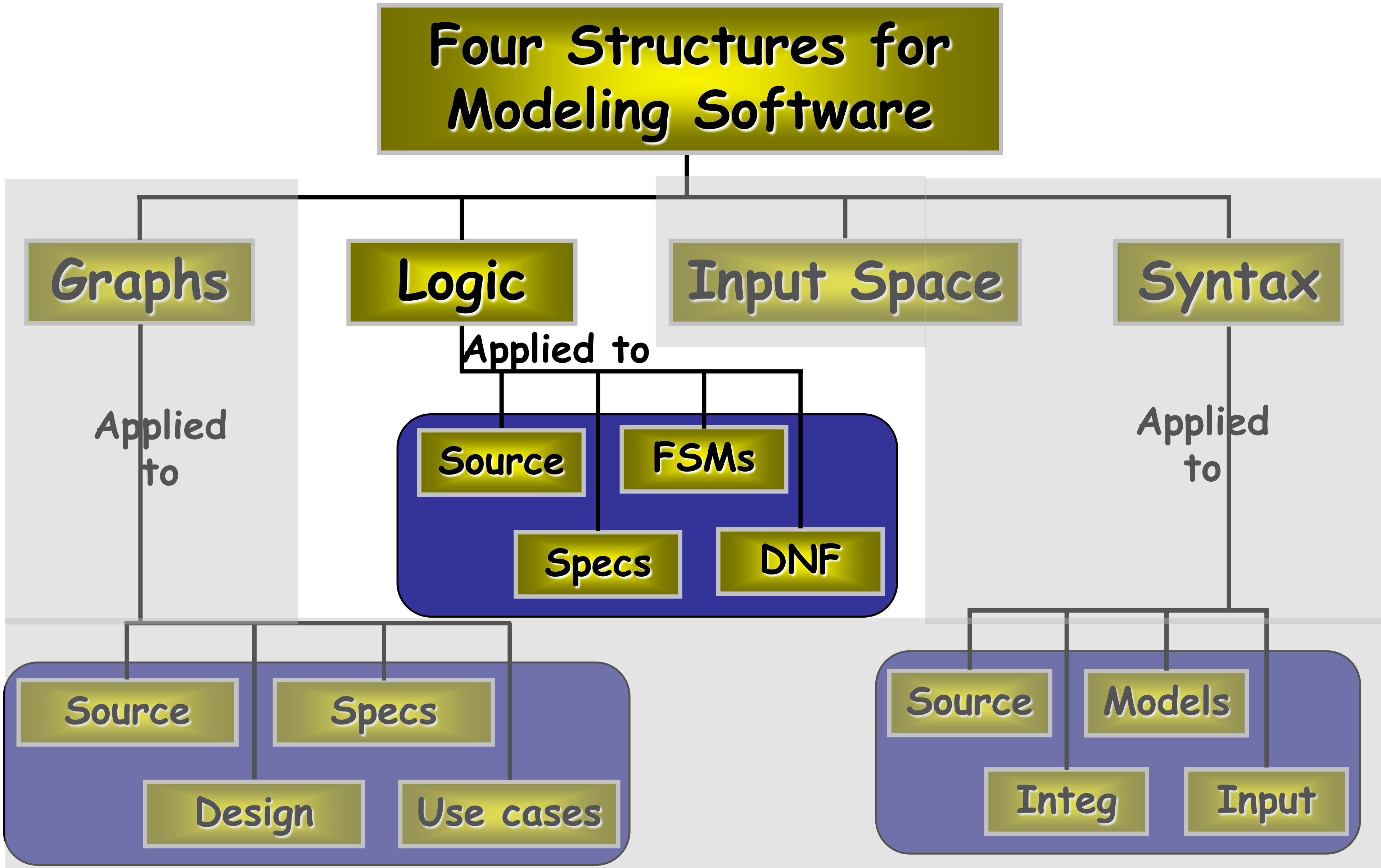
Software Engineering Testing

Logic Coverage


Four Structures for Modeling Software
Graphs
Logic
Input Space
Syntax
Applied to
Applied to
Applied to
Source
FSMs
Specs
DNF
Source
Specs
Design
Use cases
Source
Models
Integ
Input

Covering Logic Expressions

- Logic expressions show up in many situations
- Covering logic expressions is required by the US Federal Aviation Administration for safety critical software
- Logical expressions can come from many sources
  - Decisions in programs
  - FSMs and statecharts
  - Requirements
- Tests are intended to choose some subset of the total number of truth assignments to the expressions

Logic Predicates and Clauses

- A *predicate* is an expression that evaluates to a boolean value
- Predicates can contain
  - boolean variables
  - non-boolean variables that contain >, <, ==, >=, <=, !=
  - boolean function calls
- Internal structure is created by logical operators
  - $\neg$ – the *negation* operator
  - $\wedge$ – the *and* operator
  - $\vee$ – the *or* operator
  - $\rightarrow$ – the *implication* operator
  - $\oplus$ – the *exclusive or* operator
  - $\leftrightarrow$ – the *equivalence* operator
- A *clause* is a predicate with no logical operators

Examples

- $(a < b) \vee f(z) \wedge D \wedge (m >= n*o)$
- Four clauses:
  - (a < b) – relational expression
  - f (z) – boolean-valued function
  - D – boolean variable
  - (m >= n*o) – relational expression
- Most predicates have few clauses
  - It would be nice to quantify that claim!
- Sources of predicates
  - Decisions in programs
  - Guards in finite state machines
  - Decisions in UML activity graphs
  - Requirements, both formal and informal
  - SQL queries

Translating from English

- "I am interested in SWE 637 and CS 652"
- *course = swe637 OR course = cs652*

Humans have trouble translating from English to Logic

- "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"
- $time < 6{:}30 \rightarrow path = Braddock \vee time > 7{:}00 \rightarrow path = Prosperity$
- Hmm … this is incomplete !
- $time < 6{:}30 \rightarrow path = Braddock \vee time >= 6{:}30 \rightarrow path = Prosperity$

- We use predicates in testing as follows :
  - Developing a model of the software as one or more predicates
  - Requiring tests to satisfy some combination of clauses

- Abbreviations:
  - P is the set of predicates
  - p is a single predicate in P
  - C is the set of clauses in P
  - C_p is the set of clauses in predicate p
  - c is a single clause in C

Predicate and Clause Coverage

- The first (and simplest) two criteria require that each predicate and each clause be evaluated to both true and false

Predicate Coverage (PC) : For each *p* in *P*, *TR* contains two requirements: *p* evaluates to true, and *p* evaluates to false.

- **When predicates come from conditions on edges, this is equivalent to edge coverage**
- **PC does not evaluate all the clauses, so …**

Clause Coverage (CC) : For each *c* in *C*, *TR* contains two requirements: *c* evaluates to true, and *c* evaluates to false.

Predicate Coverage Example

$$((a < b) \lor D) \land (m >= n*o)$$

predicate coverage

<u>Predicate = true</u>

a = 5, b = 10, D = true, m = 1, n = 1, o = 1

$= (5 < 10) \lor true \land (1 >= 1*1)$

$= true \lor true \land TRUE$

$= true$

<u>Predicate = false</u>

a = 10, b = 5, D = false, m = 1, n = 1, o = 1

$= (10 < 5) \lor false \land (1 >= 1*1)$

$= false \lor false \land TRUE$

$= false$

Clause Coverage Example

$$((a < b) \lor D) \land (m >= n*o)$$

Clause coverage

(a < b) = true	**(a < b) = false**	**D = true**	**D = false**
a = 5, b = 10	**a = 10, b = 5**	**D = true**	**D = false**

m >= n*o = true	**m >= n*o = false**
m = 1, n = 1, o = 1	**m = 1, n = 2, o = 2**

true cases

false cases

Two tests

1) a = 5, b = 10, D = true, m = 1, n = 1, o = 1

2) a = 10, b = 5, D = false, m = 1, n = 2, o = 2

- CC does **not** subsume PC, and PC does not subsume CC
- Lets consider predicate $\mathbf{p = a \vee b}$
- Clauses C are *{a, b}*

- The four test inputs that enumerate the combinations of logical values for the clauses are:

	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Problems with PC and CC (contd.)

- Consider two test sets, each with a pair of test inputs
- Test set T_{23} = *{2, 3}* satisfies clause CC, but not PC, because p is never false
- Conversely, test set T_{24} = *{2, 4}* satisfies PC, but not CC, because b is never true
- These two test sets demonstrate that neither PC nor CC subsumes the other

Problems with PC and CC (contd.)

- From the testing perspective, we would certainly like a coverage criterion that tests individual clauses and that also tests the predicate

- The simplest solution is to test all combinations …

Combinatorial Coverage (CoC)

- CoC requires every possible combination
- Sometimes called Multiple Condition Coverage

<u>Combinatorial Coverage (CoC)</u> : For each <u>p</u> in <u>P</u>, TR has test requirements for the clauses in <u>Cp</u> to evaluate to each possible combination of truth values.

	a < b	D	m >= n*o	$((a < b) \vee D) \wedge (m >= n*o)$
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

- This is simple, neat, clean, and comprehensive …
- **But quite expensive!**
- ***2^N* tests, where *N* is the number of clauses**
  - **Impractical for predicates with more than 3 or 4 clauses**
- **The literature has lots of suggestions – some confusing**
- **The general idea is simple:**

Test each clause independently from the other clauses

- **Getting the details right is hard**
- **What exactly does "independently" mean ?**
- **We shall present this idea as "*making clauses active*" …**

Active Clauses

- Clause coverage has a <u>weakness</u> : The values do not always make a difference
- Consider the first test for clause coverage, which caused each clause to be true:
  - *(5 < 10) $\vee$ true $\wedge$ (1 >= 1*1)*
- Only the first clause ***<u>counts</u>*** !
- To really test the results of a clause, the clause should be the <u>determining factor</u> in the value of the predicate

<u>Determination</u> :	**A clause c_i in predicate *p*, called the <u>major clause</u>, <u>determines</u> *p* if and only if the values of the remaining <u>minor clauses</u> c_j are such that changing c_i changes the value of *p***

- **This is considered to *make the clause active***

Determining Predicates

$\underline{\mathbf{P = A \vee B}}$
if *B* = *true*, *p* is always true.
so if *B* = *false*, *A* determines *p*.
if *A* = *false*, *B* determines *p*.

$\underline{\mathbf{P = A \wedge B}}$
if *B* = *false*, *p* is always false.
so if *B* = *true*, *A* determines *p*.
if *A* = *true*, *B* determines *p*.

- <u>Goal</u> : Find tests for each clause when the clause determines the value of the predicate
- This is formalized in several criteria that have subtle, but very important, differences

Active Clause Coverage

<u>Active Clause Coverage (ACC)</u> : For each *p* in *P* and each major clause *ci* in *Cp*, 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.

<u>$p = a \vee b$</u>

1) **a = true, b = false**
2) **a = false, b = false**
3) **a = false, b = true**
4) **~~a = false, b = false~~**

a is major clause

b is major clause

Duplicate

- This is a form of Modified Condition/Decision Coverage software testing criterion (MCDC), which is required by the FAA for safety critical software
- <u>Ambiguity</u> : Do the minor clauses have to have the same values when the major clause is true and false?

Resolving the Ambiguity

<u>$p = a \vee (b \wedge c)$</u>

Major clause : a

a = true, b = false, c = true

a = false, b = false, c = false

Is this allowed ?

- This question caused confusion among testers for years
- Considering this carefully leads to three separate criteria :
  - Minor clauses <u>do not</u> need to be the same
  - Minor clauses <u>do</u> need to be the same
  - Minor clauses <u>force the predicate</u> to become both true and false

General Active Clause Coverage

<u>General Active Clause Coverage (GACC)</u> : For each *p* in *P* and each major clause *ci* in *Cp*, 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. The values chosen for the minor clauses *cj* do <u>not</u> need to be the same when *ci* is true as when *ci* is false, that is, *cj(ci = true) = cj(ci = false)* for all *cj* OR *cj(ci = true) != cj(ci = false) for all cj*.

- This is <u>complicated</u> !
- It is possible to satisfy GACC <u>without satisfying</u> predicate coverage
- We <u>really want</u> to cause predicates to be both true and false !

GACC does not subsume Predicate Coverage

- Lets consider predicate $\mathbf{p = a \leftrightarrow b}$
- Clause *a* determines *p* for any assignment of truth values to *b*
- So, when *a* is true, we choose *b* to be true as well, and when *a* is false, we choose *b* to be false as well
- We make the same selections for clause *b*
- We end up with only two test inputs: {*TT, FF*}
- *p* evaluates to *true* for both of these cases, so predicate coverage is *not* achieved in GACC

	a	b	$a \leftrightarrow b$
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	T

Restricted Active Clause Coverage

Restricted Active Clause Coverage (RACC) : For each *p* in *P* and each major clause *ci* in *Cp*, 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. The values chosen for the minor clauses *cj* must be the same when *ci* is true as when *ci* is false, that is, it is required that *cj(ci = true) = cj(ci = false)* for all *cj*.

- This has been a common interpretation by aviation developers
- RACC often leads to infeasible test requirements
- There is no logical reason for such a restriction

Correlated Active Clause Coverage

<u>Correlated Active Clause Coverage (CACC)</u> : For each *p* in *P* and each major clause *ci* in *Cp*, 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. The values chosen for the minor clauses *cj* must <u>cause *p* to be</u> true for one value of the major clause *ci* and false for the other, that is, it is required that *p(ci = true) != p(ci = false)*.

- A more recent interpretation
- Implicitly allows minor clauses to have different values
- Explicitly satisfies (subsumes) predicate coverage

CACC – An example

- Again, consider predicate **p = a $\leftrightarrow$ b**
- CACC can be satisfied with respect to clause *a* with the test set {*TT, FT*}
- With respect to clause *b* with the test set {*TT, TF*}
- Merging these yields the CACC test set {*TT, TF, FT*}

	a	b	a $\leftrightarrow$ b
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	T

CACC and RACC

	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

	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

major clause

major clause

CACC can be satisfied by choosing any of rows 1, 2, 3 AND any of rows 5, 6, 7 – a total of nine pairs

RACC can only be satisfied by one of the three pairs above

Inactive Clause Coverage

- The active clause coverage criteria ensure that "major" clauses <u>do affect</u> the predicates
- Inactive clause coverage takes the opposite approach – major clauses <u>do not affect</u> the predicates

<u>Inactive Clause Coverage (ICC)</u> : For each *p* in *P* and each major clause *ci* in *Cp*, choose minor clauses *cj*, *j != i*, so that *ci* <u>does not</u> determine *p*. TR has <u>four</u> requirements for each *ci*: (1) *ci* evaluates to true with *p* true, (2) *ci* evaluates to false with *p* true, (3) *ci* evaluates to true with *p* false, and (4) *ci* evaluates to false with *p* false.

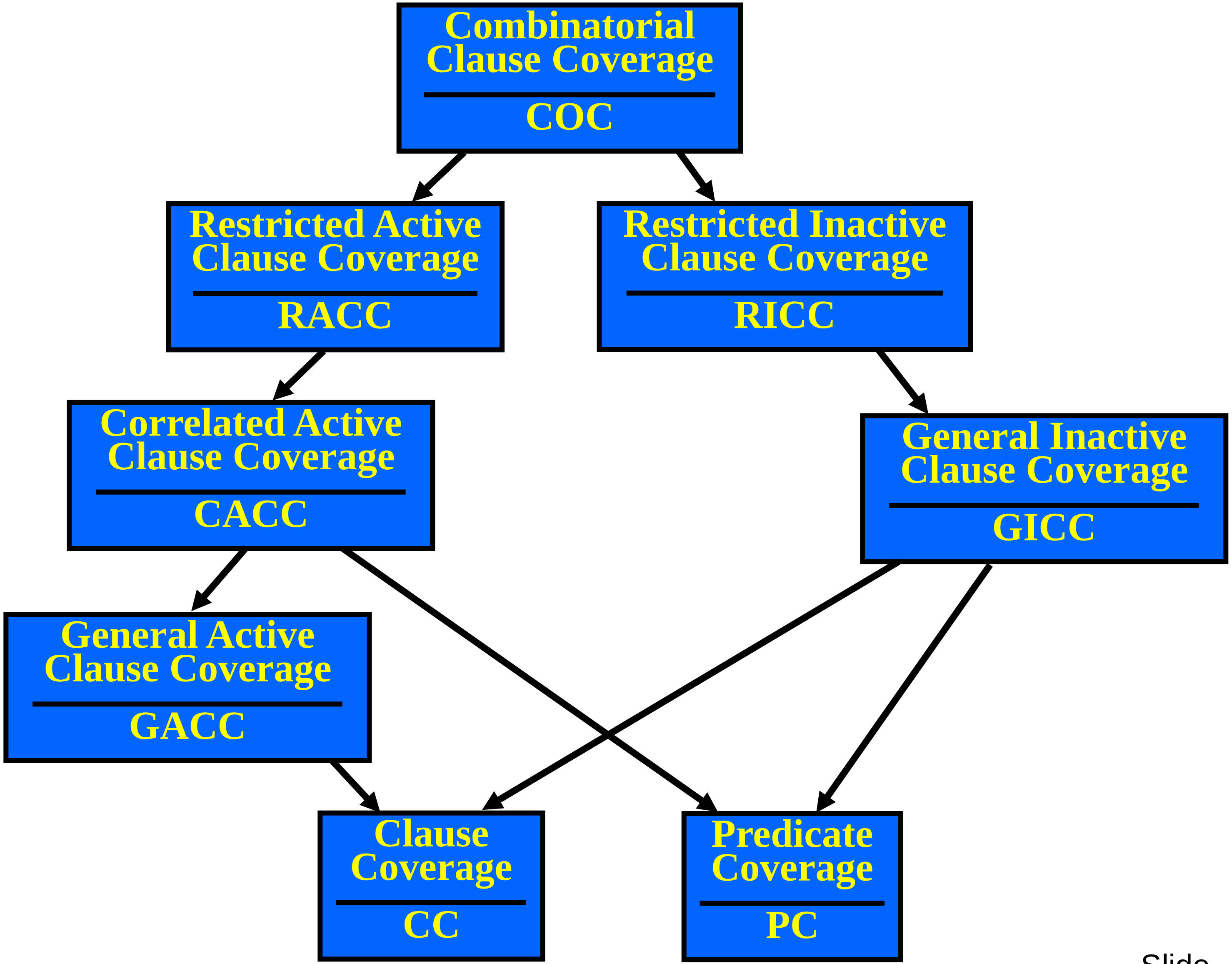
General and Restricted ICC

- Unlike ACC, the notion of correlation is not relevant
  – ci does not determine p, so cannot correlate with p
- Predicate coverage is always guaranteed

General Inactive Clause Coverage (GICC) : For each *p* in *P* and each major clause *ci* in *Cp*, choose minor clauses *cj*, *j != i*, so that *ci* does not determine *p*. The values chosen for the minor clauses *cj* do not need to be the same when *ci* is true as when *ci* is false, that is, *cj(ci = true) = cj(ci = false)* for all *cj* OR *cj(ci = true) != cj(ci = false)* for all *cj*.

Restricted Inactive Clause Coverage (RICC) : For each *p* in *P* and each major clause *ci* in *Cp*, choose minor clauses *cj*, *j != i*, so that *ci* does not determine *p*. The values chosen for the minor clauses *cj* must be the same when *ci* is true as when *ci* is false, that is, it is required that *cj(ci = true) = cj(ci = false)* for all *cj*.

Logic Coverage Criteria Subsumption



Making Clauses Determine a Predicate

- Finding values for minor clauses c_j is easy for simple predicates
- But how to find values for more complicated predicates ?
- Definitional approach:
  - $p_{c=true}$ is predicate *p* with every occurrence of *c* replaced by *true*
  - $p_{c=false}$ is predicate *p* with every occurrence of *c* replaced by *false*
- To find values for the minor clauses, connect $p_{c=true}$ and $p_{c=false}$ with exclusive *OR*

$$p_c = p_{c=true} \oplus p_{c=false}$$

- After solving, p_c describes exactly the values needed for *c* to determine *p*

Examples

$\underline{p = a \vee b}$

$$
\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \vee b) \oplus (false \vee b) \\
&= true \oplus b \\
&= \neg b
\end{aligned}
$$

$\underline{p = a \wedge b}$

$$
\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \wedge b) \oplus (false \wedge b) \\
&= b \oplus false \\
&= b
\end{aligned}
$$

$\underline{p = a \vee (b \wedge c)}$

$$
\begin{aligned}
p_a &= p_{a=true} \oplus p_{a=false} \\
&= (true \vee (b \wedge c)) \oplus (false \vee (b \wedge c)) \\
&= true \oplus (b \wedge c) \\
&= \neg (b \wedge c) \\
&= \neg b \vee \neg c
\end{aligned}
$$

- **"*NOT b* $\vee$ *NOT c*" means either *b* or *c* can be false**
- **RACC requires the same choice for both values of *a*, CACC does not**

- The definitions in this discussion yield the same tests no matter how the predicate is expressed
- $(a \lor b) \land (c \lor b) == (a \land c) \lor b$
- $(a \land b) \lor (b \land c) \lor (a \land c)$
  - Only has 8 possible tests, not 64
- Use the simplest form of the predicate, and ignore contradictory truth table assignments

A More Subtle Example

$$\underline{p = (a \wedge b) \vee (a \wedge \neg b)}$$

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} \\ &= ((true \wedge b) \vee (true \wedge \neg b)) \oplus ((false \wedge b) \vee (false \wedge \neg b)) \\ &= (b \vee \neg b) \oplus false \\ &= true \oplus false \\ &= true \end{aligned}$$

$$\underline{p = (a \wedge b) \vee (a \wedge \neg b)}$$

$$\begin{aligned} p_b &= p_{b=true} \oplus p_{b=false} \\ &= ((a \wedge true) \vee (a \wedge \neg true)) \oplus ((a \wedge false) \vee (a \wedge \neg false)) \\ &= (a \vee false) \oplus (false \vee a) \\ &= a \oplus a \\ &= false \end{aligned}$$

- ***a*** **always determines the value of this predicate**
- ***b*** **never determines the value –** ***b*** **is irrelevant !**

Infeasible Test Requirements

- Consider the predicate:

$$(a > b \wedge b > c) \vee c > a$$

- *(a > b) = true, (b > c) = true, (c > a) = true* is infeasible
- As with graph-based criteria, infeasible test requirements have to be recognized and ignored
- Recognizing infeasible test requirements is hard, and in general, undecidable
- Software testing is inexact – engineering, not science

Logic Coverage Summary – so far

- Predicates are often very simple—in practice, most have less than 3 clauses
  - In fact, most predicates only have one clause !
  - With only clause, PC is enough
  - With 2 or 3 clauses, CoC is practical
  - Advantages of ACC and ICC criteria significant for large predicates
    - CoC is impractical for predicates with many clauses
- Control software often has many complicated predicates, with lots of clauses
- Question … why don't complexity metrics count the number of clauses in predicates?

Logic Expressions from Programs

Logic Expressions from Programs

- 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)
- Applying logic criteria to program source is hard because of reachability and controllability:
  - *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
  - Variables in the predicates that are not inputs to the program are called *internal variables*
- These issues are illustrated through an example in the following slides …

Triang *(pg 1 of 5)*

```
// Jeff Offutt -- Java version Feb 2003
// The old standby: classify triangles
// Figures 3.2 and 3.3 in the book.
import java.io.*;
class trityp
{
   private static String[] triTypes = { "",    // Ignore 0.
           "scalene", "isosceles", "equilateral", "not a valid
           triangle"};
   private static String instructions = "This is the ancient
   TriTyp program.\nEnter three integers that represent the lengths
   of the sides of a triangle.\nThe triangle will be categorized as
   either scalene, isosceles, equilateral\nor invalid.\n";

public static void main (String[] argv)
{  // Driver program for trityp
   int A, B, C;
   int T;
```

Triang *(pg 2 of 5)*

```
    System.out.println (instructions);
    System.out.println ("Enter side 1: ");
    A = getN();
    System.out.println ("Enter side 2: ");
    B = getN();
    System.out.println ("Enter side 3: ");
    C = getN();
    T = Triang (A, B, C);

    System.out.println ("Result is: " + triTypes [T]);
}

// ====================================
```

Triang *(pg 3 of 5)*

```
// The main triangle classification method
private static int Triang (int Side1, int Side2, int Side3)
{
   int tri_out;

   // tri_out is output from the routine:
   //    Triang = 1 if triangle is scalene
   //    Triang = 2 if triangle is isosceles
   //    Triang = 3 if triangle is equilateral
   //    Triang = 4 if not a triangle

   // After a quick confirmation that it's a legal
   // triangle, detect any sides of equal length
   if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
   {
      tri_out = 4;
      return (tri_out);
   }
```

Triang *(pg 4 of 5)*

```
tri_out = 0;
if (Side1 == Side2)
   tri_out = tri_out + 1;
if (Side1 == Side3)
   tri_out = tri_out + 2;
if (Side2 == Side3)
   tri_out = tri_out + 3;
if (tri_out == 0)
{  // Confirm it’s a legal triangle before declaring
   // it to be scalene

   if (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
       Side1+Side3 <= Side2)
      tri_out = 4;
   else
      tri_out = 1;
   return (tri_out);
}
```

Triang *(pg 5 of 5)*

```
    /* Confirm it's a legal triangle before declaring  */
    /* it to be isosceles or equilateral  */

    if (tri_out > 3)
       tri_out = 3;
    else if (tri_out == 1 && Side1+Side2 > Side3)
      tri_out = 2;
    else if (tri_out == 2 && Side1+Side3 > Side2)
      tri_out = 2;
    else if (tri_out == 3 && Side2+Side3 > Side1)
      tri_out = 2;
    else
      tri_out = 4;
    return (tri_out);
} // end Triang
```

Ten Triang Predicates

42: (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)

49: (Side1 == Side2)

51: (Side1 == Side3)

53: (Side2 == Side3)

55: (triOut == 0)

59: (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
Side1+Side3 <= Side2)

70: (triOut > 3)

72: (triOut == 1 && Side1+Side2 > Side3)

74: (triOut == 2 && Side1+Side3 > Side2)

76: (triOut == 3 && Side2+Side3 > Side1)

Reachability for Triang Predicates

42: True

49: P1 = s1>0 && s2>0 && s3>0

51: P1

53: P1

55: P1

59: P1 && triOut = 0

62: P1 && triOut = 0
&& (s1+s2 > s3) && (s2+s3 > s1) && (s1+s3 > s2)

70: P1 && triOut != 0

72: P1 && triOut != 0 && triOut <= 3

74: P1 && triOut != 0 && triOut <= 3 && (triOut !=1 || s1+s2<=s3)

76: P1 && triOut != 0 && triOut <= 3 && (triOut !=1 || s1+s2<=s3)
&& (triOut !=2 || s1+s3<=s2)

78: P1 && triOut != 0 && triOut <= 3 && (triOut !=1 || s1+s2<=s3)
&& (triOut !=2 || s1+s3 <= s2) && (triOut !=3 || s2+s3 <= s1)

Need to solve for the <u>internal variable</u> *triOut*

Solving for Internal Variable *triOut*

At line 55, triOut has a value in the range (0 .. 6)

triOut =	**0**	**s1!=s2**	**&&**	**s1!=s3**	**&&**	**s2!=s3**	
	1	**s1=s2**	**&&**	**s1!=s3**	**&&**	**s2!=s3**	
	2	**s1!=s2**	**&&**	**s1=s3**	**&&**	**s2!=s3**	
	3	**s1!=s2**	**&&**	**s1!=s3**	**&&**	**s2=s3**	
	4	**s1=s2**	**&&**	**s1!=s3**	**&&**	**s2=s3**	***Contradiction***
	5	**s1!=s2**	**&&**	**s1=s3**	**&&**	**s2=s3**	***Contradiction***
	6	**s1=s2**	**&&**	**s1=s3**	**&&**	**s2=s3**	

Reachability for Triang Predicates
(solved for triOut – reduced)

Looks complicated, but a lot of redundancy

42: True

49: P1 = s1>0 && s2>0 && s3>0

51: P1

53: P1

55: P1

59: P1 && s1 != s2 && s2 != s3 && s2 != s3 (triOut = 0)

62: P1 && s1 != s2 && s2 != s3 && s2 != s3 (triOut = 0)
&& (s1+s2 > s3) && (s2+s3 > s1) && (s1+s3 > s2)

70: P1 && P2 = (s1=s2 || s1=s3 || s2=s3) (triOut != 0)

72: P1 && P2 && P3 = (s1!=s2 || s1!=s3 || s2!=s3) (triOut <= 3)

74: P1 && P2 && P3 && (s1 != s2 || s1+s2<=s3)

76: P1 && P2 && P3 && (s1 != s2 || s1+s2<=s3)
&& (s1 != s3 || s1+s3<=s2)

78: P1 && P2 && P3 && (s1 != s2 || s1+s2<=s3)
&& (s1 != s3 || s1+s3<=s2) && (s2 != s3 || s2+s3<=s1)

Predicate Coverage

These values are "don't care", needed to complete the test.

	T				F			
	A	B	C	EO	A	B	C	EO
p42: (S1 <= 0 || S2 <= 0 || S3 <= 0)	0	0	0	4	1	1	1	3
p49: (S1 == S2)	1	1	1	3	1	2	2	2
p51: (S1 == S3)	1	1	1	3	1	2	2	2
p53: (S2 == S3)	1	1	1	3	2	1	2	2
p55: (triOut == 0)	1	2	3	4	1	1	1	3
p59: (S1+S2 <= S3 || S2+S3 <= S1 || S1+S3 <= S2)	1	2	3	4	2	3	4	1
p70: (triOut > 3)	1	1	1	3	2	2	3	2
p72: (triOut == 1 && S1+S2 > S3)	2	2	3	2	2	2	4	4
p74: (triOut == 2 && S1+S3 > S2)	2	3	2	2	2	4	2	4
p76: (triOut == 3 && S2+S3 > S1)	3	2	2	2	4	2	2	4

Clause Coverage

	T A	T B	T C	T EO	F A	F B	F C	F EO
p42: (S1 <= 0)	0	1	1	4	1	1	1	3
(S2 <= 0)	1	0	1	4	1	1	1	3
(S3 <= 0)	1	1	0	4	1	1	1	3
p59: (S1+S2 <= S3)	2	3	6	4	2	3	4	1
(S2+S3 <= S1)	6	2	3	4	2	3	4	1
(S1+S3 <= S2)	2	6	3	4	2	3	4	1
p72: (triOut == 1)	2	2	3	2	2	3	2	2
(S1+S2 > S3)	2	2	3	2	2	2	5	4
p74: (triOut == 2)	2	3	2	2	3	2	2	2
(S1+S3 > S2)	2	3	2	2	2	5	2	4
p76: (triOut == 3)	3	2	2	2	1	2	1	4
(S2+S3 > S1)	3	2	2	2	5	2	2	4

Correlated Active Clause Coverage

					A	B	C	EO
p42: (S1 <= 0 || S2 <= 0 || S3 <= 0)	T	f	f		0	1	1	4
	F	f	f		1	1	1	3
	f	T	f		1	0	1	4
	f	f	T		1	1	0	4
p59: (S1+S2 <= S3 || S2+S3 <= S1 || S1+S3 <= S2)	T	f	f		2	3	6	4
	F	f	f		2	3	4	1
	f	T	f		6	2	3	4
	f	f	T		2	6	3	4
p72: (triOut == 1 && S1+S2 > S3)	T	t			2	2	3	2
s1=s2 && s1!=s3 && s2!=s3	F	t			2	3	3	2
	t	F			2	2	5	4
p74: (triOut == 2 && S1+S3 > S2)	T	t			2	3	2	2
s1!=s2 && s1=s3 && s2!=s3	F	t			2	3	3	2
	t	F			2	5	2	4
p76: (triOut == 3 && S2+S3 > S1)	T	t			3	2	2	2
s1!=s2 && s1!=s3 && s2=s3	F	t			1	2	2	4
	t	F			5	2	2	4

At least one pair of sides must be equal.

Program Transformation Issues

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

Transform (1)?

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

Transform (2)?

```
d = a && b;
e = d || c;
if (e) {
    S1;
}
else {
    S2;
}
```

Problems with Transformed Programs

- Maintenance is certainly harder with Transform (1)
  - Not recommended!
- Coverage on Transform (1)
  - PC on transform does not imply CACC on original
  - CACC on original does not imply PC on transform
- Coverage on Transform (2)
  - Structure used by logic criteria is "lost"
  - Hence CACC on transform 2 only requires 3 tests
  - Note: Mutation analysis (Chapter 5) addresses this problem
- Bottom Line: Logic coverage criteria are there to help you!

a	b	c	$(a \wedge b) \vee c$	CACC	PC	CACC(2)
T	T	T	T		X	
T	T	F	T	X		X
T	F	T	T	X	X	X
T	F	F	F	X	X	
F	T	T	T		X	
F	T	F	F	X		X
F	F	T	T			
F	F	F	F		X	

Summary : Logic Coverage for Source Code

- Predicates appear in decision statements
  - if, while, for, etc.
- Most predicates have less than four clauses
  - But some applications have predicates with many clauses
- The hard part of applying logic criteria to source is resolving the internal variables
- Non-local variables (class, global, etc.) are also input variables if they are used
- If an input variable is changed within a method, it is treated as an internal variable thereafter
- To maximize effect of logic coverage criteria:
  - Avoid transformations that hide predicate structure

Specifications in Software

- Specifications can be formal or informal
  - Formal specs are usually expressed mathematically
  - Informal specs are usually expressed in *natural language*
- Lots of formal languages and informal styles are available

- Most specification languages include explicit logical expressions, so it is very easy to apply logic coverage criteria

- Implicit logical expressions in natural-language specifications should be re-written as explicit logical expressions as part of test design
  - You will often find mistakes
- One of the most common is preconditions …

- Programmers often include preconditions for their methods
- The preconditions are often expressed in comments in method headers
- Preconditions can be in javadoc, “requires”, “pre”, …

Example – Saving addresses

// name must not be empty

// state must be valid

// zip must be 5 numeric digits

// street must not be empty

// city must not be empty

Rewriting to logical expression

name != “” $\wedge$ state in stateList $\wedge$ zip >= 00000 $\wedge$ zip <= 99999 $\wedge$ street != “” $\wedge$ city != “”

Shortcut for Conjunctive Clauses

- Conjunctive clauses are connected only by the and operator
- Each major clause is made active by making all other clauses true
- The tests are "all true" and then a "diagonal" of false values:

	A	B	C	…
1	T	T	T	
2	F	T	T	…
3	T	F	T	
4	T	T	F	
		.		.
		.		.
		.		.

Shortcut for Disjunctive Clauses

- Disjunctive clauses are connected only by the or operator
- Each major clause is made active by making all other clauses false
- The tests are "all false" and then a "diagonal" of true values:

	A	B	C	…
1	F	F	F	
2	T	F	F	…
3	F	T	F	
4	F	F	T	
		.		.
		.		.
		.		.

Summary : Logic Coverage for Specs

- Logical specifications can come from lots of places :
  - Preconditions
  - Java asserts
  - Contracts (in design-by-contract development)
  - OCL conditions
  - Formal languages
- Logical specifications can describe behavior at many levels :
  - Methods and classes (unit and module testing)
  - Connections among classes and components
  - System-level behavior
- Many predicates in specifications are in disjunctive normal or conjunctive normal form – simplifying the computations

References

- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008