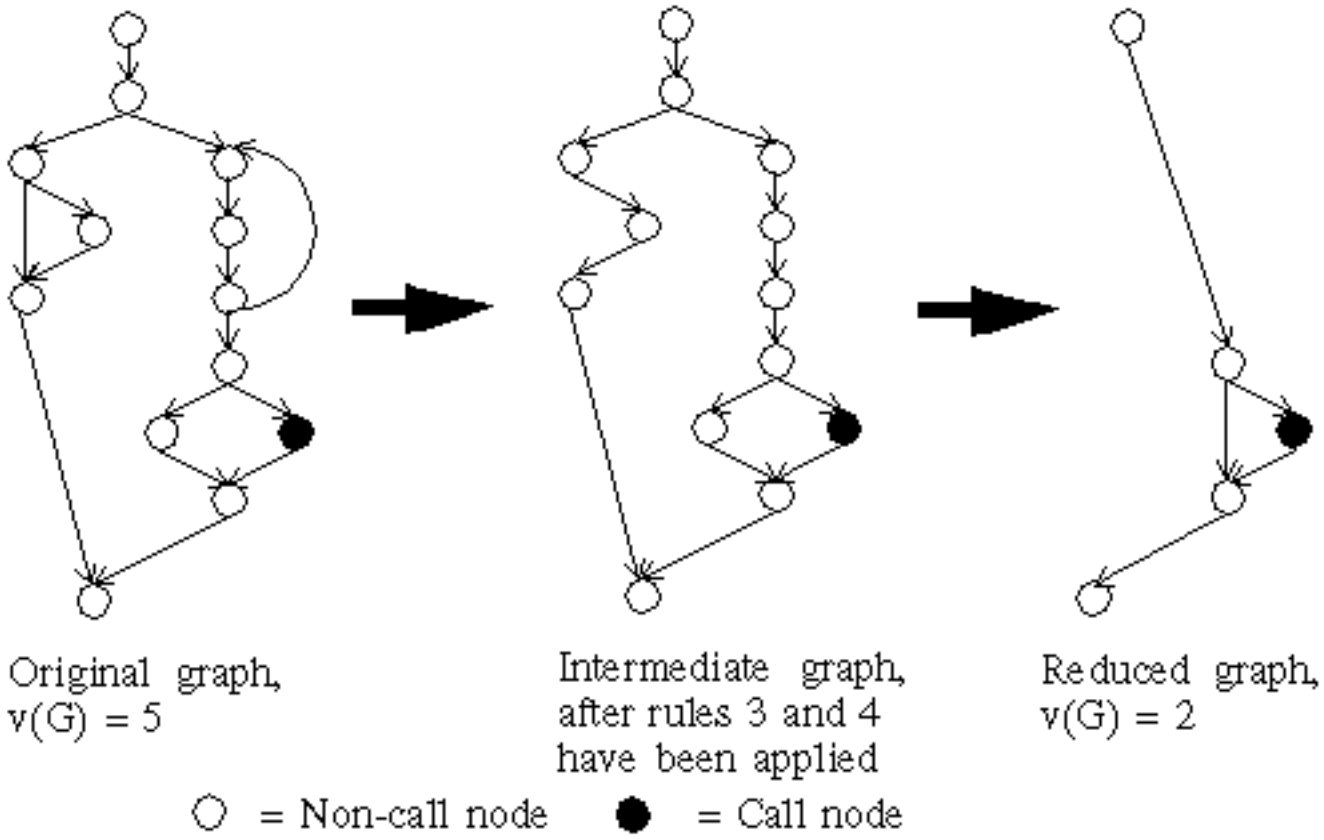
one more edge

So $\mu = \mu_1 + \mu_2$, even if the subsystems are singly connected.

"Design-Reduction" Technique: Lumping hierarchically



"Design-Reduction" Technique



Testing Based on Data Flow

Data Flow Testing

- As an alternative to basing tests on control paths, base it on data paths.
- In fact, examining data paths can reveal problems that control paths cannot.

"Definitions" and "Uses"

- "Definitions" for a variable:
  - Creating a variable and initializing it
  - Assigning a value to a variable
  - Reading the value as input
  - Initializing the variable as a parameter to a method

"Definitions" and "Uses"

- "Uses" for a variable:
  - The variable appears in an expression, such as:
    - RHS of an assignment
    - Conditional or while
    - Function call
  - The value of the variable is used in an output statement.

Definitions and Uses

Stmt. No.	Statement sequence	Definition	Use	Kill
1	u = 5;	u		
2	x = u + 9;	x	u	
3	y = x * x;	y	x	
4	x = u + 11;	x	y	x
5	y = x * y;	y	x, y	y

Note that a re-assignment is considered a *new* definition. It is said to "kill" the old one.

DU and DC Paths

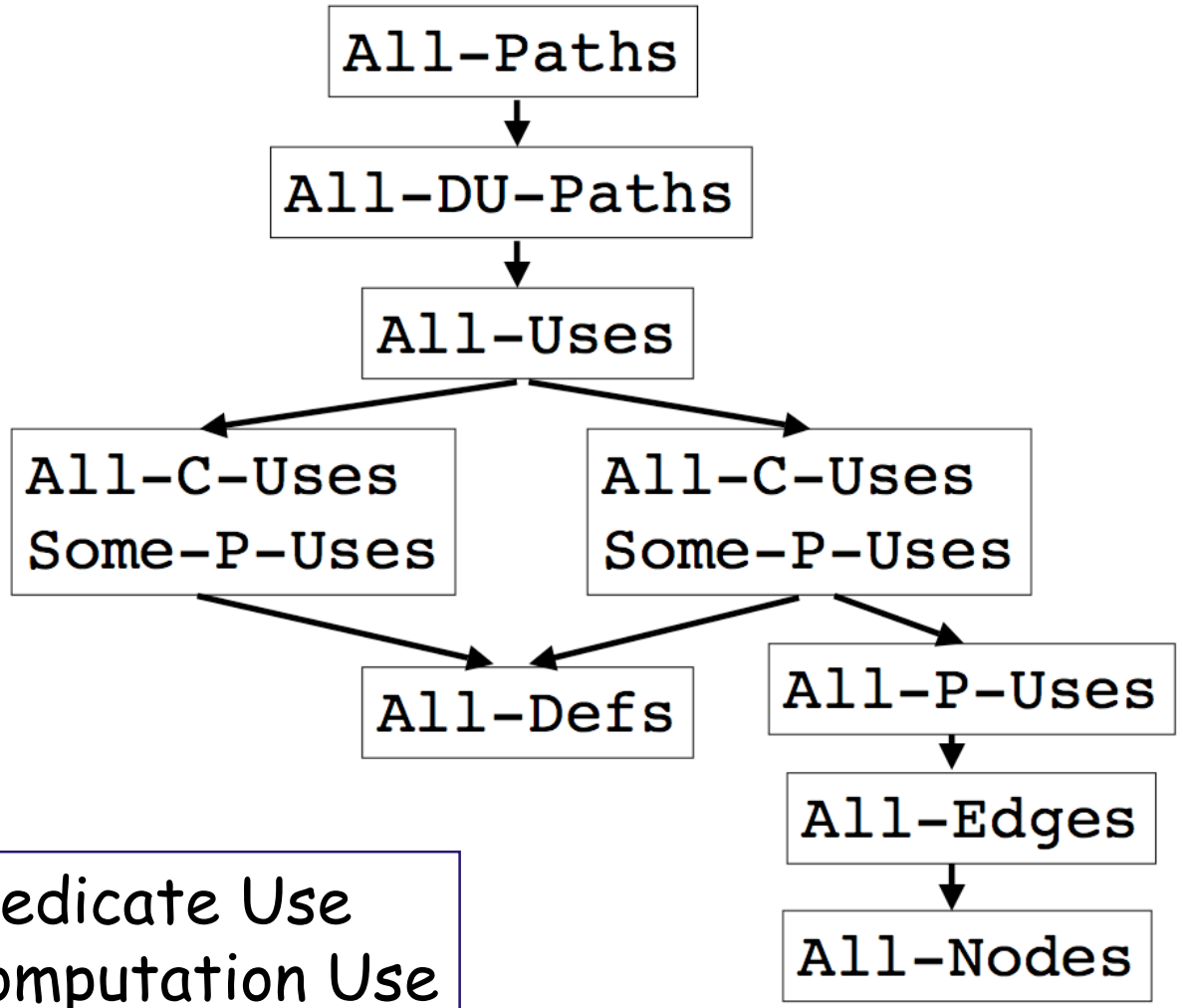
- A DU (Definition-Use) Path for a variable is a path through a contiguous sequence of statements, that:
  - Begins with a definition of the variable
  - Ends with a use of the variable

- A DC (Definition-Clear) path is a DU path that also:
  - Has no intervening definitions (but may have intervening uses).

Possible Testing Criteria

- **All-Defs** testing:
  - For every variable defintion, test **at least one** DU path from that definition.
- **All-Uses** testing:
  - For every variable definition, **and for every use**, test at least one DU path between the definition and use.

Coverage Inclusions



P-use = Predicate Use
C-use = Computation Use

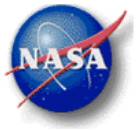
Coverage Assessment Tools

Supported languages	Tool name	Measurements			
		Statement/ line/block	Branch/ decision	Method/ function	Class
Java	Agitar [14]	X	X	X	X
	Clover [16]	X	X	X	X
	Cobertura [17]	X	X		
	EMMA [20]	X		X	X
	JCover [24]	X	X	X	X
	Koalog [25]	X			
	Jtest [26]	X	X	X	X
C/C++	Bullseye [15]		X	X	
	CodeTEST [18]	X	X		
	Dynamic [19]	X	X	X	
	Gcov [22]	X			
	Intel [23]	X		X	
	C++test [26]	X	X		X
Java and C/C++	eXVantage [21]	X	X	X	X
	PurifyPlus [27]	X		X	
	SD [28]	X	X	X	X
	TCAT [29]	X	X	X	X

Example GCOV (Gnu)

- Works with gcc, cousin of gprof (timing profile)
- Annotates a copy of source files.
- Shows times executed for each statement
  - Other stats
- Accumulates times over several runs:
  - Typically different test sets
- Try 'man gcov'
- See also:
  http://www.cs.hmc.edu/clinic/projects/2004/google/

NASA Example



Testing - Code Coverage

- Used GNU's gcov tool to determine code coverage of the current set of test cases.
- Statement coverage after running all test cases was 83%.
- Some surprising hotspots involved getting mnemonic names (~2 billion times) and string comparison on mnemonic names (~1.5 billion times).

Example LCOV

html graphical coverage from GCOV

Current view: directory

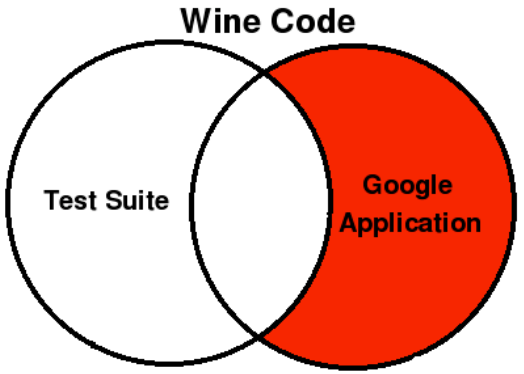
Test: Wine Regression Test Performance
Date: 2004-09-24
Code covered: 31.9 %

Instrumented lines: 304086
Executed lines: 96950

Directory name	Coverage		
/home/aarvey/winefromwinehqcvs/include		65.9 %	27 / 41 lines
/home/aarvey/winefromwinehqcvs/include/wine		49.4 %	44 / 89 lines
/home/aarvey/winefromwinehqcvs/misc		28.4 %	278 / 979 lines
/home/aarvey/winefromwinehqcvs/windows		45.7 %	5753 / 12586 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/advapi32		26.3 %	820 / 3118 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/advapi32/tests		89.9 %	453 / 504 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/cabinet		0.1 %	2 / 2821 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/comctl32		5.7 %	1559 / 27195 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/comctl32/tests		97.4 %	374 / 384 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/dbghelp		17.7 %	702 / 3967 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/dsound		13.1 %	712 / 5429 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/dsound/tests		25.8 %	603 / 2340 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/dxerr8		16.4 %	9 / 55 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/gdi		31.3 %	3777 / 12073 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/gdi/enhmfdrv/enhmfdrv		22.0 %	278 / 1264 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/gdi/enhmfdrv/enhmfdrv/enhmfdrv		22.0 %	278 / 1264 lines
/home/aarvey/winefromwinehqcvs/wine.dirty2/dlls/gdi/enhmfdrv/enhmfdrv/enhmfdrv/enhmfdrv		22.0 %	278 / 1264 lines

Differential Coverage Project
(HMC Clinic project sponsored by Google)

- What does a user run of the application cover that the test suite doesn't?
- Use the answers to extend the test suite.
- Display graphically by augmenting LCOV.



Example: Testing Notepad

Test Suite

```
572                :
573              0 :       if( lphc->dwStyle & CBS_SORT )
574              0 :         lbeStyle |= LBS_SORT;
575              0 :       if( lphc->dwStyle & CBS_HASSTRINGS )
576              0 :         lbeStyle |= LBS_HASSTRINGS;
577              0 :       if( lphc->dwStyle & CBS_NOINTEGRALHEIGHT )
578              0 :         lbeStyle |= LBS_NOINTEGRALHEIGHT;
579              0 :       if( lphc->dwStyle & CBS_DISABLENOSCROLL )
580              0 :         lbeStyle |= LBS_DISABLENOSCROLL;
581                :
```

Notepad

```
572                :
573              3 :       if( lphc->dwStyle & CBS_SORT )
574              0 :         lbeStyle |= LBS_SORT;
575              3 :       if( lphc->dwStyle & CBS_HASSTRINGS )
576              3 :         lbeStyle |= LBS_HASSTRINGS;
577              3 :       if( lphc->dwStyle & CBS_NOINTEGRALHEIGHT )
578              0 :         lbeStyle |= LBS_NOINTEGRALHEIGHT;
579              3 :       if( lphc->dwStyle & CBS_DISABLENOSCROLL )
580              0 :         lbeStyle |= LBS_DISABLENOSCROLL;
581                :
```

Differential

```
572                :
573              1 :       if( lphc->dwStyle & CBS_SORT )
574              0 :         lbeStyle |= LBS_SORT;
575              1 :       if( lphc->dwStyle & CBS_HASSTRINGS )
576              1 :         lbeStyle |= LBS_HASSTRINGS;
577              1 :       if( lphc->dwStyle & CBS_NOINTEGRALHEIGHT )
578              0 :         lbeStyle |= LBS_NOINTEGRALHEIGHT;
579              1 :       if( lphc->dwStyle & CBS_DISABLENOSCROLL )
580              0 :         lbeStyle |= LBS_DISABLENOSCROLL;
581                :
```

Summary Page

Test Suite

Date: 2004-11-11
Code covered: 33.4 %
Instrumented lines: 305705
Executed lines: 101981

Directory name	Coverage		
/home/wine/dlls/user		0.0 %	0 / 20 lines
/home/wine/include		65.9 %	27 / 41 lines
/home/wine/include/wine		49.4 %	44 / 89 lines

Notepad

Date: 2004-11-13
Code covered: 16.8 %
Instrumented lines: 190562
Executed lines: 31949

Directory name	Coverage		
/home/cpierog/build testsuite/dlls/advapi32		50.0 %	2 / 4 lines
/home/cpierog/build testsuite/dlls/comctl32		50.0 %	2 / 4 lines
/home/cpierog/build testsuite/dlls/commdlg		16.7 %	4 / 24 lines

Compare

Date: 2004-11-13
Code covered: 3.1 %
Instrumented lines: 190562
Executed lines: 5981

Directory name	Coverage		
/home/cpierog/build testsuite/dlls/advapi32		0.0 %	0 / 4 lines
/home/cpierog/build testsuite/dlls/comctl32		0.0 %	0 / 4 lines
/home/cpierog/build testsuite/dlls/commdlg		16.7 %	4 / 24 lines

When to Stop Testing?

Boris Beizer on stopping

There is no single, valid, rational criterion for stopping.

Furthermore, given any set of applicable criteria, how each is weighed depends very much upon the product, the environment, the culture and the attitude toward risk.

Software Test Standards Reference

(Order from http://www.12207.com/test1.htm)

AECL CE-1001-STD REV.2	Standard for Software Engineering of Safety Critical Software
BS-7738-1	Specification for Information Systems Products Using SSADM--(Structured Systems Analysis and Design Method)
BS-7925-1	Software Testing - Vocabulary
BS-7925-2	Standard for Software Component Testing
FDA Guidance Document 2	Guidance for Industry, FDA Reviewers and Compliance on Off-the-Shelf Software Use in Medical Devices. (Checklist available)
IEC 60601-1-4	Medical Electrical Equipment--Part 1: General Requirements for Safety-4. Collateral Standard: Programmable Electrical Medical Systems (Checklist available)
IEC 60880	Software for Computers in the Safety Systems of Nuclear Power Stations
IEC 62304	Medical device software - Software life cycle processes (Checklist available)
IEEE 829	Software Test Documentation
IEEE 1008	Software Unit Testing
IEEE 1044	Classification for Software Anomalies
IEEE 1044.1	Guide to Classification for Software Anomalies