CITS5501 Software Testing and Quality Assurance
Semester 1, 2022
Workshop 5 (week 6) – logic-based testing – solutions

Reading

It is strongly suggested you complete the recommended readings for weeks 1-5 *before* attempting this lab/workshop.

0. Notation

When writing logic expressions, we will normally use mathematical notation for "and", "or", and "not":

- $\wedge$ – "and"
- $\vee$ – "or"
- $\neg$ – "not"

If writing actual Java code, however, we use the normal Java logical operators:

- `&&` – "and"
- `||` – "or"
- `!` – "not"

(We won't be using any Python in this unit – but for reference, in Python, the logic operators are all spelled out: "and", "or" and "not".)

1. Terminology – clauses and predicates

If you need to, review the lecture material and recommended readings that explain what *predicates* and *clauses* are.

What are the *clauses* in the predicates below?

a. $((f \leqslant g) \wedge (x > 0)) \vee (M \wedge (e < d + c))$

There are 4 clauses:

i. $f \leqslant g$
ii. $x > 0$

iii. M
iv. $(e < d + c)$

b. $G \vee ((m > a) \vee (s \leqslant o + n)) \wedge U$

There are 4 clauses:

i. G
ii. $m > a$
iii. $s \leqslant o + n$
iv. U

2. Making clauses active

To make a particular clause c in some predicate *active* means to assign values to variables so that the truth-value of the whole predicate depends on c.

When coming up with test values which make clauses active, the easiest way of showing your test values is in a table.

E.g. Suppose we have a predicate $s \wedge (m \vee w)$, where

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

If asked to come up with test inputs which make each clause active in turn, and achieve Restricted Active Clause Coverage, we could show them like this:

Test description	**Inputs**	**Predicate value**
Make s active, and		
s = true	s = true, m = true, w = false	true
s = false	s = false, m = true, w = false	false
Make m active, and		
m = true	s = true, m = true, w = false	true
m = false	s = true, m = false, w = false	false
Make w active, and		
w = true	s = true, m = false, w = true	true
w = false	s = true, m = false, w = false	false

(Here, we aren't told what the expected outcome is if the predicate comes out true or false; if we were, we could add a column "Expected outcome" which listed this.)

If you aren't told the exact *types* of variables or methods used in a predicate, that means you should be able to work them out from context. For example, for the predicate

$$(x > 0) \vee (M \wedge (e < d + c))$$

you can assume that M is a boolean, and that x, c, d and e are some integral type (such as `int`).

For each of the clauses in the predicates below, identify test inputs which will make the clause *active* (that is: state what values need to be assigned to the variables in the predicate), and vary that clause so it takes on both true and false values. (In other words: write test values that achieve Restricted Active Clause Coverage.) Explain your reasoning.

a. $A \vee (B \wedge \neg C)$
b. $x > 0 \ \vee (M \wedge (e < d + c))$
c. $G \vee (m \geqslant a) \vee H \wedge U$

In the solutions, we explain our reasoning, and then give a table with test values. This is a good way to format your answers, if you're asked to come up with test values.

Often there can be *multiple* possible solutions, but we show only one.

For (a):

This one should be simple. Our predicate is $A \vee (B \wedge \neg C)$, and so our clauses here are A, B and C.

Note that strictly speaking, we don't need the parentheses in $A \vee (B \wedge \neg C)$, since "$\wedge$" is considered to bind more tightly than "$\vee$", but we add them for clarity. (This is reflected in most programming languages, where "`&&`" has higher precedence than "`||`".)

i. ***To make*** A ***active:*** Set $B \wedge \neg C$ to false.
   So, suitable test values might be: $A \in \{\text{true}, \text{false}\}$, $B = \text{false}$, $C = \text{true}$.
ii. ***To make*** B ***active:*** Set A to false and $\neg C$ to true.
   So, suitable test values might be: $B \in \{\text{true}, \text{false}\}$, $A = \text{false}$, $C = \text{false}$.
iii. ***To make*** C ***active:*** Set A to false and B to true.
   So, suitable test values might be: $C \in \{\text{true}, \text{false}\}$, $A = \text{false}$, $B = \text{true}$.

Our table of test inputs would then look like this:

Test description	**Inputs**	**Predicate value**
Make A active, and		
$A = \text{true}$	$A = \text{true}$, $B = \text{false}$, $C = \text{true}$	true
$A = \text{false}$	$A = \text{false}$, $B = \text{false}$, $C = \text{true}$	false
Make B active, and		
$B = \text{true}$	$A = \text{false}$, $B = \text{true}$, $C = \text{false}$	true
$B = \text{false}$	$A = \text{false}$, $B = \text{false}$, $C = \text{false}$	false
Make C active, and		
$C = \text{true}$	$A = \text{false}$, $B = \text{true}$, $C = \text{true}$	false
$C = \text{false}$	$A = \text{false}$, $B = \text{true}$, $C = \text{false}$	true

For (b):

Our predicate here is $x > 0 \ \vee (M \wedge (e < d + c))$, so our clauses are $x > 0$, M, and $e < d + c$.

If you look at the logical structure of this, it actually has a similar logical form as in (a): $A \vee (B \wedge C)$. But here, some of the variables are of some integral type, so our test values will be any integers we choose that make the relevant clauses true or false.

i. ***To make*** $x > 0$ ***active:*** Set $M \wedge (e < d + c)$ to false.
So, suitable test values might be: $x \in \{0, 1\}$, $M = \text{false}$, $e = d = c = 0$.

ii. ***To make*** M ***active:*** Set $x > 0$ to false and $e < d + c$ to true.
So, suitable test values might be: $M \in \{\text{true}, \text{false}\}$, $x = 0$, and $e = d = c = 1$.

iii. ***To make*** $e < d + c$ ***active:*** Set $x > 0$ to false and M to true.
So, suitable test values for x and M might be: $x = 0$, $M = \text{true}$. We then have to make $e < d + c$ true and false, each in turn. To make it true, we could select $e = d = c = 1$, and to make it false, we could select $e = d = c = 0$. (Note that any values which make $e < d + c$ true then false would be fine; but the values we have chosen happen to be easy to write down.)

Our table of test inputs would then look like this:

Test description	**Inputs**	**Predicate value**
Make $x > 0$ active, and		
$x > 0 = \text{true}$	$x = 1$, $M = \text{false}$, $e = d = c = 0$	true
$x > 0 = \text{false}$	$x = 0$, $M = \text{false}$, $e = d = c = 0$	false
Make M active, and		
$M = \text{true}$	$M = \text{true}$, $x = 0$, $e = d = c = 1$	true
$M = \text{false}$	$M = \text{false}$, $x = 0$, $e = d = c = 0$	false
Make $e < d + c$ active, and		
$e < d + c = \text{true}$	$x = 0, M = \text{true}$, $e = d = c = 1$	true
$e < d + c = \text{false}$	$x = 0, M = \text{true}$, $e = d = c = 0$	false

For (c):

Our predicate here is $G \vee (m \geqslant a) \vee H \wedge U$, so our clauses are G, $m \geqslant a$, H and U.

Note that because "$\wedge$" binds more tightly than "$\vee$", this predicate is equivalent to

$$G \vee (m \geqslant a) \vee (H \wedge U)$$

Our process for making each clause active is as follows:

i. ***To make*** G ***active.*** We need to make $m \geqslant a$ and $H \wedge U$ both false.
So, suitable test values might be: $G \in \{\text{true}, \text{false}\}$, $m = a = 0$, and $H = U = \text{false}$.

ii. ***To make*** $m \geqslant a$ ***active.*** We must make G and $H \wedge U$ both false.
So suitable values for them could be $G = H = U = \text{false}$.
Then we need to make $m \geqslant a$ true and false each in turn; to make it true, we could use $m = a = 0$, and to make it false, $m = 0$ and $a = 1$.

iii. ***To make*** H ***active.*** We need to make G and $m \geqslant a$ both false, and U true.
So suitable inputs would be: $H \in \{\text{true}, \text{false}\}$, $G = \text{false}, m = 0, a = 1, U = \text{true}$.

iv. ***To make*** U ***active.*** We need to make G and $m \geqslant a$ both false, and H true.
So suitable inputs would be: $U \in \{\text{true}, \text{false}\}$, $G = \text{false}, m = 0, a = 1, H = \text{true}$.

Our table of test inputs would then look like this:

Test description	**Inputs**	**Predicate value**
Make G active, and		
$G = \text{true}$	$G = \text{true}$, $m = a = 0$, $H = U = \text{false}$	true
$G = \text{false}$	$G = \text{false}$, $m = a = 0$, $H = U = \text{false}$	false
Make $m \geqslant a$ active, and		
$m \geqslant a = \text{true}$	$m = a = 0$, $G = H = U = \text{false}$	true
$m \geqslant a = \text{false}$	$m = 0, a = 1$, $G = H = U = \text{false}$	false
Make H active, and		
$H = \text{true}$	$H = \text{true}$, $G = \text{false}$, $m = 0, a = 1, U = \text{true}$	false
$H = \text{false}$	$H = \text{false}$, $G = \text{false}$, $m = 0, a = 1, U = \text{true}$	true
Make U active, and		
$U = \text{true}$	$U = \text{true}$, $G = \text{false}, m = 0, a = 1, H = \text{true}$	false
$U = \text{false}$	$U = \text{false}$, $G = \text{false}, m = 0, a = 1, H = \text{true}$	true

3. Scenario – trap-doors

Suppose a component under test has the following requirements:

If the lever is pulled and the chair is occupied, open the trap-door.

If the button is pressed, open the trap-door.

Represent the component as a set of logic expressions. You should explain what each variable in your expressions means. (For an example, look in section 2 at the way we gave definitions for the variables in the predicate $s \wedge (m \vee w)$.)

(Hint: if you're stuck, try writing out what the component does as one or more "`if`" statements, in pseudocode. Then recall that the set of all predicates in a system means the set of all logical expressions found in things like "`if`" statements.)

Answer:

We have two predicates in our component: the first is "the lever is pulled and the chair is occupied", and the second is "the button is pressed".

We will define three variables to represent the clauses in these predicates:

- Let L represent "the lever is pulled"
- Let C represent "the chair is occupied"
- Let B represent "the button is pressed"

Using these definitions, the set of logic expressions (i.e. predicates) in our component is therefore:

- $L \wedge C$
- B

Incorrect answers

Note that it's *incorrect* to try and represent "open the trap-door" as a clause.

This is because from the requirement we're given, "open the trap-door" is clearly not a condition we have to detect, but rather an *action* our system must take (and something we can hopefully observe as part of seeing whether we get the *expected outcome*).

To see why this is so, and why we shouldn't model "open the trap-door" as a clause, imagine writing pseudocode to represent what our component does. It might look something like this:

```
1 if lever pulled and chair occupied:
2   open trap-door
3   return
4 if button pressed:
5   open trap-door
6   return
```

Then recall that doing logic-based testing means to test the *logical expressions* in our

system – the conditions following the "if" statements – and to ensure we've thoroughly tested the clauses that make them up.

Now imagine we are writing JUnit style tests to see if our component (let's call it `TrapDoorController`) behaves as expected. We'll assume we have mock objects called `lever` and `chair` and `button` created in a `setUp` method, and that our controller has a `control()` method to which we pass the `lever` and `chair` and `button`. For one of our tests, we might have something like:

```
@Test
/** Test the case when lever pulled, chair not occupied,
  * button not pressed */
public void testA() {
  // prepare the test environment
  lever.setPosition("pulled");
  chair.setOccupancy("occupied");
  button.setPosition("unpressed");
  // invoke the method under test
  TrapDoorController c = new TrapDoorController();
  c.control(lever, chair, button);
  // assert that the behaviour is as we expect
  assertEquals( c.getStatus(), "closed",
                "trapdoor should be closed");
}
```

And we'd have a bunch of other tests to test other scenarios. So – what if "open the trap-door" were a clause, rather than an action to take? Then it would become part of the test environment. But if that were the case – what would be left to be the component under test? And what could we possibly assert in order to find out if the system behaved as expected or not? There are no sensible answers to these questions; hence it makes no sense to make "open the trap-door" a clause.

Note that for the exam and assignment, student answers that make this sort of mistake will generally be awarded very few (if any) marks – it will be taken to indicate a poor understanding of logic-based testing.

4. Scenario – login page

If you don't finish the previous sections of this worksheet in class, then attempt this in your own time.

Suppose you are part of a team developing a website called "RateMyVeterinarian", where people can log in and provide anonymous reviews of the veterinarian services they use.

Requirements for the site are currently being finalised, and one requirement is stated as follows:

> When a user enters a user ID and password into the login page and hits the "log in" button, then if that user ID is listed in the "users" database, and the password matches against the password in the record for that user, and the user record does not state that the account has been disabled, a "Welcome" page should be displayed.

a. How easy to understand do you think this requirement is? If you think it could be made easier to understand, suggest how.

b. One of your colleagues suggests that because correctly authenticating users (and keeping their details secure) is an important feature, this requirement should be thoroughly tested – so you should design a test suite that meets RAC (Restricted Active Clause) levels of coverage. Do you agree? Why or why not?

a. Requirement clarity and readability

This requirement would probably be more readable if the various conditions were given as bullet points, rather than a run-on sentence (together with a little re-phrasing):

> When a user enters a user ID and password into the login page and hits the "log in" button, then if:
>
> - the user ID is listed in the "users" database;
> - the entered password matches against the password in the database record for that user; and
> - the user record does not state that the account has been disabled
>
> a "Welcome" page should be displayed.

You may have other suggestions for how the requirement could be made clearer.

b. RACC coverage

There is no one correct answer to this question.

It's certainly true that we should probably test the login feature thoroughly. However, once we have portions of the system implemented and at least a partial test suite in place, we may discover that we in fact already *have* achieved an RACC level of coverage.

So our colleague is not incorrect in saying that thorough testing is warranted, and they're not wrong to suggest that RACC is a good level of coverage to aim for. However, if they are suggesting that the tests need to be designed *right now*, that doesn't necessarily follow. The coverage our colleague wants may arise naturally out of applying techniques like Input Space Partitioning and graph-based testing.

For instance, suppose that when we come to implement the login feature, we end up with a method that looks something like this:

```
public void handleLogin(HttpRequest request) {
  String userID   = request.getParameter("userID");
  String password = request.getParameter("password");

  UserRecord userRecord = userDb.lookup(userID);
  if (userRecord == null)
    throw new NoSuchUserException("no user called " + userID);

  if (! userRecord.getPassword().equals(password) )
    throw new InvalidPassword();

  if (userRecord.getDisabledStatus == DISABLED)
    throw new UnauthorisedAccess("account is disabled");

  // if still here, all is OK
  renderWelcomePage();
}
```

(Note that this is simplified from how a real login handler method would work – amongst other things, we shouldn't be storing user passwords in a database in their raw, original format.)

We will probably write unit tests (using the ISP method) to make sure this method does the right thing. We might also check what level of graph coverage we have of the method, and perhaps write more unit tests and/or integration tests in response to that.

By the time we've done all that, it's quite possible we will have achieved an RACC level of coverage anyway, and don't have to write additional tests.

(If you're familiar with what HTTP requests look like, you might like to consider how you'd write ISP-based tests for this method. You would probably decided to use partitioning characteristics like "HTTP request contains a valid userID" and "HTTP request contains a password matching the userID", amongst other things. Once you apply something like Base Choice Coverage as criterion for your ISP tests, it's highly likely RACC will be satisfied for the requirement.)