

Artificial Intelligence: The OPS5 Expert System Shell

Prof. Deepak Khemani

Department of Computer Science and Engineering

Indian Institute of Technology, Madras

Module – 03

Lecture - 11

So we are looking at forward chaining rule based systems. In last class we saw how rules are essentially matched with data. Today we want to look at how one decides as to what is the rule that is going to be fired next and in doing so we also want to come up with a view that what we are looking at is language called OPS5 is a programming language. So the word OPS5 it came, as I have been saying this whole word came from Carnegie Mellon university (CMU) started by Simon....., various people worked on it. So this OPS5 essentially stands for official, some people say this production system language and it gradually evolved over a period of time, it became something called OPS83 I think in 1983 and eventually transformed into something called SOAR which some people say is more than a programming language in the sense it's a cognitive architecture and maybe we will give you a tutorial on SOAR at some point.

Let's come back to looking at this as a programming language. So what are we looking at this programming language, what OPS5 does is basically it implements in, it's an inference engine which has its match, resolve, execute cycle and the match part as we have seen is made efficient by using the Rete network. But right now we are not really concerned about that. We are concerned about what kind of language can we think of, what paradigm does it follow essentially. So if you look at the languages that exist, we have imperative languages like C for example. And how does one program in a programming language, that the programmer has to state what are the actions to be done in the order in which they have to be done. So control is directly into the hands of the programmer and the programmer has to simply say that do this, then do this, then do this, and so on and so forth. Of course in the process you can put in branching and you can put in loops and you can put in all kinds of things but it's basically an imperative language. It tells the machine that this is what is to be done first, then next and so on and so forth.

Then we have seen object oriented programming which i am sure you are familiar with, languages like C plus plus and Java and there were other earlier languages like Simula. Like all good things object oriented languages also came out of the work done in AI. Little bit later in the course we will look at something called frames and those frames essentially gave rise to what we now call as object oriented systems. Basically the idea of inheritance and all that is automatically taken care of in such systems.

Then there are languages like logic programming languages which are in some sense close to what we are looking at right now. But logic programming language is a pure logical theorem proving language. Here our language says that you have a left hand side that you have to match but the right hand side is a set of actions essentially. Whereas the logic programming language tends to view the whole exercise of programming as theorem proving essentially. And we will see a little bit later how that is done essentially. And what we are studying is as we have said pattern directed systems or data driven systems.

(Refer Slide Time: 05:05)

OPS5 - a different programming language.

Official production system
↓
OPS 83
↓
SOAR

- Imperative languages
- OOP
- Logic programming
- Data driven / pattern directed

NPTEL

4/4

in the sense that what happens next or which command is executed or which sentence is executed depends upon the data. It is the data which drives the execution, of course you might argue that if you have a lot of IF statements in an imperative language then control there is also given by the data. But the programmer has to hardwire the control into the program essentially whereas in the language that we are looking at right now this is not the case. So let us see what is happening here. We have the match phase that we have studied so far and what it produces is conflict set which is essentially a set which contains pairs of rule 1 let's say data 1 and this data itself is a set so let me put it in as a set. So there is more than one pattern that a rule is matching then rule 2 with its matching data and so on. So let's say there are n rules.

(Refer Slide Time: 06:56)

OPS5 - a different programming language.

Official production system
 ↓
 OPS 83
 ↓
 SOAR

- Imperative languages
- OOP
- Logic programming
- Data driven / pattern directed

Match
 ↓
 $CS = \{ \langle rule_1, \{data_1\} \rangle, (rule_2, \{data_2\}) - \dots (rule_n, \{data_n\}) \}$

what happens next depends upon the data

NPTEL

this is input to this place called RESOLVE, so what does it resolve, it resolves the conflict between matching rules. So at the end of the match phase you have a whole set of rules with matching data and you can sort of visualise that all of that is clamouring to say execute me execute me execute me and so on, it's like getting party ticket for elections. And it's a task of the resolve phase to decide which rule to execute next essentially. So we want to look at that in today's class. And you can see that this basically embodies the problem solving strategy. So you want to look at different strategies and see what will they give us essentially.

So you want to look at this process of conflict resolution which says pick a rule with its matching data from the conflict set essentially. One property that we always want to follow is Refactoriness. This says that once selected the pair is removed from the conflict set essentially. In other words, what it means is that let's say there was a rule which got selected and let's say it had three working memory elements that it was matching and it got selected based on whatever selection strategy that we will look at in a moment. But having been selected, having executed the rule once obviously if those three working memory elements are there in the working memory, this rule will still be matching essentially. Refactoriness says that never execute the same rule with the same piece of data essentially. It's like money essentially once you spent it you are done with it.

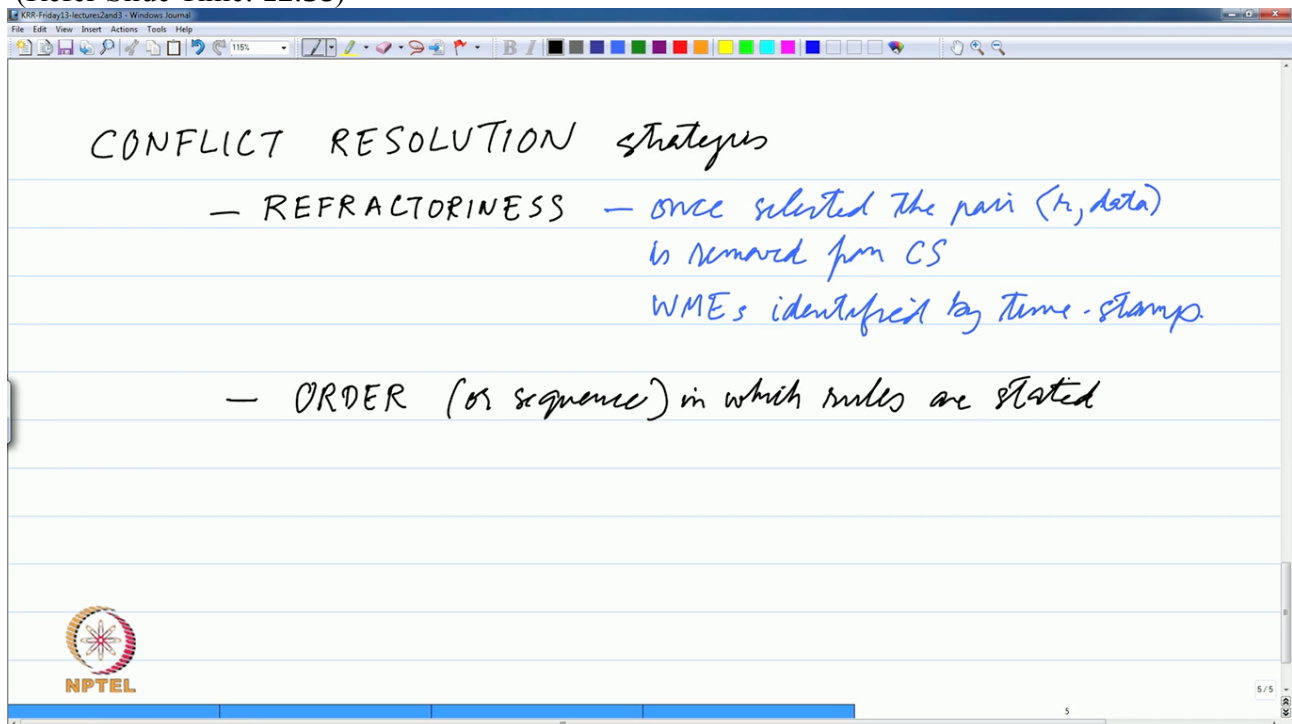
So if you have a rule with some, remember that working memory elements are identified by timestamp. So if there is a rule which is matching with data with some distinct timestamp it can fire only once. The only time it can fire with the same data again is that if one of those working memory elements gets deleted for some reason and inserted sometime later in life essentially in which case it would have a different timestamp. And then it may come into contention again but otherwise refactoriness says that with the same inputs you fire it only once essentially. So the term actually comes from looking at neurophysiology where people observe that you know that as neurons have their input building up for them, they can fire once but having fired once they cannot fire immediately again essentially. So there is a latency between one firing and the next. So that term is borrowed from there.

Essentially as far as we are concerned its simply common sense and it simply says that if you have already done what that rule is saying don't keep doing the same thing again and again. If you are doing a classification problem, then you are looking at data for certain class. So let's say you are grading students, so if you have seen a student with marks 75 and attendance good and you say the

grade is equal to let us say A, then don't reassign the grade A again and again. So once you are done with it you are done with it. So refactoriness takes care of that essentially.

Then you could do it in order or sequence in which rules are stated.

(Refer Slide Time: 12:38)



The screenshot shows a Windows Journal window with the following handwritten text:

CONFLICT RESOLUTION strategies

- REFACTORINESS — once selected the pair $(h, data)$ is removed from CS
WMEs identified by time-stamp.
- ORDER (or sequence) in which rules are stated

The window also features an NPTEL logo in the bottom left corner and a page number '5' in the bottom right corner.

you can say that okay I have an ordering of the rules and of the matching rules I will pick that rule which is first in this order, highest in this order or something like that. So obviously this gives control to programmer. The data has no impact on which rule you can select. The programmer has said that these these these rules are in contention always select always select this rule essentially. And we will see later when we look at PROLOG, it is a programming language which is based on the logic programming. That in Prolog you have to carefully write your statements in a careful order if your program is to execute properly essentially. And that's because you will see Prolog essentially does processes statements in the order in which you write them which amounts to doing breadth first search in some space essentially. But we are interested here in a different approach essentially, we say that it should be decided by the data. Incidentally Prolog does not do forward chaining it does backward chaining and it kind of embodies a totally different problem solving strategy which is goal directed essentially as opposed to pattern directed that we have here that the data decides what is to be done next, in Prolog the goal decides what patterns to be looked for essentially. Anyway we will come to that later.

So these are two basic strategies. Then there are more interesting strategies, one is called recency. This says select that pair so remember whenever i talk about the pair i mean a rule and the corresponding data which has the highest timestamp data or working memory element. In other words, most recent. And what do we mean by most recent, not the most recent working memory element, amongst the data which is matching or the working memory elements which are matching a set of rules, amongst those rules whichever is the most recent one. And select that rule which matches the most recent element. What's the advantage of this if you think of this intuitively it allows you to maintain in some sense a chain of thought or maintaining a line of reasoning essentially? In terms of logic if we had just concluded something if you had added a new working memory element which will have the latest timestamp. If there is some rule which is using that prefer that rule essentially. So in that sense whatever changes you are making so you can see that the most recent working memory elements will get attention more recently.

(Refer Slide Time: 16:54)


Conflict Resolution

- RECENCY - select that pair $\langle r, data \rangle$ which has the highest time stamped WME
 - MOST RECENT WME

↓

maintain a chain of "thought"
"line of reasoning"

—



then another important strategy is called Specificity. This says select the rule or select the pair with most specific data. And what do we mean by this. We basically mean most number of tests. So select that rule which is looking at more data for deciding what action is to be done essentially. And there is a combination is used in OPS5 which is called MEA which stands for means ends analysis and it says the following that recency for the first pattern in the rule and tie breaking by specificity.

(Refer Slide Time: 19:13)

Conflict Resolution

- RECENCY - select that pair $\langle r, data \rangle$ which has the highest time stamped WME
 - MOST RECENT WME

↓


maintain a chain of "thought"
"line of reasoning"

SPECIFICITY - select the pair with most specific data

↓

MEA (means ends analysis) - most number of tests

Recency for 1st pattern in the rule - tie breaking by specificity

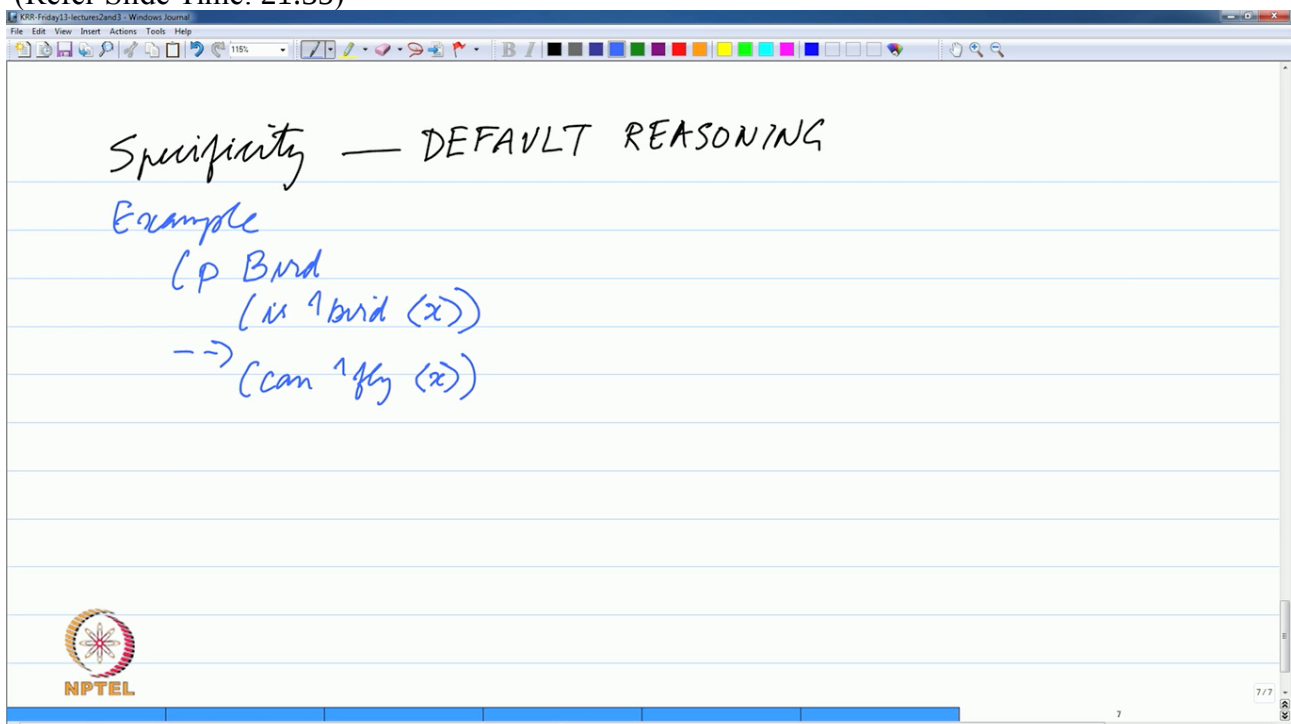


so what does this strategy say. This strategy is saying that okay you got a set of rules with matching data pick those rules whose first pattern matches the most recent data essentially. So if you go back to this idea of context which we talked about when we were talking about XCON, you set the context, you are building the foundation then you are building the wall and so on. So the first

pattern in every rule would match the context and the most recent context is the one which should be considered, the other would not fill up because it says recency for the first pattern in the rule essentially. It says more than one such rule which is matching the same data, it has to be the same data you know because it has got the same recency then you use specificity to do the truce between them, so tie breaking we do like that. So this strategy is called means ends analysis and it has been commonly used in many places.

So let's look at specificity a little bit more. It's a very interesting approach. It allows us to do something called as default reasoning which we will again look at in more detail later and as the name suggests that you can make some inferences by default unless you know better essentially. So let me do this with an example. Supposing i have these rules, the rule is called Bird and it says if something is a Bird then add a working memory element whose class name is can and the attribute name is fly and the value of the attribute is who can fly. So we are essentially saying that if x is a bird then say that x can fly.

(Refer Slide Time: 21:35)



Now i can have another rule which says let's call it a Penguin. This rule says that if x is a bird, it is also known that x is a penguin then I should really say make i forgot to do that. Let's say cannot fly. So you can think of can and cannot as things you know creatures can do and the attributes are cannot fly cannot eat cannot drink you know whatever. And the value is of individual who can or cannot do it lets say. So now we have two rules and the working memory is as follows.

(Refer Slide Time: 22:39)

Specificity — DEFAULT REASONING

Example

(p Bird
 (is ^bird (x))
 ->
 (make (can ^fly (x))))
 W

(p Penguin
 (is ^bird (x))
 (is ^penguin (x))
 ->
 (make (cannot ^fly (x))))
 ->

NPTEL

is bird tweety, is penguin..so you will find this name tweety is the most common name in AI textbooks. So there are two birds, peppy and tweety, what we know about tweety is tweety is a bird what we know about peppy is peppy is also a bird and peppy is a penguin. And i let loose this set of two rules what will happen. So i have two rules i have three pieces of data i do the match process, what will be in the match, how many rules will be in the match in the conflict set. There will be three rules, one rule which matches tweety, which says tweety is a bird so tweety can fly. Another rule which says peppy is a bird which can fly. And the third rule which matches peppy is a bird and peppy is a penguin so peppy cannot fly. According to specificity which rule will, that peppy cannot fly. So you can see that i can have if i want to determine who can fly or who cannot fly, i can have a default rule. Default rule says if anything is a bird that thing can fly. But i can also have a rule which is more specific which says that if something is a bird and something is a penguin then that something cannot fly. So in this example things will work fine, let's say the first execution will be saying that peppy cannot fly. The second time the rule fires it will say let's say tweety can fly. But unfortunately the third time the rule, penguin rule will no longer be in contention because of refactoriness so the bird rule for peppy can fly.

Can you see that. That first i will make an inference that peppy cannot fly because peppy is a penguin. But because i have this refactoriness criteria that rule will never come into contention again so i am left with two rules which can now compete with each other. One says that peppy can fly the other says tweety can fly so in some order they will execute because they are equivalent essentially. So i will first end up making an inference that peppy cannot fly, then i will end up making an inference that peppy can fly. How do I get around this problem? So my first inference is correct that I am saying that peppy cannot fly but because the other rule still exists in my knowledge base I don't want to make the inference now that peppy cannot fly. Any suggestions? So the problem is I can modify the bird rule.

(Refer Slide Time: 27:10)

Specificity — DEFAULT REASONING

Example

(p Bird
 (is ^bird (x))
 -> (make (can ^fly (x)))

W M
 (is ^bird turkey)
 (is ^penguin perry)
 (is ^bird perry)

(p Penguin
 (is ^bird (x))
 (is ^penguin (x))
 -> (make (cannot ^fly (x)))

(p Bird
 (is ^bird (x))
 - (is ^penguin (x))
 -> (make (can ^fly (x)))

so if i had rewritten my bird rule in this fashion then this problem will not occur. Now i am distinguishing between birds which are not penguins. Remember that I have put a not here essentially. If x is a bird and x is not a penguin, then say that x can fly. Now you can see that this is all very tricky business you want to simplify your knowledge representation and you want to simplify your reasoning as possible so you want to have default rules. Rules that say that anything is a bird which can fly but at some point you want to say that okay if it's a bird but if it's not a penguin then sorry if it's a bird and it's a penguin then also it should not be able to fly. Then what you have done here is that we have added an exception. So we say that all birds except penguins cannot fly. Obviously this business of adding exceptions can be very painful essentially you cannot keep on adding exceptions for any such thing essentially.

So later in the course when we see default reasoning we will see how we can get around this problem essentially. But for the moment what i really want to emphasise upon is the fact that specificity allows us to do default reasoning that if we have more information we will make different conclusion if we have less information we can make some different conclusion. In this case if we know that something is a bird we can go ahead and infer that it can fly. But if we also know that it's a bird and it's a penguin then we can make an inference that it cannot fly. So i just want to end with a small example. A program to since we are thinking of OPS5 as a programming language, a program to sort which is one of the things we do most often in computer science. If we look at industry programs most of the times we are sorting.

So the working memory let's say it looks like this. That we have a record just for simplicity we have an index as an attribute and we have a value let's say is a number as an attribute and we have a set of statements which says record 1 value 20, record 2 value 30, record 4 value 50 henceforth. You can add things like you can convert this into a student database, student name let's say rank or something or index something. And then value and marks essentially. So essentially we want to say that index 1 should have that's the problem of sorting right, that index 1 should have the smallest value, index 2 should have the next smallest value and so on and so forth. So we can do this as follows.

P sort if we have a record index i value n. If we have another record index j greater than i value m let's say less than n, let's say we are sorting in ascending order. What should i do i can swap the

indexes. Or i can swap the values it doesn't matter. So what does this rule say. That if there is a record with index i which stores a value n . If there is another record with index j which is more than i , it has the value m which is less than n then just swap the indexes. So make the first one as record j and second one as record i . and if you think about this carefully if you are given a set of 100 records then by the time this program terminates by that we mean that it stops matching anything further and then there is nothing else to be done, all the records will be in sorted order. Just think about this and it allows you to do these kinds of things quite simply and you know as computer science we are always fascinated by simple ways of doing complex things essentially. And some things can be done very simply in ideas like logic programming or rule based systems. But ofcourse it all comes with a cavier we are not talking of efficiency here, we cannot give any guarantees of saying that this algorithm will work in $n \log n$ time.

Maybe you should give some thought to it. Supposing we write such a program for sorting what is the average case complexity we expect it to be. So some small exercise to think about. So let's stop here with forward chaining and rule based systems and Rete net. We have been focusing on the algorithm side so we will swing to the other side for a little while and we will come back to first order logic. And then see how can we represent what we want to say. So we have really not paid much attention to logic we have said okay you can have arbitrary or adhoc predicate names but that will not make sense if you want to solve real problem you need to carefully choose the predicates and may be try to look at these problem of how can we express things which we really want to express. How can we choose predicate names essentially? So we will spend some time on that in next few classes. ok.

Question from audience.

Reply. That's an interesting question in the sense that what is the semantics of negation sign. The semantics is that there is no record which says that x is a penguin. If there is a statement in your knowledge base that tweety is a penguin, then this will not match. If there is no such statement it will match. So there does not exist or something like that. So you can see that testing for a negative thing is done little more carefully. I have not mentioned it when i was talking about the Rete network but you have to be a bit careful because looking at negation amounts to looking at the entire working memory to see that there is no such record existing. How do you handle that is something i have skipped?