Title: 웹 소프트웨어의 신뢰성 17

- ✓ **Instructor: Gregg Rothermel**
- ✓ **Institution: KAIST**
- ✓ **Dictated: 강단비, 김주현, 김지현, 주다은**

[00:00]

I didn't have time to do anything new for today. So I read that I...

Instead I have a talk that I it gave to the LG electronics 2 or 3 weeks ago, 3 or 4 weeks ago. And it’s about regression testing.

It's not specifically web oriented, but we've talked about some regression testing things in here. And there are a lot of issues for the web applications.

And some of you are reseaching things involving regression testing for web applications.

So I thought it'd be good to go through some of this material that is based on regression testing problems and solutions applied to the general software.

One down side is this set aside, I put online is the pdf, not the powerpoint.

So my animations is got lumped in the one slide in a few cases here.

So it won’t be as fun, but I'll have to…

There is some animations that took hours to create, showing illustrating ?[01:05].

I'd have to walk through them without depending on the animation.

But, okay. Again, this is the talk that I gave, and it's…

I've given in various places, in various forms, but this was the most recent form.

And, uhm… I'll just launch in to it.

I think I've mentioned it before.

I went into academia, I worked in software in a software company called Piolet Systems, which builts computerated drafting, and computerated manufacturings software.

And we'd create a version of a system, and then releas that and then it'd go into the Internets. In course we tested the first version, but then we'd have to release new versions for new operating systems' versions with patches.

Versions that had new features, and we go through that for a while.
And then we'd make a major release, and that would go to the same process.

And all of these subjects and versions had to be re-tested or regression tested.

And in the case of this product, we had a test that took 8 to 10 hours to run.

Automated, but a person had to look through all the results which took about that long too.
And so that was more time than we wanted people to spend.

And we thought that where we'd often time some of this changes involve very small portions of the coded system.

You shouldn't need to run all the tests.

And we looked around for ?[2:44] to do this more efficiently.

But at the time I couldn’t find it, I ended up more interested in the problem and ended up doing research on it.

So, this talk is going to talk about the various, several threads I've done on the problems of testing software as it evolved.

And I'll began with a little background on the problem.

And I'll be talking about 3 different techniques.

We've talked about both of those, I remember giving little brief distinction between these two.
A few classes ago.,What are the techniques and how do they differ.

So in this, I'll talk abou them a littl more, and talk about test suite augmentation.

And all of these ?[03:30] have applications, applications to the web application in Rom.

But they haven't looked up that much there, or people are just starting to look at.

Some of them. But I haven't seen them there any yet.

So little background.

In a regression testing we have a program.

We modify it, program P, we modify it, creating P', there is a test suite for P.

And the question is what should we do, to re-test?

To re-test P'?

And when we have this test available, the most usual answer, the answer most usually taken by companies, is to just reuse the tests.

All the tests that are there, or all the ones that are still applied.

The code's chain some may not apply anymore.

And that we call re-test all approach.

But, as I mentioned earlier, that seems like it might be more than we need.

And if your test suite takes long time for run, and if only change the small portion of code. Maybe you should be able to do better to run fewer tests.

And some other issues arise as well.

And there are more expensive test suites.

Some people I worked with at Bowing had test suite that took 7 weeks to execute, on just 2000 lines of code.

The folks said Microsoft far go have a test suite, a roling test suite that takes 4 months, excuse me 4 weeks to go through on the major system they release.

[05:00]

And so when you have things that are not expensive, you really like to find ways to save time.

So the first way you might try to save time is, to use what we call regression test selection.

And that is to find an algorism, that can look at the test suite and the changes and select a subset of a suite test that needs to be run given the changes that are made to the program.

And there are various reasons that you want to do this.

You hope to get a smaller subset than the whole set, that thereby saving time.

You also hope to get an effective subset.

So we'll be talking about that more in a few slides.

And we'll see the cases where you can do this successfully.

We'll also see the cases where it doesn't work out.

More about that later, but in some cases companies just don’t want to leave out any tests.

And so in that case, instead of selecting sub-set, you might choose to run them all.

But you might choose to put them in an order that lets you achieve your objective faster.

Like detecting faults faster in the testing process.

And that's priorization, and we've seen a little bit about that earlier as well.

So I'll talk about more about that in few slides as well.

In ironic cases, this is all about reusing the existing test suite.

And when you've actually made changes to the program, you may need new tests.

The tests in here may not be adequate for the changes.

And so, when that occurs, whether, in this case I'm using, this is like a re-test, excuse me…

regression selection approach, where I pick a sub-set,

But in any case, whether I'm running all the tests or using a sub-set or prioritizing, in addition, I might need some new test, T''.

And finding out what new test you need and adding them is called test suite augmentation.

And so that's, so how do you do that is the third question.

And that we can look at algorism quickly.

So before going into those particular techniques, I want to present, what we can call traditional process model of the maintainance activity that goes on software companies.

It's traditional in a sense that, back when I was at Piolet Systems, most companies operated this way, in terms of their maintainance cycles.

These days many still do, but, 'cause we'll see late,r many do other things.

But, in the traditional model, you have this period of time.

You release the version of product P, and you have a period of time, usually weeks or months, in which you are working toward the next release.

Fixing bugs, adding enhancements, whatever you are doing.

And you go through that period of time, until, you're more or less code complete with the changes you wanted to make.

And that time you begin your validation and verification activities, you're testing whatever else you're doing.

And from the testing point of view, that's really the critical period.

That's usually time limited, often times this maintainance thing slipped, giving you even less time. And that's where you want to maximize your testing resources.

From a testing point of view, this is really a preliminary period, it's a time where you could be doing things to set up this activity.

That'd make it go more smoothly.

Like obtaining old test information or gaining, I'll talk about some specific things in there.

But in this process, the notion is, if you can find out algorisms that do things here,

To make your life easier, here, you can gain.

And that's what these approaches I am going to look at too.

So, ah, this is a different picture of it.

In the… what am I going to have, I am going to have that…

So, once again. You have a program, new version, one thing you can do is reuse all the tests in terms of this timeline.

The other thing you can do is to do some work here to partition the test suite and then use just sub-set in there.

And that makes it more likely you can run all your tests and may leave you some time to do some other valudation things as well.

So, regression test selection.

I'm going to show you an algorism that I've developed for doing regression test selection called dejavu.

I'm going to show you the algorism by an example of it running, and then I'll show you suited code for the algorism itself.

[10:00]

The notion in this algorism is when a program changes we want to select the tests that are relevant to those chages.

And the algorism relies on some analysis frist of building control flow graphs.

You've seen maybe exactly in the slide before earlier in the semester or something very like it.

You already know what a control flow graphs.

We represent the statements as nodes and the control flow's edges.

The second thing we do, is to track execution stages which is dynamic information about what the program did when it ran.

And in these, for instance, when you run this program, here on a test constituted by an empty tile, the line shown in red execute.

And you can track those with a profiler, and that's the execution trace of the program on that test.

If you have, or you can map that onto the graph, in terms of nodes and edges that it covers.

And if you have a full test suite you can do that one by one for each test.

So this gives us what call a test history information.

So, in this algorism I am going to show you to use the control flow graphs, and test history information.

The aim of algorism is to find the tests that reach changed code.

We'll talk about why am I want to do this when you do this, a little later.

But ?[11:28] seek to do that.

And so, to illustrate this operation, here I have the little average procesure I showed in control graph for, little modified version of it.

In which I changed the line of code, added a line, and deleted a line.

They are really, kind of meaningless changes, but it still works for an illustration of algorism.

So how do we find the test to go through those changes?

Well, if you had some analysis if you knew where the test went, let me show you the next slide first.

Here the control flow graphs for the program and the modified version.

And there is a new node, because it's a new statement of missing node causing deleted statement.

And if you imagine that the content of these nodes is actually the text of the code associated with them.

This node has changed because I changed the code associated with that from the less than to stand ?[12:24], I think.

So, how do you find the test to go through the changes?

Well, if you had some mapping between this graph, if you can do iso-morphism, and find out.

Rememeber, you don't know where these changes are, I'm just showing them to you.

So, initially, you've just got a program, change program.

If you can get some iso-morphism between, I am going to determine that, oh, this node corresponds to that, and that was changed.

And this test reached. This corresponds to that, and this test reached as it.

If you can do that, you can pick the test that which changes, but that turns out to be a hard problem.

Establishing iso-morphisms between graphs, can be expensive and difficult to do particularly when there is large changes in the graphs.

It's also more work than you need, as you will see.

The algorism I developed doesn’t do any new pattern matching iso-morphism building.

Instead, begins with the 2 graphs and here is where the animation massing is going to be. Sad for me.

It begins with these 2 graphs, and walks its way down the graphs, almost if it were executing, but it's not evaluating any statements.

Looking for where changes get reached.

And so, here is what it does. Begins with enter, and enter prime node.

And looks it had S1 and S1', and asks does the code associated with those differ. And it doesn't So now, step forward to these nodes, and ask does the code associated with these differ. And it doesn’t.

Step forward again, does the code associated with these differ. And that case it does.

That's what we call a dangeroud edge, an edge leading to a change.

And the algorism selects the tests, which in the old program reached.

later.

Now we don’t have to walk any further down here, because any tests, any changes down here. The only test that can reach it, our one's that got into that area through this level.

So there is no need to analyse, the intercomponent of that.

And that's where sub-savings come in.

Instead we walk down the false branches looking for other changes there aren't in others.
And the tests we select are these 2. They're the ones that encountered change.

The test it doesn’t, T1, went around this safe space we call it where there are no changes.

[15:00]

So you can see how we stop more or less at a higher level change, without and save some analysis with that.

Pick the codes through change, tests through changed code.

As a second example, well, here we consider the case where there is no change here.

So the algorism walks down as forward, but this time it gets to these 2 nodes.

Looks it had at S5 and S5a, sees the difference, and selects that test.

Now it begins walking down the false branches, sees a difference here, and selects that test.

Doesn't have to go further beneath, though, it does continue down the false branch 3.

And doing that finds no other changes.

So again, it selects the same 2 tests, but for different reasons.

So here, this illustrates how we can find changes on multiple path with the algorism.

If they are not at the highest level, and how it deals with new code, and with deleted code.

One more case.

If we only have this change here, what would the algorism do?

It'd come down, select that test, start down this branch, comparing 6 to 6', 7 to 7', 8 to 8'.

Now it's got to look back at its 3 and 3'.

But it'd already compared those.

And you don't want it loop infinitely.

So actually what the algorism does as it walks, is keep track of which nodes compared to which nodes.

Later, if it's looking at them again, it realizes it doesn’t have to compare them again.

That's what ensures that it terminates, in the presence of loops.

So in this case, we pick only one test, the one test that reaches to change.

Here is the actual algorism.

It takes the program, modified program and test suite. Outputs the selected tests.

Begins by building the control flow graphs for the program in modified version.

And then calls recursive procedure compare on the entry nodes, are those 2 graphs.

What compare does is, compares colony, paranodes, and an N'.

It first marks as having been compared, I call that marking in N' visited, I could have said N, N' compared.

That's going to ensure that we don't compare them again.

And then, it considers the successors of N and N', on equivalently labeled edges.

So, the true successors in the false successors for instance.

If the 2 have been compared, there is nothing to do.

If the 2 haven't been compared, we are in here.

If the 2 are not lexically identical, that's the case where we run into a code change and so we pick the tests on the edge.

If the 2 are lexically identical, we walk forward and call the routine ?[18:23] between those identical children.

So that's what was going on, on those graphs that we saw.

Now, that's an algorism running on a single procedure really, I say, P and P' are really just single procedure.

Which is what you see here, graph for procedures.

And that’s not really what our goal is. It might be very cheap to test single procedure's unit testing.

We are trying to test entire programs.

So what do you do to extend this to entire programs?

To do inter-procedure test selection.

Other few ways you can do this.

You can compare all the pairs of procedures.

all assume the name equivalence between procedures and program P and program P'.

And compare the procedures.

One by one using the intra-procedural algorithm.

That might result an extra work though.

because some procedures, many procedures don't involve any changes.

So another thing you can do is to hook them together .

into an interprocedural control flow graph.

I know way back when we talked about control flow graphs.

I at least briefly mentioned you can connect ?[19:42] sites to entry nodes .

and you create an big interprocedural graph.

And then you can walk that graph.

or if you have a configuration management system, some version control system.

That will tell you things that have changed.

You can do some work and run the algorithm just on the pairs of procedures.

[20:00]

that it tells you have changed.

So any of this is a viable option.

before running this on whole programs and will see quite cheap actually.

So how does this algorithm do.

What’s its cost what's its potential for effectiveness.

Two ways to look at that analytically and empirically.

In terms of the custody algorithm In kind of big O terms.

We already know it's cheap to construct control flow graphs.

Time linear in program size.

The [20:44] of algorithm itself takes two graphs of sizes N and N' and it test set of size t.

Now there are actually two cases to consider.

One arbitary graphs.

In graphs that can have edges between any pairs of nodes.

In the worst case.

you might have to compare each node in one graph to each node in the other.

and on the order of each time do a set union of size t.

So that gives you this kind of loose upper bound t×n×n'.

That only happens in certain conditions .

It only happens, well I've called it when multiply-visited node condition hold.

What that really means is.

the only time that it happens is when in some graph.

you compare one node to more than one node in the other.

If you don't have that condition hold then the algorihtm runs in the time.

the test with times the smaller of the two graphs.

and the reason I bring this up is that.

the control graphs are not arbitary graphs, they are not fully connected graphs.

they tend to be well structured and in practice.

we've never ever seen a case where this behavior happens.

So it seems like in pratice that cost is size of test times with smaller of the graphs.

which you will see turns out to be pretty cheap in terms of run time.

Efficiency is an everything we would like our test to be effective as well.

So here too. I wish I had my... oh well kind of work with here too.

So this is your test suite what's your goal in selecting test?

And my goal when I worked for when I was at Pallette was to find.

egnore this red thing for the moment, egnore this for the moment.

was to find these test here, the test that reveals the faults in the program.

That's our goal in we are testing so we would like to find those.

Now the problem with that is that there is no algorithm that can do.

It's an undecidable problem to analyse a test suite with a program.

and locate the test that are going to reveal faults.

and if we did that we wouldn't need to run tests.

So when you are faced to something like that you try to find conditions.

under which you can solve the problem.

or relaxations that allow you to.

so we begin to think about a certain conditions.

so suppose the program ran correctly for all the tests on the test suites initially.

Working until late last night?

Suppose the program ran correctly for all the test in the test suite?

The last time we ran it.

Now that's not always realistic sometimes we often release with non-faults.

but in that case I will just take the test revaeled non-faults put them aside I run them anyway.

So in other words all i've got in T other ones are ran correctly last time.

So it's not unrealistic.

Now the second asummptions both the program

suppose test we contains no absolete tests thats a little more complicated than obsolete test.

is the test that no longer runs on the program or there is two ways.

It can no longer run along the program because maybe the inputs changed.

Now you got to throw it out make a new test or repair it.

or the oracle for that test has changed.

The expect output for the test has changed.

In which case you got to repair the oracle.

So often times there are obsolete tests.

[25:00]

But if you are goning to reuse your test at all you got to identify those.

retest all, selection whatever you to identify them.

So we are going to asumme that process you have gone through that process already.

You have identified obsolete tests.

you set them aside, you are going to repair them.

and all you are working with are the tests that still apply that are not obsolete.

In that case this program ran correctly last time and all the tests are suppossed to produce same output.

So which test can reveal faults?

Only the test that produce different output.

So the test reveal faults are the ones that have that are output altering.

So if these conditions hold and if I could find the output altering test.

I would have the fault revealing test.

The problem is there is no algorithm to find those either.

So we looked at one more condition and thats what we called controlled testing.

Controlled testing saids it like using scientific method.

When you do an experiment comparing techniques.

we have talked about this you would like to hold the other factors equal.

that might cause things to differ.

When testing you are comparing a version of a program to new version.

to try and assess whether the changes affected the output.

You would like to hold all the factors equal other than the changes.

and if you do that then the only thing that differs is the code.

and the only way you can get different output is with test that go through change code.

So when these conditions hold the test it goes to change code modification traversing.

include all those that can make different output reveal faults.

So if these conditions hold and your algorithm select those.

you can guarantee you will get the test you want.

A little bit about this it is actually about the P place hard to select those exactlly.

So the dejavu of the algorythm selects these plus potentially a few more and thats proven in a paper somewhere.

So the dejavu who has the property of being who say safe.

It can guarantee that you get the fault revealing tests.

If these conditions hold.

Now turns out there is two reasons thats interesting.

The first is the prior this work it was not known whether you can have a safe algorithm.

or whether conditions exsited for safety.

and so it is always nice to know what you can achive with algorithms.

and this showed there are conditions under which you can get safety.

but the bad news is this thing is hard to achieve.

What it really means is you've got to control all the factors that might affect your programs output. except for the change in the code

What is it mean you got to run it into on the same type of machine.

with the same type of memory with the same load on the machine.

with all non determinism removed from it's running.

So if you've got timing dependencies or on current programs.

this thing doesn't hold and so it turns out in practice.

there is lots of reasons that is difficult to make this happen.

and in that case you lose this condition and this is no longer a super set of that.

You can still select this test it's just you don't guarantee that they includ all these.

As a simple example you have probably run into the case.

well if you done a coding in C.

with dynamic memory allocation you can take a C program and run it one day and have it work finished

and then the next day run it in exactly the same way and have it fail.

simply because it was loaded in the diffrernt place in memory.

or at some dangling pointer pointed with illegal location rather than into sime legal location.

and there has been no code changes at all.

and it that case lets do the dinamic memory allocation.

and not being able to control where the pointers point.

in that case you might leave out the test.

that would have prevailed the fault.

But there still a question to how often this things happen in practice.

So we implemented this and we have tools that would create control flow graphs.

and tools with instermental code so that you can get traces profile it and traces.

and a tool that could put all thoes traces together into the test history.

so the test history in flow graphs that the algorithm uses to select tests.

so that all of the hard part here really is far as implymenting.

is building control flow graphs instrumenting programs.

[30:00]

this is only bit over five lines to see if I recall to actually that algorithm.

So we've done many studies of this by now published in various papers.

and I'll just tell you about a couple these are actually case studies rather than controled experiment.

but this is a case study of a...

Did I ever show you of this data here? I don't think so.

It's a study of a, it's actually a game software that written and see.

that implyments some play across the internet.

war game, Empire, that exsisted long time ago.

You still find it on the web but I'm not sure anyone plays it anymore.

There are still web pages about it.

It had, at that time 766 procedures or C functions really almost 50000 lines code.

I was able to find five versions of this that had been not sequential versions.

but in here was a relesaed in the system in different people it hacked it for different reasons.

so these are five different branches off of that initiall version.

and it had no tests but I used the main pages and the category partition method to build functionsal tests.

so these are specification based test the test different things you can do in the game.

Oh, I did mentioned that we did the torpedo thing from this didn't we.

that's from this game one of those main pages torpedo thing.

You can imagin doing TSL specs for all those cause we showed you one in here.

(student questioning)

Oh, no difference I don't know why I wrote procedures there they were C functions.

these days we used word functions for everything.

but when back in the days of Pascal it were procedures.

Pascal and procedures not functions, Fortran head.

I think they were procedures Fortran, what they were called anyway.

So here's the tails on the five versions some of them had relatively few changes.

couple of them had slightly larger changes in trems of functions in lines encode.

and the task here was to see these are real versions so giving this had s real changes.

and this crafted by the means of but representative test suite.

what would happen in test selective on this.

This is the percentage test selected by the algorithm before those change versions.

and you will see that this is a little over ten percent.

but as the five version there is percenrages almost all under ten percent.

of the tests go through the changes in these programs.

So that seems to suggest some savings you are leaving out.

you are able to omit over 90% of your test.

of course it also depends on the run time of the things.

If it took me an hour of analasis to save ten minutes of testing this isn't a gain.

so just numbers are the best thing to look at.

so we also looked at time.

as it happens this test suite for the system took between 7 to 8 hours to run.

If you ran all the tests and retest all approach.

thats running them and checking there outputs all automated.

but took a good deal of time.

So this is showing the five versions and how long it takes to run all the tests.

and this is showing the time it takes to build the flow graphs for the functions.

to run the dejavu algorithm and then to run the selective tests.

and thats almost all under an hours.

you are not saving 90% anymore because there are some analasis costs.

but the costs are really cheap in here...

just 10 to 20 minutes of analasis time building flow graphs and running the dejavu algorythm on these.

in order to save the testing effort.

so the time is being saved as well.

Now whather that matters depends on the 'does 8 hours matters to you'.

if thats an eight hours of human effort it probably matters.

if it's all automated you can run them over night or maybe it doesn't.

(student questioning)

You are right thats the wrong term.

very good thats been up there.

[35:00]

the term that has been on that slide in various forms for many years.

and it is not cost effectiveness at all.

It can be savings.

because it is a little more than efficiency.

Well it is savings in run time.

but there is nothing about effectiveness there because its nothing about the fault detection.

Thank you.

And I can tell you that, I don't think in this case we don't have any faults in the program.

The studies that I'm showing you are not going to look at fault detective behavior.

We always difference the output.

There were some cases where had faulty programs.

and we never missed any faults.

Now I would like to say thats because detacting safe.

but we also never had to control assumtions fully match.

We never really had control progression testing.

but it has ended up being safe in practice in this cases we have observed.

but I won't show you any of those results on the slides.

So Microsoft was interested in this.

and so we started working with them to see if we can implemented there.

and can get any gain from it.

and we had student working in ?[36:14] with them building tools to do this.

One of the first things we did was...

Actually this is before the student went to the ?

The first thing they gave us codes for their calculator.

Thich you see in all the world versions in whatever.

Which at that time at 27C functions, little over 2000 lines of code.

and they gave us 9 versions, these are sequential versions.

and they gave us three large test scripts.

Each test script had a bunch of separate little tests in it, you run those scripts.

Now between them there were 388 little tests within those scripts.

So they might organise them differently.

Right at the moment they came to us you can.

Run 1scripr, 2scripts, or 3scripts that's all you can do.

They could have broken them up into 388 scripts with one ?[37:10] thing they wanted.

So we looked at it both ways.

Because we were interested in the test design at the time too.

Here the versions.

Some cases very little code changed.

Which is exactlly when we were interested in the selecting subsets to test.

Some cases more code 12 out of 27 functions.

Just three functions for 245 lines to code.

and how we do a selection there.

Here are the results looks quite different than Empier.

bacause, let me tell you how to read this first, % tests selected

The 9 versions.

Here is the three test perapactive and there is 388 test perspective.

So the three main differences from the Empire results are there are cases where there in no savings.

…there's cases where there's no savings.

Here, you need all 3 scripts or all 388 tests, they all go through changes.

Here, you need all 3 scripts or most of the tests go through the changes.

So it's not always a win.

At the script level, there were cases where you could basically leave out a script.

One, one script didn't go through changes or leave out two scripts.

And some cases here that I'll get to in a minute.

But the, the kind of the finer grains tests, if you broke up the test into little tests, you could save more.

Even there you save more, here you save more, here you save more, here you save more.

Which is saying that within a single script, there could be a small percentage of the individual tests that went through the changes.

So this suggests that small tests may be better than large ones, it's really much more complex than that though, because you also have to consider, if you put,

If you take three test scripts, and they start up the calculator and apply a bunch of commands, you got that startup time and the commands.

If you…if you take them into 388 little tests, each test has the startup time which could be substantial.

And so, in terms of savings, sometimes, finer granularity helps selectivity in terms of numbers but whether it helps in terms of time savings well depend on other factors.

So we did later look at things involving test size in another piece of work.

But then there are these two cases where no tests were selected.

It's not any error by the algorithm, why do you think there were no tests selected in these cases?

(Student speaking)

[40:00]

I'm sorry what? Oh.

(Student speaking)

That's right.

Exactly, there are no tests to go through the changed code.

And maybe that's, that's why they needed to fix the code, right?

They hadn't tested it, they ran into an error in the field, and they had to test it.

So this just goes back to my point earlier then.

Just working with the existing test set may not be enough.

First, because you make changes to the program, second, because it may not be adequate to begin with.

I can also point out that these two cases were, oops, where all the tests, or most of the tests were updated, versions 4 and 7.

Where the cases where, lots of changes had happened.

It's not always the case, I mean, you can have one change and it happens to be, to the startup method of the program, and every test goes through it, you'll need every test with only one function, with only one line of code change, you could do that.

But in this case, that's what happened.

Now, they then tried this on NT, this is in their early days of NT.

And unfortunately, most of the times they got results like this.

Because they almost always changed the startup code.

And startup code, all the tests are startup code, they're always worried about how long it takes to startup, it always takes too long so they're always trying to, to make it faster.

So they ended, so that's when with them, we started looking at prioritization instead, because they couldn't select a subset, so let's put them in an order, so I'll talk about that.

But we did also start working with Boeing on this, and they were able to get savings much more like the empire case.

And that was working with, on their critical aviation software using a subset of eight, a subset of eight to eighty three which is used for critical systems and which is used for those because it contains no dynamic memory allocation,

it's very well-behaved, it's very safe and we were able to prove that, that subset of eight to eighty three allowed them to achieve the conditions for safety.

Which meant that they could use this algorithm, select a subset of tests, and know they hadn't missed any, which is important for them.

So, the, I guess the bottom line is sometimes it works, sometimes it doesn't, depends on what things change, where your tests go, test sweep designs, and other factors.

So that's selection.

Now prioritization is ordering tests.

As I showed you earlier, it's doing this.

I think we've talked about it earlier in class.

That's where I'd relate it.

I can select the subset of my tests and then prioritize them, fine.

I can prioritize and then select the subset, the first half so that, they can achieve similar things but in their, in their formal statements, selection's about picking a subset and not ordering them and prioritization is about ordering without doing any subsetting.

So prioritization has couple, a couple of things going with it and this you've heard something about this in here so I don't have to say too much but here's an objective function f, the thing you're trying to maximize,

and here's the heuristics you're trying to use to maximize it, I know we've talked about that.

Many different possible objectives you could have.

If you've got some regul, some agency or regulation that says you need to cover all the code, well the faster you do that, the sooner you can meet the regulation.

If you're doing reliability analysis, there's ways to get your estimates, your confidence levels and reach them earlier.

But the things I'm interested in is revealing faults earlier in testing.

So, you need a metric for that and the rate of fault detection metric is one of those.

And measures how fast you detect faults over time.

So if you have these ten faults detected by those five tests, for instance, test A to test those two faults, if you put the tests in order A-B-C-D-E and begin running them, when two-tenths of the test feed has been run, you detect 20% of the faults.

And you can map that curve and that area under it is the rate of fault detection.

Put the tests in this order, you get a higher area, a higher rate.

Here's the best order for that scenario, 84%.

How do I know it's the best, well I only know it because I already know which test results reveal which faults.

So that's not something we can achieve in practice, but afterward we can look at what a best order would have been, so that's a measure to compare things to.

[45:00]

So there's a formula for that in some papers but basically it comes down to measuring this area under this curve, this line.

So, I think I mentioned this earlier, we can't meet that function exactly, so we end up finding heuristics such as [?45:27]we can cover things.

So you'll probably recall that there's a, the, one of the notions is let's try and cover statements, the tests that covers the most statements might cover the most faults.

And if that's true then, if I have coverage of statements by two or three tests, here's what they cover.

The one that covers the most is t3, the one that covers the next most is t1.

So, I'm trying to cover as much code as I can.

So, the intuition is the more code you cover, the better.

One thing about this is, this test, excuse me, this test covers nothing new.

Everything in here was covered by this, this test does cover something new, s5.

So you might say it's not how much you cover, it's how much things that haven't yet been covered, that you cover.

It's the additional coverage you get that's better.

And that leads to additional statement coverage where you pick the one that covers the most initially and then the one that covers the most new things.

And now you've covered everything and just start again, you just keep doing that.

Yes.

(Student speaking) …I think covering many statements [?46:46] but I think some force are related to the, some specific sequence of statements.

Yes.

(Student speaking) Is it reasonable to select such test cases for five and four which are…

You might be able to come up with, I don't know if I've seen this, you might be able to come up with a prioritization metric based on sequences covered.

(Student speaking) Is there, are there work related…

I haven't seen that, I haven't seen sequences.

I'm not sure how to find these sequences.

One place, I guess one reaction I have is, there'd be a lot of different sequences so you'll run into a combinatorial problem and how do you rank sequences as better than others.

So we might be able to find that and do well with that.

Another place that apply with though, would be when, people talk about testing sequences of calls to methods, for instance when testing a class library, you want to test sequences of calls. And there are really is sequences like testing a stack, I want to do pop pop push, pop push push, things like that.

So their sequences become, that might be a firmer way, a place to look at sequences and prioritize by sequences.

It's an interesting thought.

So you could do the same, you could do it at the statement level, or you could do it at the function level.

I mean if this is a call graph representing functions, calling functions, if you've got an enormous program if you don't want to cover, want to measure statement coverage, you can measure functions covered.

And the same two techniques apply.

So there's lots of different sources of data you could use.

Code coverage data, I've just talked about statement coverage, function coverage here, but sequence coverage is another form of code coverage.

You could look at where chages are.

You could consider the cost of tests.

Different tests may have different costs, some tests may take an awful long time to run so you woulnd't want to necessarily treat them the same as cheap tests.

You could look at the criticality of tests, or how important they are really, do they test security critical code or safety critical code or not.

So, and, and you could look at the history information and people have looked at all of these things in one way or another but there's probably other ways that could be looked at.
So I'm not gonna show a lot of empirical results.

I'm just gonna show a case study we did and then some study from the literature.

And this is again, with the Empire program.

In this case, I, oops, this is later on in time and I found this really is 11 sequential released versions of the Empire program.

And we seeded faults in those versions, that's how they ended up having several faults each

And this is the same functional test suite I designed for the first experiment, for, with a test selection experiment.

[50:00]

And we compared prioritization techniques, their ability to improve the rate of fault detection.

We did them at the functional level, we did both total function coverage and additional function so additional is with feedback, essentially.

Additional coverage means pick a test, see what it covers, and now, now you've got feedback on that, you can choose the next best test.

So that means total and additional there.

And we compared them to random orderings of tests.

And this chart, I've simplified this to remove a couple other things we also compared to because I don't want to have to go into those but…

This is some of the, this is, across the whole, this is just one summary chart across all 11 versions, how did the techniques do.

Random's APFD on average was 72%, total function coverage eighty-something, additional function coverage close to 90.

And we've seen similar results…in general, in many, many different experiments, controlled experiments.

The heuristics out-perform random almost always.

The technique with feedback, a large percentage of the time does better than technique without, not always.

It really depends on what the coverage of the tests are and where the faults are.

So this has also been used, so Microsoft then, then started using this and they created, there's actually a paper on this where they created this whole prioritization system.

And this, I won't tell you the details, basically they created a system.

In some differences, they were actually looking on coverage of binaries not source code.

But they studied its use on binaries, here's a statement about it: "able to operate on large binaries built from millions of lines of source and produce results within a few minutes."

Bill Gates gave a keynote address at, I forget, it was a software engineering conference in, and actually said at Microsoft we're using prioritization to do blah-blah-blah-blah.

And so they published, this is some data from a case study talked about in the paper listed at the bottom there.

I've just summarized one piece of data just to show you.

And this is, what is this showing.

Well, first off, it's a case study on a 400kb binary built from over 3,000 functions.

There were known defects in it.

Test cases seems rather small to me but maybe these were like scripts or something, 221 test cases.

And they, what they did was, additional coverage, you pick the test that covers the most and then the one that covers the most new things, eventually you've covered everything and then you start again.

So you end up with these sequences of tests.

And that's what they looked at here.

The first sequence of tests, this is prioritization, this is a sequence of tests that covers everything that could be covered.

And, what is this showing, unique defects and defect, I'm sorry that's hard to see.

This first sequence of, of tests found 81% of the known defects, just the first sequence.

And of course, those were all unique defects, the first time.

The second sequence found another 8% new defects and also a bunch of defects that had already been found and after that, no more new ones were found.

So basically prioritization let them find faults a lot earlier here in this case study.

So, so it's being used by other companies that I know of but I still don't know of any commercial tools that implemented it, I think people implemented it themselves.

Last thing I'll talk about much more briefly will be augmentation.

Hello, here we go.

In augmentation is where after you've rerun tests or some tests, you figure out where you need new tests.

So the more formally, we've got a program P prime, test suite T.

There's two steps involved, determine which, this is usually coverage testing, determine which coverage elements aren't yet covered and create tests to cover those.

That's what augmentation is all about.

It relies on automated test generation approaches.

There are many to choose from.

We've been working with two different varieties, search-based algorithms and concolic testing

If you've seen something about concolic testing already.

So let me, these slides talk about the concolic approach.

[55:00]

This you've already seen, illustrated here quite recently.

Concolic testing just creates, you begin by generating an input, see what it executes, pick the last predicate on the chain, get the path constraint forth, the gate, the last part of the path constraint, via to a constraint solver and get a new input.

And go on like that and I'm just going to flip through these slides because we've already talked about that.

This shows how concolic works.

When using this in augmentation, we start with, we start augmentation we've got a program

modified, modified programs and tests that we start by picking the test cases that are affected by the changes.

The ones that aren't, we're not interested in.

For those, we update coverage and constraint information and we use the concolic approach using those.

So, here's an example.

Suppose you have a test suite there of five tests and they're coverage-adequate for this little program.

Suppose you change this, I've just changed it, well how did I change it, I changed greater than to greater than or equal.

Given the test inputs I chose here, this is a very precisely chosen example to illustrate.

Given this change, it causes tests to take different paths, your test suite is no longer adequate.

And so your job in augmentation is to cover, excuse me, these things that aren't, that now aren't covered.

First thing we do is pick the tests that matter to us, the affected test cases.

In this case, we use dejavu to find the tests that reach changes.

Which happens to be those four.

And so we work with those.

And now, in regression testing, we could begin by running those tests and that would let us get their…learn what paths they went through and get, calculate constraints for them.

Their...learn what path they went through and get calculate constraints for them.

So I want going to that tail to tail but we want to run them and get information need on those that led us now run a concolic approach.

And so augmenting the test suite means finding target branches, ordering them, and applying this, I'll show you rather than reading this to you.

We're in this situation.

There are the target branches.

There're many branches, they're no longer covered by the uh...we run those effective test find to what's not covered.

These are our targets.

Now, one thing we've always thought is there might be a better ordering which to consider.

Alcohol's to cover each of these.

And so we're going to iterate through them one by one and try and cover.

And we've always thought that maybe starting with this top one would be better, cause we can cover it, we'll kind of get something below it, we might get something below it for free.

And so we ordered the target branches, and sort of a debt first order.

Turns out as not quite as easy to gain using that but that's another story.

but at this approach we ordered them, and now we start dealing with them one by one.

and so pick the first branch, get its path condition, now we're just using standard concolic.

get path condition, that's condition to go down there, gate it, and now you have condition to go down there.

feeded that control server, hopefully you come up with a new test case.

Test case gets that target at all so gets another ones for free.

now pick the next target, do the same thing with it, pass condition, gate, new test case.

that's what we do.

so the big difference here, from say, regular concolic testing is we have this test suite we can start with.

and holishes how you can take best advantage of existing test suite if you have one.

and so this what approach does.

so we've done a few studies of this now, this is just going to report on one relatively simple case study that was looking at how efficient it is.

that completing coverage and how effective is this, we call this directed test suite augmentation, DTSA.

this case study was on the Grep utility, Grep C utility and K lines.

C5 versions, relatively large number of branches, about 800 test cases.

Those little cases don't achieve all that much coverage though initially of the program.

[60:00]

and performances here all at once, I just going to show just one table of a performance data but basically with augmentation.

this is showing how many additional braches we are able to cover with the augmentation

approach.

so there's 600 or some more.

so increase the coverage by 28 to 30%.

however there's still many uncovered branches there.

and concolic was not enough to find them.

so we'll later look to combining them with another approaches, actually I have one slide there.

but at least able to do those automatically, it took a lot of time between 9 and 14 hours of machine time to generate these tests.

still for humans to do that it would take much huger moment of time, this automated.

so that some suggestion that the computer time's be usual.

so I've told you about 3 different things that are going on that I do research on.

and we already know that there are some work on prioritization, realm of web apps that you see, see little more about...

That should say test case selection, I don't know it all aligns that, but prioritization...

but a lot of under stuff hasn't looked at yet, web application realm, and the same regression testing problems happen there.

the question is what has to change and we certainly see that things had to change for the prioritization.

things had to be altered to apply to that certain types of programs.

so I'll tell you just a little bit...actually I left out this is omitted from here, never mind that.

but I'll tell you why that's there.

just a little bit about each of these 3 things.

approving augmentation.

applying approaches to embedded software and adapting techniques to new process model.

but I did except the...LG I wanted to show them to people had successfully use concolic testing on real industrial programs and what Dr.Kim has done that in Samsung so...

I have couple a slides to show you that took much the same, that's not unrealistic, think you could use this, but I'm going to show here.

okay improving techniques.

the main things we've been looking at is ways to combine techniques, because the different algorithm for generating tests has different strengths, like concolic testing relies on constraints sever which don't always work.

they're limited what they function on.

and so there's other techniques that don't have those limitations.

and so for search-based test case generation approaches use other call search techniques, things like genetic algorithms.

simulating nearly in various, what should I say...they call search based algorithm, they come from other realms but people are using them in software [1:3:26] lot now.

they can be better in covering certain thing, so were combining algorithm to try to see what happens...

and this actually graph that shows some of the stuff you can gain.

ignore the dotted line, I just want to show you one thing here.

I think it's on GREP again.

And we're trying to cover branches.

And this is overtime, and this line right here is showing what happens if you run concolic.

If you run the concolic, augmentation approach to cover new branches.

and then after you consider all the branches, switch to genetic.

the genetic algorithm, another test generation approach.

and then genetic is able to cover many more.

and so doing concolic followed by genetic, you build up your coverage in this case, 400 to 600 branches.

by of course adding more time.

so that's just a simple linear sequence of algorithms.

so we try putting them together differently, we tried basic approach where we run concolic, itterating through targets to branches, so conside this branch, this branch, this branch...
and a minute concolic failed we switch to other algorithm.

and we run it until it failed, then we switch back.

so that's a simple way to interlive them, not the intelligent maybe.

but in that case, these both of them interlives, concolic for while, genetic for while and we quickly

reach, much more quickly reach...

[65:00]

were first off at the time where concolic was no longer covering anyting new, and we wait way above that.

and even in the end we are able to get more things that one alorithm followed by the other.

That might seem a little counter-intuitive but reason is concolic algorithms take...

well anytime you consider a new path, you might...new test to the program, you might can get someting out of it and so going through here genetic algorithm would generate new task which concolic call can use to do better.

here the genetic was all after concolic, so all this new test were never tried again.

It ended up not reaching the same height.

but that is a just early results that show that some forms of interliving with these should be useful. that's where hard to find what's the best way to do this.

embedded software, well that's a whole different realm.

here are work that's focuse on some different stuff and involves using different oracles.

oracles are agian well...what is the same...

monitoring systems have difficult wide range of more difficult defaults to ditect to do the concurrency, races things like that.

it turned that what we found is that instead of checking your outputs, out put based oracles.

if you monitor internal states you can find defaults more effectively.

do I have more...yeah this is little bit of data on that, where we put in some internal monitoring program states to trying to detect various defaults.

and compare our newly detected defaults of different classes to the case we're just using outputs. so if you use properties internally you can get much higher default detection if you use output.

so like, one block management defaults internal properties you can detect 83% of those, where just checking output you're just getting 24%.

this class defaults buffer management defaults we can get them all with property based, we get non without the base.

that's so the...we have a big ?[1:7:37].

the next thing we're looking at this is using these properties and regression testing.
and trying to...while you can imagine prioritizing test based not just on what they cover but also

something related to properties they test.

(student questioning)

I had to refer you papers to details on those there, but there...what do I mean by property.

It really kind of means, they were [1:8:16] means dissortions.

so with respect to crinical sections.

okay you can make an assortion by crinical sections.

the variables suppose to be detectd in a critical sections should not be accessed by some other thread or task when you're currently on deception in the first task, so that's a propety.

and now we follow up to implement that property to check things and program.

(student questioning)

you don't need to...the work for the researcher is to determine what assortions will test that property.

so what student working on this did once we state the property and what can we look for to code you imagine.

you can imagine an analysis to determine what variables are accessed, critical sections and runtime.

checks whether it's being accessed by others.

so you end up putting the checks to activate the runtime, but you can automatically ride a tool that what's the assortions had entered.

so for instance, it's a ?[1:9:39] but looks for critical sections and put the right assortions.

so if you just ask me I can [1:9:50].

so that's something we're doing there.

and then new process models.

[70:00]

are we look at...I was talking about this model.

that's what people always used to, there're still a lot of companies to do that.

but those were in the days when you made a new system, it has a release, you deload it onto a some physical device [1:10:19], deloaded on CD or the early days called TK50 where were the tapes, you couldn't do updates frequently off the code.

but these days of course we get updates all the time.

so if you are in the company, you can do much shorter on maintenance cycles and much shorter testing cycles if you want.

so this is just a illustration of that, this represents a month, this represents a week.

and a traditional model you could have cycles of 5 days of maintenance followed by the 2days of testing, not a Samsung where they were 7days a week but 5 days of maintenance and the 2days of testing in America company might work.

then at the end they'll do the system testing.

even do even more you can do lot of many [1:11:18] test everynight.

even when people aren't still working, still run test of course.

whatever had been checked in out.

code during the day, they test at night.

what interesting here is it really changes the landscape, the situation for regression testing.

cause as I said ?[1:11:41] looking at do a lot of analysis here, such as gathering, we gather all the infornation for tests.

you going to run all test and measure the coverage, and you don't want it do it on there.

you're not trying run your test, so you can run all the tests you figure out this coverage in there.

so you have to do it there.

so this gets a lot of preliminary time, but that reduces that time and reduces the amount of time we have for test.

so you have to look at new versions of that algorithms in this case.

and trying to finding the things fit in the times you have and so the [3:12:17] are different.

It ?[1:12:21] brings up a lot of options for different and interesting algorithms that might work in this situations.

And we're working those.

I think that mos ?[1:12:32] illustration is, that just a paper that looks at the running tests between ?[1:12:41].

The notion here ?[1:12:43].

That programmers making their edits program, and they'll stop for couple minutes, rolling a coffee, pushing and start out a test.

but how you do it that between edits, maybe it's not a stable but ?[1:13:1].

that's what they really call continue, this is actually what the continuouses.

so that discrete.

but that's pretty how ?[1:13:14].

you know, Jeong Seok was my main student in North Dakota state.

those are my current [1:13:27].