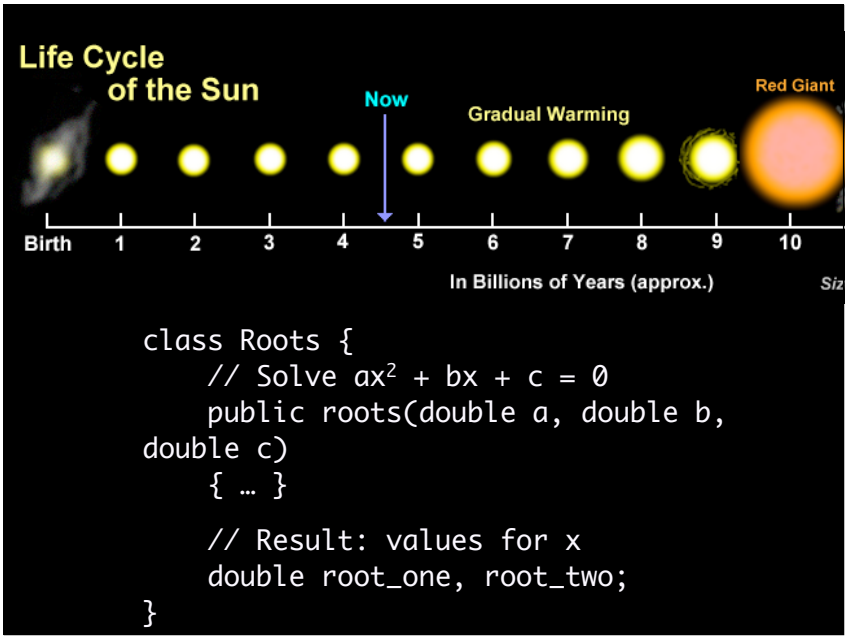
Advanced Coverage Criteria

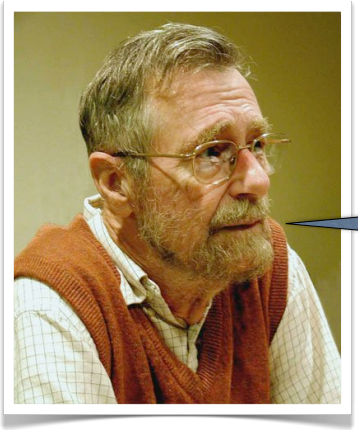
Software Engineering
Gordon Fraser • Saarland University




```
class Roots {
    // Solve ax² + bx + c = 0
    public roots(double a, double b,
double c)
    { … }

    // Result: values for x
    double root_one, root_two;
}
```

Remember the Roots example? Having a million computers doing a million tests per second would be sufficient to test the Roots example four times during the lifetime of the sun. Clearly, exhaustive testing is not feasible in practice.



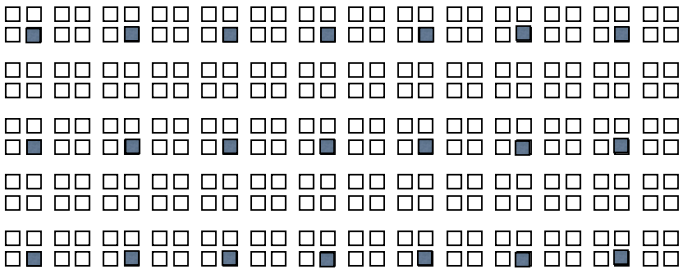
Dijkstra's law

Testing can show the *presence* but not the *absence* of errors.

Because we cannot do testing exhaustively, we can only sample test cases. Therefore, we can never be sure that our program is free of bugs. Because showing the absence of bugs is impossible, the aim of testing is to show that there are bugs. Testing is successful if we find bugs (even though this is sometimes indicated with a red light in a GUI, suggesting otherwise).

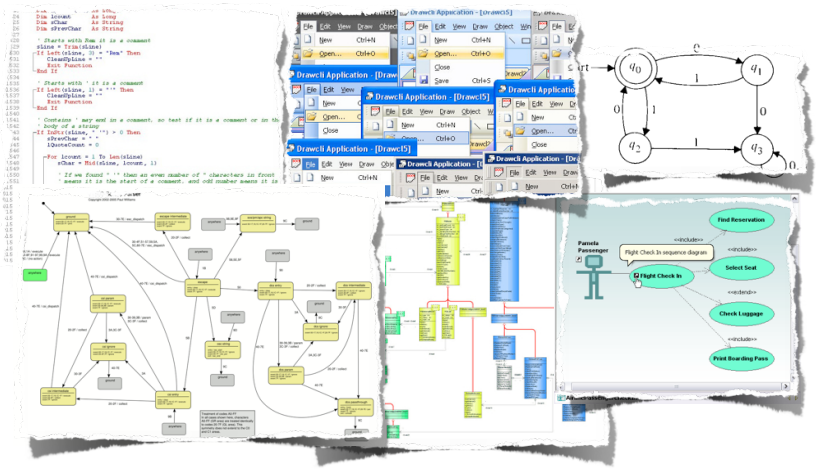
Coverage Criteria

- Possible test case



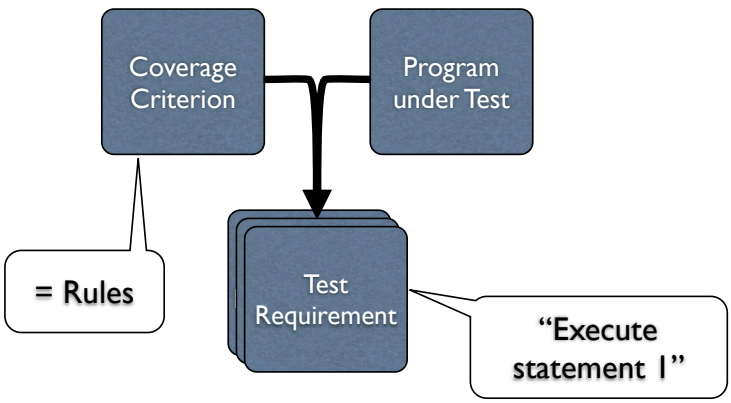
A coverage criterion describes a finite subset of test cases out of the vast/infinite number of possible tests we should execute.

Coverage of...



We can measure coverage on any artifact produced during software development, e.g., structural coverage of source code, coverage of input space, coverage of complex inputs – e.g., grammar based, coverage of specification, coverage of test models, coverage of requirements, coverage of GUI elements, ...

Coverage criteria



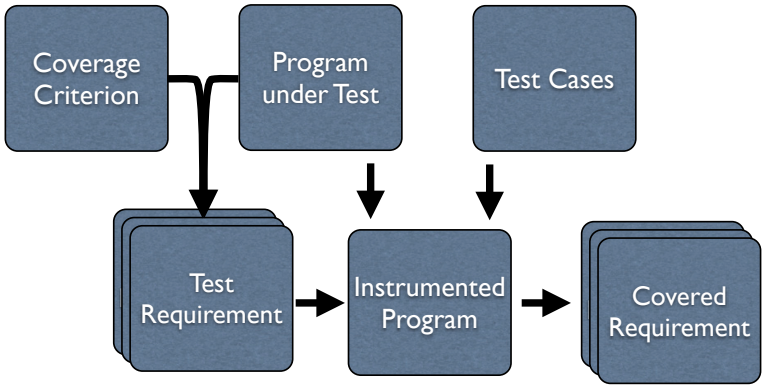
A coverage criterion can be seen as a finite set of test requirements that a test suite should fulfill. There is usually more than one way to cover a test requirement, so a coverage criterion is not a unique description of a test suite.

Using coverage criteria

1. Adequacy: Have I got enough tests?
2. Guidance: Where should I test more?
3. Automation: Generate test that satisfies a test requirement

Coverage criteria serve two main purposes: To measure adequacy of existing test suites, and to guide generation of new test cases. Even though coverage is often used to measure the quality of an existing test suite, coverage is not a good measurement for this. Generally, coverage is only good at telling you which parts haven't been covered.

Measuring Code Coverage



To make use of coverage in practice we need to measure it. This is done by instrumenting the source code with an instrument for every single test requirement, described by the coverage criterion. When test cases are run on the instrumented program the instrumentation keeps track of what has been executed and what hasn't, and so at the end of the execution we can analyze this information to point to uncovered areas and quantify the coverage.

Instrumentation

- Instrument: Additional code that does not change functional behavior but collects information

```
public int min(A, B) {
  int m = A;
  if(A>B) {
    m = B;
  }
  return m;
}
```

```
public int min(A, B) {
  int m = A;
  if(A>B) {
    Mark: "if body reached"
    m = B;
  }
  return m;
```

In general, instrumentation adds program code that does not change functional behavior but collects information. This instrumentation might, however, change other aspects of the program, such as timing, interleaving, etc.

Instrumentation

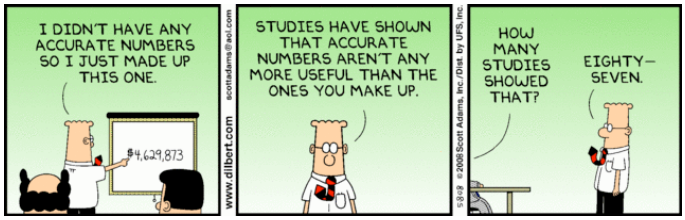
```
public int min(A, B) {
  int m = A;
  if(A>B) {
    m = B;
  }
  return m;
}
```

```
public int min(A, B) {
  statement[0]++;
  int m = A;
  statement[1]++;
  if(A>B) {
    statement[2]++;
    m = B;
  }
  statement[3]++;
  return m;
}
```

Here is an example of how to measure statement coverage: Before executing a statement we simply trace that the statement has been executed, for example by adding a static method call, incrementing a counter, etc.

Coverage Value

$$\text{Coverage value} = \frac{\#\text{ Covered test requirements}}{\#\text{ Total test requirements}}$$



Coverage is usually quantified as the percentage of test requirements satisfied. But what does that mean?

Coverage Value

$$\text{Coverage value} = \frac{\#\text{ Covered test requirements}}{\#\text{ Total test requirements}}$$

I've got 100% statement coverage on my program. How many bugs are left?

Coverage is usually quantified as the percentage of test requirements satisfied. But what does that mean?

Weyuker's Hyopthesis

The adequacy of a coverage criterion can only be intuitively defined.

...at the end of the day, we still don't know what we've achieved by satisfying a coverage criterion.

Coverage is dangerous

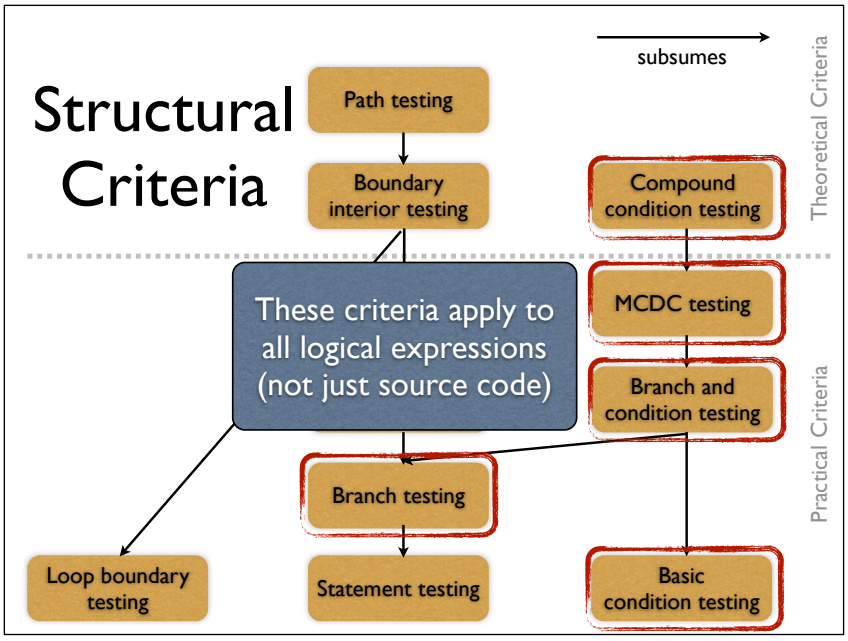
- Developers write test only to satisfy coverage
- 100% coverage can detect no faults:
  Coverage measures what is *executed*, not what is *checked*
- Coverage metrics tell you what code is not tested, but cannot accurately tell you what code is tested:
  - Low coverage means code is not well tested
  - But high coverage does not mean code is well tested

The use of coverage has some dangerous aspects, that might even reduce the quality of testing. If success is only quantified in a coverage metric, developers will get very efficient and writing test cases that satisfy the coverage goals, but not at finding bugs. Also, it is possible to cover the entire program without detecting a single bug – testing is more than just input generation (see Mutation testing lecture).

Coverage is useful

- It always tells you where you haven't tested
- Testing everything a bit is better than not testing most of the program - unless you know where the faults are
- Coverage != coverage
  Stricter criterion → more tests
- More tests = more chances of hitting bugs

Despite it's downsides coverage has some useful sides: It is very efficient at telling you which parts of a program you haven't tested at all. Intuitively, testing everything a little bit should be better than testing some aspect thoroughly and neglecting the rest – unless you already know where the bugs are (which you don't in general).



Coming back to the family of structural coverage criteria, we can see that several of these criteria focus on the logical expressions in the source code. This is worth having a closer look at, since this is a recurring problem in software testing.

Logic Coverage

Predicate

```
if(((a>b) || C) && p(x))
  o.m();
else
  o.n();
```

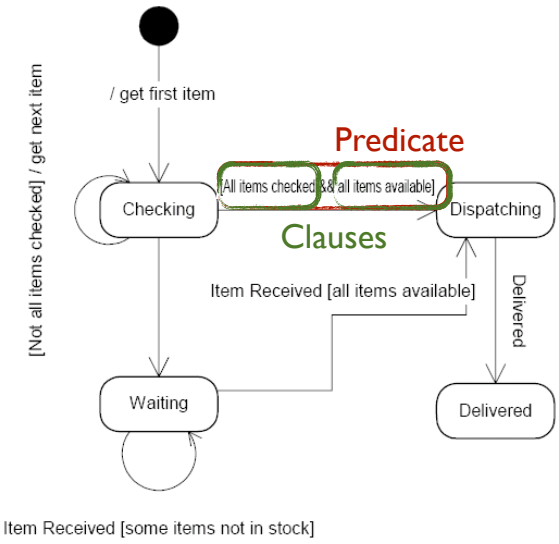
Clauses

Because the criteria we are going to consider now are independent of source code we will adopt a slightly different nomenclature, to add some confusion. A logical expression is a predicate, and the predicate consists of clauses, conjoined by boolean operators (and, or, ...). A clause contains no boolean operators.

	Education		Individual					
Education account	T	T	F	F	F	F	F	F
Current purchase > Threshold 1	–	–	F	F	T	T	–	–
Current purchase > Threshold 2	–	–	–	–	F	F	T	T
Special price < scheduled price	F	T	F	T	–	–	–	–
Special price < Tier 1	–	–	–	–	F	T	–	–
Special price < Tier 2	–	–	–	–	–	–	F	T
Out	Edu discount	Special price	No discount	Special price	Tier 1 discount	Special price	Tier 2 discount	Special Price

Clauses Predicate

Predicates and clauses occur everywhere, not only in source code. For example, test models often consist mainly of logical expressions.



UML state charts, as another example, have predicates in terms of OCL expressions.

Translating from English

"If you leave before 6:30 AM, take Braddock to 495, if you leave after 7:00 AM, take Prosperity to 50, then 50 to 495"

$$time < 6{:}30 \rightarrow path = Braddock \vee$$
$$time > 7{:}00 \rightarrow path = Prosperity$$

Incomplete!

Why is testing logical expressions important? That's because they are difficult to get right. Translating from natural language to predicates is a process that is error prone, and natural language requirements are often incomplete.

Predicate Coverage (PC)

- For each predicate:
  - Have at least one test where it evaluates to true
  - Have at least one test where it evaluates to false
- Also known as:
  - Branch coverage
  - Decision coverage
  - Basic criterion

Predicate Coverage

```
if(((a>b) || C) && p(x))
  o.m();
else
  o.n();
```

Predicate coverage is the most basic logical coverage criterion, and there are usually many different ways to satisfy it.

Predicate Coverage

	a>b	C	p(x)	((a>b)||C)&&p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

Predicate coverage is the most basic logical coverage criterion, and there are usually many different ways to satisfy it.

Predicate Coverage

	a>b	C	p(x)	((a>b)||C)&&p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

Clause does not change value

Predicate coverage is the most basic logical coverage criterion, and there are usually many different ways to satisfy it.

Clause Coverage (CC)

- For each clause in a predicate:
  - Evaluate to true
  - Evaluate to false
- Also known as:
  - Condition coverage
  - Basic condition coverage

Clause Coverage

	a>b	C	p(x)	((a>b)||C)&&p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

Clause Coverage

	a>b	C	p(x)	((a>b)||C)&&p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

Predicate is not covered

Note that clause coverage does not guarantee that predicates also evaluate to true and false.

Combinatorial Coverage (CoC)

- For each predicate:
  - All possible valuations for the clauses
- Also known as:
  - Multiple condition coverage
  - Compound condition coverage

The most thorough logical coverage criterion is combinatorial coverage (CoC). This simply requires to explore all possible combinations of truth values.

Combinatorial Coverage

	a>b	C	p(x)	((a>b)||C)&&p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

2^N tests for N clauses

In terms of our truth table this means all rows have to be executed. Note that this does not mean that the input space is completely covered: for example we only need to consider two possible outcomes for a>b.

CoC is thorough but problematic. The number of necessary tests to satisfy CoC is exponential to the number of clauses, and it is therefore not a practical criterion. To overcome this problem what we do is to

	a>b	C	p(x)	((a>b)||C)&&p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f			
8	f			

The outcome of a<b does not matter

The most interesting case for a clause is when the clause determines the outcome of the predicate. A clause determines the predicate if changing the truth value of only the clause will change the truth value of the predicate.

Clause Determination

- A clause c_i in predicate p, called the major clause, determines p if and only if the values of the remaining minor clauses c_j are such that changing c_i changes the value of p

$P = A \vee B$

if B = true, P is always true.

so if B = false, A determines P.

if A = false, B determines P.

$P = A \wedge B$

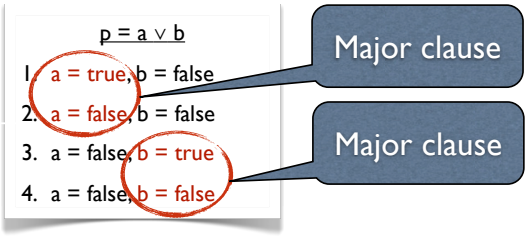
if B = false, P is always false.

so if B = true, A determines P.

if A = true, B determines P.

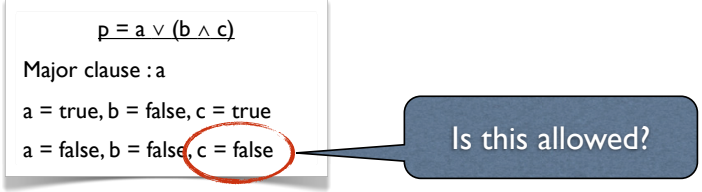
Active Clause Coverage

For each major clause ci,
choose minor clauses cj, j != i, so that ci determines p.
TR has two requirements for each ci:
ci evaluates to true and ci evaluates to false.



ACC is basically what you already know as MCDC. For each clause ACC requires that there are test cases where the clause determines the outcome of the predicate, and the clause is true and false.

Ambiguity



- This question caused confusion among testers for years
- Three separate criteria :
  - Minor clauses do not need to be the same
  - Minor clauses do need to be the same
  - Minor clauses force the predicate to become true and false

But it is not clear how exactly to interpret this. In fact, there are three different possible interpretations, and programmers have for years been uncertain which one of these versions is meant by "MCDC".

General Active Clause Coverage

- For each clause c, choose minor clauses such that c determines the predicate
- Clause c has to evaluate to true and false
- Minor clauses don't need to be the same

In the simplest case (GACC) there are no restrictions other than that the clause has to determine the predicate.

General Active Clause Coverage

Minor clauses can be different

	a>b	C	p(x)	((a>b)||C)&&p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

Correlated Active Clause Coverage

- For each clause c, choose minor clauses such that c determines the predicate
- Clause c has to evaluate to true and false
- Predicate p has to evaluate to true and false
- Minor clauses don't need to be the same
- Also known as:
  - Masking MCDC

GACC does not guarantee PC, therefore a stricter version of GACC is CACC. This criterion simply adds the PC requirement to GACC.

Restricted Active Clause Coverage

- For each clause c, choose minor clauses such that c determines the predicate
- Clause c has to evaluate to true and false
- Predicate p has to evaluate to true and false
- Minor clauses have to be the same
- Common interpretation in avionic domain
- Why keep minor clauses identical?

Finally, the third interpretation of MCDC is RACC, in which the minor clauses between two test cases for a major clause have to be identical.

CACC vs RACC

Major clause: a

	a	b	c	a && (b || c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

9 possibilities

	a	b	c	a && (b || c)
1	T	T	T	T
5	F	T	T	F
2	T	T	F	T
6	F	T	F	F
3	T	F	T	T
7	F	F	T	F

3 possibilities

It is not clear what the benefit of keeping minor clauses fixed really is, even though this has in the past been the most common interpretation. A main effect is that it makes testing much harder, so RACC is clearly preferable in practice.

Inactive Clause Coverage

- If a clause should not affect outcome (=inactive), then test whether it really doesn't
- Again, question of identical minor clauses
- Also known as:
  - Reinforced Condition/Decision Coverage (RCDC)

Inactive clause coverage is complementary to ACC, in that it tests for the cases where a clause should not affect a predicate. For example, if we activate the brakes and accelerate at the same time we want to make sure that the acceleration has no effect on the system.

General Inactive Clause Coverage

- For each clause c in predicate p choose minor clauses such that c does not determine p
  - c evaluates to true with p true
  - c evaluates to false with p true
  - c evaluates to true with p false
  - c evaluates to false with p false
  - Minor clauses may differ

There are two different flavors of ICC: With and without the requirement on fixed minor clauses.

General Inactive Clause Coverage

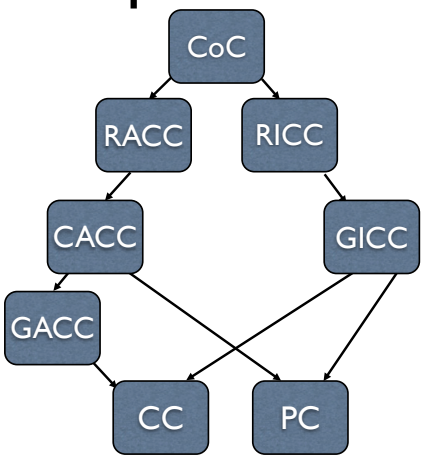
Major clause a>b

	a>b	C	p(x)	((a>b)||C)&&p
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

Restricted Inactive Clause Coverage

- For each clause c in predicate p choose minor clauses such that c does not determine p
  - c evaluates to true with p true
  - c evaluates to false with p true
  - c evaluates to true with p false
  - c evaluates to false with p false
  - Minor clauses may not differ

Subsumption Hierarchy



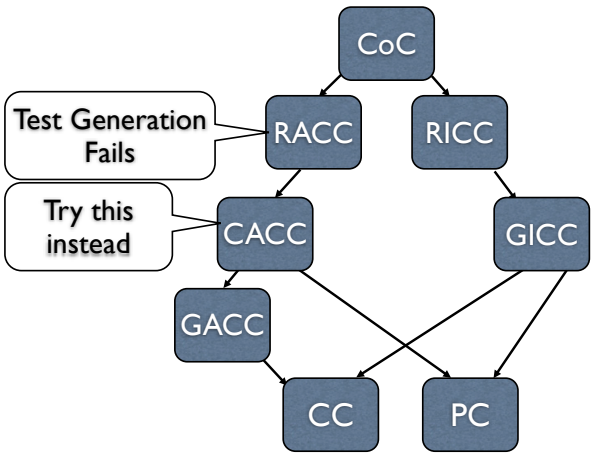
As always when defining coverage criteria, these criteria are related to each other. An arrow from one criterion to another means that the former subsumes the latter. This means that if we test for CoC, we will automatically satisfy all other coverage criteria as well.

Infeasibility

- $(a > b \wedge b > c) \vee c > a$
- $(a > b) = true, (b > c) = true, (c > a) = true$ is infeasible
- Infeasible test requirements have to be recognized and ignored
- Recognizing infeasible test requirements is hard, and in general, undecidable
- More complex criteria also produce more infeasible test requirements

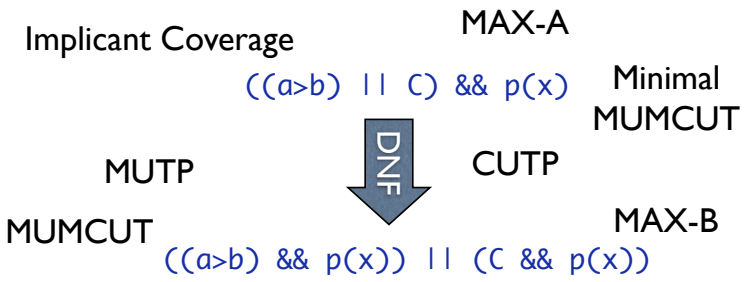
Subsumption is, however, not always given: Sometimes test requirements are simply unsatisfiable.

Best Effort Strategy



Determining infeasibility is undecidable, so what can we do in practice if we can't find a test case for a particular test requirement? A simple solution is to use a best effort approach: If, after some time, we cannot find a test case for a test requirement, we simply turn to the next simpler version of the same predicate in the subsumption hierarchy.

There's more!



Corresponding Unique True Point and Near False Point Pair Coverage (CUTPNFP)

We have seen a number of different logical coverage criteria, but there's more. For example, if you assume that predicates are given in DNF or you convert the predicates to DNF there's a whole new world of coverage possibilities. These criteria will not be subject of the course :-)

Logic Criteria on Source Code

- Predicates are derived from decision statements in programs
- In programs, most predicates have less than four clauses
- Wise programmers actively strive to keep predicates simple
- When a predicate only has one clause, COC, ACC, ICC, and CC all collapse to predicate coverage (PC)

Predicates in source code are usually simple. If a predicate consists of only one clause, all of the coverage criteria collapse to PC.

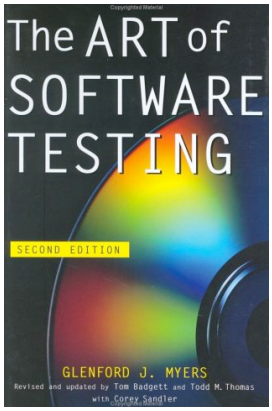
Logic Criteria on Source Code

- Reachability : Before applying the criteria on a predicate at a particular statement, we have to get to that statement
- Controllability : We have to find input values that indirectly assign values to the variables in the predicates

Generating tests for logical coverage criteria offers some challenges: First, we need the program to execute the path that leads to the predicate (reachability). Second, we need input values that indirectly assign values such that the clauses evaluate as needed. For example, for the clause a<b we need to find suitable values for a and b.

Triangle Example

"A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral."



Let's have a look at this problem in the context of a simple program. The triangle example is sort of the testing "Hello world" example, and dates back to Myers's classical book on software testing.

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

This is an example implementation of the triangle example. If one of the triangle sides is negative or the inputs don't satisfy the triangle invariant, then we return invalid (4). If they're equilateral we return 1, 2 if two sides are isosceles, and 3 if the triangle is scalene.

```
    P1
if (a <= 0 || b <= 0 || c <= 0) {
    return 4; // invalid
}   P2
if (! (a + b > c && a + c > b && b + c > a)) {
    return 4; // invalid
}   P3
if (a == b && b == c) {
    return 1; // equilateral
}   P4
if (a == b || b == c || a == c) {
    return 2; // isosceles
}
return 3;  // scalene
```

```
    P1
if (a <= 0 || b <= 0 || c <= 0) {
    return 4; // invalid
}   P2
if (! (a + b > c && a + c > b && b + c > a)) {
    return 4; // invalid
}   P3
if (a == b && b == c) {
    return 1; // equilateral
}   P4
if (a == b || b == c || a == c) {
    return 2; // isosceles
}
return 3;  // scalene
```

Always reachable

!P1

!P1 && !P2

!P1 && !P2 && !P3

These four predicates are not independent. In fact, to reach any of the predicates we depend on the outcome of the preceding predicates. This is the problem of finding the right path through the control flow graph, viewed differently.

Where's the error in the program?

Code transformation

```
if(a && b)
  S1;
else
  S2;
```

Branch:
(a,b), (!a, b)
MCDC:
(a, b), (a, !b), (!a, b)

```
if (a)
{
  if(b)
    S1;
  else
    S2;
}
else
  S2;
```

Branch coverage:
(a, b), (a, !b), (!a, b)

Branch coverage:
(a, b), (a, !b), (!a, !b)

not MCDC on other code

We can influence coverage criteria by the way we implement expressions in source code. For example, the two code snippets represent the identical code. In the first case, we need three test cases to achieve MCDC, and two to achieve branch coverage. In the second, branch coverage requires three test cases – branch coverage of the right snippet therefore means better testing then branch coverage of the left snippet.

Code transformation

```
if(a && b)
  S1;
else
  S2;
```

MCDC:
(a, b), (a, !b), (!a, b)

```
4:  iload_1 // a
5:  ifeq 28
8:  iload_2 // b
9:  ifeq 28
17: invokestatic #3; // S1
20: goto 31
28: invokestatic #4; // S2
31: ...
```

An example of when this is relevant is when coverage is measured on bytecode. This is actually sometimes required in industrial settings – but it affects the way coverage is defined and measured. In this example, coverage of the bytecode gives you less testing than coverage of the source code. (But does it make a difference?)

Logic Instrumentation

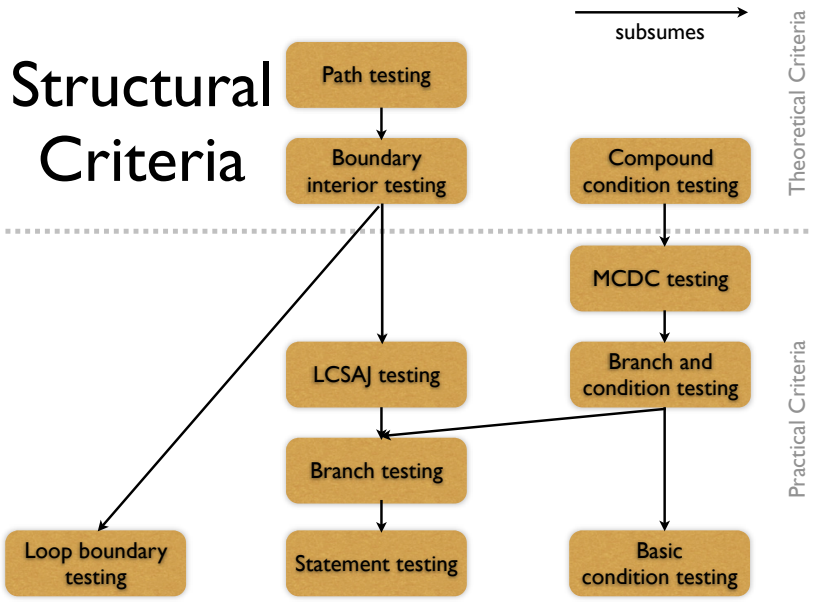
```
if(a && b) {
  // ...
}
```

```
Mark_1(a,b);
if(a && b) {
  // ...
}
```

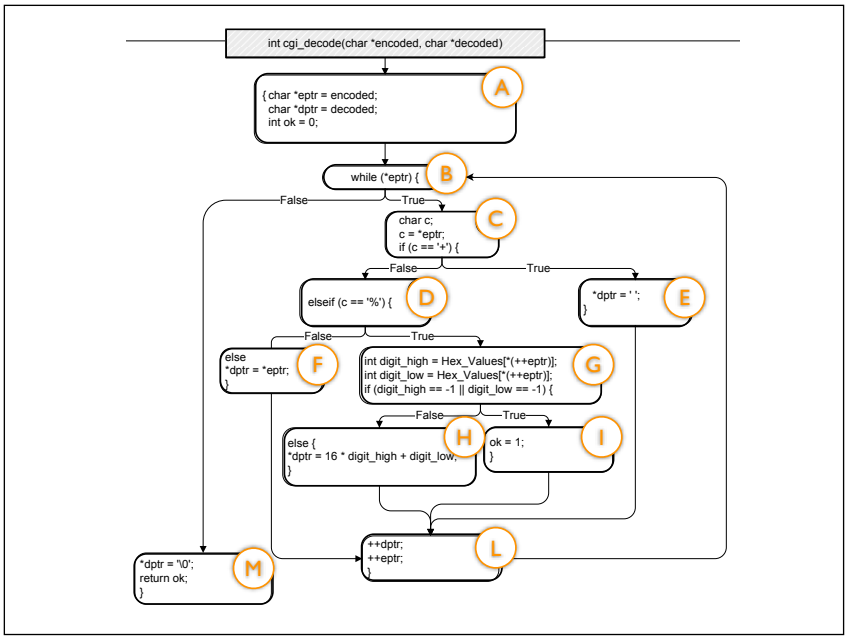
```
void Mark_1(a,b) {
  if(a)
    if(b)
      covered[1]++;
    else
      covered[2]++;
  else
    // ...
}
```

To instrument for logical criteria we need some more instrumentation. Because of short circuit operators not all clauses might be evaluated in an expression. We therefore record the values of all clauses before entering a conditional statement.

Structural Criteria



- Even if the control flow is correct...
- ...data objects might not be available
- ...silly things can be done to data objects

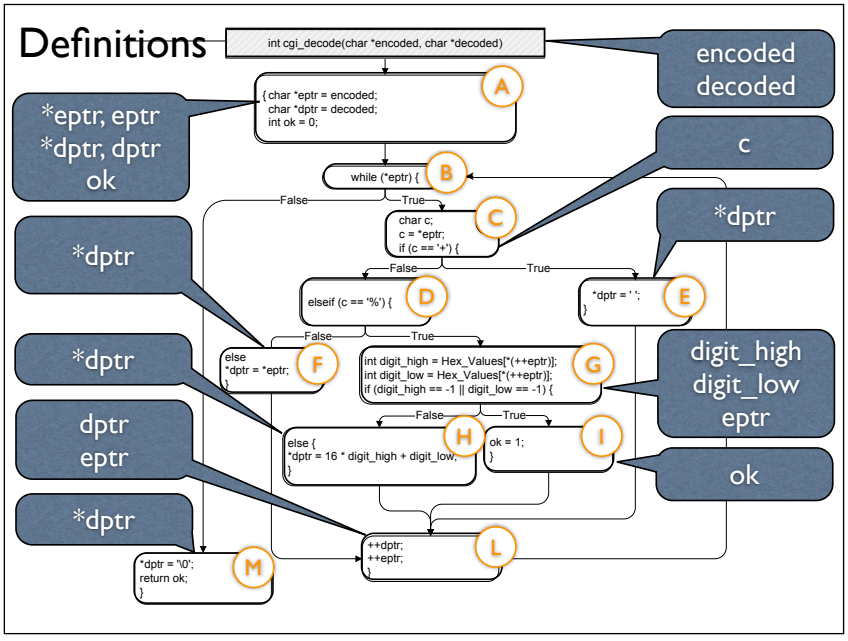


Example control flow graph

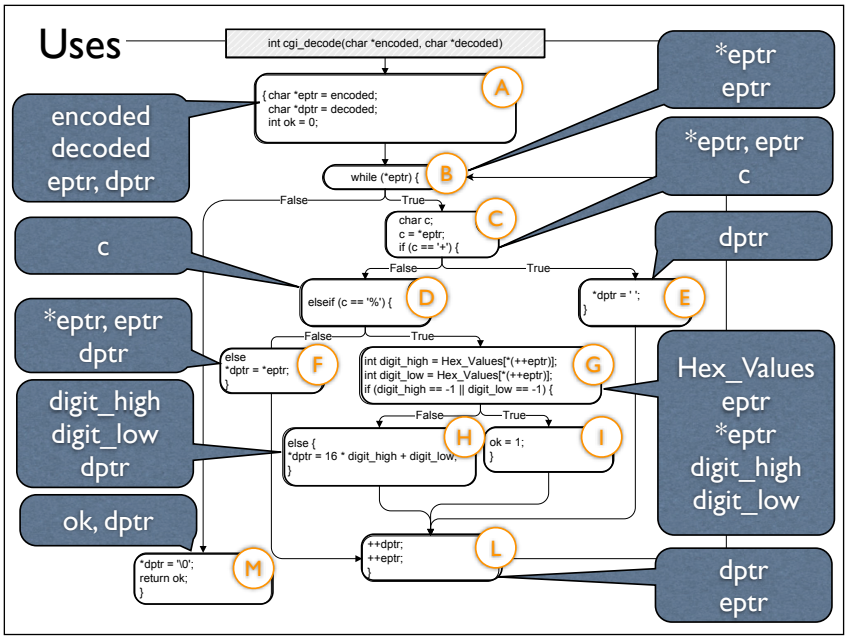
Data Flow

- Definition
  Variable declaration
  Variable initialization
  Variable assignment - left hand side of an expression
  Values received by a parameter
- Use
  Expressions
  Parameter passing
  Conditional statements
  Returns
- P-use: Predicate-use (if, while, ...)
- C-use: Computation-use (anything else)

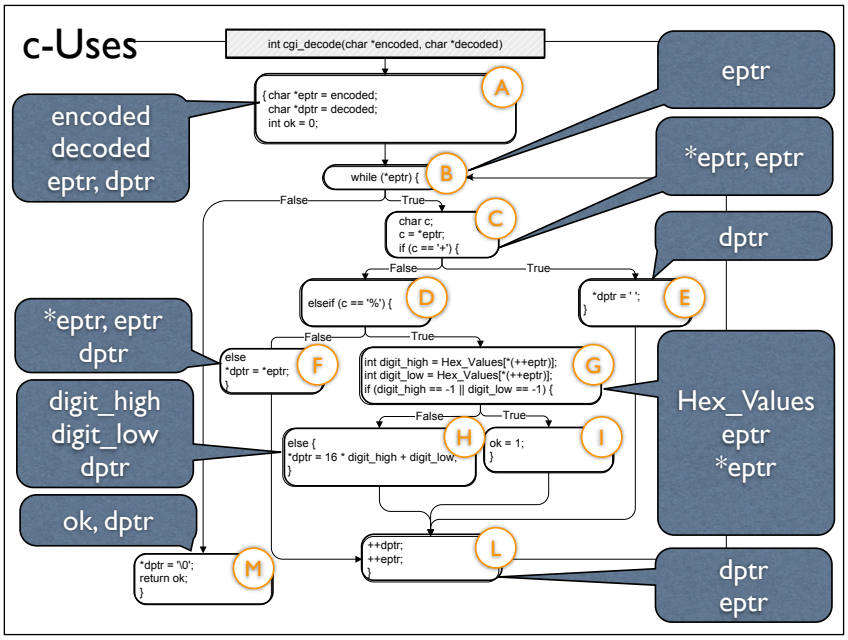
During the life time of a variable, it can be defined and used. We further distinguish between predicate and computational use of data.



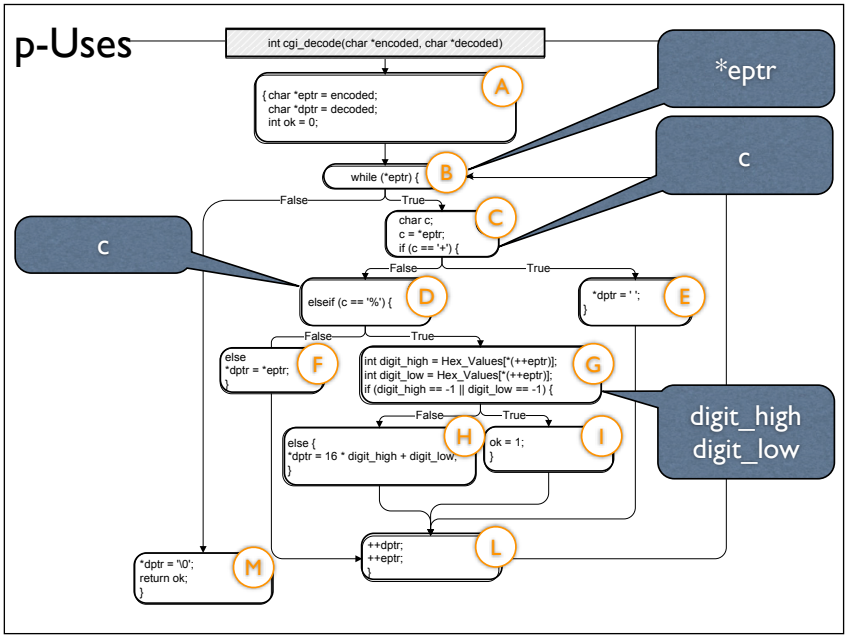
Here are all the definitions in the example control flow graph. Note that for pointers, eptr and *eptr count as two different variables.



These are all the uses in the program.



The majority of the uses are computational uses.

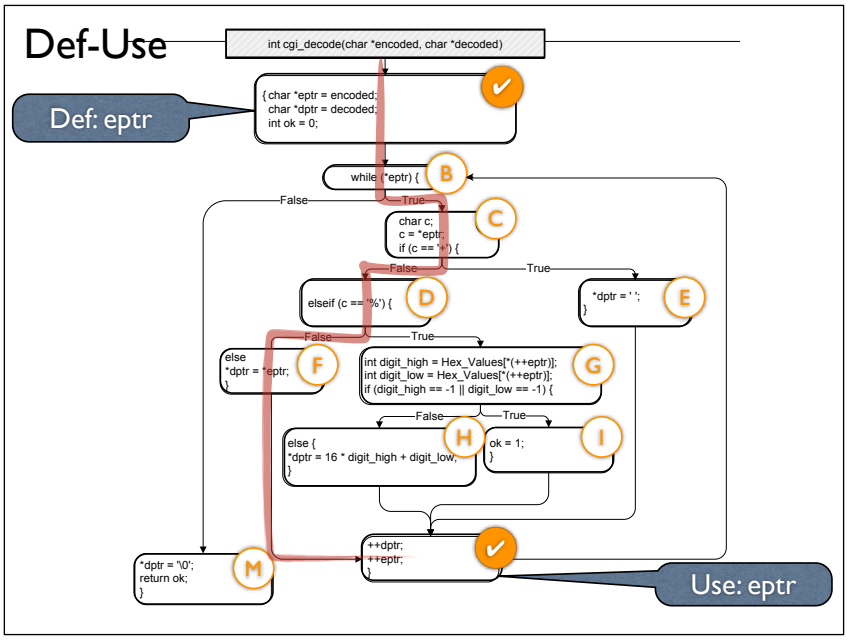


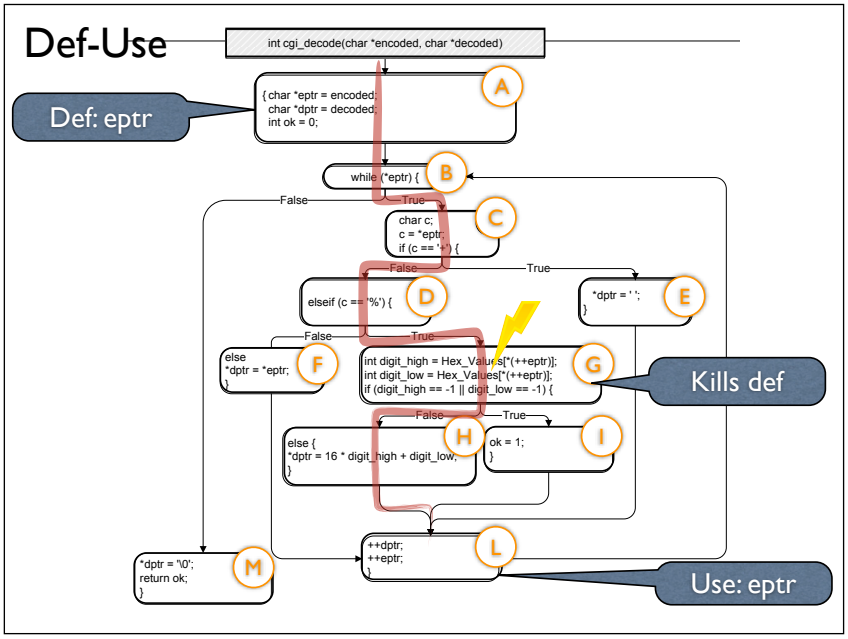
Only few uses are p-uses – they can be found in if, while, etc statements.

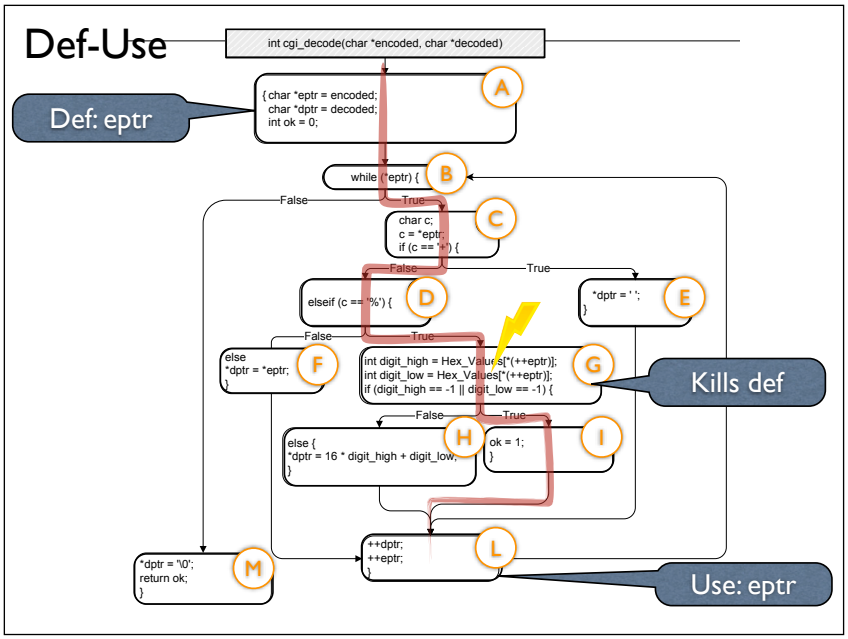
Definition-Use Pairs

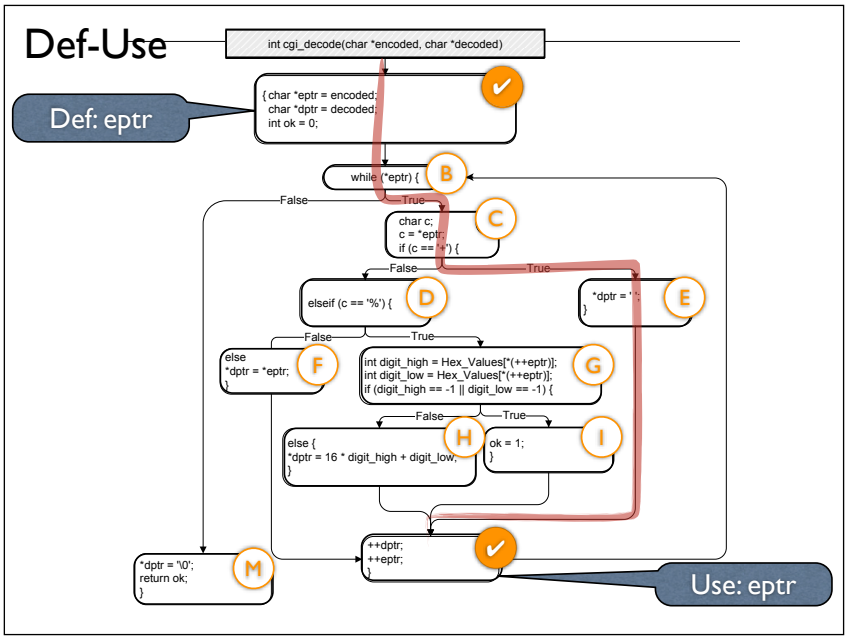
- Definition clear path
  Path from def to use without another def
- Definition-Use pair (DU pair)
  Definition+use with def clear path
- Definition-Use path (DU path)
  A DU pair can have several different definition clear paths

The main concept in data flow testing is a Definition Use pair. A definition use pair consists of a definition of a variable, a use of the same variable, and at least one definition-clear path from with the definition reaches the use without being redefined.



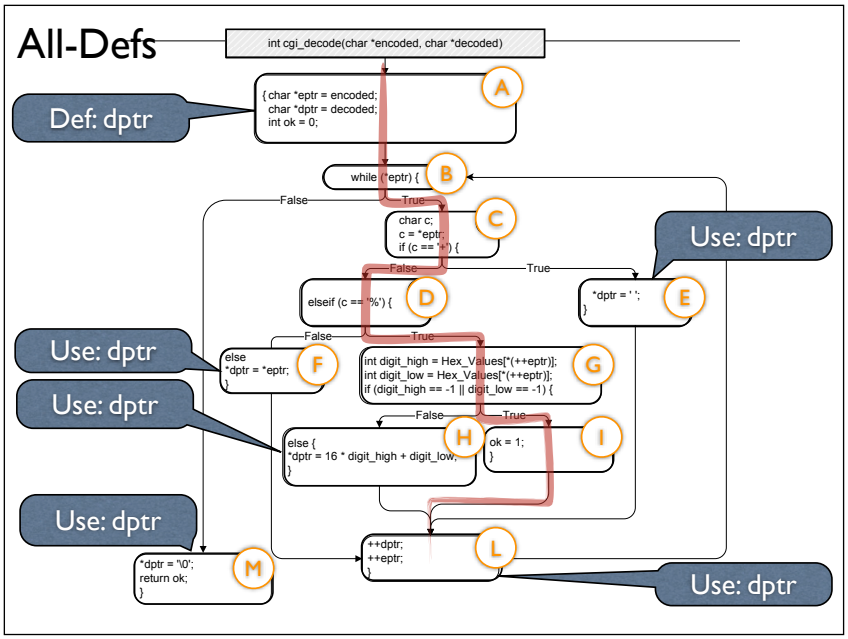






Coverage criteria

- All-definitions
  One DU path for each definition



All-Def coverage is satisfied by choosing one definition-clear path for any of the def-use pairs for each definition.

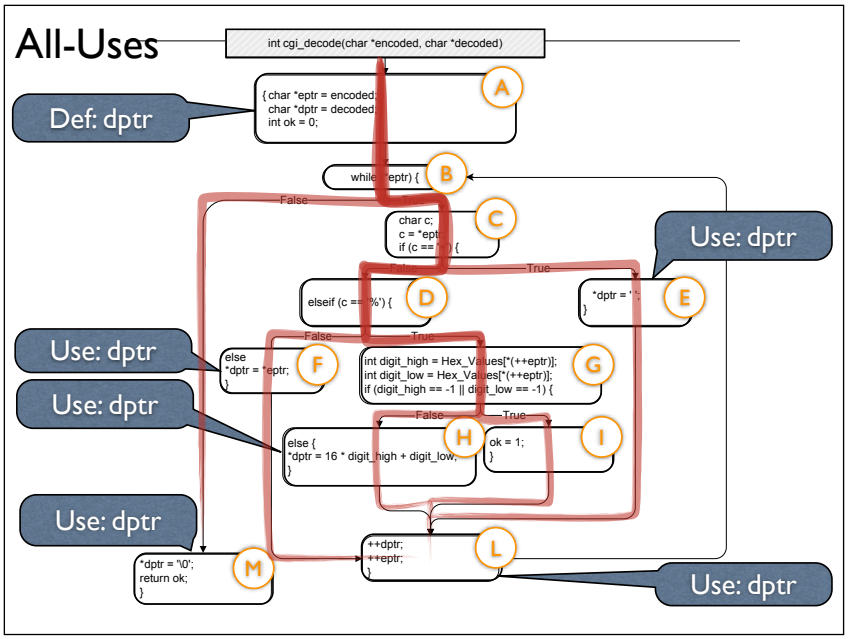
Coverage criteria

- All-definitions
  One DU path for each definition
- All-c-uses
  One DU path for each definition-c-use pair
- All-p-uses
  One DU path for each definition-p-use pair
- All-c-uses-some-p-uses
  One DU path for each definition-p-use pair
  If there is no p-use, then one c-use
- All-p-uses-some-c-uses

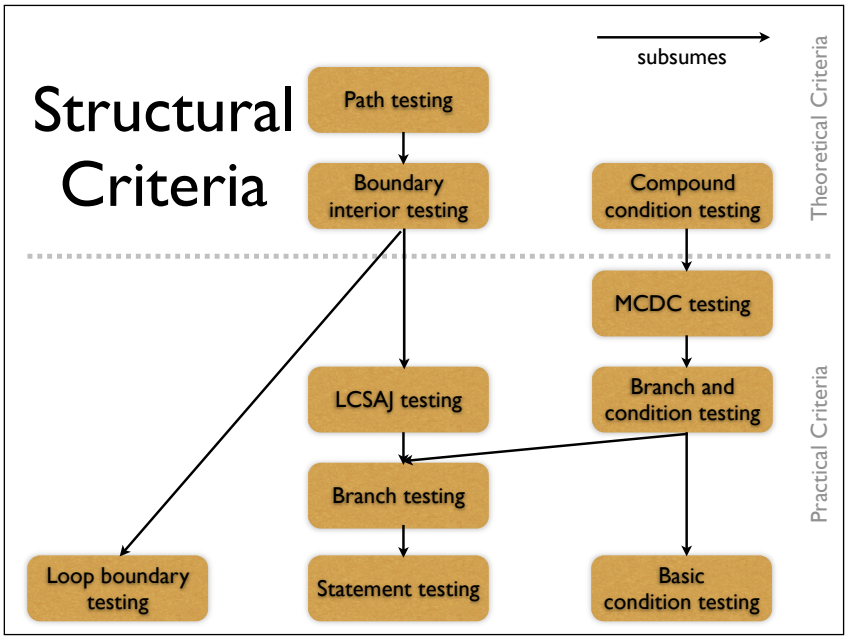
Coverage criteria

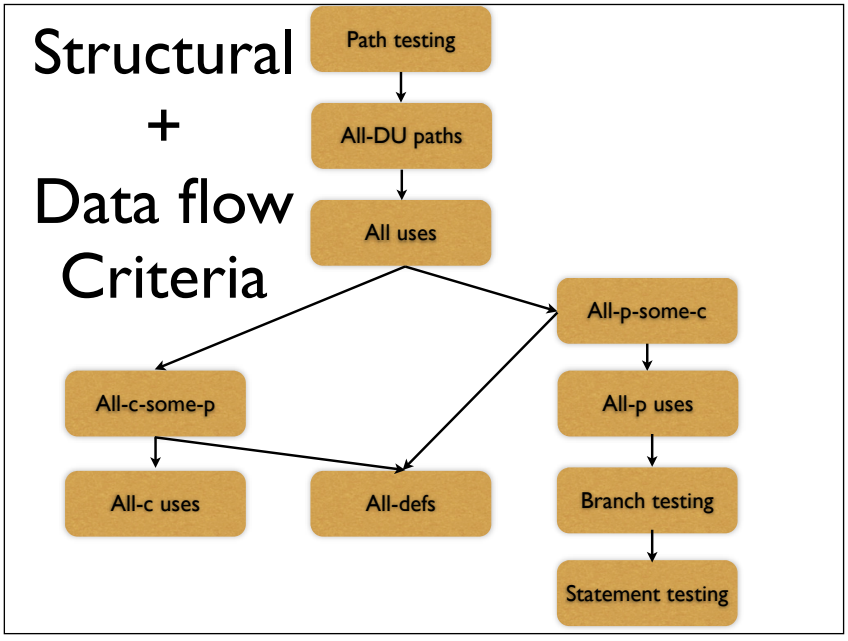
- (All-Uses)
  One DU path for each use
- All-DU-pairs
  One DU path for each def-use pair
  = All-p-uses +all-c-uses
  also known as All-Uses
- All-DU-paths
  All (simple) DU paths for each def-use pair

All-Uses is sometimes interpreted as requiring one def-use path for each use, but the definition we are sticking to is the combination of All-P-Uses and All-C-Uses.



Here are all the definition-clear paths we need for All-Uses coverage of the definition of dptr in block A.





Data-flow criteria are related to the structural criteria we already heard about.

Calculating DU pairs

- Searching all paths is not feasible
- Without loops, number of paths is exponential to number of nodes
- With loops....forget it
- →Reaching definitions

Reaching Definitions

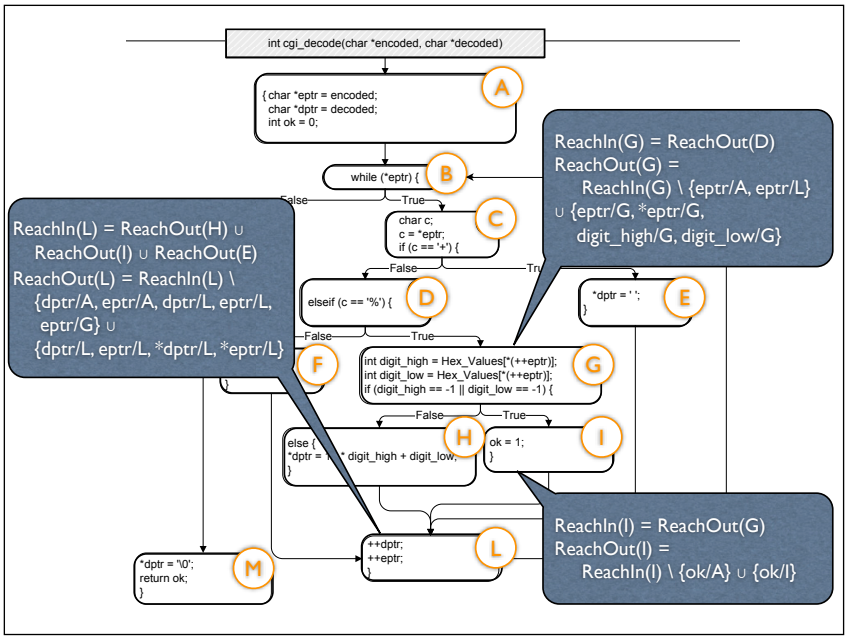
- Forward analysis
- Definition d reaches use u if there is a definition clear path from d to u

ReachIn(Node) = ReachOut(Predecessors)
ReachOut(Node) = (ReachIn(Node) \ {Killed}) ∪ {Defined}

Reaching Definitions

ReachIn(Node) = ReachOut(Predecessors)
ReachOut(Node) = (ReachIn(Node) \ {Killed}) ∪ {Defined}

- Initialize ReachOut for all nodes as {}
- working set = all nodes
- Repeat until working set is empty:
  - Pick some node and recalculate
  - If changed, add all successors to working set




```
a[i] = 13;
k = a[j];
```

Is this a DU pair?

```
a[i] = 13;
if(i == j)
  k = a[i];
else
  k = a[j];
```

We can think of the snippet as follows to identify the possible def-use pair.

```
int[] a = new int[3];
int[] b = a;
a[2] = 42;
i = b[2];
```

Is this a DU pair?

Aliasing

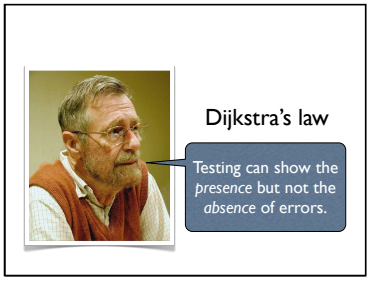
- Aliasing of variables causes serious problems!
- Working things out by hand for anything but small methods is hopeless
- Compiler-based tools help in determining all DU paths

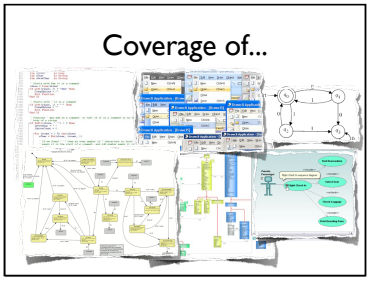
Instrumentation

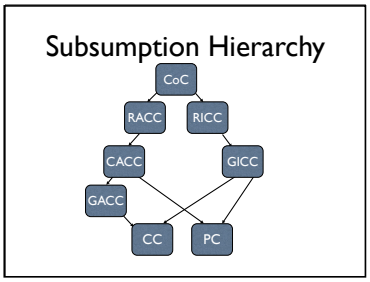
```
1: int x = 0;
2: if(a && b) {
3:   // ...
4:   y = 2*x;
5: }
```

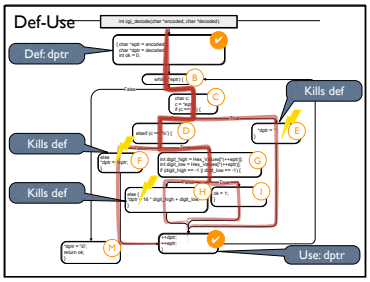
```
1: int x = 0;
   defCover[x] = 1;
2: if(a && b) {
3:   // ...
4:   y = 2*x;
     useCover[4, x,
     defCover[x]]++;
5: }
```

Instrumentation for data flow criteria is simple, but requires to instrument at two spots: First we need to keep track of all currently active definitions, and then we keep track of the actual uses that were found.


Dijkstra's law
Testing can show the *presence* but not the *absence* of errors.


Coverage of...


Subsumption Hierarchy
CoC
RACC
RICC
CACC
GICC
GACC
CC
PC


Def-Use
Def: dptr
Kills def
Kills def
Kills def
Use: dptr