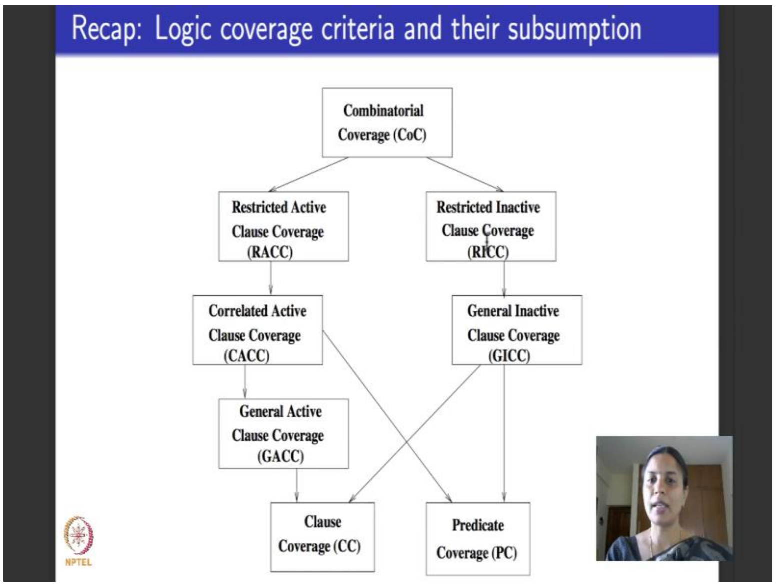
Software Testing
Prof. Meenakshi D’Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 27
Logic Coverage Criteria: Issues in applying to test code

Hello again. Welcome to the third lecture of the sixth week, we are in logic coverage criteria. Last two classes, I told you about how logic coverage applies to source code.

(Refer Slide Time: 00:26)



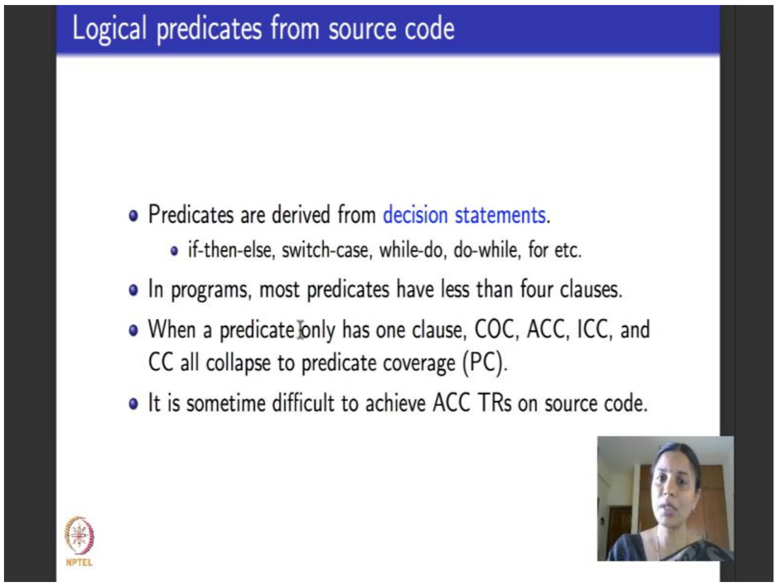
Recap: Logic coverage criteria and their subsumption

$$\begin{tikzcd}
 & \text{Combinatorial Coverage (CoC)} \arrow[dl] \arrow[dr] & \\
\text{Restricted Active Clause Coverage (RACC)} \arrow[d] & & \text{Restricted Inactive Clause Coverage (RICC)} \arrow[d] \\
\text{Correlated Active Clause Coverage (CACC)} \arrow[d] \arrow[drr] & & \text{General Inactive Clause Coverage (GICC)} \arrow[dl] \arrow[d] \\
\text{General Active Clause Coverage (GACC)} \arrow[r] & \text{Clause Coverage (CC)} & \text{Predicate Coverage (PC)}
\end{tikzcd}$$

This is a recap slide. We have seen so many logic coverage criteria and using examples of 2 kinds of programs thermostat and detecting the type of a triangle. Using those example we saw how to practically apply predicate coverage, clause coverage and correlated active clause coverage. These 3 are considered to be the most useful amongst the various logic coverage criteria.

You would have hopefully realized through the examples that we had to do a good amount of work for even a fairly reasonably sized program like triangle type which had a good number of predicates to be able to reach the inner predicates, and solve for the internal variables, rewrite the predicates purely in terms of inputs. Then they looked very big and then the tables getting the tables for active clause coverage criteria was a bit of an effort.

(Refer Slide Time: 01:27)

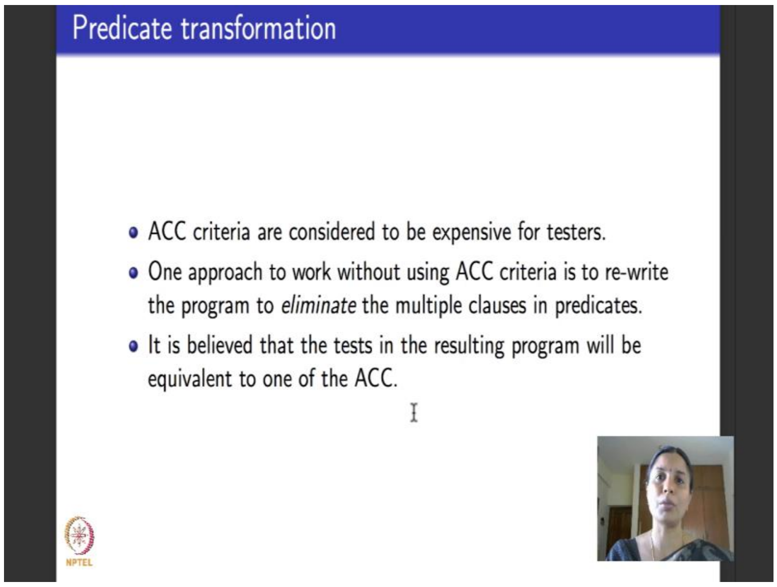


Logical predicates from source code

- Predicates are derived from decision statements.
  - if-then-else, switch-case, while-do, do-while, for etc.
- In programs, most predicates have less than four clauses.
- When a predicate only has one clause, COC, ACC, ICC, and CC all collapse to predicate coverage (PC).
- It is sometime difficult to achieve ACC TRs on source code.

So, imagine few thousand lines of code, which is what a standard program is all about and applying logic coverage criteria to that. It is going to be pretty non trivial. So, there is always been a bit of a debate, you know when predicates come from these kinds of statements and predicates have less than 4 clauses and if a predicate has more than one clause then, we said in the last class in the slide that active clause coverage criteria is very useful. But we saw through examples that its usually a little difficult task right. It needs a lot of human intervention and explicit knowledge about the code to be able to apply active clause coverage criteria and derive the test requirements for these criteria.

(Refer Slide Time: 01:59)



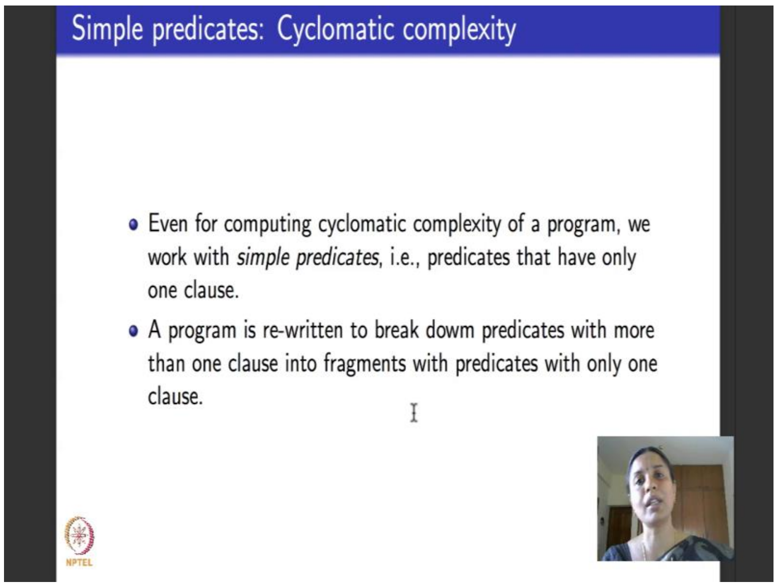
Predicate transformation

- ACC criteria are considered to be expensive for testers.
- One approach to work without using ACC criteria is to re-write the program to *eliminate* the multiple clauses in predicates.
- It is believed that the tests in the resulting program will be equivalent to one of the ACC.

So, there is been a lot of debates in the programming community. In fact, there is been a fairly large school of thought which says that why do we consider active clause coverage criteria because predicates has more than one clause. At least in those 2 examples we saw that predicate had several clauses, sometimes up to 4 clauses and then solving for the variables in that clause substituting back to them, taking each clause in turn be a major clause, filling up the truth table for correlated active clause coverage took a bit of time. So, there is been this school of thought that why not to rewrite the program to eliminate these many clauses in the predicate. Rewrite it to make sure that each predicate has one clause. I will show you some examples of what it means to rewrite a program.

So, lot of people have been thinking. And in fact, at some point in time there was a conjunction going around that if you take a program which has a predicate with let say 2, 3 or 4 clauses then using the semantics of operators like and nor which connect the clauses you will be able to rewrite the program by breaking the if statements into smaller statements. And it was believed that the resulting program, if I apply predicate coverage it will be the same as applying active clause coverage in the given program. Very soon it was proved that this is not correct.

(Refer Slide Time: 03:38)



Simple predicates: Cyclomatic complexity

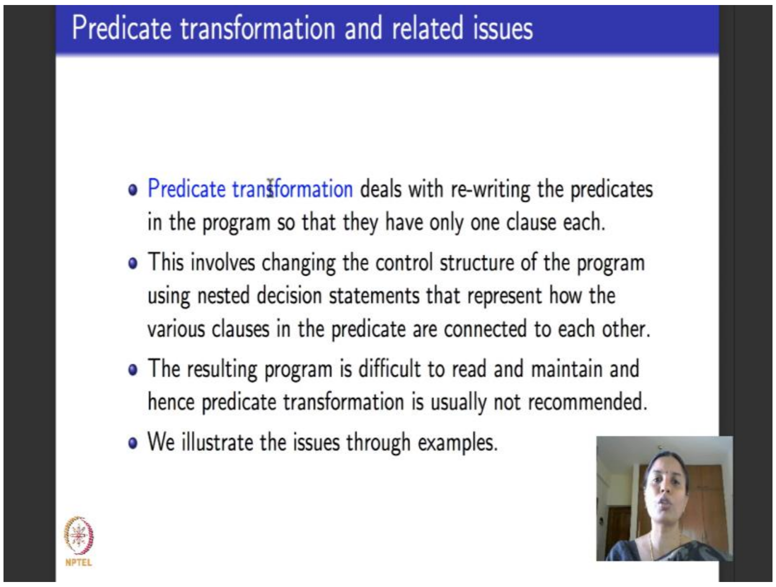
- Even for computing cyclomatic complexity of a program, we work with *simple predicates*, i.e., predicates that have only one clause.
- A program is re-written to break dowm predicates with more than one clause into fragments with predicates with only one clause.

I will show you a examples that illustrates that predicate coverage in the transformed program is not the same as the desired active clause coverage. It could be correlated, it could be general either cases it may not be the same, we will see it through examples.

Another reason why people believed that predicate transformation is necessary is, remember I told you what how to do source code classical testing for source code write in the third week of or something like that. So, there we saw this notion of cyclomatic complexity which determined the number of independent paths in the program. One of the basic conditions in the cyclomatic complexity is when I consider control flow graph of the program all the predicates in that should be what are called simple predicates which means now, in terms of logic coverage criteria that we have seen, they should have exactly one clause.

So, it was recommended that you rewrite the program to be able to do that. So, there are lots of schools of thoughts which keep recommending that if there is a predicate which has 3 or 4 clauses make it simpler, rewrite it. What I want to do in today's lecture is using examples show you that it may not always be a good idea to rewrite the program to break down the several clauses in the predicate. It usually does not work all the time. We will see 2 examples, try to rewrite the predicate which come in this examples and see what are the issues that we will get, right.

(Refer Slide Time: 04:48)



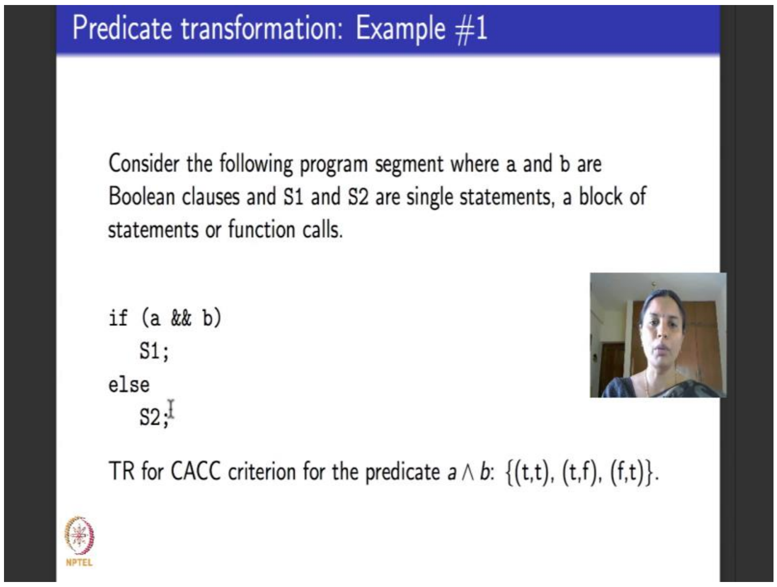
Predicate transformation and related issues

- Predicate transformation deals with re-writing the predicates in the program so that they have only one clause each.
- This involves changing the control structure of the program using nested decision statements that represent how the various clauses in the predicate are connected to each other.
- The resulting program is difficult to read and maintain and hence predicate transformation is usually not recommended.
- We illustrate the issues through examples.

So, what is the notion of predicate transformation ? The process of predicate transformation deals with rewriting the predicates in the program so that they have, in the resulting transformed program each predicate has only one clause. This obviously involves change in the control structure of the program. Like for example, if I have if a and b which has 2 clauses a and b, I break it up I have to put nested statements. First test for if a: if a is true then you test for if b: if b is true then it means both a and b are true then you write the then part of the original predicate here.

So it obviously means adding more control structure to the program. Lot of nested if statements lot of else statements right. The resulting program becomes very difficult to read. It has too many if statements too many nested if else statements then you do not know what is happening where, what is going where and usually because of this people do not like predicator transformation. Also programs large programs are written and supposed to be maintained for several years. So, for maintainability which means a third programmer or a developer or a maintenance engineer who is going to be able to look at your program and analyze it we will find it very cumbersome if you had written these complicated nested if then else to be able to just simplify for the sake of avoiding predicates with multiple clauses. It does not work, I will show you why using 2 examples.

(Refer Slide Time: 06:14)



Predicate transformation: Example #1

Consider the following program segment where a and b are Boolean clauses and S1 and S2 are single statements, a block of statements or function calls.

```
if (a && b)
   S1;
else
   S2;
```

TR for CACC criterion for the predicate $a \wedge b$: {(t,t), (t,f), (f,t)}.

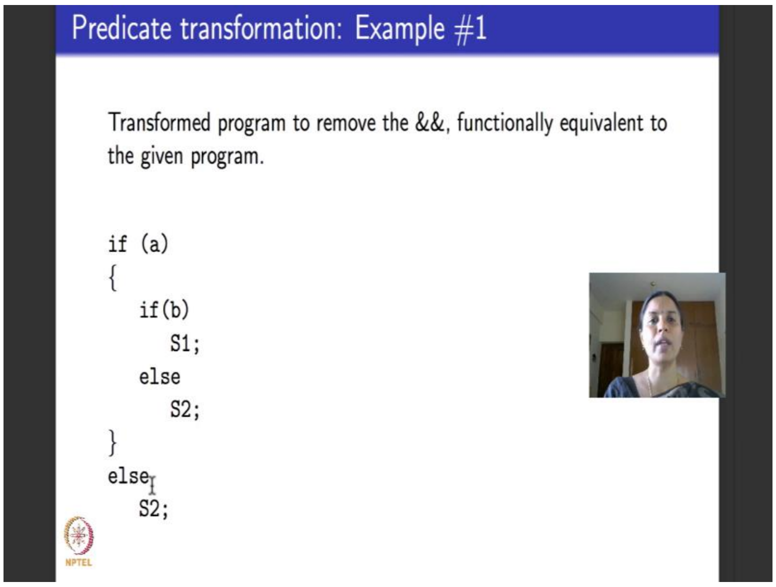
Here is a small program fragment, it is only a program fragment not the entire program, meant to illustrate issues with predicate transformation.

So, this program fragment has 2 clauses a and b there is a predicate here in the if statement which says if a and b then you do statement or set of statements S1. S1 could be any statement, single statement, a block of statements, could have if again inside that, could have other decision statements like while could have function calls. We do not care about it we just abstract it out call it S1. So, this clause says if a and b do S1, otherwise you do S2. This is just a small fragment. Now the predicate under consideration is this one a and b, as you can see it has 2 clauses.

So, I want to let say be make apply correlated active clause coverage criteria to this predicate I want to apply CACC criteria for this predicate. Now what you have to do? You have to do the same old exercise a be the major clause work out pa that is when a determines p, and then consider b to be the major clause workout p b which says when b determines p put them together to obtain correlated active clause coverage criteria where a becomes true, false once b becomes true, false once. We worked up the same example p is equal to a and b when we did this in the slides and inferring from those slides that we did for CACC for a and b here is the TR. I am not re doing it you can refer back to those slides.

The final test requirement for CACC for a and b will be this. How do you read this ? The first pair says make a and b true, second pair says make a true b false, third pair says make a false b true. Between these 3, correlated active clause coverage is completely satisfied for the predicate p which is a and b in our case. Now the goal is we will looking at program transformation this predicate has this 2 clauses. For some reason I do not like it, I do not want to do this CACC criteria, I want to break this 2 clauses and make rewrite this program into a program that has predicates that contain exactly one clause.

(Refer Slide Time: 08:35)



Predicate transformation: Example #1

Transformed program to remove the &&, functionally equivalent to the given program.

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

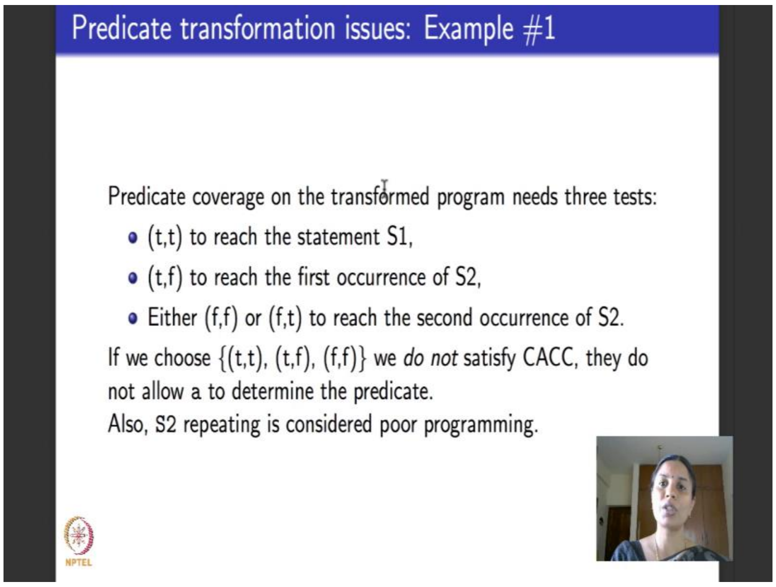
So, here is an attempt at rewriting. This program, in this slide, if a and b do S1 else do S2 is rewritten like this. How do we understand what this is? So, if you read this program it says if a then you go here then you test for if b.

So, let say both these fs pass then what is it mean it means a is true and b is true or going back by the semantics of the given program a and b are true then what do I do? I do S1 it is a same thing that I do here. If a is true and then b is true I do S1 else, this else is for this if, which means a is true and b is false. So, a and b is false then I do S2. What does this one say? This one is the else for the first if here it says if a is false then you do S2. So, what I have achieved here, I have taken the program in this slide which says if a and b are both true do S1 else do S2 and I have broken it up like this I said if a is true and b is true do S1 else which means if a is true and b is false do S2. If a itself is false do not bother about checking b because a and b is going to be false directly do S2. So, this captures exactly the same semantics as this program, but it is rewritten to ensure that each predicate here has exactly one clauses.

So, there are 2 predicates one just has clause a one just has clause b. I have eliminated the predicates containing 2 clauses here and this is the new control structure that I have arrived at. Now this does not look too bad but typical experienced programmers will not like this control structure. There are several disadvantages to it. One is people do not like the fact that this entire code segment S2 gets repeated here. That is not considered good programming practice at all and we cannot avoid it here, because we had our goal was to remove and that was there in the previous predicate and that process we ended up creating another copy of the code for S2. It is not consider a good programming practice.

And the other thing is now let us consider predicate coverage for this transformed program. Remember the conjecture was this is the transformed program, I have removed predicates with multiple clauses. If I achieve predicate coverage in this it will mean that achieving a appropriate active clause coverage in this original program. What we will show is that predicates coverage in this transformed program will not satisfy CACC criteria for the original program. What is predicate coverage for this transformed program? Predicate coverage for this transformed program will need three test cases: first one is a is true, b is true that will reach this statement S1.

(Refer Slide Time: 11:07)



Predicate transformation issues: Example #1

Predicate coverage on the transformed program needs three tests:

- (t,t) to reach the statement S1,
- (t,f) to reach the first occurrence of S2,
- Either (f,f) or (f,t) to reach the second occurrence of S2.

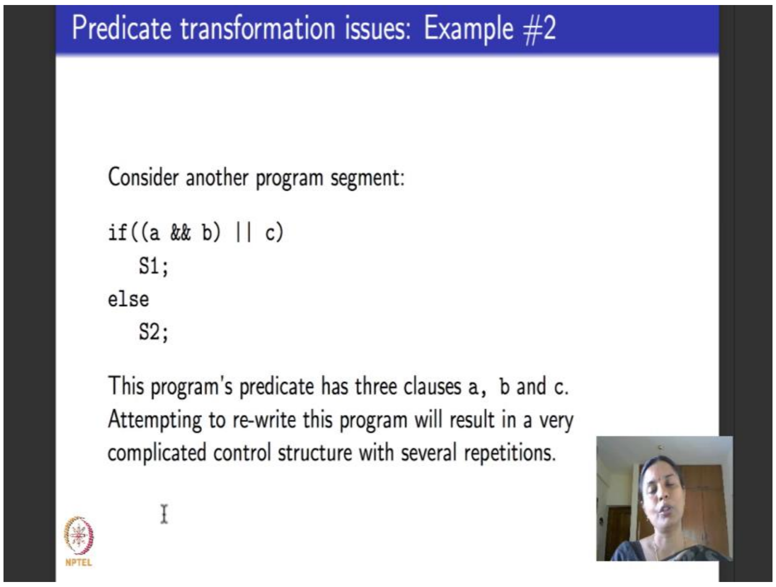
If we choose {(t,t), (t,f), (f,f)} we *do not* satisfy CACC, they do not allow a to determine the predicate.
Also, S2 repeating is considered poor programming.

Second one is a is true, b is false, that will reach this statement S2. The third one which is a is false as I told you, but b could be either false or true which means a false and b false or a false and b true. Both these test cases will reach this S2. So, the first test case for predicate coverage is both a and b are true, reach S1. Second test case is a is true b is false, so, reach this S2. The third test case which needs to reach this copy of S2 can be achieved in 2 ways: when is both a and b could be false or a could be false and b could be true both of them will reach this second copy of S2.

So, that is what is given here. But let say I have a choice here in the third case right for either taking both of them to be false or a to be false and b to be true, but let us say I take both of them to be false. So, the final set of test cases that I choose for predicate coverage are: true, true to reach S1, true, false from the second item to reach S2, the first occurrence of S2 and from the third one I choose false, false, to reach this second occurrence of S2. So, this satisfies predicate coverage for the transformed program, but it does not satisfy CACC for the original program. CACC for the original program as we have discussed, had needed 3 test cases a true, b true; a true, b false and a false, b true. This was the only option left where as in predicate coverage I had the choice between choosing this or this and I decided to choose both to be false because of that I do not satisfy CACC, but I satisfy predicate coverage.

So, clearly the conjecture that you transform the program, breakup predicates with more than one clauses into predicates with single clause and try to do predicate coverage on the transformed program that will be equivalent to CACC or GACC on the given program that is not true for this example. Also the transformed program has this not so welcome feature of the code fragment S2 being copied ditto twice, that is usually not accepted.

(Refer Slide Time: 13:32)



Predicate transformation issues: Example #2

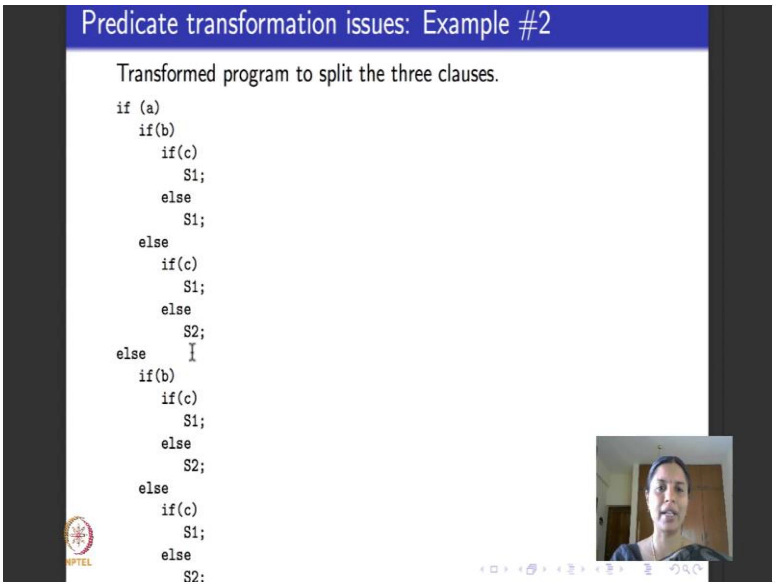
Consider another program segment:

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

This program's predicate has three clauses a, b and c. Attempting to re-write this program will result in a very complicated control structure with several repetitions.

So, I show you one more example to understand the same issue, but we will look at it from a slightly different prospective. Here is an example that has a predicate with 3 clauses that are connected like this: a and b or d with c. If a and b or c is true then you do S1 else you do S2, right. So, this is easy to read, if you notice because I just can read this directly any programmer will understand what is happening here. It says if e a and b is true or c is true then you do S1. If it the case that both of them are false in which case the whole thing is false then you do S2. Now let us attempt to rewrite this program. You can attempt in several different ways. Whatever ways you attempt, we will realize that you will get a very long program with several occurrences of S1 and S2 repeated and it will have a very ugly looking control structure.

(Refer Slide Time: 14:26)



Predicate transformation issues: Example #2

Transformed program to split the three clauses.

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

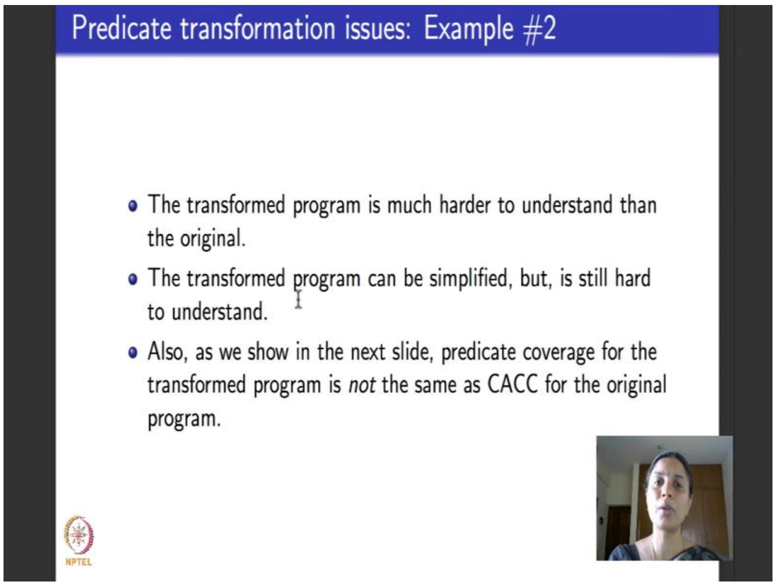
So, here is an attempt, one attempt at rewriting the program. So, if you look at this itself you will realize that this looks very complicated right and quite difficult to understand unlike this, this is neat. The only thing that you had here was you had a predicate with 3 clauses, but otherwise its quite neat and simple. For some reason you wanted to eliminate this predicate with 3 clauses, but you wanted to retain a new program where each predicate as a single clause with exactly the same meaning and here is an one attempt to rewrite it. How do I read this? I read this as follows I say if a is true then you check if b is true b is also true. Then you check c is true if which means if I am here if c becomes true, then what does it mean? It means all 3: a, b and c are true then what do you do you do S1. If c is not true, but a and b both true then you still do S1. That correctly fix to the semantics of this program right. If a and b are both true and c is either true or false then this whole thing will be true all the cases I do S1.

So, I have simulated till here. Now let us look at this part. This part has an else that corresponds to this b. So, we are in a situation where if a is true, but b is false which is this else then you can still do S1 if c is true because that is what the semantics of this predicate is. So, you check if c is true then you do S1. Tf c is false and b is false which means his whole predicate is going to be false then you have to do S2. So, then you S2, is that clear. So, now, let us go continue, we have not finished all the cases. So, now, let us look at this else, if you go straight up corresponds to this first if which checks if a is true.

So, it means if you reach this else it means that a is false. So, I check if b is still true a and b will be false, but if c is true then the or statement of the predicate in this slide becomes true. So, I can still do S1, but if c is false then I do S2 and again I check if b is false if c is true then I do S1, but if c is also false then I do S2. After this long drawn effort I have finally achieved to rewrite the program to capture the exactly the same meaning as this predicate.

But look at this program. You would agree with me that it looks completely ugly and unreadable. Just because I did not want this predicate, I try to attempt a transformation and that resulted in this. By the way this is not the only transformed program. There are several other ways of transforming it I just wrote this longest transformation in some sense to illustrate how ugly a transformed program can get.

(Refer Slide Time: 17:06)

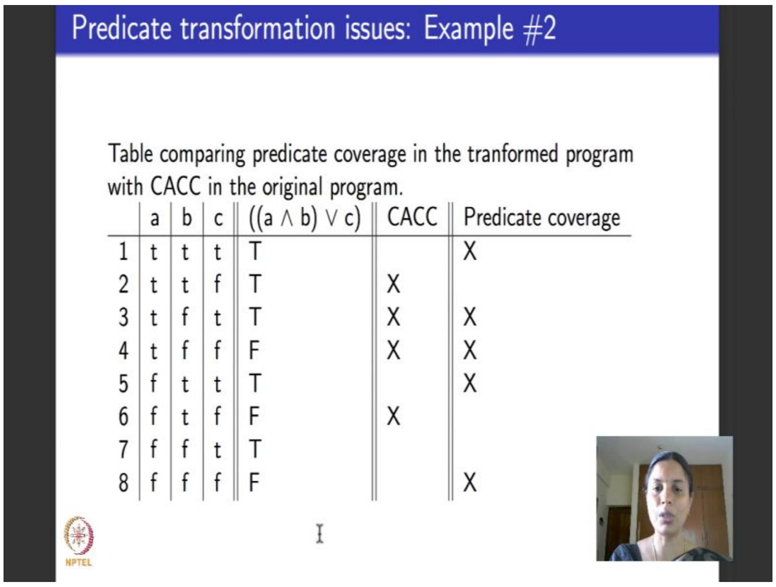


Predicate transformation issues: Example #2

- The transformed program is much harder to understand than the original.
- The transformed program can be simplified, but, is still hard to understand.
- Also, as we show in the next slide, predicate coverage for the transformed program is *not* the same as CACC for the original program.

But the point that I wanted to illustrate is that whatever is the transformed program its not unique, this is not the only transform program you could try to do it and you could probably get a simpler one. The transformed program is usually much harder to understand than the given one. You would agree with me because just by seeing his example and going back to our logical coverage criteria conjecture which said that predicate coverage on this transformed program should be the same as CACC on the given program, that is also false for this program as it would illustrate.

(Refer Slide Time: 17:39)



Predicate transformation issues: Example #2

Table comparing predicate coverage in the tranformed program with CACC in the original program.

	a	b	c	$((a \wedge b) \vee c)$	CACC	Predicate coverage
1	t	t	t	T		X
2	t	t	f	T	X	
3	t	f	t	T	X	X
4	t	f	f	F	X	X
5	f	t	t	T		X
6	f	t	f	F	X	
7	f	f	t	T		
8	f	f	f	F		X

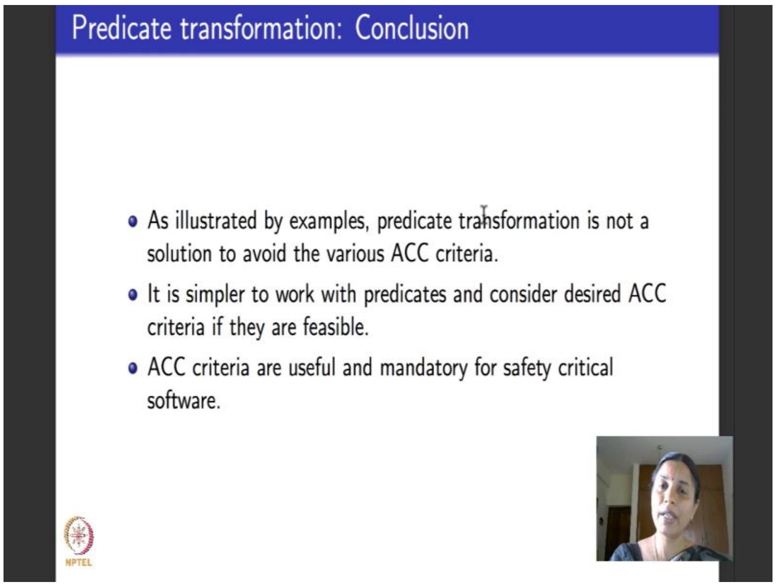
So, here in this slide. I have given you details of correlated active clause coverage for the predicate which was this. I have written it in terms of logical notation 'and' and 'or', but it is the same predicate a and b or c that was here, a and b or c. Here its written in programming notation, here is written in logical notation, but it is a same predicate. This part the first five columns illustrates the truth table for the predicate. Predicate had 3 clauses a, b and c each clause can take true false values in turn and this is the entire truth table that assigns true false values to a, b and c and this is how the predicate evaluates it evaluates to true and all of them are true. It evaluates to false when b and c are false resulting in the whole thing being false it also evaluates to false when a and c are false or all 3 are false.

So, they tell you what the predicate evaluates to. Please do not get confused to this capital t and small t they both mean true. I have written it as capital t just to illustrate what the resulting truth false values of the predicate is. Now if you see to achieve predicate coverage on the on the resulting program I can choose rows 1, 3, 4, 5 and 8 that will completely achieve predicate coverage. Why is that so, because if you see row 8 corresponds to predicate being false 5, 3 and 1 respond to predicate being true, row 4 also corresponds to predicate being false.

So, predicate coverage is achieved on the transformed program where each clause takes turns to be true and false resulting in the predicate to be true and false. But correlated active clause coverage on the original program, if you work out all the details by computing each clause to be the major clause do p a, p b, p c. Write out the truth table and do that exercise which I have not given here because we have done it for enough examples if you do that and do the final marking, the TRs for CACC will happen to be rows 2, 3, 4 and 6. So, if you compare the last 2 columns, you will realize there is immediately a miss match right, because row one satisfies predicate coverage, is not needed for correlated active clause coverage, row 2 satisfies correlated active clause coverage, but does not achieve dedicate coverage.

Similarly, row 6 satisfies correlated clause coverage does not achieve predicate coverage and row 5 satisfies predicate coverage, rows 5 and 8 satisfy predicate coverage, but are not needed for CACC. So, there is a lot of mismatch between these 2 columns which means that the TRs for CACC and for predicate coverage do not match for this example.

(Refer Slide Time: 20:26)



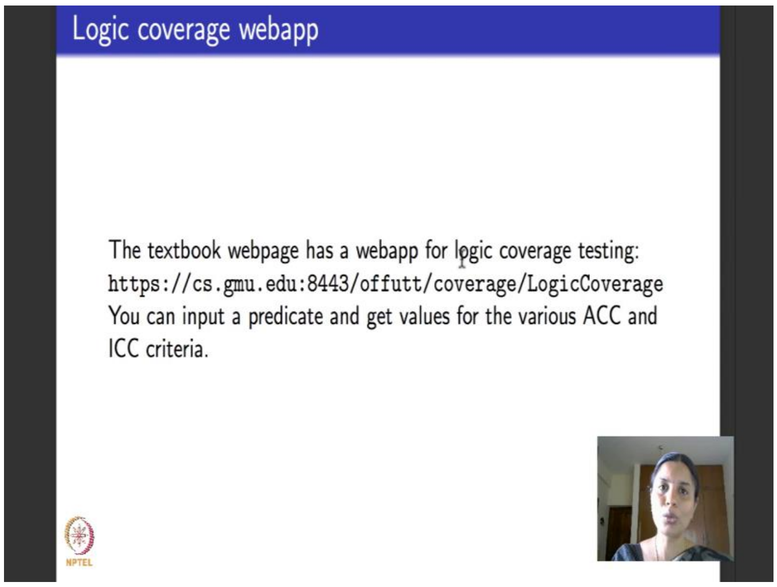
Predicate transformation: Conclusion

- As illustrated by examples, predicate transformation is not a solution to avoid the various ACC criteria.
- It is simpler to work with predicates and consider desired ACC criteria if they are feasible.
- ACC criteria are useful and mandatory for safety critical software.

So, what I want you to illustrate to you using these 2 examples is that predicate transformation is not a good solution to avoid using active clause coverage criteria. Active clause coverage criteria is mandatory for testing safety critical software and has to be proven effective for finding lot of errors, especially for predicates that have 3 or 4 clauses.

So, it is a good idea to be able to go ahead and use it and transforming a program to remove those extra clauses in the predicate with the aim of avoiding active clause coverage criteria will not work. They are useful, they mandatory and it will be good to get used to it.

(Refer Slide Time: 21:08)



Logic coverage webapp

The textbook webpage has a webapp for logic coverage testing:
https://cs.gmu.edu:8443/offutt/coverage/LogicCoverage
You can input a predicate and get values for the various ACC and ICC criteria.

In fact, to help you get some more practice, I encourage you to look at this book website. Most of the material as we have seen is derived from this text book called introduction to software testing. The book has a very good web page. By now you all should know the webpage. In that webpage there is a page that contains web apps. So, there is a web app for logic coverage criteria here is the URL for that app.

So, here you can input a predicate and get values for the various active clause coverage criteria and the various inactive clause coverage criteria. I do not think they do predicate in clause coverage because they are easier coverage criteria to meet, but they have this app that works well for ACC and for various ICC also. Try use it for a some predicate of your choice with 3 clauses or 4 clauses, work out the solution manually then use the app to see if your answer matches with the answer given by the app. The app slows down a bit for predicate with larger clauses because they underlying problems that it tries to do to get these coverage criteria of each clause determining a predicate are non trivial problems.

So, try and use it for predicate with 3 or 4 clauses. That way you can work out the solution on your own and try and see if the solution that you have got is the same as the solution that the web app provides, and feel free to ask me if you have any doubts in the forum.

Thank you.