Alison Cawsey
Department of Computing and Electrical Engineering
Heriot-Watt University
Edinburgh EH14 4AS, UK

Rule-Based Systems

Instead of representing knowledge in a relatively declarative, static way (as a bunch of things that are true), rule-based system represent knowledge in terms of a bunch of rules that tell you what you should do or what you could conclude in different situations. A rule-based system consists of a bunch of IF-THEN *rules*, a bunch of *facts*, and some *interpreter* controlling the application of the rules, given the facts.
There are two broad kinds of rule system: *forward chaining* systems, and *backward chaining* systems. In a forward chaining system you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts. In a backward chaining system you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new subgoals to prove as you go. Forward chaining systems are primarily data-driven, while backward chaining systems are goal-driven. We'll look at both, and when each might be useful.
[Note: I previously used the term production system to refer to rule-based systems, and some books will use this term. However, it is a non-intuitive term so I'll avoid it.]

Forward Chaining Systems

In a forward chaining system the facts in the system are represented in a *working memory* which is continually updated. Rules in the system represent possible actions to take when specified conditions hold on items in the working memory - they are sometimes called condition-action rules. The conditions are usually *patterns* that must *match* items in the working memory, while the actions usually involve *adding* or *deleting* items from the working memory.
The interpreter controls the application of the rules, given the working memory, thus controlling the system's activity. It is based on a cycle of activity sometimes known as a *recognise-act* cycle. The system first checks to find all the rules whose conditions hold, given the current state of working memory. It then selects one and performs the actions in the action part of the rule. (The selection of a rule to fire is based on fixed strategies, known as *conflict resolution* strategies.) The actions will result in a new working memory, and the cycle begins again. This cycle will be repeated until either no rules fire, or some specified goal state is satisfied.
Rule-based systems vary greatly in their details and syntax, so the following examples are only illustrative.
First we'll look at a very simple set of rules:

1. IF (lecturing X)
   AND (marking-practicals X)
   THEN ADD (overworked X)
2. IF (month february)
   THEN ADD (lecturing alison)
3. IF (month february)
   THEN ADD (marking-practicals alison)
4. IF (overworked X)
   OR (slept-badly X)
   THEN ADD (bad-mood X)
5. IF (bad-mood X)
   THEN DELETE (happy X)
6. IF (lecturing X)
   THEN DELETE (researching X)

Here we use capital letters to indicate variables. In other representations variables may be indicated in different ways, such as by a ? or a ^ (e.g., ?person, ^person).
Let us assume that initially we have a working memory with the following elements:

(month february)
(happy alison)
(researching alison)

Our system will first go through all the rules checking which ones apply given the current working memory. Rules 2 and 3 both apply, so the system has to choose between them, using its conflict resolution strategies. Let us say that rule 2 is chosen. So, `(lecturing alison)` is added to the working memory, which is now:

```
(lecturing alison)
(month february)
(happy alison)
(researching alison)
```

Now the cycle begins again. This time rule 3 and rule 6 have their preconditions satisfied. Lets say rule 3 is chosen and fires, so `(marking-practicals alison)` is added to the working memory. On the third cycle rule 1 fires, so, with X bound to alison, `(overworked alison)` is added to working memory which is now:

```
(overworked alison)
(marking-practicals alison)
(lecturing alison)
(month february)
(happy alison)
(researching alison)
```

Now rules 4 and 6 can apply. Suppose rule 4 fires, and `(bad-mood alison)` is added to the working memory. And in the next cycle rule 5 is chosen and fires, with `(happy alison)` removed from the working memory. Finally, rule 6 will fire, and `(researching alison)` will be removed from working memory, to leave:

```
(bad-mood alison)
(overworked alison)
(marking-practicals alison)
(lecturing alison)
(month february)
```

(This example is not meant to a reflect my attitude to lecturing!)
The order that rules fire may be crucial, especially when rules may result in items being deleted from working memory. (Systems which allow items to be deleted are known as *nonmonotonic*). Anyway, suppose we have the following further rule in the rule set:

7

```
IF (happy X)
THEN (gives-high-marks X)
```

If this rule fires BEFORE (happy alison) is removed from working memory then the system will conclude that I'll give high marks. However, if rule 5 fires first then rule 7 will no longer apply. Of course, if we fire rule 7 and then later remove its preconditions, then it would be nice if its conclusions could then be automatically removed from working memory. Special systems called *truth maintenance systems* have been developed to allow this.
A number of conflict resolution strategies are typically used to decide which rule to fire. These include:

- Don't fire a rule twice on the same data. (We don't want to keep on adding `(lecturing alison)` to working memory).
- Fire rules on more recent working memory elements before older ones. This allows the system to follow through a single chain of reasoning, rather than keeping on drawing new conclusions from old data.
- Fire rules with more specific preconditions before ones with more general preconditions. This allows us to deal with non-standard cases. If, for example, we have a rule ``IF (bird X) THEN ADD (flies X)'' and another rule ``IF (bird X) AND (penguin X) THEN ADD (swims X)'' and a penguin called tweety, then we would fire the second rule first and start to draw conclusions from the fact that tweety swims.

These strategies may help in getting reasonable behaviour from a forward chaining system, but the most important thing is how we write the rules. They should be carefully constructed, with the preconditions specifying as precisely as possible when different rules should fire. Otherwise we will have little idea or control of what will happen.
Sometimes special working memory elements are used to help to control the behaviour of the system. For example, we might decide that there are certain basic stages of processing in doing some task, and certain rules should only be fired at a given stage - we could have a special working memory element `(stage 1)` and add `(stage 1)` to the preconditions of all the relevant rules, removing the working memory element when that stage was complete.

Backward Chaining Systems

[Rich &Knight, 6.3]

So far we have looked at how rule-based systems can be used to draw new conclusions from existing data, adding these conclusions to a working memory. This approach is most useful when you know all the initial facts, but don't have much idea what the conclusion might be.

If you DO know what the conclusion might be, or have some specific hypothesis to test, forward chaining systems may be inefficient. You COULD keep on forward chaining until no more rules apply or you have added your hypothesis to the working memory. But in the process the system is likely to do alot of irrelevant work, adding uninteresting conclusions to working memory. For example, suppose we are interested in whether Alison is in a bad mood. We could repeatedly fire rules, updating the working memory, checking each time whether `(bad-mood alison)` is in the new working memory. But maybe we had a whole batch of rules for drawing conclusions about what happens when I'm lecturing, or what happens in February - we really don't care about this, so would rather only have to draw the conclusions that are relevant to the goal.

This can be done by *backward chaining* from the goal state (or on some hypothesised state that we are interested in). This is essentially what Prolog does, so it should be fairly familiar to you by now. Given a goal state to try and prove (e.g., *(bad-mood alison)*) the system will first check to see if the goal matches the initial facts given. If it does, then that goal succeeds. If it doesn't the system will look for rules whose conclusions (previously referred to as *actions*) match the goal. One such rule will be chosen, and the system will then try to prove any facts in the preconditions of the rule using the same procedure, setting these as new goals to prove. Note that a backward chaining system does *NOT* need to update a working memory. Instead it needs to keep track of what goals it needs to prove to prove its main hypothesis.

In principle we can use the same set of rules for both forward and backward chaining. However, in practice we may choose to write the rules slightly differently if we are going to be using them for backward chaining. In backward chaining we are concerned with matching the conclusion of a rule against some goal that we are trying to prove. So the 'then' part of the rule is usually not expressed as an action to take (e.g., add/delete), but as a state which will be true if the premises are true.

So, suppose we have the following rules:

7. IF (lecturing X)
   AND (marking-practicals X)
   THEN (overworked X)
8. IF (month february)
   THEN (lecturing alison)
9. IF (month february)
   THEN (marking-practicals alison)
10. IF (overworked X)
    THEN (bad-mood X)
11. IF (slept-badly X)
    THEN (bad-mood X)
12. IF (month february)
    THEN (weather cold)
13. IF (year 1993)
    THEN (economy bad)

and initial facts:

(month february)
(year 1993)

and we're trying to prove:

(bad-mood alison)

First we check whether the goal state is in the initial facts. As it isn't there, we try matching it against the conclusions of the rules. It matches rules 4 and 5. Let us assume that rule 4 is chosen first - it will try to prove `(overworked alison)`. Rule 1 can be used, and the system will try to prove `(lecturing alison)` and `(marking practicals alison)`. Trying to prove the first goal, it will match rule 2 and try to prove `(month february)`. This is in the set of initial facts. We still have to prove `(marking-practicals alison)`. Rule 3 can be used, and we have proved the original goal `(bad-mood alison)`.

One way of implementing this basic mechanism is to use a stack of goals still to satisfy. You should repeatedly pop a goal of the stack, and try and prove it. If its in the set of initial facts then its proved. If it matches a rule which has a set of preconditions then the goals in the precondition are pushed onto the stack. Of course, this doesn't tell us what to do when there are several rules which may be used to prove a goal. If we were using Prolog to implement this kind of algorithm we might rely on its *backtracking* mechanism - it'll try one rule, and if that results in failure it will go back and try the other. However, if we use a programming language without a built in *search* procedure we need to decide explicitly what to do. One good approach is to use an agenda, where each item on the agenda represents one alternative path in the search for a solution. The system should try `expanding' each item on the agenda, systematically trying all possibilities until it finds a solution (or fails to). The particular method used for selecting items off the agenda determines the search strategy - in other words, determines how you decide on which options to try, in what order, when solving your problem. We'll go into this in much more detail in the section on search.

Forwards vs Backwards Reasoning

Whether you use forward or backwards reasoning to sove a problem depends on the properties of your rule set and initial facts. Sometimes, if you have some particular goal (to test some hypothesis), then backward chaining will be much more efficient, as you avoid drawing conclusions from irrelevant facts. However, sometimes backward chaining can be very wasteful - there may be many possible ways of trying to prove something, and you may have to try almost all of them before you find one that works. Forward chaining may be better if you have lots of things you want to prove (or if you just want to find out in general what new facts are true); when you have a small set of initial facts; and when there tend to be lots of different rules which allow you to draw the same conclusion. Backward chaining may be better if you are trying to prove a single fact, given a large set of initial facts, and where, if you used forward chaining, lots of rules would be eligible to fire in any cycle.