Testing

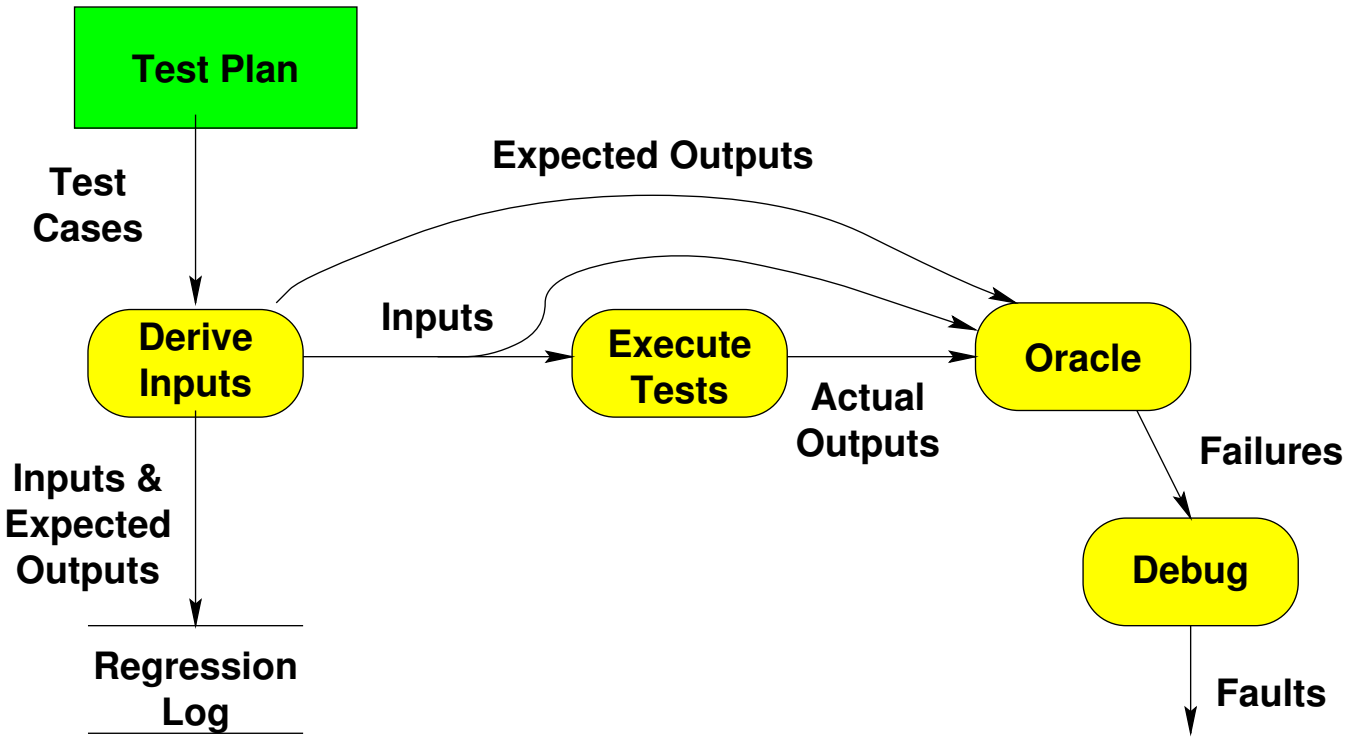
Steven Zeil

February 13, 2013


Outline


Outline I


The Testing Process



Testing Process Components

- *Test case*: a general description of a required test
  - There may be many possible inputs that would serve
- *Regression Log*: Database of tests and expected outputs
  - Used during regression testing to quickly rerun old tests
- *Oracle*: The process, person, and/or program that determines if test output is correct
- *Failure*: An execution on which incorrect behavior occurs
- *Fault*: A defect in the code that (may) cause a failure
- *Error*: A human mistake that results in a fault


Why do we test?

- To show that the program works


Why do we test?

- To show that the program works

  Dijkstra: "Testing can show the presence of errors, but never their absence."

Why do we test?

- To show that the program works

  Dijkstra: "Testing can show the presence of errors, but never their absence."

- To reveal faults


Why do we test?

- To show that the program works

  Dijkstra: "Testing can show the presence of errors, but never their absence."

- To reveal faults
  - A test is "successful" only if it reveals a fault
  - A bit silly when large numbers of tests have executed without failure
- To gain confidence in the program


Why do we test?

- To show that the program works

  Dijkstra: "Testing can show the presence of errors, but never their absence."

- To reveal faults
  - A test is "successful" only if it reveals a fault
  - A bit silly when large numbers of tests have executed without failure
- To gain confidence in the program
  - prior experience with similar testing methods
  - statistical confidence


Choosing Test Data

From these motives come different approaches to choosing test data

- Representative testing
  - tests designed to reflect the frequence of user inputs. Used for reliability estimation.
- Directed testing
  - Tests designed to discover system defects.
  - A successful defect test is one which reveals the presence of defects in a system.


Outline I


Representative testing

Choose data that is representative of the way the end users will exercise the software.

- Advantages:
  - realism — catches the kinds of failures that the users would have encountered
  - relatively cheap to generate tests
  - time to failure during test reflects time to failure in operation
    - useful in statistical reliability modeling


Producing Representative Tests

- collected data from existing system
  - further selection may be needed
  - cannot accomodate new functionality


Producing Representative Tests

- collected data from existing system
  - further selection may be needed
  - cannot accomodate new functionality
- random generators
  - non-uniform, to match desired distribution
  - can be hard to generate non-numeric ADT values


Representative Testing

- Requires *operational profile* for its definition
  - The operational profile defines the expected pattern of software usage

Sample Op Profile

Input Category	*Percentage*
Transaction Proc.	85%
Balancing	14%
Year-end Report	1%


Sample Op Profile

Input Category	*Percentage*
Transaction Proc.	85%
New account	7%
Close account	3%
Debits	70%
Credits	20%
Balancing	14%
Year-end Report	1%


Sample Op Profile

Input Category	*Percentage*	
Transaction Proc.	85%	
New account	7%	
new		90%
already exists		10%
Close account	15%	
non-existant		15%
exists		85%
Debits	70%	
non-existant		25%
exists		75%
Credits	20%	
non-existant		25%
exists		75%
Balancing	14%	
Year-end Report	1%	


Representative testing difficulties

- Uncertainty in the operational profile
  - This is a particular problem for new systems with no operational history. Less of a problem for replacement systems
- High costs of generating the operational profile
  - Costs are very dependent on what usage information is collected by the organisation which requires the profile
- Statistical uncertainty
  - Difficult to estimate level of confidence in operational profile
  - Usage pattern of software may change with time


Reliability Growth Models

- Growth model is a mathematical model of the system reliability change as it is tested and faults are removed
- Used as a means of reliability prediction by extrapolating from current data


Reliability modeling procedure

- Determine operational profile of the software
- Generate a set of test data corresponding to the profile
- Apply tests, measuring amount of execution time between each failure
- After a statistically valid number of tests have been executed, reliability can be measured


Reliability metrics

- Probability of failure on demand
  - This is a measure of the likelihood that the system will fail when a service request is made
  - $POFOD = 0.001$ means 1 out of 1000 service requests result in failure
  - Relevant for safety-critical or non-stop systems
- Rate of fault occurrence (*ROCOF*)
  - Frequency of occurrence of unexpected behaviour
  - *ROCOF* of 0.02 means 2 failures are likely in each 100 operational time units
  - Relevant for operating systems, transaction processing systems

Reliability metrics

- Mean time to failure
  - Measure of the time between observed failures
  - $MTTF = 500$ means that the time between failures is 500 time units
  - Relevant for systems with long transactions e.g. CAD systems


Reliability metrics

- Availability
  - Measure of how likely the system is available for use. Takes repair/restart time into account
  - $Avail = 0.998$ means software is available for 998 out of 1000 time units
  - Relevant for continuously running systems e.g. telephone switching systems


Reliability measurement

- Measure the number of system failures for a given number of system inputs
  - Used to compute *POFOD*
- Measure the time (or number of transactions) between system failures
  - Used to compute *ROCOF* and *MTTF*
- Measure the time to restart after failure
  - Used to compute *Avail*


Time units

- Time units in reliability measurement must be carefully selected. Not the same for all systems
- Raw execution time (for non-stop systems)
- Calendar time (for systems which have a regular usage pattern e.g. systems which are always run once per day)
- Number of transactions (for systems which are used on demand)


Jelinski-Moranda Model

Assumptions:

- Software contains N faults (N is unknown)
- Each fault *manifests* (causes a failure) at rate ϕ
- Faults manifest independently
- Faults are fixed perfectly, without introducing new ones

If we have repaired i faults, the program's failure rate λ is

$$\lambda_i = (N - i)\phi$$

Observed reliability growth

- Simple equal-step model but does not reflect reality
- Reliability does not necessarily increase with change as the change can introduce new faults
- The rate of reliability growth tends to slow down with time as frequently occurring faults are discovered and removed from the software

Musa Logarithmic Poisson Model

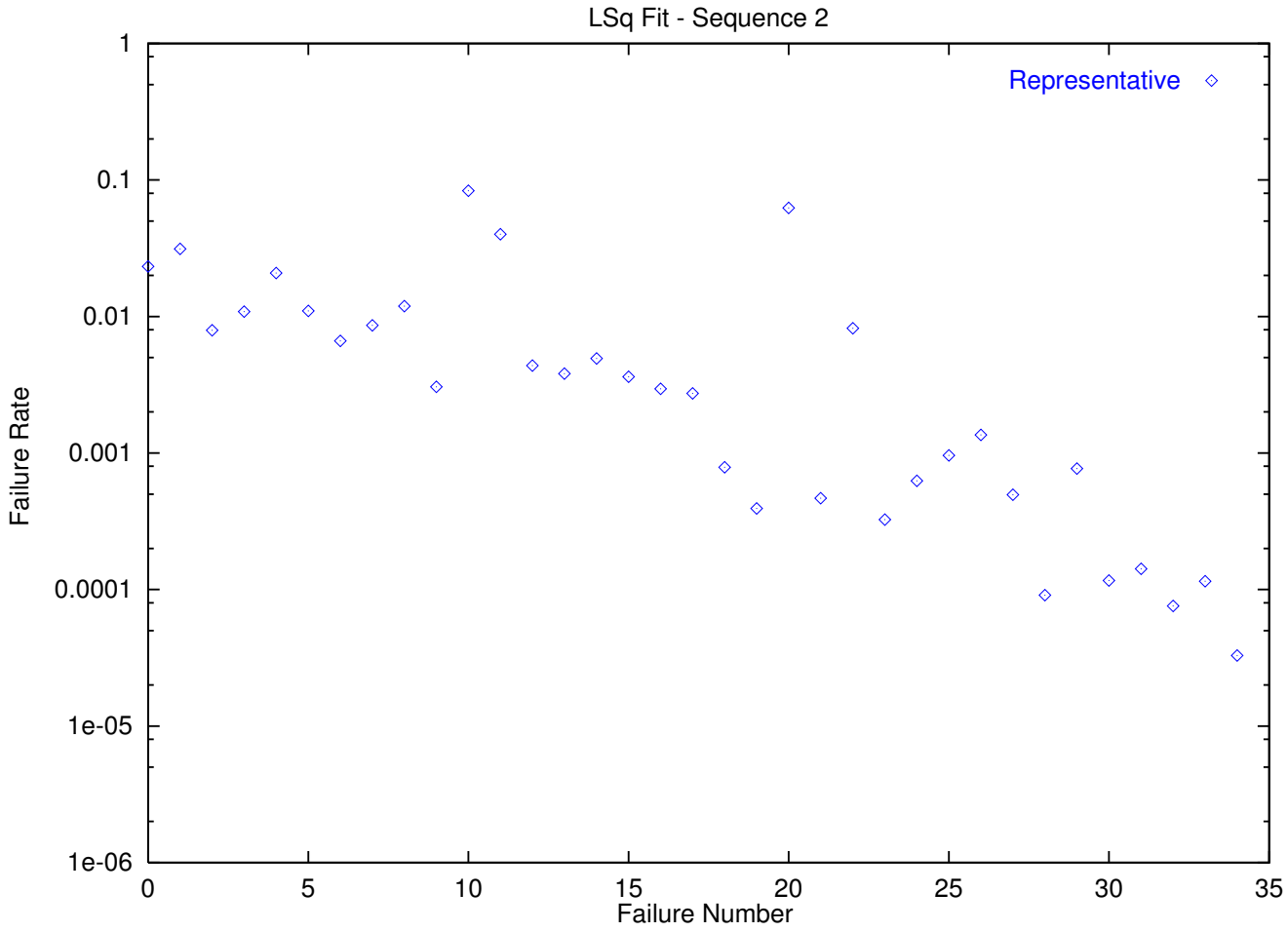
Assumptions:

- Software can never be completely free of faults.
- Faults manifest independently
- Faults are found in decreasing order of failure rate.
- The program failure rate before repairing any faults is λ_0
- Faults are fixed perfectly, without introducing new ones

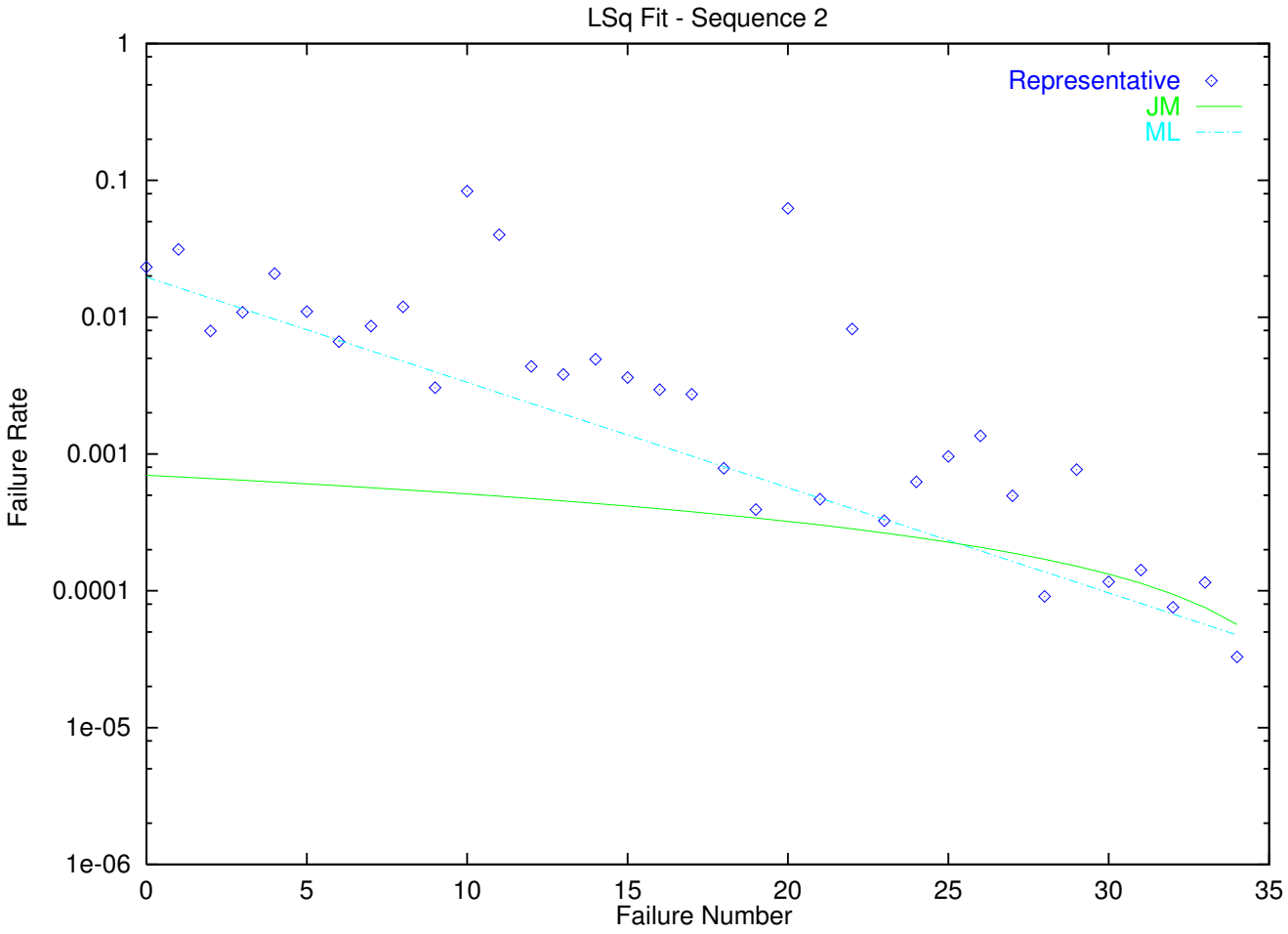
If we have repaired i faults, the program's failure rate λ is

$$\lambda_i = \lambda_0 e^{-\theta i}$$


Fitting Example



Fitting Example



Outline I


Directed testing

Choose tests designed to reveal

- many faults
- as quickly as possible


Choosing Good Test Data

Techniques for selecting test data are generally termed either

- white-box, or
- black-box


White-Box Testing

White-box testing is the choice of test data based upon the details of the algorithms in the code under test.

- Formally coverage measures, E.g., statement coverage, branch coverage
- Informally, design tests specifically to exercise "tricky" parts of the code


Black-Box Testing

Black-box (a.k.a. *specification-based*) testing chooses tests without consulting the implementation.

- based simply upon our understanding of what the unit is supposed to do.


Functional Coverage

a.k.a *Equivalence partitioning*

- Choose at least one test that covers each distinct "behavior" described in the requirements.
  - Different functions performed by the program
  - Different types or ranges of input that get treated or described separately.
  - Different types or ranges of output that get treated or described separately.
- Large, structured projects place emphasis on tracking requirements to functional test cases


Boundary Values Testing

a.k.a., *Extremal Values Testing*

- Choose as test data the largest and the smallest values for each input and for each "functional" case
  - Often misunderstood as simply choosing largest & smallest possible inputs
  - Assumes that we have started by attempting functional coverage


Special Values Testing

Choose as test data those certain data values that just tend to cause trouble.
Programmers eventually develop a sense for these. They include

- For integers: -1, 0, 1
- For floating point numbers: -e, 0, +e, where "e" is a very small number
- For strings: the empty string, strings containing only blanks, strings containing no alphabetic characters


What is Special?

- As we move to other data structures, we may develop a suspicion of other special values, e.g.,
  - For times of the day: midnight, noon
  - For containers of data: an empty container
- Special values and Boundary values often overlap
- (F14 example)


Should You Test With Illegal Values?

Many people advocate testing programs with illegal inputs. For example, if a program is supposed to accept a positive integer as its input, they would suggest running it on a zero and a negative input value.


Should You Test With Illegal Values?

Many people advocate testing programs with illegal inputs. For example, if a program is supposed to accept a positive integer as its input, they would suggest running it on a zero and a negative input value.

This is *nonsense*.


You Can't Test an Illegal Input

A test is only useful if it is possible for the program to either pass or fail the test.

So just what is a program *supposed* to do on an illegal input?

1. Should it abort/crash/stop running?


You Can't Test an Illegal Input

A test is only useful if it is possible for the program to either pass or fail the test.

So just what is a program *supposed* to do on an illegal input?

1. Should it abort/crash/stop running?
   - If so, how should it inform you that it has stopped?


You Can't Test an Illegal Input

A test is only useful if it is possible for the program to either pass or fail the test.

So just what is a program *supposed* to do on an illegal input?

1. Should it abort/crash/stop running?
   - If so, how should it inform you that it has stopped?
   - Perhaps even more importantly, how can you possibly test for this behavior if you don't *know* what the program is supposed to do?

You Can't Test an Illegal Input

A test is only useful if it is possible for the program to either pass or fail the test.

So just what is a program *supposed* to do on an illegal input?

1. Should it abort/crash/stop running?
   - If so, how should it inform you that it has stopped?
   - Perhaps even more importantly, how can you possibly test for this behavior if you don't *know* what the program is supposed to do?
2. Should it continue running?


You Can't Test an Illegal Input

A test is only useful if it is possible for the program to either pass or fail the test.

So just what is a program *supposed* to do on an illegal input?

1. Should it abort/crash/stop running?
   - If so, how should it inform you that it has stopped?
   - Perhaps even more importantly, how can you possibly test for this behavior if you don't *know* what the program is supposed to do?
2. Should it continue running?
   - If so, how should it recover from the illegal state that it will find itself in?


Illegal versus unusual Inputs

There's a big difference between the following two specifications:

Negatives are Illegal

The function sqrt(x) should be run on non-negative x. It returns a number z such that z∗z is approximately equal to x.

Negatives are Legal, but Unusual

The function sqrt(x), when run on non-negative x should return a number z such that z∗z is approximately equal to x. When run on a negative x it should abort execution with status code 144.

Illegal Inputs

> The function sqrt(x) should be run on non-negative x. It returns a number z such that z*z is approximately equal to x.

- In this spec, a negative input is illegal.


Illegal Inputs

> The function sqrt(x) should be run on non-negative x. It returns a number z such that z*z is approximately equal to x.

- In this spec, a negative input is illegal.
  - It's the programmer's responsibility to never do this.


Illegal Inputs

> The function sqrt(x) should be run on non-negative x. It returns a number z such that z*z is approximately equal to x.

- In this spec, a negative input is illegal.
  - It's the programmer's responsibility to never do this.
  - We can't use a negative input as a test because, no matter what the function does, it's *correct* by definition.


Unusual Inputs

```
   The function sqrt(x), when run on non-negative x
should return a number z such that z*z is
approximately equal to x.
   When run on a negative x it should abort
execution with status code 144.
```

- In this spec, a negative input is legal. It is probably intended to be an unusual case.


Unusual Inputs

```
   The function sqrt(x), when run on non-negative x
should return a number z such that z*z is
approximately equal to x.
   When run on a negative x it should abort
execution with status code 144.
```

- In this spec, a negative input is legal. It is probably intended to be an unusual case.
  - Programmers are allowed to supply this input if they are ready to deal with the specified "error" handling condition.


Unusual Inputs

```
   The function sqrt(x), when run on non-negative x
should return a number z such that z*z is
approximately equal to x.
   When run on a negative x it should abort
execution with status code 144.
```

- In this spec, a negative input is legal. It is probably intended to be an unusual case.
  - Programmers are allowed to supply this input if they are ready to deal with the specified "error" handling condition.
  - We can (and should) test the function with a negative input, because we know exactly what the desired behavior is supposed to be.


White-Box Testing

White-Box (a.k.a. *Implementation-based* testing) uses information from the implementation to choose tests.

- Structural Testing (a.k.a., "path testing" (not per your text) Designate a set of paths through the program that must be exercised during testing.
  - Statement Coverage
  - Branch Coverage
  - Cyclomatic coverage ("independent path testing")
  - Data-flow Coverage
- Mutation testing


Statement Coverage

Require that every statement in the code be executed at least once during testing.
Special programs ("software tools") will monitor this requirement for you.

- **gprof** in GNU gcc suite
- Java?


Example

```
cin >> x >> y;
while (x > y)
  {
   if (x > 0)
      cout << x;
   x = f(x, y);
  }
cout << x;
```

What kinds of tests are required for statement coverage?


Branch Coverage

Requires that every "branch" in the flowchart be tested at least once

- Equivalent to saying that each conditional stmt must be tested as both true and false
- Branch coverage implies Statement Coverage, but not vice versa

```
if X < 0 then
  X := -X;
Y := sqrt(X);
```


```
cin >> x >> y;
while (x > y)
  {
   if (x > 0)
      cout << x;
   x = f(x, y);
  }
cout << x;
```

What kinds of tests are required for branch coverage?

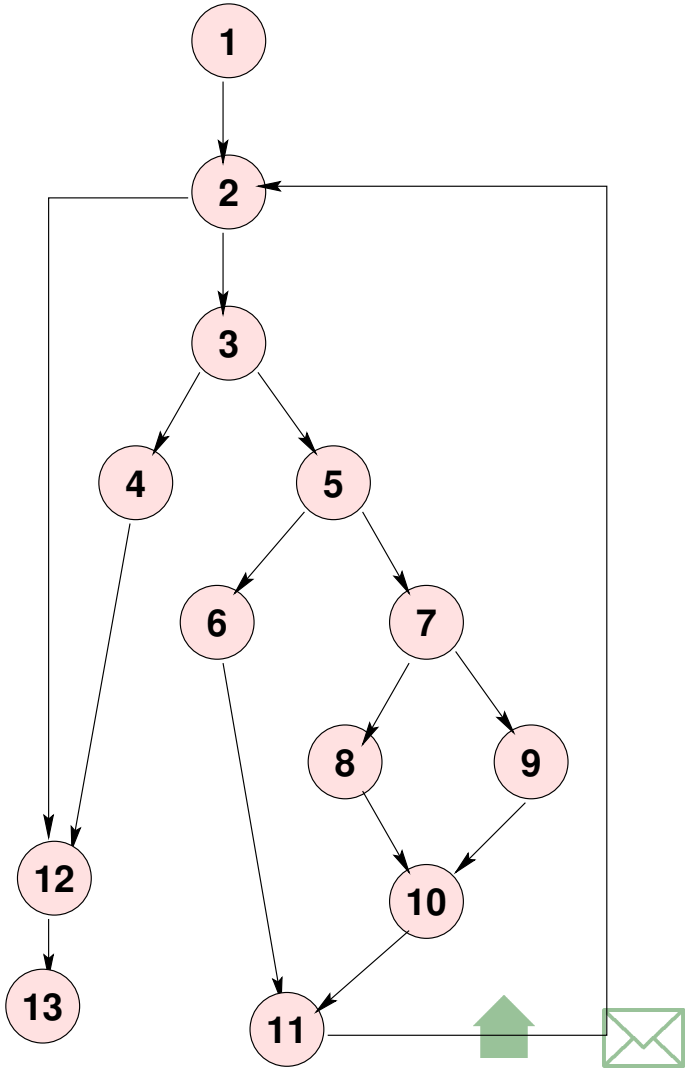

Cyclomatic Coverage

(a.k.a "independent path coverage", "path testing")

- The latter term (used in your text) should be discouraged as it is both vague and means something entirely different to most of the testing community
- Each independent path must be tested
  - An *independent path* is one that includes a branch not previously taken.

A Control Flow Graph

What are the independent paths?



A Control Flow Graph

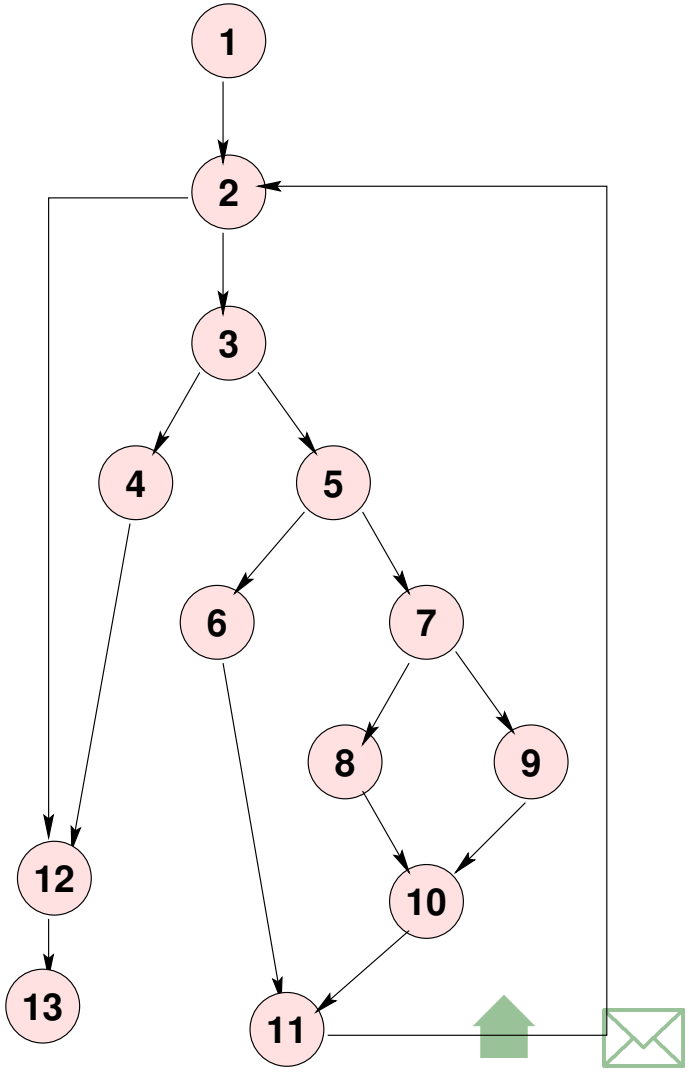
What are the independent paths?

1, 2, 3, 4, 12, 13

1, 2, 3, 5, 6, 11, 2, 12, 13

1, 2, 3, 5, 7, 8, 10, 11, 2, 12, 13

1, 2, 3, 5, 7, 9, 10, 11, 2, 12, 13



Cyclomatic Complexity

the number of independent paths in a program can be discovered by computing the cyclomatic complexity (McCabe, 1976) ...

$$CC(G) = \text{Number(edges)} - \text{Number(nodes)} + 1$$

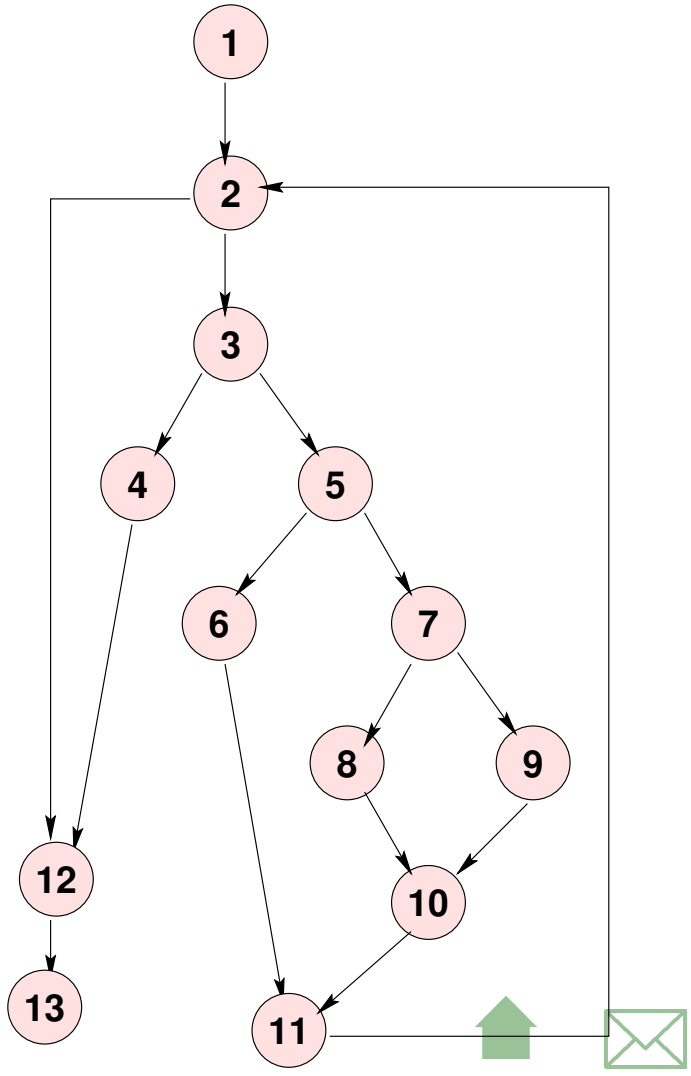
- This is a popular metric for module complexity.
- Actually pretty trivial: for structured programs with only binary decision constructs, equals number of conditional statements +1
- relation to testing is dubious
  - simply branch coverage hidden behind smoke and mirrors


Uniqueness

Sets of independent
paths are not unique, nor is their size:

1,2,3,5,6,11,
2,3,5,7,8,
10,11,2,3,
5,7,9,10,11,
2,12,13

1,2,3,4,12,13



Data-Flow Coverage

Attempts to test significant combinations of branches.

- Any stmt `i` where a variable `X` may be assigned a new value is called a *definition* of `X` at `i`: *def(X,i)*
- Any stmt `i` where a variable `X` may be used/retrieved is called a *reference* or *use* of `X` at `i`: *ref(X,i)*

def-clear

- A path from stmt `i` to stmt`j` is *def-clear with respect to X* if it contains no definitions of `X` except possibly at the beginning (`i`) and end (`j`)

all-defs

The *all-defs* criterion requires that each definition def(X,i) be tested some def-clear path to some reference ref(X,j).

```
1:  cin >> x >> y;          d(x,1) d(y,1)
2:  while (x > y)           r(x,2), r(y,2)
3:    {
4:     if (x > 0)           r(x,4)
5:        cout << x;        r(x,5)
6:     x = f(x, y);         r(x,6), r(y,6), d(x,6)
7:    }
8:  cout << x;              r(x,8)
```

What kinds of tests are required for all-defs coverage?

all-uses

The *all-uses* criterion requires that each pair (def(X,i), ref(X,j)) be tested using some def-clear path from `i` to `j`.

```
1:  cin >> x >> y;          d(x,1) d(y,1)
2:  while (x > y)           r(x,2), r(y,2)
3:    {
4:     if (x > 0)           r(x,4)
5:        cout << x;        r(x,5)
6:     x = f(x, y);         r(x,6), r(y,6), d(x,6)
7:    }
8:  cout << x;              r(x,8)
```

What kinds of tests are required for all-uses coverage?


Mutation Testing

Given a program P,

- Form a set of *mutant* programs that differ from P by some single change
- These changes (called *mutation operators*) include:
  - exchanging one variable name by another
  - altering a numeric constant by some small amount
  - exchanging one arithmetic operator by another
  - exchanging one relational operator by another
  - deleting an entire statement
  - replacing an entire statement by an abort() call


Mutation Testing (cont.)

- Run P and each mutant P_i on a previously chosen set of tests
- Compare the output of each P_i to that of P
  - If the outputs differ on any test, P_i is *killed* and removed from the set of mutant programs
  - If the outputs are the same on all tests, P_i is still considered *alive*.

Mutation Testing (cont.)

A set of test data is considered *inadequate* if it cannot distinguish between the program as written (P) and programs that differ from it by only a simple change.

- So if any mutants are still alive after running a set of tests, we augment the tests until we can kill all the mutants.

Mutation Testing Problems

- Even simple programs yield tens of thousands of mutants. Executing these is time-consuming.
  - But most are killed on first few tests
  - And the process is automated
- Some mutants are actually *equivalent* to the original program:

```
X = Y;                X = Y;
if (X > 0) then       if ( Y > 0) then
  ⋮                     ⋮
```

  - Identifying these can be difficult (and cannot be automated)


Perturbation Testing

Perturbation testing (Zeil) treats each arithmetic expression $f(\bar{x})$ in the code as if it had been modified by the addition of an error term $f(\bar{v}) + e(\bar{v})$, where v are the program variables and e can be a polynomial of arbitrarily high degree (can approximate almost any error)

- Monitor the variable values actually encountered during testing
- Solve for the possible e functions that would have escaped detection on the tests done so far
  - If there are none, we're done.
  - If some exist, can generally be expressed as "coincidental" relations between variables
    - e.g., x and y have always been equal in all tests, so substitutions of one for the other could yet not have been detected

Reliability Modeling with Directed Tests

Most literature on reliability models assumes that it can only be done with representative testing, because

- Directed tests' time-to-failure is unrelated to operational time-to-failure
- Directed tests may find faults "out of order"


Order Statistic Model

Zeil & Mitchell (1996) presented a model for reliability growth under either representative or directed testing.
Assumptions:

- Software contains N faults, whose failure rates are described by a distribution F.
- Faults manifest independently
- The test process is biased towards finding faults with higher failure rates.

Measurement Process

- Fault failure rates are measured when the fault has been identified and corrected.
- Fault failure rate data is then sorted by failure rate.
- The sorted data is fitted to an *order statistic* distribution.
  - Order statistics is the study of the probability of selecting certain values when the selection process is biased.

Fitting Example