Rules in production system

Production systems formalize knowledge in a certain form called production rules and use forward-chaining reasoning to derive new things. They are used in many practical applications (e.g. expert systems).

A production system maintains a working memory (WM) which contains assertions that are changing during the operation of the system.

A production rule consists of an antecedent set of conditions and a consequent set of actions:

IF conditions THEN actions

The antecedent conditions are tests applied to the current state of the WM; the consequent actions are a set of actions that modify the WM.

The production system operates in a three-step cycle that repeats until no more rules are applicable to the WM:

1. recognize - find the applicable rules, i.e. those rules whose antecedent conditions are satisfied by the current WM.
2. resolve conflicts - among the rules found at the first step (called conflict set), choose those to fire.
3. act - change the WM by performing the consequent actions of all the rules selected at step 2.

Working Memory

WM consists of a set of working memory elements (WMEs).

A WME has the form:

(type attribute_1: value_1 ... attribute_n: value_n)

where type, attribute_i and value_i are atoms.

Examples (person age: 21 home: bucharest)
(student name: john dept: computerScience)

Declaratively, a WME is an existential sentence:

$$\exists x\,[\,type(x) \wedge attribute_1(x) = value_1 \wedge \ldots \wedge attribute_n(x) = value_n\,]$$

WMEs represent objects and relationships between them can be handled by reification (i.e. transforming a sentence into an object).

$$\left[\begin{array}{l} Purchases(john, bike, nov\,9) \\ \quad\quad\downarrow \\ Purchase(p3) \wedge agent(p3) = john \wedge object(p3) = bike \ldots \end{array}\right.$$

For example, (basicFact relation: olderThan firstArg: john secondArg: mary) may be used to express the fact that John is older than Mary.

<u>Production rules</u>

The conditions of a rule are connected by conjunctions. A condition can be positive or negative (written as -cond) and it has the form:

(type attribute$_1$: specification$_1$ ... attribute$_k$: specification$_k$)

where each specification is one of the following:

- an atom
- a variable
- an evaluable expression, within [ ]
- a test, within { }
- conjunction, disjunction or negation of a specification ($\wedge, \vee, \neg$)

For instance, (person age: [n+1] height: x) is satisfied if there is a WME whose type is person and whose age attribute is n+1, where n is specified elsewhere.
if the variable x is not bound, then x will be bound with the value of the attribute height; otherwise, the value of the attribute height in the WME has to be the same as the value of x.

$-$(person age: $\{<20 \wedge >6\}$) is satisfied if there is no WME in WM whose type is person and age value is between 6 and 20 (negation as failure).

A WME matches a condition if the types are identical and for each attribute/specification pair in the condition, there is a corresponding attribute/value pair in the WME and the value matches the specification. The matching WME may have other attributes that are not mentioned in the condition:

condition (control phase: 5)
matching WME (control phase: 5 position: 7 ...)

The consequent set of actions of production rules have a procedural interpretation. All actions are executed in sequence and they can be of the following types:

1. ADD — adds a new WME to the WM.
2. REMOVE i — removes from WM the WME that matches the i-th condition in the antecedent of the rule; not applicable if the condition is negative.
3. MODIFY i (attribute specification) — modifies the WME that matches the i-th condition in the antecedent by replacing its current value for attribute by specification; not applicable if the condition is negative.

<u>Obs</u>. In ADD and MODIFY, the variables that appear refer to the values obtained when matching the antecedent of the rule.

<u>Example 1</u> IF (student name: x) THEN ADD (person name: x)
(the equivalent of $\forall x.\, \mathrm{Student}(x) \supset \mathrm{Person}(x)$)

<u>Example 2</u> - assume that a rule added a WME of type birthday

IF (person age: x name: n) (birthday who: n)
THEN MODIFY 1 (age [x+1])
REMOVE 2
↖ to prevent the rule from firing again

Example 3 - to indicate the phase of a computation

```
IF (starting) THEN REMOVE 1
                   ADD (control phase : 1)

---

IF (control phase : x) ... other conditions
      THEN MODIFY 1 (phase [x+1])

IF (control phase : 5) THEN REMOVE 1
```

Example 4 We have three cubes, each of different size. With a robotic hand, we want to move the cubes in the positions called 1, 2 and 3. The goal is to place the largest cube in position 1, the middle one in 2 and the smallest in position 3.

```
WM  [ 1. (counter value : 1)
    [ 2. (cube name : A  size : 10  position : heap)
    [ 3. (cube name : B  size : 30  position : heap)
    [ 4. (cube name : C  size : 20  position : heap)

      [ 1. IF (cube position : heap  name : n  size : s)
      [        - (cube position : heap  size : {> s})
      [        - (cube position : hand)
Rules [    THEN MODIFY 1 (position hand)           there are no conflicts
      [ 2. IF (cube position : hand)               because only one
      [        (counter value : i)                 rule can fire at a time
      [    THEN MODIFY 1 (position i)
      [         MODIFY 2 (value [i+1])
```

Rule 1 → n = B, s = 30 → WME 3 changes
(cube name : B size : 30 position : hand)

Rule 2 → i = 1 → WME3 changes
(cube name : B size : 30 position : 1)
WME 1 changes (counter value : 2)

```
Rule 1 → n=C, s=20 → WME 4 changes
             (cube name: C  size: 20  position: hand)
Rule 2 → i=2 → WME 4 changes
             (cube name: C  size: 20  position: 2)
           WME 1 changes (counter value: 3)
Rule 1 → n=A, s=10 → WME 2 changes
             (cube name: A  size: 10  position: hand)
Rule 2 → i=3 → WME 2 changes
             (cube name: A  size: 10  position: 3)
           WME 1 changes (counter value: 4)
```

```
   ⎡ 1. (counter value: 4)
   ⎢ 2. (cube name: A  size: 10  position: 3)
WM ⎢ 3. (cube name: B  size: 30  position: 1)
   ⎣ 4. (cube name: C  size: 20  position: 2)
```

No rule is applicable now, so the production system halts.

<u>Example 5</u> - compute how many days are in any given year.

```
if (year % 4) then (normal year)
   else if (year % 100) then (leap year)
      else if (year % 400) then (normal year)
         else (leap year)
```

```
WM [ (wantDays year: 1953)

      ⎡ 1. iF (wantDays year: n)
      ⎢       THEN REMOVE 1
      ⎢            ADD (year mod4: [n%4] mod100: [n%100] mod400: [n%400])
      ⎢ 2. iF (year mod4: {≠0})
Rules ⎢       THEN REMOVE 1
      ⎢            ADD (hasDays days: 365)
      ⎢ 3. iF (year mod4: 0  mod100: {≠0})
      ⎢       THEN REMOVE 1
      ⎢            ADD (hasDays days: 366)
      ⎢
```

```
Rules [ 4. iF (year mod100 : 0  mod400 : {≠0})
            THEN REMOVE 1
                 ADD (hasDays days : 365)
        5. iF (year mod400 : 0)
            THEN REMOVE 1
                 ADD (hasDays days : 366)
```

WM [(hasDays days : 365)

All conditions are disjoint, so one rule fires at a time.

Example 6 - compute the greatest common factor of two integers.

```
[ if a>b then gcf(a,b) = gcf(a-b, b)
          else gcf(a,b) = gcf(a, b-a)
  gcf(0,a) = gcf(a,0) = a
```

WM [(gcf val1 : 6 val2 : 9)

```
Rules [ 1. iF (gcf val1 : 0  val2 : 0)
              THEN REMOVE 1
                   ADD (res val : 0)
        2. iF (gcf val1 : 0  val2 : {≠0})
              (gcf val2 : x)
              THEN REMOVE 1
                   ADD (res val : x)
        3. similar to 2, for val1 : {≠0} and val2 : 0
        4. iF (gcf val1 : {≠0}  val2 : {≠0})
              (gcf val1 : a  val2 : b)
              (gcf val1 : {>b})
                 THEN MODIFY 2 (val1 [a-b])
        5. similar to 4, for a ≤ b.
```

WM [(res val : 3).

All conditions are disjoint, one rule fires at a time.

Solving conflicts

There are conflict resolution strategies to eliminate some applicable rules, if necessary. The most common approaches are:

- order: choose the first applicable rule in order of presentation (this strategy is implemented in PROLOG).
- specificity: choose the applicable rule whose conditions are most specific.

IF (bird) THEN ADD (canFly)

more specific then the first [IF (bird weight: {>100}) THEN ADD (cannotFly)
IF (bird) (penguin) THEN ADD (cannotFly)

another criterion should be applied further to choose between the two rules

- recency - choose the applicable rule that matches the most/least recently created/modified WME.
- refractoriness - reject a rule that has just been applied with the same value of its variables - it prevents going into loops by repeated firing of a rule because of the same WME.

The efficiency of production systems

The rule matching operation consumes up to 90% of the operating time of a production system.

Two key observations led to a very efficient implementation:

- WM modifies very little during an execution cycle;
- many rules share common conditions.

The RETE algorithm (1974) creates a network from the rules antecedents. This network can be created in advance, as the rules do not change during the system operation.

During operation, new or changed WMEs are checked if they satisfy the conditions of a rule. In this way, only a small part of WM is re-matched against the conditions of the rules, reducing drastically the time to calculate the applicable rules.

The advantages of production systems

- modularity – because the rules work independently, they can be added or deleted relatively easily.
- control – a simple control structure.
- transparency – the rules use a terminology easy to understand and the reasoning can be traced and explained in natural language.

In real-life applications, these advantages tend to diminish. Nevertheless, production systems are successfully used in a wide variety of practical problems (e.g. medical diagnosis, checking for eligibility for credits, configuration of systems).

MYCIN – expert system for diagnosis of bacterial infections, developed at Stanford University in '70
- 500 production rules for recognizing 100 causes of infection
- the most significant contribution was the introduction of a level of certainty of evidences and confidence in hypotheses.

XCON – rule-based system for configuring computers, developed at Carnegie-Mellon University in 1978
- 10000 rules to describe hundreds of types of components.
- contributed to a growing commercial interest in rule-based expert systems.