1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

Control structures for rule-based systems

Rule-based systems for artificial intelligence (also called *production systems*, but that sounds like an assembly line) can work quite differently from the usual (built-in) way that Prolog interpreters work. We'll now discuss some of these ways. The differences are in the *control structure* or *conflict resolution* or *inference engine*, the way and order in which facts, rules, and parts of rules are used. Choices in the order for these things can enormously affect the success and efficiency of a rule-based system--Prolog interpreters aren't always best.

Control structures are somewhat like the algorithms in other parts of computer science. Control structures, however, are more general and less precise in their effects. As we said in Section 4.13, artificial intelligence programs tend to consist of many small pieces of code, and the control structure usually serves as a coordinator or manager for all these pieces instead of trying to impose a sequence on them all.

The control structure is critical for an important and practical class of rule-based systems. Called *rule-based expert systems*, these rule-based systems try to mimic the performance of human experts on specialized tasks. Their rules are usually more like recommendations than definitions, often like the traffic lights example of Section 4.11. Rule-based expert systems typically need large numbers of rules about a problem area or *domain* in order to approach skilled human performance. Because of their size, efficiency is important, and choice of control structure strongly affects efficiency. We'll frequently have expert systems in mind in this and the next two chapters, if not explicitly.

Just like everything else in computer science, there's a tradeoff of generality and flexibility against speed with control structure ideas. The ideas we'll discuss in this chapter can be placed on an flexibility/efficiency spectrum (see Figure 6-1). Those that are flexible, "high-level", and slow appear at the top, while those that are inflexible, "low-level", and fast (*compiled control structures*) appear at the bottom.

Backward-chaining control structures

Many control structures impose a single sequential ordering on everything that happens. But sequential control structures differ in the relative importance they assign to rule order, fact order, and query (*goal*) order. The usual Prolog-interpreter control structure described in Chapter 4 puts goal (query) order top priority, and then treats rule and fact order of equal importance after that, using the database order as a way of assigning priorities to individual rules and facts. This is *backward chaining* or *goal-directed reasoning*. It is told something to prove, and it tries to find a way, binding variables as necessary. If alternative conclusions are possible (as in the traffic lights example in which several actions are considered until one is found), backwards chaining can try to prove the first, then try the second if the first fails, and so on like an "or". Backward chaining is often a good control structure when there are many more facts than final conclusions (goals).

As an example, consider these rules, labeled with codes for easy reference:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
```

```
/* R5 */ b :- d.
/* R6 */ b :- e.
/* R7 */ c(2) :- not(e).
/* R8 */ d :- fact2, fact3.
/* R9 */ e :- fact2, fact4.
```

Suppose the goals in this rule-based system are **goal1** and **goal2(X)**, and suppose we consider them in that order. In other words, it's like a query was:

```
?- goal1; goal2(Z).
```

Suppose further that only facts **fact2** and **fact3** are true. R1 is tried first, but **fact1** isn't true so it fails. Next R2 is tried, invoking R4 and then R8. R8 succeeds because **fact2** and **fact3** are both true. So the subgoal **d** succeeds, and therefore R4 (and **a**) fails because of the meaning of **not**. So finally **goal2** is tried, which invokes R3. This invokes R7, then R9. The latter fails because **fact4** isn't true. So R7 succeeds, hence R3 succeeds with **X**=2, and hence **goal2(2)** succeeds.

There are several useful enhancements of backward chaining. A simple trick that can often greatly improve efficiency is to *cache* or enter as facts some or all of the conclusions reached. For instance, once we prove conclusion **b** with the preceding rules, we could add a **b** fact to the database. We should put fact **b** in front of rules that can prove it, so a subsequent query will find it before the rules. The more times **b** is queried later, the more effort this caching can save. The disadvantage is more facts to search through to answer questions.

Conclusion caching is simple with Prolog: just use the **asserta** built-in (that is, you don't have to define it) predicate of one argument, a fact to be asserted. This predicate **asserta** is like **not** in that the argument must be a predicate expression; querying it always succeeds, but has the side effect of adding that expression as a new fact to the database, in front of the facts with that same predicate name and rules with that name on their left side. Like most of Prolog's built-in predicates, **asserta** always fails on backtracking: there's no second way you can assert a fact. Prolog also has two related built-in predicates: **assertz** which caches a fact *after* the last fact or rule in the database having the same predicate name, and **retract** which removes (or "un-caches") a fact from the database.

Another trick to improve efficiency of backward chaining is not to require a complete starting database, but ask for facts as needed--that is, designate *virtual facts*. For instance, if there's something wrong with a car's electrical system, you could check for loose wiring. But that's a lot of work, and some wires are hard to find. We shouldn't require checking wires before running a car program, only if more obvious problems like a dead battery have been ruled out. Backward chaining works well with virtual facts because it only queries facts immediately relevant to its conclusions. The control structure we discuss next, forward chaining, does not have this advantage, but it has some others.

Forward chaining

Often rule-based systems work from just a few facts but are capable of reaching many possible conclusions. Examples are "sensory" and "diagnosis" expert systems, like those that identify an object from a description of what you see at a distance, or those that tell you what to do when your car breaks down from a description of what isn't working. For these, it often makes more sense to start with the facts and reason to goals (conclusions), what is known as *forward chaining* or *data-directed computation* or *modus ponens* reasoning.

As an example, take the rule

```
a :- b, c.
```

and suppose **b** and **c** are facts. Then we can conclude **a** is a fact, and add it to the facts we have. This is called *modus ponens* inference in logic. There's no query, no goals; we just use the facts we know to infer some new fact. (The built-in predicate **asserta** can help implement this too; see the next chapter.)

To use modus ponens as the basis of a control structure, take the facts in order. For each fact, find all rules whose right sides contain a predicate expression that can be matched to it. (We can index predicate names mentioned on right sides to speed this up.) Now "cross out" the predicate expressions matched in the rules; that is, create new rules like the old rules except without these expressions. But wherever a fact matched the last expression on the right side of some rule, the left side of the rule has been proved a new fact, after substituting in any bindings made on the right side; so cache that new fact in the database. The last paragraph gave an example. For another, consider:

```
a(X,3) :- b(X).
```

Then if **b(20)** is a fact, modus ponens would conclude **a(20,3)** is a fact.

It matters where we put a new fact among the others, since the facts are pursued in order. Usually it's good to put the new fact in front of those not yet considered, a *most-recent-fact* or *focus-of-attention* idea. So the fact just proved will be the next fact examined; the fact list then is much like a stack (last thing "in" is the first thing "out"). This might be good if we want to reach some particular conclusion as fast as possible. But, on the other hand, if we want to systematically find every conclusion possible from the facts, we should put new conclusions at the end of the set of facts; the fact list is then like a queue (last thing "in" is the last thing "out").

Any **not**s in rules require special handling. Since we want to follow the closed-world assumption for forward chaining too (it's simplest), we want **not** to mean "it can't be proved". So forward chaining must first assume all **not**s to be false, prove all possible facts, and only then consider as true those **not**s whose arguments are not now facts. Those **not**s may then prove new facts with new consequences. (To avoid such awkwardness, some artificial-intelligence systems let you state certain facts to be false, and we'll discuss how to handle such "unfacts" in Chapter 14, but this creates its own complications.)

Here's the formal algorithm for (pure) forward chaining:

1. Mark all facts as unused.

2. Until no more unused facts remain, pick the first-listed one; call it F. "Pursue" it:

    (a) For each rule R that can match F with a predicate expression on its right side, ignoring **not**s, and for each such match in the rule (there can be multiple match locations when variables are involved):

        (i) Create a new rule just like R except with the expression matching F removed. If variables had to be bound to make the match, substitute these bindings for the bound variables in the rule.

        (ii) If you've now removed the entire right side of rule R, you've proved a fact: the current left side. Enter that left side into the list of facts, and

mark it "unused". (The focus-of-attention approach here puts the new fact in front of other unused facts.) Eliminate from further consideration all rules whose left sides are equivalent to (not just matchable to) the fact proved.

(iii) Otherwise, if there's still some right side remaining, put the new simplified rule in front of the old rule. Cross out the old rule if it is now redundant. It is redundant if the old rule always succeeds whenever the new rule succeeds, which is true when no variables were bound to make the match.

(b) Mark F as "used".

3. Create a new fact list consisting of every **not** expression mentioned in rules whose argument does not match any used fact. Mark all these as "unused", and redo step 2.

A forward chaining example

Let's take an example of forward chaining with focus-of-attention handling of new facts. Consider the same rules used for backward chaining:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R5 */ b :- d.
/* R6 */ b :- e.
/* R7 */ c(2) :- not(e).
/* R8 */ d :- fact2, fact3.
/* R9 */ e :- fact2, fact4.
```

Suppose the rules are taken in the given order; and that as before, only **fact2** and **fact3** are true, in that order (see Figure 6-2).

1. We start with **fact2**, and find the matching predicate expressions R8 and R9. This gives the new rules

```
/* R10 */ d :- fact3.
/* R11 */ e :- fact4.
```

Rules R8 and R9 are now redundant since no variables were bound, and R8 and R9 can be eliminated.

2. No new facts were discovered, so we pursue next **fact3**. This matches an expression in R10. So now R10 succeeds, and the new fact **d** is put in front of any remaining unused facts (though there aren't any now). R10 can be eliminated.

3. We pursue fact **d**, and find that rule R5 mentions it in its right side. (R4 mentions it too, but as **not(d)**, and we're saving **not**s for last.) Matching R5 gives the new fact **b**. Rules R5 and R6 can now be eliminated.

4. Fact **b** matches in R2, giving

```
/* R12 */ goal1 :- a.
```

and rule R2 can be eliminated. The current set of rules is:

```
/* R1 */ goal1 :- fact1.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R7 */ c(2) :- not(e).
/* R11 */ e :- fact4.
/* R12 */ goal1 :- a.
```

5. We have no more facts to pursue. But we're not done yet, since R4 and R7 have **not**s.

6. Fact **d** is true, so rule R4 can't ever succeed. But fact **e** has not been proved. Hence add **not(e)** to the list of facts.

7. This matches the right side of R7. Hence **c(2)** is a fact too. Eliminate R7.

8. This matches the only expression on the right side of R3, when **X=2**, and hence **goal2(2)** is a fact. We can't eliminate R3 now because we had to bind a variable to make the match, and we can still use R3 for other values of **X**.

9. That's everything we can conclude.

Though Prolog interpreters don't *automatically* forward chain, it's not hard to teach them--see Section 7.10.

This "pure" forward chaining is rarer in applications than backward chaining. There's a story about a huge metal robot that came clanking into a bar one day. "I'm a pure-forward-chaining robot, and I can do a complete analysis of the quality of any liquor-dispensing establishment with a single sample. Please mix a martini for me, and pour it down the analysis chute in my chest." The bartender did so, and said, "That'll be eleven dollars. Say, we don't get too many forward-chaining robots in here." "At your prices I'm not surprised," replied the robot.

Hybrid control structures

Different control structure ideas can be combined in *hybrid* control structures. Hybrids of forward and backward chaining, compromising on the advantages and disadvantages of both, are often used, The most common is the *rule-cycle hybrid* | REFERENCE 1| because it is easy to implement (see Section 7.9). .FS | REFERENCE 1| The rule-cycle hybrid is often confused with pure forward chaining. .FE

With the rule-cycle hybrid, rules are tried in order as with backward chaining, but each rule is used in a forward chaining (modus ponens) way to assert new facts. The rule list is cycled through repeatedly, first ignoring any rules with **not**s. If the conditions on the right side of some rule all match facts (that's *facts*, not just something provable), then the rule succeeds and its left side (with appropriate bindings) is added to the database as a new fact. When no new rules succeed on a cycle through all of them, rules with **not**s are now considered; cycling resumes at the top of the rules, with the **not** expressions now succeeding if their arguments aren't now facts. Again we continue until no new rules succeed on a cycle. So with the rule-cycle hybrid, rule order takes precedence over fact order, but it's different than with backward chaining. Figure 6-3

summarizes the differences between forward chaining, backward chaining, and the rule-cycle hybrid.

Here's a more formal algorithm. Warning: it will only work with the restriction that no **not(p)** occurs in the right side of a rule before a rule having **p** as its left side, for any **p**, but that's usually easy to satisfy.

> Cycle through the rules repeatedly until no new facts are found on a cycle, ignoring rules with **not**s.
>
> > For each cycle, consider the rules in order.
> >
> > > For each rule R, treat its right side as a query about the facts (without using any other rules via backward chaining). If R succeeds, add its left side (with substitution of bindings made) as a fact at the front of the list of facts. And then eliminate from further consideration all rules whose left sides are equivalent to this new fact.
>
> Now repeat the previous step with *all* the original rules, taking also as true the **not**s whose arguments are not facts.

Take our standard example:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R5 */ b :- d.
/* R6 */ b :- e.
/* R7 */ c(2) :- not(e).
/* R8 */ d :- fact2, fact3.
/* R9 */ e :- fact2, fact4.
```

With the rule-cycle hybrid when **fact2** and **fact3** are true (see Figure 6-4):

> 1. R1, R2, R3, R5 and R6 are tried (we skip R4 and R7 because they have **not**s). None succeed because nothing on any right side matches a fact that is stated to be true.
>
> 2. R8 is tried, and it succeeds. Fact **d** is asserted. Eliminate R8.
>
> 3. R9 fails.
>
> 4. We return to the top of the rule list and start a new cycle. Rules R1 through R3 fail as before, and R4 is ignored.
>
> 5. But R5 now succeeds since **d** is a fact. Fact **b** is asserted. Eliminate R5 and R6.
>
> Now the rules are:
>
> ```
> /* R1 */ goal1 :- fact1.
> /* R2 */ goal1 :- a, b.
> /* R3 */ goal2(X) :- c(X).
> /* R4 */ a :- not(d).
> ```

```
/* R7 */ c(2) :- not(e).
/* R9 */ e :- fact2, fact4.
```

6. R7 and R9 fail (R8 was eliminated). And all the remaining rules fail on the next cycle.

7. Possibilities are exhausted, so we must now include rules with **not**s. R1, R2, and R3 fail as before, and R4 fails because **d** is a fact.

8. R5 and R6 were eliminated, and the **not** in R7 succeeds because **e** is not a fact. So **c(2)** is a fact. Eliminate R7.

9. None of R8, R9, R1, and R2 succeed. But R3 succeeds, with **X**=2, and **goal2(2)** must be a fact. We can't eliminate R3 because **goal2(2)** is more specific than the left side of R3. But we're done now if we only want to reach one goal.

A different hybrid of forward and backward chaining alternates (*time-shares*) between forward and backward chaining steps. It picks a fact, and finds what rule right sides mention it; it does backward chaining in those right sides to try to establish the left side as a fact. Then it picks another fact and does the same thing over again. This hybrid sometimes pays off when neither forward, backward, nor rule-cycle hybrid chaining works well.

Order variants

Query, fact, and rule ordering is important in backward, forward, and rule-cycle hybrid chaining, and control structures can apply many criteria to do it.

With backward chaining, predicate expressions in the query can be sorted by priority. In an "and", the expressions easiest to process or least likely to succeed can be done first. Sections 13.2 through 13.7 discuss these ideas at length.

With forward chaining, facts can be sorted by priority. The facts most useful (matchable in the right sides of the most rules) can go first, to help reach interesting conclusions faster. Or the facts most powerful in reaching conclusions can go first (statistics on queries can be kept for this).

Rule order is important to both forward and backward chaining. One common ordering is by *specificity*. Specificity doesn't mean how "concrete" the rules are, but whether conditions on the right side of rule 1 are a subset of conditions on the right side of rule 2; or in other words, whether the success of rule 2 means success of rule 1. The specificity ordering puts rules for the most narrowly described situations (or *exceptions*) first, then those for less narrow situations, then those still less narrow, and so on up to very broad *default* rules. (If the most broadest rules came first, the others would never get a chance to work.) Since catching subset relationships between rule right sides requires some analysis, a variant is to put the *longest* rules first, but this doesn't work quite as well.

Here's an example for rule-cycle-hybrid chaining in which all the rule left sides are goals:

```
/* R1 */ u :- b.
/* R2 */ v :- c.
/* R3 */ w :- b, c, d.
/* R4 */ x :- d, e.
```

```
/* R5 */ y :- b.
/* R6 */ z :- b, d.
```

A rule-specificity ordering would insist that R3 be in front of R1, R2, R5, and R6, and insist that R6 be in front of R1 and R5.

There are two problems with specificity ordering. The first is that there are often few such subset relationships among rules, leaving undecided how to complete the ordering. The second is that the narrowest rules apply rarely, meaning wasted work if they're first. Chapter 13 will study these issues. But usually a quick fix is available in the adding of extra **not** expressions to the broader rules, and putting those rules first. For instance, for our last example it may be possible to rewrite R1 as

```
/* R1 */ u :- b, not(c), not(d).
```

and then it doesn't matter where it goes.

This is often what the original rules really meant, since we often forget the minor exceptions to rules--there are just too many. For example, with the rule "if a car won't start and the radio won't play and the lights don't light, then the battery is dead", we don't rule out the chance that aliens from outer space have nullified all electric fields.

Partitioned control structures

When there are a thousand or more rules as in major current expert systems, the rules can interrelate in many ways, and a bug can be hard to trace. So just like large computer programs, it's a good idea to divide rules into groups, modules, or partitions for which members of each group have minimal interactions with members of other groups. You can think of each group as a separate rule-based system that may occasionally decide to call on another for a separate analysis. An advantage is that the rule groups can be written and debugged mostly separately. This idea is called a *partitioning* or *context-limiting* control structure.

Diagnosis expert systems provide good examples. For instance in diagnosis of a malfunctioning car, there are major systems of the car that don't interact much with one another. If electrical devices aren't working, you can be pretty sure that the problem is in the electrical system; or if the car goes forward but not backward, you can be pretty sure the problem is in the transmission. So you can put rules for electrical problems in one partition, and rules for transmission problems in another, rules for the engine and fuel system in another, rules for the car body in another, and so on. You'll need one other partition of rules, a "startup" partition, to look at key evidence and decide which partition appears most relevant to a problem. And partitions can choose to transfer control to another partition, if say none of their own rules succeed.

Meta-rules

A general approach can encompass all the control structure ideas so far: specification of control by a rule-based system itself. *Meta-rules* are just rules whose domain of knowledge is the operation of another rule-based system; they're a kind of *heuristic*, a topic we'll investigate more thoroughly in Chapter 9. Rules deciding to load partitions (Section 6.6) are one simple example of meta-rules, but they can do many other things. Remember that one goal for putting knowledge into computers was to make explicit the complicated "common-sense" knowledge people have but don't realize they have. How to order rules and use them is another kind of common-sense knowledge, also formalizable. Here are some example meta-rules for a

backward-chaining-like rule-based system, to control selection of the next rule to try to satisfy instead of following the database order of rules:

--Prefer the rule that handles the most serious issue.
--Prefer the rule that was written by the most knowledgeable human.
--Prefer the rule that is fastest to execute.
--Prefer the rule that has been used successfully the most times.
--Prefer the rule with the most things in common with the last rule successfully applied.

The big advantage of meta-rules is their flexibility and modifiability, which allows precise control of a rule-based system.

Meta-rules can express things besides rule orderings and partition referrals. Prolog interpreters make the *closed-world assumption* or *lack-of-knowledge inference*: if you can't prove something true, assume it false. This may be unfair for some predicates; a meta-rule could then override normal reasoning. For instance, a meta-rule could say to use the closed-world assumption only when querying predicates from a certain list, and to assume a failure means **yes** otherwise.

Meta-rules seem to be important in human reasoning. People aren't generally systematic enough to use any of the chaining methods successfully, but instead they rely on problem-specific meta-rules for deciding what to do next. So to reason more naturally, meta-rules are critical. But figuring out just what meta-rules people do use is hard.

Decision lattices

We'll now consider some lower-level control structure ideas: decision lattices, concurrency, and and-or-not lattices. In the terminology of computer science, these are *compiled* structures. But they're compiled in a different sense than what programming-language "compilers" produce: they represent a simplifying first step before the traditional compiler operates. Some people don't consider these compiled structures truly artificial intelligence, but they're so closely linked to artificial intelligence that we'd better explain them.

Choosing a good sequence for rules can be important and hard, as we discussed in Section 6.5. But computers can use storage structures besides sequences (see Appendix C). They can organize rules in a hierarchy, what is called a *decision lattice* or *discrimination net*. Decision lattices do a restricted but very efficient kind of reasoning, a kind of classification. The idea is to always specify where to go next in the computer based on question answers. In other words, a kind of *finite-state machine*. (Sometimes they're called *decision trees*, but technically they're lattices since branches that diverge can converge or "grow back together" later. Any graph without cycles in which this convergence can happen is a lattice and not a tree; cycles wouldn't make much sense here because you'd be asking the same question twice.)

For instance, consider an expert system to diagnose malfunctions of small household appliances (see Figure 6-5). It is important first to distinguish problems within the appliance from problems outside the appliance. A good way is to ask if the appliance works at all. If it doesn't, ask if it is plugged in. If it isn't, that is the problem. If it is, ask if other electric devices nearby (lights, clocks, etc.) work. If they don't, the problem sounds like a blown fuse. If other appliances definitely work, the problem must be internal to the faulty appliance. If no such observations can be made (as when there are no electrical appliances nearby), try plugging the faulty appliance into another outlet to see if the problem reappears.

On the other hand, if the appliance partially works, then it matters what kind of appliance it is. That's because interpretation of partial-failure clues is quite appliance-dependent, like smoke when the device has a heating element. As another example, strange noises are more serious in a device with no moving parts than in a blender. So the next question for a partially-working appliance should classify it.

So decision lattices impose a classification hierarchy on the universe based on observations. They are useful for simple expert systems, with several advantages:

1. Implementation is easy: just use pointers (memory references). They can even be implemented without a computer, as printed text with cross-references.

2. They need not explicitly question a human being: they can examine buffer contents or sensor readings. Then they can be fast, faster than the chaining methods, because no matching, binding, or backtracking is needed.

3. They can be designed to ask the absolutely minimal number of questions necessary to establish conclusions, unlike chaining methods for which such optimization can be difficult.

But decision lattices have major disadvantages as a compiled or "low-level" control structure:

1. They can't reason properly or efficiently for many applications because they don't easily permit variables or backtracking.

2. They are difficult to modify and debug, since later questions must assume certain results to earlier questions.

3. They can't easily reuse query answers since they don't explicitly cache.

4. They may be hard to build, because at each point you try to determine the best question to ask, something not so easy to judge.

Decision lattices were around long before computers. Expert-system technology only made significant progress when decision-lattice control structures were mostly abandoned, due to the limitations mentioned.

Concurrency in control structures

If speed of a rule-based system is important (as in a real-time application), and multiple processors are available, a control structure can use concurrency. Usually the processors must share and access the same database of facts and rules for this to work well. For a Prolog-style rule-based system, four types of parallelism for concurrency are identified (see Figure 6-6): (1) *partition parallelism*, (2) *or parallelism*, (3) *and parallelism*, and (4) *variable-matching parallelism*. These parallelisms are useful with all three kinds of chaining.

Partition parallelism means running different partitions of the rules simultaneously. Each partition can reason separately, though they can explicitly pass conclusions to one another, or cache into a global database. This is good if we've got groups of rules that don't interact much, each group relevant to a problem.

"And" parallelism is parallelism among expressions "and"ed on the right side of a rule or a query. Usually it

is only done for the predicate expressions that do not bind variables, the "tests" in the generate-and-test concept (see Section 3.12). These tests can be done concurrently on separate processors; if any test fails, the whole "and" should fail, and the other processors should be sent a message to stop work. Otherwise, the "and" should succeed. "And" parallelism is probably not a good idea when some tests are much harder to satisfy than others; then the hard tests should go first (see Chapter 13).

"Or" parallelism usually means parallelism between rules with the same left-side predicate name. It is good when there are many such rule groups. Or-parallel rules are sometimes called *demons* because they're like little people each independently waiting for a particular set of conditions to be satisfied. "Or" parallelism can also mean parallel pursuit of facts in forward chaining.

Variable-matching parallelism is parallelism in the argument matching done when matching two predicate expressions to one another. It makes each match attempt faster, but it doesn't change the usual sequential examining of the database. It only pays off when you have a significant number of predicates with two or more arguments.

Concurrency can be simulated on a sequential machine. This gives a new class of control structures, the sequential reductions of concurrent process descriptions. This idea is often associated with the "agenda" search methods in Chapter 10.

Parallelism is not just an efficiency trick, however. Parallelism is necessary to model many real-world phenomena. These phenomena are often addressed in *object-oriented programming*, for which the world is divided into clusters of facts representing *objects*, each with its own partitioned module of rules governing its behavior. Object-oriented programming is especially useful for simulations. For instance, objects (and their facts) can represent organisms in an ecosystem, and rule modules for each kind of organism can govern the behavior of each object. Another application is to modeling components of a car, where each object represents a part of a car. We'll talk more about object-oriented programming in Chapter 12. While it emphasizes rule-partition parallelism, it can also involve the other three kinds. For instance in modeling organisms in an ecosystem, "and" and "or" parallelism can reflect the ability of organisms to do and think several things simultaneously.

And-or-not lattices

The extreme case of parallelism in rule-based systems is the *and-or-not lattice* representation of rules, in which each rule can be thought (or maybe actually is) a hardware logic gate incessantly computing a certain logical combination of input logic signals representing facts and intermediate conclusions. (It's sometimes incorrectly called an *and-or tree*, but like in the decision lattice, the paths can recombine after splitting, so it isn't a tree necessarily.) "And"s in a rule become "and" gates, "or"s become "or" gates, and "not"s becomes inverter gates.

For instance, for the rules used previously:

```
/* R1 */ goal1 :- fact1.
/* R2 */ goal1 :- a, b.
/* R3 */ goal2(X) :- c(X).
/* R4 */ a :- not(d).
/* R5 */ b :- d.
/* R6 */ b :- e.
/* R7 */ c(2) :- not(e).
```

```
/* R8 */ d :- fact2, fact3.
/* R9 */ e :- fact2, fact4.
```

the and-or-not lattice is shown in Figure 6-7. Here we follow the usual conventions for gate shapes (see Appendix A); facts are the far-left-side "inputs" and goals are the far-right-side "outputs" of this logic network. Truth of a fact or conclusion is represented by a high or "on" voltage along a line, falsity by a low or "off" voltage. The output (right-side) voltage of a gate is determined by the logic function of the voltages representing input(s) on the left side; "and" gates determine a logical "and", "or" gates logical "or", and inverter gates logical "not". A given set of facts sets the input voltages for a group of initial gates, which determine voltages of others, and so on, with everything proceeding in parallel. The and-or-not lattice is a useful interpretation of rules and facts for the many applications in which order within "and"s and "or"s doesn't mean anything. It also provides a specification for an integrated-circuit chip that could do this very fast.

This unordered interpretation of rules takes us to the opposite extreme of the classical Eckert-Mauchly | REFERENCE 2| model of the computer as a sequential processor. .FS | REFERENCE 2| Sometimes called the Von Neumann model, but evidence now suggests that Von Neumann had little to do with it. .FE The main advantage of the and-or-not lattice is processing speed. Another advantage is the partitionability into modules: we can identify subnetworks and their inputs and outputs, and then build them easily into large networks. We don't need to worry much about *redundancies*, gates whose effect is also accomplished by other gates; they can't slow computation. But it is true we can't have contradictory gates or results will be meaningless.

Like decision lattices, and-or-not lattices do have the disadvantage that they can't handle variables well (you'll notice the example has none). Gates can't directly "bind" variables, though we could indirectly represent alternative bindings by having multi-position switches on each input to the logic network, switches that could connect the inputs to one of several lines representing the truths of different facts with the same predicate name. Then binding means selecting a switch position, and trying possible bindings means turning the switch and watching the conclusions reached. But this is awkward, and--especially when there is more than one variable--can greatly increase the time to get an answer, a main reason for using and-or-not lattices in the first place.

Randomness in control structures

A control structure need not be deterministic (that is, always work the same way on the same rules and facts). Human beings seem to have some randomness in their actions. So a control structure can make some random choices, especially when alternatives seem of equal worth. For instance, maybe in backward chaining when there are more than ten rules applicable to a situation, choose one at random. Randomness is most appropriate for rule-based systems trying to model skilled real-time performance by people. Randomness can prevent infinite loops, because when you choose randomly you can't get stuck in a rut.

Grammars for interpreting languages (*)

Human and computer languages can be handled and processed with *grammars*. Grammars are rule-based systems different from those considered so far. For one thing, grammar rules don't deal with facts, but with strings of words or symbols in a particular order. All grammar rules are of a particular narrow form: a statement that one string of words can be substituted for another string of words. Though we didn't say so, we saw an example of a grammar in Section 5.9: those substitution pairs, though disguised as facts. But

grammars can also have *nonterminal* words, words that symbolize grammatical categories and are not part of the language itself. For instance, the nonterminal "noun" can be substituted for the word "man" or the word "computer". Nonterminals can also describe sequences of words, like identifying a "noun phrase" as a determiner followed by an adjective followed by a noun. Linguists have lots of these categories.

Grammars are essential for getting computers to understand sentences in a language, because they make it possible to determine the structure of a sentence. Used this way, grammars are analogous to the rule-based systems in this chapter. The "facts" are the words of a sentence, and the goal is to change that sentence, by substitutions, into the sentence consisting of the single nonterminal called "sentence". Such a process is analogous to forward chaining, and is called *bottom-up parsing*. It usually involves substituting shorter sequences of words for longer sequences of words. But you can also work in the opposite direction from "sentence" to a list of actual language words, analogously to backward chaining, and this is called *top-down parsing*. It usually involves substituting longer sequences of words for shorter sequences of words, and so is likely to require more backtracking than bottom-up parsing, but less work per backtrack since shorter sequences are searched for. Figure 6-8 shows an example; upward arrows represent bottom-up parsing, downward arrows top-down.

Hybrids that compromise on the advantages and disadvantages of both bottom-up and top-down parsing are possible. Grammar rule order is important for efficiency: the most likely rules should be tried first. Parts of the sentence can be processed concurrently. Partitioning can be used to group information about related words and related parsing rules into modules. Meta-rules can be used for sophisticated control more like what people do.

Today most artificial-intelligence work in understanding sentences from human languages (or *natural languages*) uses a variation on top-down parsing called *augmented transition networks* (ATNs). These are a "smarter" kind of top-down parsing that attaches additional conditions to the parsing rules, so that only the most "reasonable" rules that apply will be tried. They also use recursion in a complicated way similar to the one in Chapter 11.

Keywords:

rule-based systems
control structure
conflict resolution
expert systems
compiled control structures
backward chaining
goal
virtual facts
the ***asserta predicate***
the retract predicate
forward chaining
focus-of-attention feature
rule-cycle hybrid control structure
rule and fact priorities
rule specificity
partitioned control structures
meta-rules
decision trees
partition parallelism

"or" parallelism
demons
"and" parallelism
variable-matching parallelism
object-oriented programming
and-or-not lattice
grammar
top-down parsing
bottom-up parsing
augmented transition networks (ATNs)

Exercises

6-1. (A) Here's a rule-based system:

```
/* R1 */ k(X) :- j(X), b(X).
/* R2 */ f(X) :- a(X), not(g(X)).
/* R3 */ a(X) :- b(X), i.
/* R4 */ d :- i.
/* R5 */ d :- e(X), c.
/* R6 */ g(X) :- h, a(X).
/* R7 */ g(X) :- l.
/* R8 */ b(X) :- c.
```

***Assume the goals are f(X), d*, and k(X), in that order, and the facts are c, l, e(a), and j(b), in that order. Assume no extra caching. Assume we stop when a goal is proved.**

(a) Suppose we use pure forward chaining. List the rule invocations, successes, and failures, in order as they occur. Use the rule numbers to indicate rules.

(b) Now list the rule invocations, successes, and failures for backward chaining, in order.

(c) Does fact order affect which goal is proved first with forward chaining? Why?

(d) Does fact order affect which goal is proved first with backward chaining here? Why?

6-2. Consider this database:

```
top(X,Y,Z) :- bottom(Z,W,Y,X).
bottom(A,B,7,D) :- data(A,0,B), data(A,D,1).
data(3,0,1).
data(3,2,1).
```

List in order the facts that are proved by pure forward chaining using focus-of-attention placement of new facts. Don't stop until everything provable is proved.

6-3. Suppose we are doing pure forward chaining on a set of R rules, rules without semicolons ("or"s), nots, arithmetic, and variables. Suppose these rules have L distinct predicate expressions on the left sides, and S distinct predicate expressions, each occurring no more than M times, on the right sides. Suppose there are T total predicate expressions on right sides. Suppose there are F facts, F>0.

(a) What is the maximum number of locations immediately matchable to facts (that is, without having

to prove new facts) on rule right sides?

(b) What is the maximum number of new facts that can be eventually found by forward chaining?

(c) What would be the effect on your answer to part (a) of allowing one-argument variables in rules and facts?

(d) What would be the effect on your answer to part (b) of allowing one-argument variables in rules and facts? (A conclusion with an unbound variable still counts as one new fact.)

6-4. Consider the following Prolog database:

```
a(X) :- b(X).
b(X) :- c(X), d(X).
d(X) :- e, f(X).
c(X) :- g(X).
g(2).
f(5).
g(5).
e.
```

(a) What new facts are proved in order by the rule-cycle hybrid of forward and backward chaining? Continue until all possible facts are found.

(b) What new facts are proved in order by the pure form of forward chaining with focus-of-attention conflict resolution for facts? Continue until all possible facts are found.

6-5. (R,A) Consider the following rules and facts:

```
a :- v, t.
a :- b, u, not(t).
m(X) :- n(X), b.
b :- c.
t :- r, s.
u :- v, r.
r.
v.
c.
n(12).
```

(a) Suppose we do pure forward chaining with focus-of-attention placement of new facts. Suppose we want all possible conclusions. List in order the new facts derived from the preceding. Remember we must save nots for last.

(b) Suppose we do rule-cycle hybrid chaining with focus-of-attention placement of new facts, saving nots for last, and we want all possible conclusions. List in order the new facts derived.

(c) One problem with rule-cycle hybrid chaining is that it repeatedly checks the same rules again on each cycle. Describe a way to know when not to check a rule because of what happened on the last cycle, besides the idea of deleting a rule without variables when it succeeds. (Hint: using this idea you only need check nine rules total for the preceding rules and facts.)

6-6. (E) Consider a variant of the rule-cycle hybrid that only cycles through the rules once. How is this like an assembly line in a factory?

6-7. (A) Suppose we are doing rule-cycle hybrid chaining on R rules, rules without variables and nots. Suppose there are L distinct predicate names on rule left sides, and S distinct predicate names on rule right sides out of T total predicate names on right sides. Suppose there are F facts, F>0. What is the maximum number of cycles needed to prove everything that can be proved?

6-8. (A,H) Reasoning about rules and facts written in English instead of Prolog can be tricky, and you must be careful. Consider the following rule-based system for actions of a small reconnaissance robot. Suppose the following facts are true, in this order:

F1: There is an object with branches to the right of you.
F2: This object is 2 feet tall.
F3: This object occupies 20 cubic feet.
F4: This object is stationary.
F5: Another object is moving towards you.
F6: You hear speech from that object. (Assume speech is not a loud noise.)

Assume all other facts mentioned in rules are false. Assume any new facts, when added, will be put at the front of the list of facts. Assume the rule-based system can result in the following actions, in this order:

A1: Turn around.
A2: Stop and wait.
A3: Turn towards something.
A4: Move a short distance forward.
A5: Turn 20 degrees right.
A6: Move a long distance forward.

Here are the rules:

R1: If you hear a loud noise in front of you, then turn around and move a long distance.

R2: If you want to hide, and there is a bush nearby, then turn towards the bush, and move a short distance.

R3: If you want to hide, and are beneath a bush, then stop and wait.

R4: If an object is moving towards you, and it is a person or vehicle, then hide.

R5: If an object is moving, and it is an animal, then stop and wait.

R6: If an object is an obstacle and you are moving, and the object is blocking your path, then turn right 20 degrees, and move a short distance.

R7: Move forward a long distance. [notice no "if" part here]

R8: If an object has long branches, and the branches are moving, and it does not have

wheels, then it is an animal.

R9: If an object makes irregular noises, then it is an animal.

R10: If an object makes regular noises, and it is moving, it is a vehicle.

R11: If an object has wheels, then it is a vehicle.

R12: If an object is stationary, and occupies more than 1 cubic foot, it is an obstacle.

R13: If an obstacle has branches, and is less than 3 feet tall, it is a bush.

R14: If an obstacle has branches, and is more than 3 feet tall, it is a tree.

R15: If an obstacle has no branches, it is a rock.

R16: If an animal has four branches in two pairs, and one pair supports the animal, it is a person.

R17: If an animal speaks, it is a person.

(a) List in order the rule invocations, successes, and failures with backward chaining. Assume conflict resolution based on the rule order given. Assume caching of proved facts, so once something is concluded it need never be figured out again.

(b) List in order the rules invoked with forward chaining, ignoring rule R7. Again, take rules in the order given,

(c) Give a different kind of conflict resolution that would work well for this rule-based system.

6-9. (A) A rule-based system is monotonic if anything that can be concluded at one time can be concluded at any later time. Consider a consistent (that is, non-self-contradictory) "pure" backward-chaining rule-based system, one that doesn't use asserta or retract or any built-in predicates besides not. Such a system is necessarily monotonic.

(a) Suppose the system is partitioned into modules whose rules cannot "see" the rules of other modules until those modules are specifically loaded. Must the new scheme be monotonic?

(b) Suppose we cache intermediate and final conclusions reached by the system, using asserta. Must the new scheme be monotonic?

(c) Suppose we add to right sides of rules in the system asserta predicate expressions with arbitrary arguments. Must the new scheme be monotonic?

6-10. (E) (a) Despite their similar names, decision lattices and and-or-not lattices are quite different things. List their major differences.

(b) List their major similarities.

6-11. Consider the following rule-based system for self-diagnosis and treatment of colds and flu:

A. To sleep, go to bedroom.
B. To drink fluids, go to kitchen.
C. To drink fluids, go to bathroom.
D. To take temperature, go to bathroom.
E. To take aspirin, go to bathroom.
F. To telephone, go to living room.
G. To telephone, go to kitchen.
H. If feel headache or nasal congestion, then feel sick.
I. If feel sick, then take temperature.
J. If have fever, then take aspirin.
K. If have fever, then call boss.
L. If have fever, then go to bed.
M. If have nasal congestion or fever, then drink fluids.

(a) Order the rules in a good way. Show how backward chaining would work applied to the circumstance when you wake up with a headache, nasal congestion, and a fever. (Use the letter in front of each rule to identify it.)

(b) Suggest a fundamentally different control structure than the preceding, one good for this problem. (Find a general control structure that could work in many daily-living rule-based systems, not just this problem.) Show how it would work on the first eight rules applied to the same situation.

6-12. (R,A) Consider using a rule-based system as the brains of a "smart house", a house that automatically does a lot of things that people do for themselves in other houses. Assume the smart house can control power to all the electrical sockets, so it can turn lights, heat and appliances on and off at programmed times. The smart house can also monitor many different kinds of sensors--for instance, light sensors to turn off outside lights during the daytime, infared sensors to detect when people are in a room, audio sensors to detect unusual noises, and contact sensors to detect window openings (as by a burglar). Sensors permit flexible, unprogrammed action, like sounding a burglar alarm in the master bedroom only if someone is there, or turning off appliances when they seem to have been left on accidentally and no one is in the room anymore. Priorities could be established, so routine activities won't be allowed when sensors say a fire is progress. Assume the rule-based system is invoked repeatedly to check for new conditions.

(a) Which is better for this problem, backward chaining or forward chaining? Why?

(b) Is caching a good idea? Why? If so, how would it work?

(c) Are virtual facts a good idea? Why? If so, how would it work?

(d) Is rule partitioning a good idea? Why? If so, how would it work?

(e) Are decision lattices a good idea? Why? If so, how would they work?

(f) Are and-or-not lattices implemented on integrated circuits a good idea? Why? If so, how would they work?

6-13. On the popular fictitious television show *The Citizenry's Courtroom*, Judge Wimpner hears small-claims cases and makes monetary awards of varying amounts to the plaintiff. But the award is misleading because the plaintiff and defendant are also paid for their appearance on the program, as one will discover on reading the small print of a disclaimer flashed on the screen for a second during the closing credits. The plaintiff is given 25 dollars plus the amount of the award plus half of a pool. The defendant is given 25 dollars plus the other half of the pool. The pool is zero if the amount of the award is 500 dollars or more, otherwise the difference between 500 dollars and the amount of the award.

(a) Suppose the award amount is represented in Prolog as a fact with predicate name **award** and one argument representing the amount of the award in dollars. Write three Prolog rules representing the stated calculations necessary for the plaintiff's amount, the defendant's amount, and the pool amount. Don't simplify; represent exactly what was stated.

(b) Now "compile" the three preceding rules into two rules for the plaintiff amount and the defendant amount. Refer to the award amount only once in each rule.

(c) Discuss what is needed to do this compilation for any set of rules involving arithmetic. Will a few simple tricks do it, or is a large rule-based expert system of its own necessary?

(d) What are the disadvantages of this form of compilation?

(e) Compare and contrast this kind of compilation with and-or-not lattices.

6-14. (E) *Occam's Razor* is the principle that when several alternative explanations are possible for something, you should pick the simplest one. Suppose we try to apply Occam's Razor to a rule-based system in which the left sides of rules represent explanations of phenomena.

(a) Which of the terms in Figure 6-1 best describes Occam's Razor?

(b) Why would it be very difficult to apply Occam's Razor to a practical problem?

6-15. (E) Compare and contrast the control structures discussed in this chapter with control of activities (by directive) in bureaucracies.

6-16. (P) Write a Prolog program to drive a car. Assume the following sensor facts are known to you, as "data":

```
S1: you are traveling below the speed limit
S2: you are traveling at the speed limit
S3: you are traveling above the speed limit
S4: you are on a two-lane road
S5: you are on a four-lane road
S6: you see an intersection coming up
S7: a car is less than 100 meters in front of you
S8: the road changes from two-lane to four-lane shortly
S9: the road will change from four-lane to two-lane shortly
S10: the brake lights of the car in front of you are on
S11: you are getting closer to the car in front of you
S12: you are passing another car going in the same direction
```

Assume the only actions are

```
A1: speed up
A2: slow down
A3: maintain speed
A4: pass a car in front of you (or keep passing one you're passing)
```

Here are examples of what you want the program to do. (These are not good rules--they're too specific.)

```
S2,S4,S6: slow down (A2)
S2,S5,S8,S10: slow down (A2)
S1,S5,S11: pass (A4)
S1,S5: speed up (A1)
S3,S4,S7: slow down (A2)
```

Write a Prolog program that decides what to do for every possible combination of the sensor facts. (Since there are 12 kinds, you must handle |2 sup 12|, or 4096 different situations). Assume your program is called every second to decide what to do that second. Choose a good control structure, and discuss (justify) your ordering of rules. Try to drive safely.

6-17. (E) Sometimes the ideas of science fiction writers seem pretty wild, but sometimes the ideas that seem wild really aren't when you think about them a bit.

(a) In *Camp Concentration* by Thomas Disch (1968), it's suggested that a disease that rearranges nerve patterns in the human brain could cause people to become a lot smarter, more able to reason and make inferences. Why is this not reasonable based on the assumption that people reason using the methods of this chapter? Suggest a small change to ideas from this chapter that would permit this phenomenon.

(b) In *Brain Wave* by Poul Anderson (l954), it's suggested that a small increase in speed of the nerve impulses in the human brain could lead to a much larger increase in the speed at which humans reason and make inferences. Why is this not reasonable based on the assumption that people reason using the methods of this chapter? Give a small change to ideas from this chapter that permit this phenomenon.

Go to book index