

Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

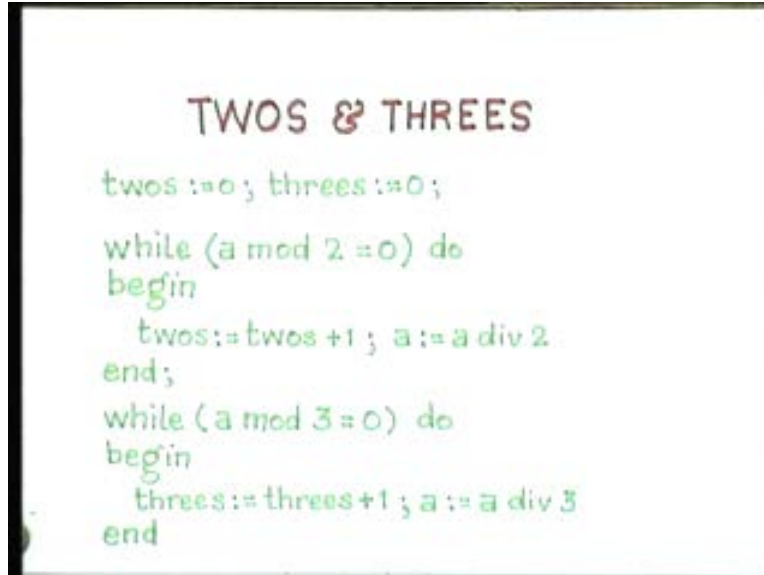
Lecture no 24
Lecture Title: Non-Determinacy

Welcome to lecture 24. We will have a short talk on non determinacy. There is a lot of confusion about the word, non determinacy. There is non determinism as opposed to determinism and determinism is an 'ism'. It is a philosophy which talks about destiny and fatality. Determinacy is a property of a system which is predictability. They are closely related but they are not the same. There is correspondingly also indeterminacy which often means you do not know. It might be deterministic but indeterminacy usually just means that you do not know.

Non determinacy means that there are several possible choices and you cannot predict which choice would actually be taken. For example division by 0 is indeterminate. So, non determinacy is a property of systems by which there are several possible alternatives may be an infinite number or a finite number but there is usually at least more than one possibility and the property of determinacy itself is a degenerate case of non determinacy where there is exactly one option. We were looking at the case statement and this non determinacy or choice is really a generalization of the case statement. But let us motivate it by an example. Let us say given an arbitrary positive integer, you have to find the largest m and n such that $2^m \cdot 3^n \mid \alpha$. It is a small part of unique prime factorization where you are restricting yourself to just the powers of 2 and 3.

Let us just look at this. α is '0'; m and n are infinite and indeterminate. But when ' α ' is positive, you will have some definite number m and n . Here is a simple Pascal like program segment which just counts the numbers of 2s and 3s found so far. One possibility is just that while you have found that 2 is still a divisor of ' α ' just keep incrementing 2s by 1 and keep dividing ' α ' by 2 and once you are through with all the divisions by 2 you start with the divisions by 3 and you find all the number of times 3 divides this number.

(Refer Slide Time: 4:30)



```
TWOs & THREEs

twos:=0; threes:=0;

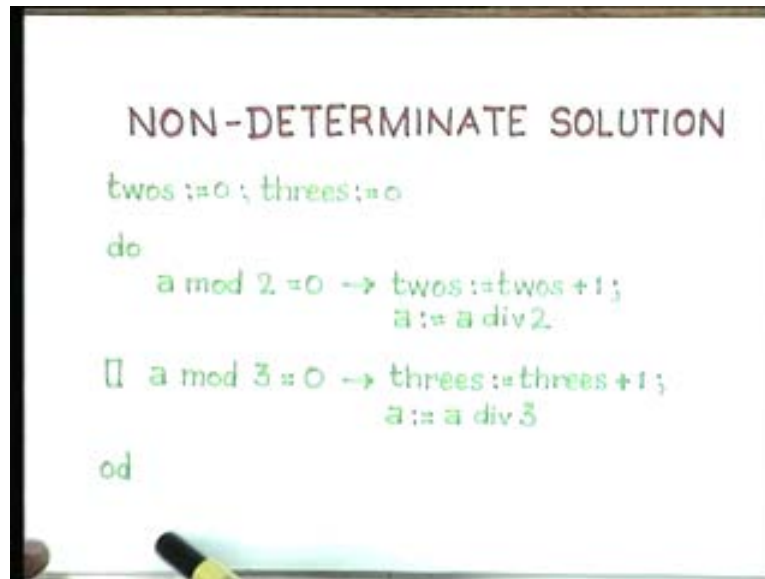
while (a mod 2 = 0) do
begin
  twos:=twos+1; a:=a div 2;
end;
while (a mod 3 = 0) do
begin
  threes:=threes+1; a:=a div 3;
end;
```

If you look at this there is an equivalent program of course in which I could have flipped the two 'while' loops and it would not have mattered at all. But there is also another equally viable alternative in which I do not care to first find the number of times 2 divides the number and the number of times 3 divides the number. The two numbers are really quite independent. So, I could decide to divide it twice by 2 if 4 is in fact the divisor of that number and then start looking at a few 3s and then come back to looking at a few more 2s and then come back to looking at a few more 3s till the process ends and till the number is no longer divisible by 2, 3 or 6. This is a perfectly valid computation. I divide by arbitrarily by 2s sometimes, 3s some other time and I continue this process. After all why is that in any sense less important as a computation than going about it in such a rigid and straight jacketed manner? Dijkstra once coined a non deterministic choice mechanism which works as follows. Here is a non determinate solution.

Here the way the control goes is as follows. You have this choice between two alternatives. You can look upon this as checking whether a Boolean condition is true, another Boolean condition is true and what makes it non determinate is that both Boolean conditions could be true at the same time. It is not necessarily that only one Boolean condition will be true at one time. Both could be true at the same time and so I take all these choices and put them all together. I just separate them by what on a keyboard would just probably be square brackets placed close together. This actually models this phenomenon of mine.

I feel like dividing by 2 and so I divide by 2 then I feel like dividing by 3 I divide by 3 and without exhausting the 2s I do these things in any order. If you actually look at the computation defined by this program then what it means is that there are several possible orders. There is a multiplicity of orders. In fact if m and n are the number of 2s and 3s that divide the number then it is a shuffling of $m + n$. Let us say the 2s are coloured with red match sticks and 3s are coloured with black match sticks and it is exponential in the number of m and n . In how many different ways can you take m card of one colour and intersperse them with n cards of another colour?

(Refer Slide Time: 9:10)



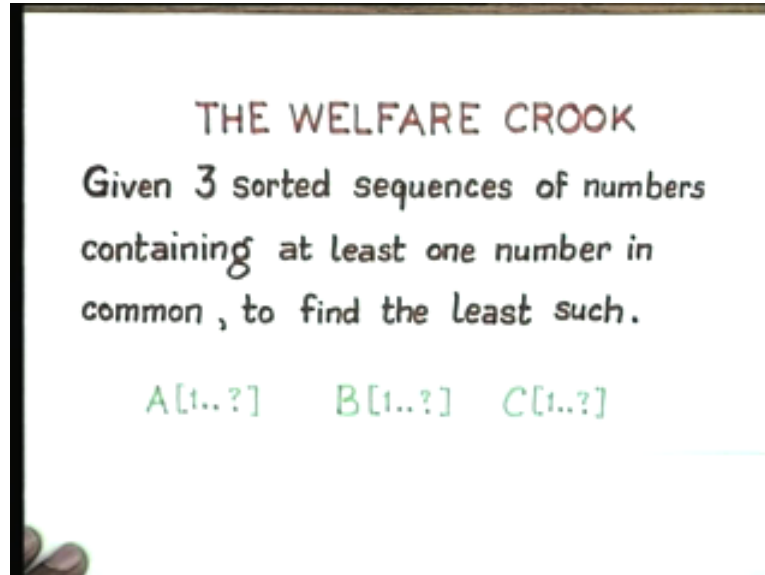
The number of choices therefore is the number of different possible computations that you can have which is actually phenomenally large. But the whole program itself is deterministic in the sense that there is a single number m and a single number n which the program computes. If the program were wrong and in the general case the fact that you have lots and lots of different options open at every stage means that you could also have lots and lots of different answers coming at the end of the program if it ever does terminate.

It actually shows is that a programmer can at many times not want to be bothered about imposing an artificial sequentiality in his computations. The sequentiality that this program imposes because of the design of the language is entirely artificial. It has got nothing to do with the notion of computation itself. There is nothing dictated by the notion of computation which necessarily says that it should be done deterministically only in this order or in the other order. This is a freedom which a programmer might want for various reasons.

Let us look at another solution which is actually a more famous solution for this and it was coined by David Gries and it is described in his book, 'the science of programming'. It is called the welfare crook problem.

I have stated a very prosaic and dull version of the problem but the original problem is actually that there are three lists of employees at IBM Thomas J Watson research center, students at Cornell University and people in New York State who are drawing dole that is government welfare.

(Refer Slide Time: 12:30)

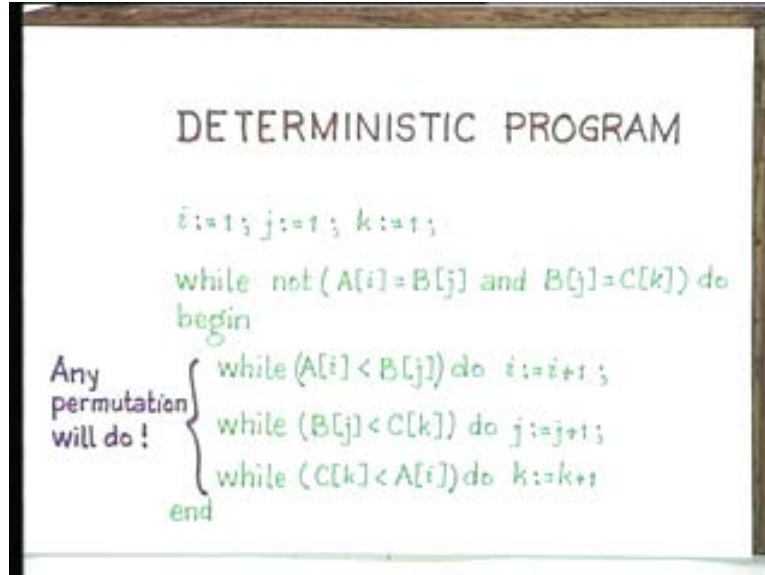


Unfortunately the laws they have in the country are such that you cannot be a full time student and an employee at the same time. You cannot be an employee and draw dole. You cannot be a full time student and draw dole but apparently there is one individual who is actually in all the three lists and the idea is just to find the first such person. Bringing the problem down to earth is always saying that there are three sorted sequences of numbers with at least one number in common in all the three sequences and you want to find the least number that is common to all the three sequences.

I am not particularly bothered about the bounds and the sequences which is why I have put a question mark here. But let us assume for simplicity that we have got some large arrays. If you were to program this in any of the standard programming languages then you will get a deterministic solution of this kind as long as you have not found the common element which is what is represented by this NOT.

```
while (A[i] < B[j]) do i: = i + 1;
while (B[j] < C[k]) do j: = j + 1;
while (C[k] < A[i]) do k: = k + 1
end
```

(Refer Slide Time: 13:39)



These three while loops could be written in any order and the whole point is that when you terminate from this while loop you can claim that for the current values

You cannot have a cyclic \geq relation unless they are all equal.

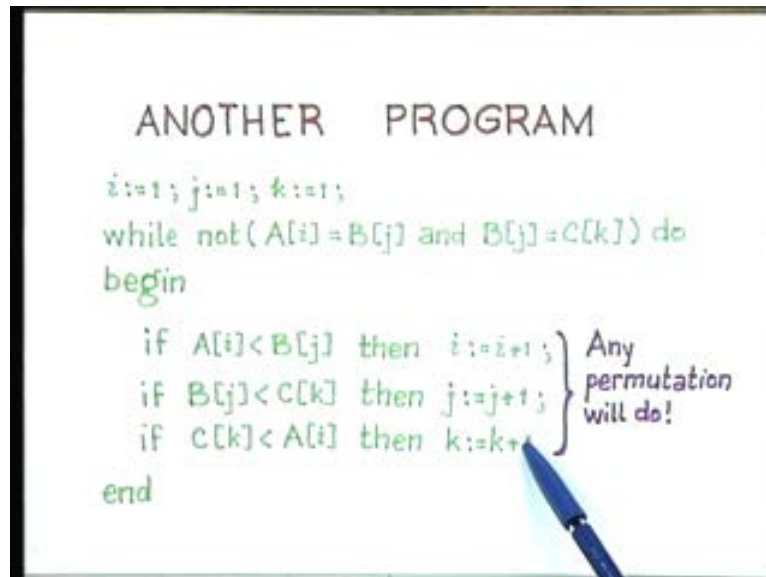
$A_i \geq B_j; B_j \geq C_k$

$C_k \geq A_i$

$\therefore A_i = B_j = C_k$

So, any permutation of this would do and one problem is that there are too many 'while' loops in this program. We should probably locate another program. If you look at another program I could transform all those 'while' loops into just 'if then' statements. After all there is this condition that guards the entry into the loop and so I cannot enter the loop if I have indeed found i, j and k such that $A_i = B_j = C_k$. In this case this allows for a far more number of possibilities than the previous one. The previous one was really straight jacketed.

(Refer Slide Time: 14:17)



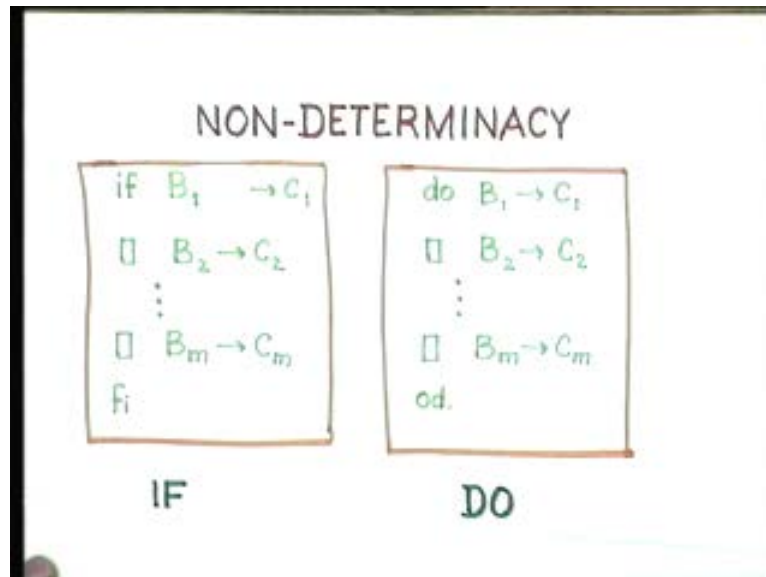
Within one iteration of the main loop you had to first find 'i' such that $A_i \geq B_j$.

Then you have to find j such that $B_j \geq C_k$ and you have to find k such that $C_k \geq A_i$ and then repeat the whole process till you have found i, j and k of that kind. It allows for possibilities like changing i, j and k in a sequential fashion. You first change i if it is necessary then you change j if it is necessary, you change k if it is necessary and go through the process all over again but it is still totally deterministic. It is totally deterministic and therefore it is easier to reason with such deterministic programs.

A non deterministic program for this problem would just be the following. There are three conditions. That outermost complicated condition is not necessary because if all these three conditions are false it means that $A_i \geq B_j$; $B_j \geq C_k$; $C_k \geq A_i$ and therefore they are all equal. The loop exits when all these conditions are false; when it is impossible to execute any of the conditions or when it is impossible to go past these guards. They are called guards. If you cannot go past these guards then the loop terminates. If there is at least one guard that is true then you should be able to go past it. The square bracket is a piece of syntax to separate out the guards. Square bracket is just a syntactic notion but it also means that it is a symmetric operation.

The square brackets indicate that there are so many different choices. So, if there are m square brackets then there are 'm + 1' different choices. Supposing you are adding m + 1 numbers then you have m + signs in between the infix notation. You can think of this as an infix notation to separate out the various choices. In particular let us look at this syntax. There are two analogous kinds of constructs for non determinate choices. One is the 'if' construct and this 'if' construct just says that there are Boolean conditions and Boolean choices and corresponding to each Boolean choice there is an appropriate command to be executed.

(Refer Slide Time: 18:32)



In case when control enters this 'if' guarded command one of these Booleans which is true is executed. You can go past a guard if that Boolean is true otherwise you cannot go past the guard. We do not know how you cannot go past it at the moment but if B_i is true then C_i can be executed. Since it is much harder to visualize all the different possibilities, the programmer has to be careful enough to ensure in the case of this 'if' that he has to do an exhaustive case analysis. You can look upon this construct as a generalization of the Pascal case and the 'if then else' put together. What would you do if you generalize both the 'if then else' and the case to get a new construct? The 'if then else' is an asymmetric construct and it specifies a clear line of control. The case is a symmetric construct which is restricted to enumerated values.

From Pascal and from all the semantics that we have done we know that they are all deterministic. So, there is only one possibility in the case which can actually be executed. From our expression language semantics it is clear that it can have only one value for a given state and if it is going to have just one value then there is only one possibility of the case which is true. If you generalize the case and the 'if then else' then it means that you get something like this which has the symmetry of the case statement but it is not necessarily restricted to enumerations. It is not necessarily restricted to ordinal values. It is a full blown Boolean condition as you would use in any conditional statement and the fact that you are using full blown conditionals and that there could be any Boolean expressions means that more than one could be true in a given state.

If more than one could be true then it means that more than one of these C_i 's could be executed but the 'if' only states that you can execute only one of them which means that if more than one is true there are several different possibilities. If you execute that 'if' once may be the C_i will be taken or you will take some C_j for some B_j which is true. I mean to convey that is really non deterministic. It is 'an act of God' which guard is going to be taken. When I say 'an act of God' it means that there are reasons which are unpredictable because of which one particular execution might be chosen.

In the case of the do as long as at least one of these guards is true the loop keeps executing and it exits the moment all the guards are false. Let us just look at the operational semantics of these two things. At the moment I will assume that 'if' stands for this which guards B_1 to B_m and commands C_1 to C_m and the 'do' similarly has guards B_1 to B_m and command C_1 to C_m . The operational rule for the 'if' is as follows and this is the first instance of a rule which actually makes your transition system. We can add these constructs to the 'while' language for which we have defined a deterministic set of rules. The moment you add these constructs to the 'while' language and give this set of rules you get the first instance of a non deterministic transition system.

All this rule says is that if there is some B_i which is true then this statement 'if' can transfer control to the command C_i period. If there are two or more different values i and j for which the B_i and B_j are all true it does not say which one is going to be executed. It only says that if both B_i and B_j are true since there are m different rules for the construct and all those m different rules, one for each i , only tell you m different possibilities. If every guard is true for example, then all that these m rules tell you is that one of them can be executed. Only one of these C_i 's can be executed. It does not say anything more nor does it say anything less.

The moment you introduce a non deterministic set of possibilities then reasoning about the program becomes far more difficult. After all you do not know what policy to choose the command corresponding to that guard which is to be executed. You look upon all these guards as forming not a sequence. They are written in sequence only because textually you have to write it all in a sequence. They actually form a set of possibilities. Any possibility from the set could be picked up and the corresponding command could be executed.

(Refer Slide Time: 27:04)

$$\begin{array}{c}
 \text{IF - OPERATIONAL RULE} \\
 \\
 \frac{\langle B_i, \sigma \rangle \rightarrow_B^* \langle t, \sigma \rangle}{\langle \text{IF}, \sigma \rangle \rightarrow_C \langle C_i, \sigma \rangle} \quad , 1 \leq i \leq m \\
 \\
 \frac{\langle B_1 \text{ or } B_2 \text{ or } \dots \text{ or } B_m, \sigma \rangle \rightarrow_B^* \langle f, \sigma \rangle}{\langle \text{IF}, \sigma \rangle \rightarrow_C \text{error}}
 \end{array}$$

Since the reasoning process is much harder in this case because of the fact that there is an explosion of possibilities the whole transition system now becomes non deterministic and so you have to ensure that if your 'if' statement is non exhaustive for some reason then it should come

out to be an error. Note that I had said that we never have any negative rules. As I was explaining it, it looked like it was going to be a negative rule that if none of the guards is true then the 'if' statement gives an error. But actually it can be expressed as the positive rule.

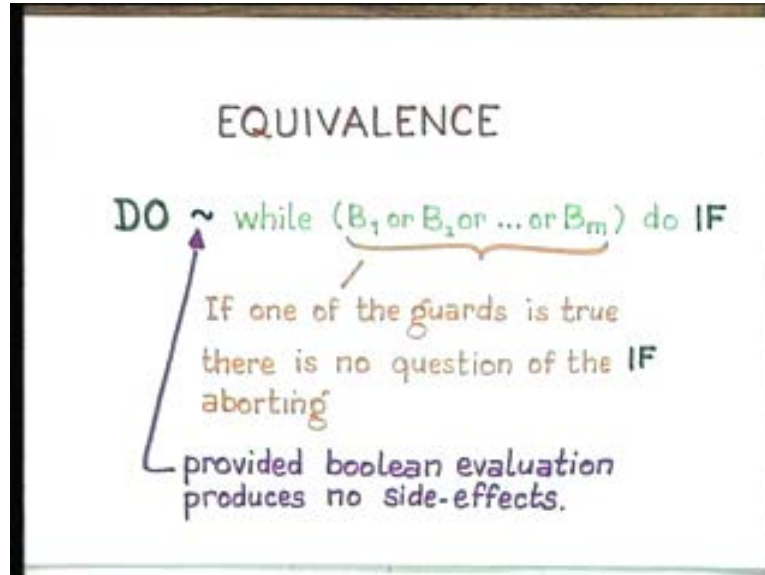
If this disjunction is false, then none of these possibilities is true and then this leads to an error. This should ensure that any programmer has to be careful to make an exhaustive list of all the possibilities. It is not necessary that B1 or B2 or the disjunction of all the guards has to be necessarily a tautology. But it is necessary that for the set of possible states it should be possible to prove that only those states are possible which make at least one of these guards true. The two things are different. If you look at this as a logical formula then it need not necessarily be a tautology but I might be able to prove about the program before entering the 'if' statement that there are only limited possibilities and that the state has to satisfy one of these properties.

So, it means that this disjunction need not be a tautology in the sense that it need not always be true but for that set of states alone it should always be true. That means there are only a certain assignments of values possible for those set of states. For example; take the case of these 2s and 3s problem. Let us suppose I require this condition. The meaning of the do is similar except that it is a matter of exiting. In the case of the loop it is a matter of an infinite computation. That is the only difference. In the case of this 'if' I know that the state is always one in which $a > 0$ and I have an 'if then else' of the form of something then this itself is a sufficient condition so that the guards need not form a tautology.

Let us take a negation of these guards also. They need not form a tautology. It could just be that for that set of states for an invariant property 'a' will always be greater than 0 and if the state satisfies that invariant property then I am guaranteed that the loop will exit without any problem. The loop will exit only when all these conditions are true. It is not necessary to have this forming a tautology. It is only necessary to ensure that it satisfies an invariant property of the possible states that will actually be encountered before entering this 'if' condition so that it does not exit in an error. It is exhaustive in that sense but it need not be a tautology. The rule for the loop is very similar to our 'while' loop and it just tells you that as long as some 'Bi' is true you execute the body and repeat the loop. In the case of the 'do' there is no question of an error and you have to exit from loop. If all the guards are false and the fact that there is no rule which specifies that, it follows that the loop will be exited.

The 'do' is actually equivalent to this 'while' loop and of course the conditions in all these cases are that there are no side effects caused by the evaluation of Booleans. If Boolean evaluation produces side effects then it matters how many times you evaluate the Boolean. If the Boolean evaluations cause side effects then that adds its own complexion to the program that is it adds its own non determinacy to the program which is often unpredictable and which is often implicit rather than explicit. Whereas the 'do' construct makes the non determinism explicit and if you do not have side effects then irrespective of the number of times you evaluate these Booleans they are going to yield the same value.

(Refer Slide Time: 33:08)



Note that you are going to evaluate some Boolean inside this 'if' which means that you are actually evaluating some Boolean at least twice in this equivalence. Whereas in an actual operational definition of the 'do' you are just evaluating Booleans as long as they are false till one is true and then you follow that guard. Side effects are always difficult aspects especially if you have to prove the correctness of your program and surprisingly for a long time when Dijkstra actually produced these things this is used for constructing perfectly correct toy programs. But now this is actually being used in the Ada language. Let us summarize the properties of non determinacy in general and in particular. It is a generalization of the case. You can use arbitrary Boolean expressions as guards. More than one guard could be true at a time and it aborts, if no guard is true.

This is for the 'if' and it allows you a specification mechanism. So, it is an excellent specification mechanism because very often the programmer does not want to be bothered about the nitty-gritty details of how exactly a computation should be performed and what most of the languages do is that they force you to specify a certain sequence in which computations have to be performed whereas often in many cases there is absolutely no reason why the computation should be performed in that order. You can look upon the 'if' 'fi' as a symmetric generalization of the 'if then else'. The 'if then else' is itself a deterministic construct and it is also a generalization of the case because it allows arbitrary Boolean expressions and it frees the programmer from imposing an unnecessary sequencing of operations.

(Refer Slide Time: 34:40)

IF $\square \dots \square FI \dots 1$

- ★ Generalization of **case**
 - + arbitrary boolean expressions as guards
 - \Rightarrow more than one guard could be true at the same
 - \Rightarrow aborts if no guard is true
Ensures exhaustive case analysis.

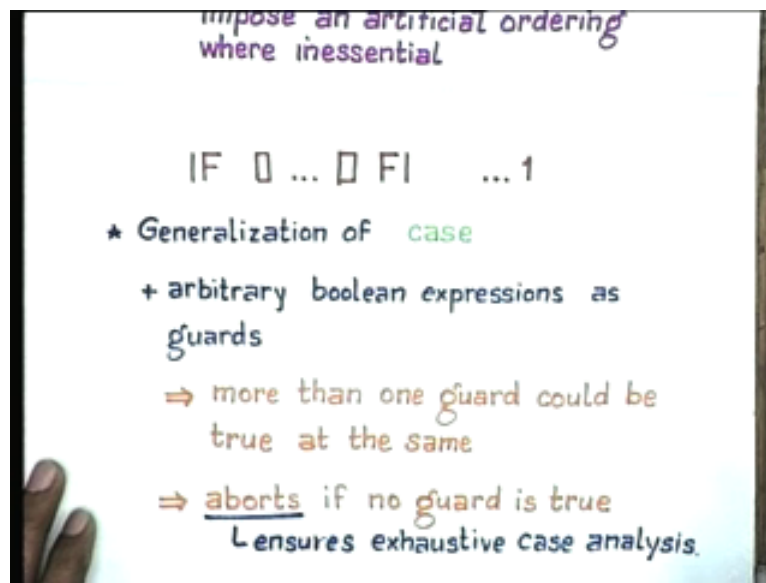
(Refer Slide Time: 34:58)

IF $\square \dots \square FI \dots 2$

- ★ Excellent specification mechanism
 - when programmer does not care which command may be executed.
- ★ Symmetric generalization of **if-then-else**
 - \uparrow does not force programmer to impose an artificial ordering where inessential

But it also makes it more difficult in the sense that it opens a variety of possibilities. Unless your program is proved correct there usually is no guarantee that your program will work as you intended to.

(Refer Slide Time: 36:01)



For example; nowhere we specify to the programmer in what order the Boolean expressions have to be evaluated. One possible implementation is that I evaluate all the Boolean expression values and once I have created the set of indices of the true Booleans then I run a random number generator program and pick out one. That is one possibility. Another very simplified possibility is that I just go through the Boolean expressions in sequence till one is true but there can be any kind of complicated scheduling mechanisms and it is a specification mechanism in the sense that since the programmer is not forced to impose an artificial ordering it means that the actual implementation is to be hidden from the program. The program is not going to know how exactly this 'if' statement will be implemented. If you want the programs written in this language to be portable it should work under all possible scheduling policies. This implies the way you schedule the evaluations of the Booleans in a guard in 'if' or a 'do' statement.

What happens if it is going to work under all possible scheduling assumptions and it does not use any other kind of mechanisms? Both the constructs are symmetric which means that if you change the order the program should not change. If instead of writing them in the order $B_1 \rightarrow C_1$ to $B_m \rightarrow C_m$, I reverse or shuffle the order in any way, the meaning of the program does not change and if the program was proved correct then it should not matter either in what order these guards are written or in what order they are taken by the runtime system to execute.

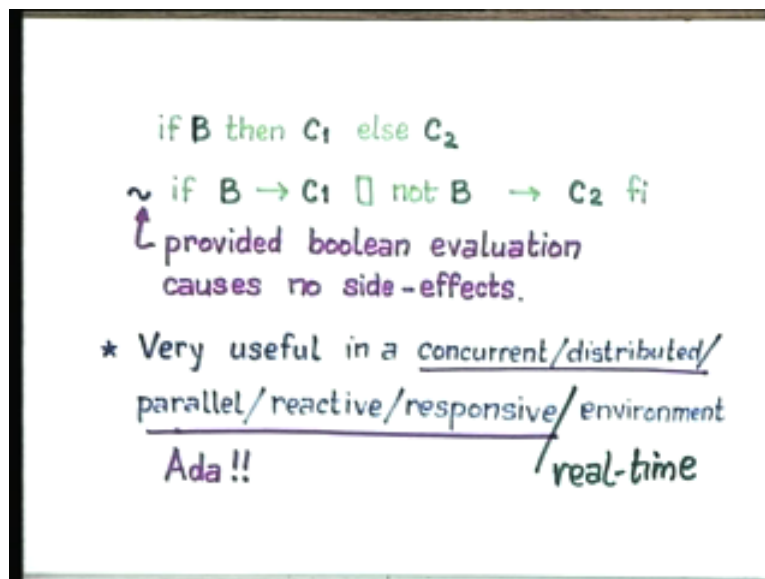
It is a specification mechanism which frees you from imposing an artificial constraint but then it also brings in its own unpredictability of the result. A program may be wrong in the sense that different implementations could give you different wrong results and one of them might actually give the right result because the implementation took a certain decision which happened to match with the programmer's intention or his presumption of how the guards are going to be evaluated. All this goes to show that it does not mean anything if you actually get the right results from an execution. Therefore the obligation for proof then becomes absolutely essential in such a situation. The moment you have non determinacy you have to have a proof.

If the proof is going to be of the form which tries out each and every possibility then it is not much of a proof but just a hand execution.

What should the proof be like? The proof should exploit the invariant properties regardless of what choice you take. You should be able to specify one single general invariant property of the loop and regardless of what choice is taken it should be possible to show that your eventual goal is going to be obtained through that invariant property. Only then the proof has some meaning. This construct was actually created by Dijkstra to develop programs in such a way that program and proof are developed side by side with the proof leading the way to development of the program.

The whole philosophy is that it is not that you should write a program and then prove it correct. That is an after thought and a proof after the fact is really of no importance except of an academic kind only. But what is essential is to use the proof as a guiding tool to develop the program. You have to specify the invariant properties. You have to do a case analysis at each time. They should annotate your programs and then based on that your programs should be developed which is what we tried to teach in the early programming courses. The whole point is that you have to exploit the invariant properties so that regardless of the number of options you have you are guaranteed that a certain theorem or a certain goal can be proved. All this was actually motivated by Dijkstra's own experience in the design of multiprocessor and multi-user operating systems.

(Refer Slide Time: 43:00)



If you look upon bringing down various operating system concepts and encoding them in programming languages, which is very useful to do, then you can write the entire operating system in that programming language. You can bootstrap an operating system on top of another. For example; for an ordinary general purpose operating system you could write a real time operating system. If you wrote a general purpose operating system using non determinacy then the portability of your programs also increases. Somebody might just decide to attach one more

processor to the current machine and your operating system should be able to take care of this extra component.

It should not be necessary to rewrite the entire operating system. You should be able to write the entire operating system also in a high level programming language and not depend upon assembly language all the time which is what is being done right now. But the area programming language for example designed bomb in nuclear installations. What motivated the design of Ada is that the US department of defense found that it has probably a thousand installations and at least a hundred different programming languages being used in those installations. In defense you keep getting transfer to different places. What it meant was that when programmers were transferred from one installation to another even though the system architecture was identical just because the programming language was different they had to spend a few months learning the new programming language, learning how to use a system and a lot of essential time was wasted. This is a documented fact by the US department of defense.

Secondly, you could not move port programs across installations in the sense that I am having a new Persian missile installation somewhere and I cannot move the old programs because that system was by a different company and it supports different compilers and it does not support any of the compilers for which I have installations. This meant re writing the entire software in a different language and when you transfer personal to and fro they spend the first six months just coming to grips with a new installation, a new piece of hardware, a new piece of software, a new language, a new system called conventions, a new procedure called conventions and a whole lot of aspects. It is a whole society in itself in each installation and there is very little interaction between different installations.

What they decided was that except for their business data processing in terms of their accounting and pay roll they found that COBOL was sufficient. It was scientific where they do a lot of research in pure science. For all their control systems and for all their non business applications; scientific, numerical, control, concurrent and distributed you have a wide variety of installations which have to work in a distributed fashion. You require fast computations with parallel computers, vector processors etc. You have control systems which have to react to external stimuli in the form of sensors for nuclear reactors, pressure controllers, temperature controllers and transducers. You have to respond may be to a cosmic ray shower and so you require a wide variety of different kinds of applications and you have an explosive variety of programming languages in which all of them are written. So, for everything other than their business data processing what they wanted was a single unified language.

They went further to say that it should be a Pascal like language which supports concurrency, modules, reactivity and responsiveness and therefore if you are talking of reactive or responsive software embedded in some hardware like a controller for a nuclear installation or a chemical reactor then what that controller gets as signals from various transducers is totally non deterministic. It is not predictable in a highly decentralized mixture of mechanical, electrical, electronic hardware and software to actually predict when some signal might come from somewhere and therefore it is necessary for these controllers or even client servers in a more business like environment such as railway reservation system to look at the design of the server or a controller in isolation. When you do that all you know is that you might get signals from this

wide variety of systems. What you do not know is the order in which you are going to get those signals.

The basic purpose of the controller is that if the pressure goes beyond some limit you have to activate some mechanism, relays, some electronic switches in order to bring the pressure down. You have to keep pressure, temperature, volumes and flow rates and some balance within certain tolerances. When one variable increases at the design of the controller, you will know the kinds of signals that you are going to get. But you do not know in what order they are going to come.

Sitting as a controller, monitoring an entire distributed installation, I am subjected to non deterministic pressures from a variety of places. It means I have to have a facility to deal with this and it is not going to be done by a human being. Imagine a human being sitting in a nuclear installation trying to co-ordinate the entire process. It is all going to be done by a controller written in a mixture of hardware and software. You go into hardware. There is really no difference between hardware and software except that when you want some faster executions that are time sensitive and responsive you have to activate various procedures within a given time frame otherwise the whole chemical reactor might burst.

They have to be time-sensitive and they have to react to stimuli which come in a non deterministic fashion in non deterministic order and therefore you require a convenient construct which is symmetric and since you cannot predict in what order you are going to get these signals your guards in addition to being Boolean conditions would also include signal values.

If a certain signal comes then this response has to be sent somewhere not necessarily back to the same place. You have to activate something else. So, the fact is that for any isolated program in a large system the only way to cope with the entire system is not to know the nitty-gritty's of each and every system. After all that system can keep changing. I might replace something by a faster machine; I might replace some mechanical relays by some electrical relays and I might replace something with more capabilities. Then the only way to cope with the entire system is not to know the nitty-gritty's of each component of the system but to know only broadly what kinds of stimuli you are going to get and therefore what should be your suitable response to it. It is an abstraction that you have to perform.

Let me write real time. These are all buzz words of which many are closely related and overlap but they are different words. In a real time concurrent distributed environment you have to be in a reactive system. You have to be responsive and you have to do it in real time. For that you require such a construct. The only difference is that the construct also includes communications from outside and also it includes a timing mechanism which ensures that if something does not happen within a certain amount of time then something else has to be done about it.