웹 소프트웨어 신뢰성

✓ **Instructor: Gregg Rothermel**
✓ **Institution:** 한국과학기술원
✓ **Dictated: OES** 봉사단

Last time we started talking about some fundamentals of testing.

Now we're going to start talking about white box testing although it will bring us back to a point near the end of last class.

We talked about the fact that, well wait let's see

I presented something called the testing, the definition of the testing problem and that.

That whatever we're doing when we're testing, it's not attaining correctness of the program and so question is, what can we do as testers, how do we know what inputs to try since there's no infinite set we can try.

And how do we know when to stop.

And one class of answers to those questions involves adequacy criteria which are predicates if you want that determine when to stop testing or let's say when testing has been sufficient.

And as I mentioned there are black box criteria that don't look inside the code that look at things like specifications.

And there are white box that do look inside the code.

And it's white box that we are going to start looking at today.

There are a whole huge number of code based white box adequacy criteria that could be considered.

But we'll look at the five listed here starting with the most basic statement coverage.

The notion in statement coverage is to find test cases that exercise all the executable in the program.

Now not all statements are executable else and if and while and are not really executable.

We talked about that one building control flow graphs as well.

So statement coverage.

We're going to try an exercise statement here.

And let's see. The statements that can be covered here are numbered as one, two, seven.

Now okay that'll work.

So let's just see what happens when we start running test on this program.

The first test x equals 8, y equals 4.

Begins executing the program.

Cover statement 1, cover statement 2, it covers execute statement 3.

What does it cover next?

Statement 4 because x does not equal Y.

Where does it go next?

X is greater than y so it goes to 5. Now x is going to equal x -y, so xand4, x and y are

both going to be 4.

We head back to the while statement, essentially execute 3 again and jump out of the loop to 7.

And those are the statements then that are exercised for those inputs.

Rule for that if you want test case.

Test case. Now I marked this twice because it was essentially executed twice but it covers criteria well at least the ones we're thinking about.

Once is the same as multiple times.

Once is considered enough.

So really all we need bullied information.

Was it covered or was it not?

We're we checked if it was covered one or more times and that execution.

Next test x equals 2, y=2. Of course it starts off covering 1 and2 and 3 and then what?

7. Jumps to 7,, okay? So it doesn't cover anything new compared to what this covered.

It does take a different path.

I mean it's interesting what happens if you get to that y loop and never even entered the loop so there's some different behavior in this testing where that's not anything covering new in terms of state ones.

And so and it's we still have a covered statement 6.

So we haven't yet achieved statement coverage.

So now x=2, y=4 . Goes to the first 3 of course. Where does it go next?

4. It enters the loop. Where does it go next? 6. And that makes y=y-x, they're both 2.

Goes back to 3 jumps to 7.

And now we have with these three tests covered all the statements.

At least once.

If our test suite consists of these three test cases, in terms of just coverage statements, this one is redundant.

And we could eliminate it and with these two achieve full coverage.

Is there a smaller sized test suite that will cover all the statements? Smaller than this one plus this one? Smaller than two test cases?

You sure? If you go around that loop if you pick the values right.

I don't know maybe 12 and 5 are something like that.

You'll get it in there with x grader and y.

You'll do that a couple of times but then you'll loop around and x will be less than y and so you will cover the other one.

So you could come up with a minimal sized test suite that covers all the statements consisting of a single test okay? So what I've just followed here could be thought of as a greedy algorithm

For getting statement coverage.

And it says you're giving it a program you're going to open a test suite.

We're going to do this thing here. Let's talk about what that is in just a minute.

Not just yet.

But what I did was what we did was we repeated this, created the test, executed the program with it and in our case walked through the program.

Made note of what was covered by that test and if the test adds to the accumulated coverage, we added it to the test suite.

Now in other words I'm saying consisted accumulative I wouldn't have added it.

Okay. And we do that until we've gotten adequate coverage.

That's a greedy algorithm.

As we've observed I could have done this in one test if my first test had been x=12, y=5.

I would have been done after the first test.

Greedy algorithms are an optimal And we keep doing that until statement coverage adequate.

Now what we've just discussed is T necessarily minimal? No. There are a lot of issues that come up.

Even when doing something as simple as statement coverage.

And the first one is what about unreachable statements? Sometimes cod has unreachable statements.

Usually why engineers would put reachable statements in code it makes no sense.

But they do. A simple relatively stupid example of a unreachable statement is that one. Okay? There's no way you can satisfy the conditions that cause that execute. That does look rather silly.

But if this were something more complex.

If f of p and f of q or well no let's say f prime of p and you had to go off and look in those functions it can happen that something can't be reached.

Even more so as programs evolve and you're talking about interacting functions and methods and so on that's even more possible to end up with unreachable statements also known as dead code.

Or sometimes when you use when you build a program that uses some other suprotine .

There may be part of that ___ that your program never invokes.

Like a calculator invoking a square root routine. If the calculator only takes positive numbers, and the square root routine handles the square's negative numbers that code dealing with negative numbers will never be invoked.

Because your calculator never passes in negative numbers.

So there could be unreachable code and the issue there is you don't want to require yourself to execute code that's unreachable that gives you a criterion that is not workable.

So there's an unreachable statement I say you got to cover your statement you're never going to satisfy your criterion.

So in general all of the criteria are re cast restated in terms of things that can be executed.

So we say you need to reach all the executable statements in code.

Another kind of statement is hard to reach statement.

And what do I mean by hard to reach?

I forget my syntax, but I know in c there's a mallec routine, right?

If maybe someone knows the syntax for this but equal mallec.

Let me define ____

If mallec fails do something,. And the syntax is if something equal mallec not equal I forget the syntax but obviously mallec can fail to return memory to you to use.

And when that happens you like to text that a problem has happened and recover or give an air message or whatever you have to do.

Now a tester suppose you're testing a program that contains this.

How do you cause that statement or those statements to execute.

What kind of test case do you write?

How do you what's the condition that has to be met to execute this?

Mallec has failed so somehow you got to make mallec fail.

I mean somehow you got to use up all available memory. How are you going to do that?

Well, yeah and right this may vary with various things too.

I mean may vary with the system you're on, obviously the amount of memory it has that kind of thing. So that's what I call a hard to reach statement.

Now different programs are different and if this is I don't know world processing program or something of that sort

Maybe I won't even care about that.

I'll just inspect it and say, "Yeah, it prints mallec failed.

That's all I wanted to do.

The print's going to work and use a code inspection approach rather than executing it. "

But if its recovery code on say the mars roper and you want to make sure that the air's recovered from you probably do want to physically test it.

And now the question is how do you do that? Can anyone think of a tricky way to do it?

Without having to make memory fill up? And remember that mallec is a call to a routine.

Make a dummy mallec put in original mallec that returns in code.

Or you can rewrite this to do my mallec and put in my mallec and as a tester this is the kind of thing we do all the time.

The actual code we have may be difficult to for us to test and by some substitutions we're able to test the code we want to test, okay? So that's one thing we can do

But you want to make the choice is it important to you to test that code or what inspections to do. And there are various types of hard to reach statements.

That might be considered in there

So those are two problems with statement coverage.

And then probably well one of the biggest problems is how do you actually pick the test input to use.

Now we just went through I provided some inputs and the last one you could come up with

Inputs manually come up with inputs that's fine.

But as programs get more complex and the pass to them gets more complex and the

conditionals get more complex it's harder and harder to find inputs that will reach statements.

And to get up to that 100 percent coverage point can be very difficult.

So there's been a lot of work in the research realm and you'll see this supplying to web applications as well on automatically generating test cases.

And so we will probably look at some papers that deal in that as apply to web apps.

Another practical approach is to industries to start with your blackbox test.

And blackbox tests we'll see a little later are built based on this specifications without looking at the code.

They can be built right up front before there is any code or at least to find up front

And you can start by running those and seeing what covers they get.

And that's what we usually recommend. Start with those and now find test that cover the code that isn't covered by your functional test.

Code instrumentation.

There is that statement about instrument code.

When we executed this program in our heads we kept track of what executed it.

Obviously that's not what we're going to do when we're testing a large program.

We want some way to automatically keep track of which statement execute it.

And that involves code instrumentation. Putting probe into the code.

And here's one simple way to do that. We'll just put in print statements.

Right after the line.

Print line 1 executed. Line2 executed. Where do we put the thing that says line 3 executed?

We put it here. Here and after you could do. That looks suffice.

You could put it before.

There are some issues here of course.

If I put it before and the system for some reason crashes there it says line 3 executed line 3 really didn't but it fits approximation but still what you could do that and in c you can put in multiple statements in there too.

You can put it in here.

Print line or call some function and then do the comparism.

But anyway that's one way do that prints statements.

Of course now you run the program you get all this ugly outputs all the screens so and that's not really best way do it.

And in practice then there much more sophisticated each instrumentation of approaches that find ways to put monitering code in.

And you can imagine that another form multiple be a code function that's keeping matrix what's covered.

And the we are very full to do things like that.

And there is lots of resource on how to do that more cost effectively.

Because even if you do have function calls here like after line 1, you will have the call to your function cover the passes the no.1 and after line 2 covered passing it 2 so you are passing all after function which is keeping track of things or maybe the function name is in there the program name.

So I could put them in there the prints are coming out it's going to a function.

Any problems with that? That you could think of? Adding code.

Say after if A line into the program? Think about real time software. performance issues okay? If it's real time your changing the performance of the software.

There may even be well dependencies in the code and timing issues in changing the timing.

So in some sense you're not running the same program or the program that you mean test on to do that.

You don't want your instrumentation to change the behavior of the program to change the semantics of program.

And in some cases it can. So that's one thing to be aware of.

Also the volume of information of gets generated can be quiet substantial.

Oh, I should mention this.

If I am using prints, and this loop executes several thousand times.

That's a lot of information.

Now if I am just keeping coverage table which fits in it the information doesn't grow with the executions in terms of size.

We're keeping track of.

You have to think about how to keep information is what I am saying.

Okay so we got let's see if we covered all this.

Using function calls instead of prints. We haven't covered this.

In some sense it isn't really necessary remember the whole concept of basic blocks in control flow graphs.

Basic box single entry single exits.

So you could confine your coverage information to basic blocks.

I can essentially put one put a print here saying lines one and two or execute of call a thing saying you know make note one and two or done.

I can do it per a basic block and cut down the number of probes used.

And there has been work on finding the minimal number of probes or not minimum but minimal number of probes needed to actually give you covers information.

But in any those cases you can run into problems with runtime or determining behavior programs.

There has been work on instrumenting binary code as well as source code like I could tell you that with JAVA has been work on instrumenting the bycode or instrumenting the source code and you find those instrumentals.

If it's interpreted you can track what's executed inside interpreter or such as in the JBM.

But now you're specific to that particular system that's a drawback to that.

There are all sorts of choices here and all sorts of different things people have looked.

Statement coverage also has a weakness beyond those that I have mentioned so far.

And let's see if we can identify the weakness.

Give me A tests suite.

Is statement coverage adequate for this program.

A minimal test suite. [X=2, y=1] Everyone agree to the covers all the statements? What is its weakness? There are some aspects to program that hasn't been tested even when you cover all the statements with that one test.

Well it's an interesting point.

In some test maybe that's the weakness of the no code in there to handle incorrect. But I will try another one.

Overflow.

There is another one that I wasn't thinking of. But that's valid too.

There is one more that's more related to the coverage criteria itself though.

And to what it basically statement coverage you run that test you cover all statements.

You say I am done.

And there is something some part of a logic of the program that's left undone in that case.

Involve statement 3.

It does not cover the branch out of 3.

I mean there is no statement there is no else here, ok?

So there is no statement associated with the else. But there is still a branch out of this. That occurred.

When this is false, you jump up to there. What if this is incorrect?

It's supposed to be greater than equal to or something else like that?

You haven't really tested that behavior.

So conditionals the outcome of conditonals are very important part in determining whether code is operating properly.

And statement coverage doesn't necessarily require that your cover all branches.

It might or it might not.

Let's look back at GCD with the same test.

And look at what we will call decision coverage which is find test cases that ensure coverage of every outcome of each predicate or each decision in the program.

So every outcome is two outcome in the false done in this case.

So [X=8, Y=4] and this case we are not convering statements we're covering outcomes predicates.

3 true, 3 false, 4 true, 4 false.

So [X=8, Y=4] comes in gets to line 3 where is it go? 3 true.

What's next? 4 true. Now the both flow so what happens next? 3 false, in the that execution. [X=2, Y=2] What does it do? 3 False, nothing new.

So in my greedy algorithm. I just keep it and go on.

And [X=2, y=4] What does it do? 4 false, 3 False, okay? So similar to the statement coverage in this case.

We've showed that out.

These two tests yielded branch coverage they also yield statement coverage.

So what I said previously was!

I don't even know what to click here anything.

Where do I find my slide-show? Of course there.

That's an icon. That's good any language.

When I said what's the weakness and said that statement coverage main I get you decision coverage it's truly it's may not.

In this case within the either outcome each predicate leads to some code and so if you cover the statements, you'll cover all the decisions.

But it's the case where we had the else less if there is a case where covering the statement does not imply covering all the outcomes of predicates.

So the outcome for and the algorithm for that is pretty much same as the greedy algorithm for statement coverage.

So I just generalized this instrument for any sort of code base coverage instrumental program that report that coverage and then just create run tests keeping that one coverage until you've got not the coverage.

And let's say pretty algorithm forgetting any type of code based coverage. So Replace that with various types.

The problem that exists for statement coverage exists here as well.

You can have unreachable outcomes of predicates.

The same problem might have the same silly example I had P bigger zero and NP less than zero.

You can't take the true branch.

There are going to be hard to reach decisions again that's true outcome of the mallec failure.

You still have to choose the tests.

You stop the instruments to code.

So all those problems are there.

What will we do two statements? Little louder! That's one way to do it change If to else, I mean you can [28:05] you get all those branches.

Another way without changing it? Essentially there are branches out of these.

And treat the branch outcome.

One another thing we do for convenience is relates to what do I have procedure that consists of flat line code.

And this is a big program.

And I want to cover all the code in the program.

Well branch coverage doesn't necessarily require me to cover anything in here.

So just tribually we treat the entrance to each method or precedure as the branch as well.

So you do have to enter it and get you coverage of all the top level things as well.

So in terms of program logic, give me a decision coverage adequete test we do for this program. Minimal one.

Give me one test first tell me what it covers. Kicheol.

Give me one test and tell me what it covers. [2,1] and what does it covers, I'm sorry? [x>y, y>0] So covers 4, covers all the statements.

Which branch or decision did it cover? [3,2] okay. So what are we still cover? So give me some inputs cover that. [x=2, y=3] oh did you say zero? Y is not greater than zero. And therefore we do 3 False.

What haven't we covered? We've covered both branches out of that.

What if I said that first predicate are supposed to be I don't know greater than are equal to. That's not the best examples we used but.

Think about the ways in which statement 3 can be true or false. How many are their combinations? 4 combinations.

Basically and what we did there [x=2, y=1] this case and that's you have the big that one because that's it only way you cannot you can get the two branch on that, right?

Right? And x=2, y=0 was the true false case.

But there are two other combinations that weren't tried there.

And trying those might expose an error in the logic.

Though that's what that leaves and we thought about statement coverage, decision coverage these have some different names to sometimes but I'm stick of this.

To condition coverage which says find test cases that ensures coverage of every outcome of each predicate in the program, under each assignment of truth values to the individual conditions in that predicate. So I'm not going to go through this on the board.

No, I'm going to ask a question. Those this test suite gives me condition coverage of this program? Why? That's right. That's right. Okay?

There are no multi part predicates there. So, all you got to do make it go true and false. And so this is an equal's condition.

But if you have got compound the predicates that's not going to be the case.

Okay?

Same problems as before. Have these to start.

There will unreachable conditions.

Sometimes 3 out of 4 of these can be reached but one of them can't.

Just given the truth value that you have signed to it.

So unreachable conditions hard-to reach ones.

You still have to pick the tests but now it's even more complicated trying to make certain combinations happen can be even harder from a tester's point of view.

And then how many tests do we need for two parts predicate?

Already told me that. 4 For p or q or r, how many? 8? The next? 16? So rapidly

grows large.

Now there aren't there are probably aren't that many I think in some studies aren't that many programs that have predicates they have that many parts but still it does get large quickly and it's expensive.

That! So like short circuiting, right? Yeah.

And the .. that's a complicated for testers, too.

Because what it reaches some of these truth value things never never can be covered because.. I have to figure this out, but.. in one of those you wouldn't need one.

Like false, false true you wouldn't be, right? The first one's false you done.

So they become the same. That's a good point, too.

That's part of the you can you got have to find what is it covered this every coverable one and in the case of short circuiting that changes cover ability.

So we have talked about this adequacy criteria in terms of code.

I mentioned covering statements covering predicate outcomes covering conditions.. You can also define them in terms of graphs.

In terms of control flow graphs. Oh.. let's see. Here's a control flow graph for GCD and can you .. let me say at this way. Statement coverage covers every executable statement in the program. Can you define that on the graph? In terms of graph? Parts? Cover every node.

Every node should be executed. Code associable knows should be executed if you want.

We can define most of the things in terms of graph.

So statement coverage becomes what we called node coverage.

When define and you will sometimes you will lose or the node be basic blocks basic block coverage.

Branch coverage? What is a.. well we use the term branch coverage applied to the graph what I called decision coverage on the graph is branch coverage and corresponds to taking the edges out of predicates or you could call all label edges if you want with predicate edges related to.

You can talk about edge coverage too.

Edge coverage would deal the same as taking all labeled things I really should have true there remember cause I am going to treat this as a labeled edge as entering procedure.

What about condition coverage? And I have to find that terms of graph?

What if this code is if ((x>y)&&(y>0))? That's my node.

That's right. How could I change the structure of graph?

That's right. So you could take this you can break this down into a ..break that down into that.

So if you would pick that is that.

Now covering all the label edges executes me condition coverage.

You can also incorporate short-circuiting in those graphs by omitting edges that would not exist in case of short-circuiting.

So that agrees to find the graph, now you could define the coverage criteria in terms of this form of graph.

We do the graphs to help us do different algorithm.

So there's nothing sacred about this graph formats let’s change it to what we need.

Now one thing when we are looking at some tests like the statement coverage tests.

First we tried.. what was it..x=8, y=4, and we tried x=2, y=2. And that didn't give us any new coverage.

But it did do something different.

The net test namely, it came down here and when around like that.

That's a very different pack then coming in here and going like that or coming in here and going like that.

And it may expose errors often times there are there could be airs associated with not entering the loop at all.

And as testers we often want test for the condition and so the statement coverage and branch coverage for statement decision condition don't get that notion of paths.

In fact, how many paths are there through this program? There's an infinite number.

What if I had the.. what if this was just never mind what the program does.. but was that's for I equals 1 to 5 do, then how many paths are the to the program?

It's a little hard to calculate because I think they are 2 to 5 plus maybe skipping entire.. no you can't cause of 4. Okay.

But so it's final number then, but it could be a large number obviously.

Still each of those paths could behave differently.

If I don't know too much about the code and I'm not an expert in semantics.

There's a sense in which as testers we might wish we could exercise all the paths because there might reveal more then just exercising all the branches.

But for the none trivial programs that's going to be invisible there too many paths.

But still theoretically path coverage is something that ensures coverage or every entry-to-exit in the program.

Now the problem there obviously is infinite number paths.

People have look that ways to define a subset of that infinite number of paths.

That's stronger than the paths that give you all the branches.

Again. I can get all branches in this program with 2 tests.

None of which skipped the y loop but with a criterion that requires you could say all an acyclic paths to the program.

There's a finite number.

It still might be a lot if you got big nested ifs.

And we won't go look at that detail just be aware that there are attempts to the find path based criteria.

We will get at acyclic different way just a minute.

So you can cover all acyclic paths. Cover all set basis paths.

I'm not going to talk about that. Just skip one.

Now there's a .. there are ways to compare criteria and one way is just analytical comparison of them.

What are the relative strengths criteria? When are criteria better than others?

And we can say that A subsumes B more less means includes is greater than subsumes b if for any program P and test suite T. T being A-adequate means T is B-

adequate. So let's put some names there for statement.

I mean for A and B. Does T being statement adequate ensure that T is branch adequate imply? For every program every test suite if I get statement adequate test suite is it branch adequate? No and we saw an example one it was.

What about the other way?

If my test suite is branch adequate is it statement adequate?

This is actually trick your question than I might say but the answer is yes.

You got a manipulated things a little bit you do have to treats to entry to the program as a branch because otherwise you can..if there no branches technically speaking you don't even have to cover it.

But using read definition branch is ..subsumes statement.

So actually subsubmition relations are drawn like that.

That is same branch subsumes statement okay? What about condition coverage?

Does statement coverage subsume condition? Is every statement coverage adequate test suite condition coverage adequate?

No. Is every branch coverage adequate condition adequate?

No, that was one of the weaknesses.

Is every condition coverage adequate test suite branch coverage adequate?

Yeah.

If you cover all the combinations of the outcomes of the predicate ..

Did someone say no?

But if you cover all the combinations you’re certainly going to cover the overall outcome being true and false.

Again you have to little careful there could be issues with well obviously there could be issues with we are talking about there feasible and infeasible things to cover if you are talking about condition where like the false branch never be taken then..

You might add some problem but avoiding nodes where's path suite here.

If you cover path all paths even those are only theoretical to cover all paths you cover all branches? Yeah so of course then transitive cover new all statements?

If you cover all paths do you cover all coditions? no..not giving the first format of CFG anyway.

If I redefine my graph as we saw then I could ensure that paths equal them all but we will just put this like this path condition decision statement.

One of the issues here is in some sense of branches are kind of weak conditions stronger but it still doesn't force you to get certain paths that might be important in the program we can't do all the paths.

So one thing people have looked for is some criterion in here.

That's stronger than branch not infeasible. but yields we hope..better fault revealingness

And that brings us to dataflow coverage.

Now the notion here since we done dataflow analysis since we know what that's all about we know that data dependency is in programs and places where definitions a variables can reach uses and affected competition so one thing we might realize as a tester is that -- requiring that I execute this statement and this statement may not be enough there's a path through in here and path through in here in program and I take this path to execute this one and this one to execute this.

I haven't executed the case where this definition reaches this use.

And that explodes airs.

So people reason that focusing on these data dependencies might be a way to do better testing.

It certainly is less than all paths because there might be paths on we still don't use that you don't need to focus on

Now when you do this you do the definition use pairs but we know how to get that DU chain, data dependencies.

So in some sense this is asking you to do something we've already done.

What are the definition use pairs? By which I mean definition?

Definition pair is a definition and a use that can be reached from a definition.

Meaning there is a definition path to the definition to use.

So what are the definition use pairs involved in the read in statement 1?

What uses can it reach? Three, yeah, it can reach the use of exit three.

The use of exit four, five, six, seven.

It reaches all the uses in this case in the program.

And you can calculate that for all of your definitions.

And now as a tester, you get to find tests that exercise all those pairs.

Oh, there they are.

So DU pairs repetition of although I haven't used the term definition use-pair maybe we call them DU chains, but it is definition D and a use.

Now in this definition I said of a variable v.

If we're being more precise of memory location because you can have various memory location access by various variable particularly if you have alieas in effect, so it's really a memory location at reaches.

I think we've seen this before and the DUpairs

I think we've seen this before and the DU pairs.

This is DU-pairs just involving X in this case.

How can we compute them? We've already done that.

Well just refresh the IN sets, the OUT sets.

You do the data flow analysis, get the IN sets and Hook, the definitions of IN sets up to the uses that are in the blocks.

So in Dataflow testing we're going to focus on these pairs.

But one thing that we're going to do is here, is distinguish a couple of different types of uses as there's reason for it.

We're going to say it, a predicate use.

The use in the predicate in a variable is predicate use or P use, the use of a variable in a computation is a C-use.

And in terms of coverage, let's see, here you got a definition of X, so there's a definition use pair, and there's a definition use pair.

But in terms of coverage, we're going to require that if the uses in the predicate, you

have to reach the uses and make the predicate take both conditions.

So you have to have one test that reaches it, and it goes true and one that goes and reaches false.

There's loops that might be same as the test.

The point is you get to branches out.

That's done just specifically, we'll ensure covering all the branches.

Cause otherwise you might not cover all branches.

And we're looking for something in between branch and path.

Now once you've defined that, there are a whole bunch of different criteria you can come up with.

These are just three of them.

Well I get to all path there, I think it should be all DU definition use paths but all definition says test each definition to some use, to at least one use.

So in that in there, all definitions would say, well if you've made X reach that use, that's enough.

And you get to make Y reach some use.

So that's kind of weak.

And in fact that doesn't, that isn't stronger than branch coverage all definitions.

But all definition use pairs, is going to say make each definition reach each use, each reachable use, we still run into feasibility problems.

I'll get back to that.

And then one step further, all DU paths, here we go again.

That should really say, ALL-DU-meaning definition paths, okay?

There may be many paths by which you can get the definition to use.

And so we could require that we get definition to each use, by each path, that could be taken.

And these are increasingly strong.

Now without putting all DU paths in there, turns out that you can do all as I marked you can get all definitions without getting all decisions.

So that's hanging off here.

But if you do all uses it's all definitions to all uses, you're in between path, and decision.

Which is one thing that we're interested in getting.

And all DU-paths would come in up there.

So the data flow gives you some criteria, whole family of criteria really, that can get you something stronger than branch and less stronger than path.

So that's all determine that's all strength stuffs and this is an analytical determination of strength.

I can say, if I cover all the paths and I cover all the uses, I can prove that, given certain structures of programs.

There still remains the practical question.

What are we trying to do when we test, we're trying to detect false.

So if you do these things, how well will you do for detecting false?

Because these are stronger in the sense that they test more than the ones beneath them.

But they also require, that you develop more tests.

And there are infeasibility issues that get more difficult as you get further up.

So theoretically they seem stronger, since they test more theoretically, they should be better at detecting false, but they are also more costly to use.

And has in all things when there's cost and benefit tradeoffs, you wonder just what are the cost benefit tradeoffs.

As you step up how much more do you get in terms of false detection and is it worth the extra cost?

And that's I don't know how to quantify that and analytically.

And it really has to be looked at empirically.

And there's been some studies of that.

One of the earliest ones that actually is one of the first examples of a control experiment that I know of in testing literature was a 1994 paper by well Hutchins was the first author but ____ was one of the main ones was the leader.

They looked at they compared empirically.

They created or got several small programs.

And testing techniques it's like comparing medicines right?

And I've got programs with false how well do my testing techniques do it detecting the false?

That's essentially what they're trying to do.

So they had programs so there were 7 of these.

But I'll just see here's the program. And they went and put false in them and a bunch of people create false.

Engineers who created make false statements hid them there experiments were similar to false found in practice.

So now there are various different false that could be in here.

Then they created now what you really want to compare these things is you want to create a bunch of branch adequate test suites a brunch of all use is adequate test suites.

You can do that with all the any criteria but they were just in comparing these two branch and all uses.

And then you might want to compare to just random testing.

There are a monkey testing or random testing.

Because certainly of these control testing techniques aren't better than that there's something to be said.

So to get a bunch of test suites they actually created a huge pull of inputs.

They had various people use various techniques in testing inputs.

And this represents a kind of I'll call it u in universe.

Represents it's not every input that could be to this program but like one of these had 5000 500 or so inputs.

For one of the program P.

And obviously there's an infinite number of inputs that could be for the program but this is a very large universe and from this you could use that greedy algorithm for making coverage adequate test suites.

You could sample, you could pick a test, add it to your test suite if it adds coverage you can do that until you've got a 100% coverage.

And now you got a branch adequate test suite. And then do that again.

And you've got another one.

So they did this and made a 1000 branch adequate test suits in a 1000 all uses adequate test suites.

By sampling the poll. And then one by one they ran those suites on each of the false versions and saw which ones could detect which false.

So that's one of the early experiments and there's been many more all of that since then.

And just various summary form we've got more data on this.

They were finding that you've got two things here the expense vs. the effectiveness.

And for expense, to measure that they just looked how many past cases had to be created to get the coverage.

Now random well they chose just to make random test suites of a 100 tests.

There's another way they could have done that.

But basically on average, they could get branch test adequacy.

This may be I forget now whether this is across all steps and probably across all the programs.

But they could get branch testing adequacy with 34 test cases but all the uses require more than twice as many.

And the branch was detecting 85 percent and the all uses 90%. So in one sense I could be negative about this, I'm only detecting 4 percent more faults.

For more than twice the effort in terms of test cases.

So is that worth it to me? If it's the mars rover, probably.

If it's a pace maker softer probably.

For others it's not so clear.

Now this is just one experiment on 7 programs and you can't generalize too far.

They were finding that random test suites of a 100 test cases of a 100 were not doing as well as the coverage criteria.

Even that was just a 5 percent less than the branch.

An interesting thing that's not looked at here is in experiment we're going to look at this more.

Let's see if you can think of this.

See. Let's see.

I've got I have branch coverage adequate test suites of size 34 on average.

They get 85% and all uses they get of size 84 on average they get 90%.

And I'm going to tell you all uses coverage causes me to get better fault detection.

It's the use of all uses coverage that causes me to get this.

This may be a hard question to ask but am I necessarily right or could something else have a cause for me to get 4 and a half percent more faults?

It is necessary the use of the use of coverage criteria? See the two numbers.

What if it's just the number tests? What if I gain 4 percent just because I had a lot more tests here?

Now okay the criteria caused me to get more but maybe if I had taken 34 and added 50 random tests I could have done just as well.

Is it the number or is it the criteria.

So there has been later studies that tried to look at that.

And if I wanted to look at that question I could have done just what I just said.

I can take branch testing.

I'll take the branch testing adequate test suites and I'll add test to them so that they're the same sized but they won't be all uses adequate and now I'll check the difference.

Another thing I could have done here is I could have created a 1000 random tests of size 34 or a 1000 I had a there were branch covers of many sizes that might have been a 30 a 31 a 32 but I could make random ones of those sizes.

And now I get a fair comparison it's eliminating.

In experiments you want to eliminate experiments try to focus on particular factors on whether a causes b and to do that you'd like to eliminate other potential things that might cause b.

And you don't want don't want to conclude that a causes b or something c that causes b.

And so we control things in experiments.

And we'll talk about that more but it's a thing to start thinking about.

How do you do experiment that really shows what you're aiming to show.

But back to this, this is just one study but it does suggest marginal size effects that you can get better fault detection at more cost.

I guess another question you can ask in experiment is do you believe that this is the right way to measure the cost of these techniques? A number of tests?

If I'm a Manager of a product team, what's my bottom line really?

In terms of the cost of the technique? I've got 10 engineers.

A bottom line is how much I'm paying to get this done.

And numbers may not reflect the amount of effort required.

And so that may not be the best measure.

What you probably want to do is get a bunch of engineers to apply these things.

What amount of cost is it for them to do the coverage because it may take substantial mount of work to cover some of these uses.

More so than the branches.