CITS5501 Software Testing and Quality Assurance
Logic-Based Testing

Unit coordinator: Arran Stewart

Logic-based testing

- One simplification we made when looking at source flow control was that a decision point was just treated as a "black box" – it might contain a complex boolean condition, but all we cared about was the fact that graph edges went in, and graph edges came out.
- Modeling the internal structure of the boolean conditions leads us to *logic testing* – modeling the logical structure of conditions, and checking how well we have exercised different parts of them.

Logic expressions

- Logic expressions show up in many situations in software systems.
  - Decisions in programs
  - State charts (a system can move to a different state when particular conditions are satisfied)
  - Use cases (a user can take different actions)
  - . . . pretty much all the things we could model as a graph, in fact, where there's a choice of path based on logical conditions
  - Requirements, both formal and informal (e.g. something must satisfy requirement A **AND** either or both of requirements B and C)
  - SQL queries

Uses of logic expressions

- As with graph models, we can use logical expressions to identify new tests, assess the coverage of existing tests, and sometimes spot problems in our system
  (e.g. we might have written redundant clauses in a logic expression, that can never in fact be executed – the logic equivalent of "dead code")

- Plus, it's useful to look at logic-based testing, because sometimes it's mandated by legislation.
- e.g. A certain level of logic expression coverage is required by the US Federal Aviation Administration for safety critical software.
- (I don't know offhand what requirements are imposed by CASA, the Civil Aviation Safety Authority in Australia. But very few people design planes without considering what requirements are imposed by, for instance, the USA and the EU.)

Structure of logic expressions

- Here's a sample logic expression:

$$(i < j) \wedge isAlpha(c) \vee b \vee (m \geq n * o)$$

Structure of logic expressions

- Here's a sample logic expression:

  $(i < j) \wedge isAlpha(c) \vee b \vee (m \geq n * o)$

- Things it contains:
  - boolean variables ("b")
    (technically, it might also include boolean literals, `true` and `false`, but there's no point in doing so)
  - function calls that evaluate to booleans ("$isAlpha(c)$")
  - *predicates* ("tests") that operate on values, and evaluate to booleans ("$i < j$", "$m \geq n * o$")

Structure of logic expressions

- Here's a sample logic expression:

  $(i < j) \wedge isAlpha(c) \vee b \vee (m \geq n * o)$

- Things it contains:
  - boolean variables ("b")
    (technically, it might also include boolean literals, `true` and `false`, but there's no point in doing so)
  - function calls that evaluate to booleans ("$isAlpha(c)$")
  - *predicates* ("tests") that operate on values, and evaluate to booleans ("$i < j$", "$m \geq n * o$")
- We can think of all those as *inputs* to the expression.
- They are joined by various *connectives* . . .
  ("and", "or", "not", etc., plus parentheses)
- . . . and ultimately, produce a single boolean output – so logic expressions are a function from the inputs, to a boolean.

Structure of logic expressions

- If we have some number n of inputs, each of which can be true or false, then there will be 2^n possible inputs in total
- e.g. For the expression $a \wedge (b \vee d)$, a, b and d can all be true or false, giving eight possibilities...

a	b	c
F	F	F
F	F	T
F	T	F
F	T	T
T	F	F
T	F	T
T	T	F
T	T	T

Structure of logic expressions

a	b	c
F	F	F
F	F	T
F	T	F
F	T	T
T	F	F
T	F	T
T	T	F
T	T	T

- We would like to choose some subset of the total number of truth assignments, since 2^n quickly gets large.
- 2^6 (a modest number of inputs) is 64, and even if we only have, say, a few hundred logic expressions in our program, that's still potentially 19200 tests (assuming 300 expressions).
- It would be nice if we could do with fewer.

Definitions

$$(i < j) \wedge isAlpha(c) \vee b \vee (m \geq n * o)$$

- A *predicate* is an expression that evaluates to a boolean value. (So the expression as a whole is a predicate, as are some of its parts – e.g. $i < j$)
- Predicates can contain
  - boolean variables (e.g. b)
  - non-boolean variables, operated on by relational operators (e.g. "$>$", "$<$", "==", ">=", "!=" all can operate on non-boolean variables)
  - function or method calls that return a boolean

Connectives

- Predicates can be joined by connectives – logical operators.
- Common programming connectives:

$\neg$, `!`	–	logical not
$\wedge$, `and`, `&&`	–	logical "and"
$\vee$, `or`, `||`	–	logical "or"

- Less-common connectives:

$\rightarrow$	–	logical implication
$\oplus$	–	XOR (exclusive or)
$\equiv$	–	logical equivalence

Clauses

$$(i < j) \wedge \mathit{isAlpha}(c) \vee b \vee (m \geq n * o)$$

- A *clause* is a predicate with no logical connectives (i.e., it's at the "bottom level").
- b is a clause, as is $i < j$; they contain no *logical* connectives

Clauses

So, how many clauses here?

$$(i < j) \wedge isAlpha(c) \vee b \vee (m \geq n * o)$$

Clauses

- In practice, most predicates have few clauses
- (It would be nice to quantify that claim...
  Doing so is the task of *empirical software engineering*, which amongst other things studies corpuses of code to see how programmers write code in practice.)

Use of predicates in logic-based testing

- 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

Some abbreviations

We'll use:

- P is the set of predicates
- p is a single predicate in P (that is, $p \in P$)
- 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 we consider require that each predicate and each clause be evaluated to both true and false

Predicate Coverage (PC): For each p in P, our test requirements include that:
(1) p evaluates to true, and (2) p evaluates to false.

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

Clause Coverage (CC): For each c in C, our test requirements include that:
(1) c evaluates to true, and (2) c evaluates to false.

Predicate coverage example

Consider the predicate

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

Assume a, b, m, n, o are integers, and d is a boolean.
What are some input values for getting predicate coverage of this predicate?

Predicate coverage example

Consider the predicate

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

Assume a, b, m, n, o are integers, and d is a boolean.
What are some input values for getting predicate coverage of this predicate?

What about clause coverage?

Clause coverage example

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

Clause coverage example

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

1. a = 5, b = 10, d = true, m = n = o = 1
2. a = 10, b = 5, d = false, m = 1, n = 2, o = 2

Problems with PC and CC

- PC does not fully exercise all the clauses, especially in the presence of short circuit evaluation

Problems with PC and CC

- PC does not fully exercise all the clauses, especially in the presence of short circuit evaluation
- CC does not always ensure PC
  - That is, we can satisfy CC without causing the predicate to be both true and false
- A simple solution (well, simple to state, anyway) is to test all combinations . . .

Combinatorial Coverage (CoC)

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

Combinatorial Coverage (CoC): For each p in P, our test requirements include requirements for the clauses in C_p to evaluate to each possible combination of truth values.

Combinatorial Coverage (CoC)

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

Combinatorial Coverage (CoC): For each p in P, our test requirements include requirements for the clauses in C_p to evaluate to each possible combination of truth values.

	$a < b$	d	$m \geq n * 0$	$((a < b) \vee d) \wedge (m \geq n * o)$
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

Combinatorial Coverage

- This idea is "simple", neat, clean, and comprehensive ...
- But quite expensive!
- 2^N tests, where N is the number of clauses
  $\rightarrow$ Impractical for predicates with more than 3 or 4 clauses

Combinatorial Coverage

- This idea is "simple", neat, clean, and comprehensive ...
- But quite expensive!
- 2^N tests, where N is the number of clauses
  $\rightarrow$ 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?
- The text presents this idea as "making clauses active" ...

Active clauses

- Clause coverage has a weakness: The values do not always make a difference
- Consider the first test for clause coverage, which caused each clause to be true:

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

test 1 values:

$(5 < 10) \vee \mathit{true} \wedge (1 >= 1 * 1)$

- Since $5 < 10$ is true, only the first clause counts – doesn't matter what the rest were.
- To really test the results of a clause, the clause should be the *determining factor* in the value of the predicate

Determining predicates

Informal explanation of when a clause "determines" the value of some predicate:

- Suppose we have some predicate p, with a bunch of clauses in it.
- Let's suppose we're looking at one particular clause, call it c.
- Whether c "determines" the value of p depends on the values of all the *other* clauses in p.
- *If* the values of all the other clauses are such that, whenever we flip c, the value of the whole predicate changes, *then* we say that c determines the value of p.

Determining predicates

The textbook definition of "determination" is as follows:

Determination

Given a major clause c_i in predicate p, we say that c_i determines p if the minor clauses $c_j \in p$, where $j \neq i$, have values so that changing the truth value of c_i changes the value of p

- By "major clause", Ammann and Offutt just mean "the clause we're currently considering", and by "minor clause", they just mean "all the other clauses".

Determination examples – "OR"

Determination definition

- Given a major clause c_i in predicate p, we say that c_i determines p if the minor clauses $c_j \in p, j \neq i$ have values so that changing the truth value of c_i changes the truth value of p.

- So suppose p is $A \vee B$.
  - A determines p when B is false.
    (i.e. The value of p depends on A.)
  - But when B is true, A does not determine p.

Determination examples – "AND"

- Suppose p is $A \wedge B$.
- When will A determine p?
- Let's consider possible values for B.
  - When B is false, then $A \wedge B$ will be false, regardless of the value of A. So A doesn't determin p in this case.
  - But when B is true, then $A \wedge B$ will be true when A is true, and false when A is false.
  - Which is exactly the definition of "determination".
- So: A determines p when B is true.

  That is – the value of p depends on A.

Active clause coverage

- So, we have seen several sorts of logic coverage criterion, and some problems with them.
  - clause and predicate coverage – these are usually too coarse-grained
  - combinatorial coverage/multiple condition coverage – this may be too expensive
- Now that we have defined what it means to determine the value of a predicate, we can define a more useful coverage criterion: *Active Clause Coverage* (ACC).
- Informally: for each clause c in each predicate p, we find values for all the *other* clauses in p such that c determines p.

Active clause coverage definition

Active Clause Coverage (ACC)

- For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i determines p.
- For each c_i, we require that c_i evaluates to true and c_i evaluates to false.

- In other words: for all the predicates we are considering (P), we want to make each clause in each predicate p *active*
- We choose values for the *other* clauses so that the clause under consideration (c_i) will determine the value of the predicate as a whole
- And then we select test inputs that will make that clause true and false, in turn.

A problem with active clause coverage

Active Clause Coverage (ACC)

- For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i determines p.
- For each c_i, we require that c_i evaluates to true and c_i evaluates to false.

- Given this definition of ACC, there is some ambiguity about how values for "other clauses" may be selected.
- Namely: when considering the minor clauses (c_j), do they need to have the same value when our major clause (c_i) is false, as when it is true?

Active clause coverage ambiguity

- Consider the following predicate p:

  $A \leftrightarrow B$

- This is equivalent to the Java code `a == b`, where a and b are booleans

Active clause coverage ambiguity

p: $A \leftrightarrow B$

- p contains two clauses, A and B
- Now, regardless of what B is, A determines p.
- So let's make A the major (active) clause, and select test values:
  - Test 1: $A = true$, $B = true$, and consequently $p = true$
  - Test 2: $A = false$, $B = false$, and consequently $p = true$.
- Now let's make B the major clause;
  We could select exactly the same values as when A was active.
- So we end up with only two test inputs, and we never actually make p false; so ACC does not subsume predicate coverage.

General active clause coverage (GACC)

p: $A \leftrightarrow B$
- Test 1: $A = true$, $B = true$, and consequently $p = true$
- Test 2: $A = false$, $B = false$, and consequently $p = true$.

- In this case, our minor clauses have different values for when the active clause is true, as when it is false.
- This is the most general case of ACC – we call it "General active clause coverage" (GACC)
- As we have seen, it *misses* some cases that predicate coverage would catch.

General active clause coverage (GACC) definition

The definition is the same as ACC, with an addition:

Active clause coverage

- For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i determines p.
- For each c_i, we require that c_i evaluates to true and c_i evaluates to false.

We add:

- The values chosen for the minor clauses c_j do not need to be the same when c_i is true as when c_i is false.

Restricted active clause coverage

Alternatively, if we insist the values for minor clauses c_j must be the same when c_i is true as when c_i is false, we end up with Restricted Active Clause Coverage (RACC).

Restricted active clause coverage

- For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i determines p.
- For each c_i, we require that c_i evaluates to true and c_i evaluates to false.
- The values chosen for the minor clauses c_j must be the same when when c_i is true as when c_i is false

Relevance of active clause coverage

- The Amman and Offut text describes various other flavours of ACC, and other sorts of logic coverage – we will consider only GACC and RACC
- You may also see logic coverage referred to as "condition coverage"
- One application: "modified condition/decision coverage" (MC/DC), mandated for some avionics software.

Modified condition/decision coverage (MC/DC)

MC/DC

- Each entry and exit point must be invoked
- Each decision takes every possible outcome
  - Equivalent to what we have called branch or edge coverage
- Each condition in a decision takes every possible outcome
  - A "condition" means an atomic-level Boolean expression – what we have called a *clause*
  - So this is equivalent to clause coverage
- Each condition in a decision is shown to independently affect the outcome of the decision.
  - Equivalent to active clause coverage

- Since ACC is ambiguous, it follows MC/DC is too – this has led to confusion.

Logic expressions

- We will typically extract logic expressions from specifications or source
- When a predicate has only one clause, CoC, CC and ACC 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*)

Specifications

- Specifications can be formal or informal
  - Formal specs are usually expressed mathematically
  - Informal specs are usually expressed in natural language
- Many formal languages and informal styles are available
- Most specification languages include explicit logical expressions, so it is easy to identify the predicates and clauses
- 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 sources is preconditions

Example

- "Release the monster when the sky is clear; in addition, the moon must be full, and/or the wind calm"
- Re-write as an explicit logical expression (checking with client this is correct):

  Let:

  - m be "the moon is full"
  - s be "the sky is clear"
  - w be "the wind is calm"

  and

  p be $s \wedge (m \vee w)$

Example

p is: $s \wedge (m \vee w)$

Suppose we are asked to come up with tests which will satisfy the Restricted active clause coverage criterion.

We must make each of s, m and w active in turn. And the values for the *other* two clauses must be the same when the clause under consideration is true, as when it is false.

Example – clause s

p is: $s \wedge (m \vee w)$

Making s active:

- s will be active when $m \vee w$ is true; we just need to pick values of m and s which will make this so. Let's choose $m =$ true, and $w =$ false.
- Now, the value of s determines p.

Example – clause m

p is: $s \wedge (m \vee w)$

Making m active:

- We need s to be true, otherwise p will always be false.
- And we need w to be false, otherwise $m \vee w$ will always be true.
- So we set $s =$ true, and $w =$ false.
- Now, m determines the value of p.

Example – clause w

p is: $s \wedge (m \vee w)$

Making w active:

- This will be very similar to the last case (since $\vee$ is what is called "commutative" – like "$+$" in arithmetic, where $a + b = b + a$).
- s must be true, and m must be false.
- And then, w determines the value of p.

Example

p is: $s \wedge (m \vee w)$

So in this case, a set of tests that give us RACC are:

Test description	**Inputs**	**Expected output**
Make s active, and		
s = true	s = true, m = true, w = false	Monster is released
s = false	s = false, m = true, w = false	Monster not released
Make m active, and		
m = true	s = true, m = true, w = false	Monster is released
m = false	s = true, m = false, w = false	Monster not released
Make w active, and		
w = true	s = true, m = false, w = true	Monster is released
w = false	s = true, m = false, w = false	Monster not released