Chapter 3

Knowledge-Based Systems

Among the software systems that have been created in the area of artificial intelligence (AI), some are heavily oriented toward emphasizing sophisticated approaches to their control structures, while a majority of them emphasizes the role of the knowledge related to a specific subject. Successful systems are characteristically "knowledge-rich even if they are methods-poor [Feigenbaum 1977]. The latter category of systems are called knowledge-based systems (KBS) because they are heavily oriented toward specific and narrow knowledge domains. Common knowledge representations used are:

- Predicate logic — For relatively small systems where the data and rules are known with certainty.
- Semantic networks, frames, and object-oriented approaches — For systems where inheritance of attributes among classes is important.
- Rule-based approaches — For systems where the control structure (known as "inference engine") determines the direction of the deduction.

Rules or implications, stated in the form of if-then rules, are one of the most common ways of expressing the domain knowledge, so much so that many people associate knowledge-based systems with rule-based systems. The rules in a KBS are mostly heuristic rules. Heuristic rules are statements that we accept as true based on our real-world experience, and that can be generally applied to solve problems pertaining to a particular domain. Heuristic rules are often referred to as production
rules, and a system that operates using a set of production rules is known as a production system.

Figure 3.1. An overall Classification of AI Programmes

The overall classification of AI programmes is shown in Figure 3.1. Any programme that exhibits performance that may be called "intelligent", come under the category of AI programmes. When procedures for reaching a solution are not well-defined, we introduce "heuristics" which, while not guaranteeing solutions at all times, do find them often enough—and quickly enough—to warrant their use and, as stated earlier, KBSs form a subclass of these, and rule-based systems are a specialisation of KBSs. As shown in Figure 3.1, expert systems may exist in all the three categories.
3. 1. SYMBOLIC LOGIC AND AI

IN DEVELOPING knowledge representations, one of the most useful logical connectives is the implication (IF) operator. The implication operator is used to formulate *conditional statements*, that is, statements of the form

```
if antecedent
    then consequent
```

Symbolic information encoded via (possibly compound) ‘antecedent’ and ‘consequent’ statements, together with the implication connective, form the basis for many AI implementations, including rule-based systems.

The implication or conditional operator is written as $\leftarrow$ or $\rightarrow$ depending upon the direction of the implication. When written as $p \rightarrow q$, the term on the left, that is, the antecedent $p$ implies the term on the right, that is, the consequent $q$. If the implication is TRUE when the term on the left is TRUE, the term on the right must also be TRUE.

3. 1. 1. Logical Properties of Implication

The logical properties of implication are summarised in the accompanying truth table (Table 3.1).

```
<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p → q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
```

*Table 3. 1. Implication Truth Table*
Chapter 3. Knowledge-Based Systems

Implication yields a fundamental basis for rule-based inference systems. If we assume that implication is TRUE, this knowledge may be used to constrain the allowable truth values of the antecedent and consequent. Consider the antecedent \( p \) being TRUE and the value of the implication \( (p \rightarrow q) \) also being TRUE. (This is the first row of Table 3.1) This constrains or forces \( q \) to assume the value TRUE.

It is noteworthy that implication is used in a weak sense, in logic. For example,

1. \( p \) and \( q \) together do not need to make sense.
   
   \[
   \begin{align*}
   & \text{if } 4 + 2 = 3 \\
   & \text{then America is within India.}
   \end{align*}
   
   This corresponds to row 4 of Table 3.1. A FALSE hypothesis (antecedent) implies any conclusion.

2. A simple statement (that is, containing no connectives) could be viewed as an implication with no antecedents.

3. The implication statement, \( p \rightarrow q \) does not signify that \( p \) causes \( q \); there is no cause-effect relationship. From the truth table, the fact that the implication is TRUE signifies that
   
   a.) \( p \) implies \( q \).
   
   b.) \( q \) is TRUE if \( p \) is TRUE.
   
   c.) \( p \) is TRUE only if \( q \) is TRUE.
   
   d.) \( p \) is a sufficient condition for \( q \).
   
   e.) \( q \) is a necessary condition for \( p \).

4. A Karnaugh map for the implication, \( p \rightarrow q \), yields the result,
   
   \( (p \rightarrow q) \equiv (\neg p) \lor q \). The significance of this result is that, given an implication statement is known to have the value TRUE, the fact that the antecedent is also TRUE forces the consequent to be TRUE. This is the logical basis for forward chaining in a rule-based system. The antecedent statement may, however, be itself a compound statement, involving the conjunction of several simple statements.

3. 1. 2. Inferencing

   Inferencing is the process of deducing (new) facts from (other) existing facts.

The process of reaching a conclusion from a set of propositions is called deductive reasoning. One deduction strategy is known as modus ponens, expressed as a
compound statement involving implication: \((p \land (p \rightarrow q)) \rightarrow q\) Modus ponens provides a rigorous and justifiable mechanism for making inferences.

Another mathematically tractable process is known as **abduction**. Abduction is a mathematically justifiable, practical, and reasonable way of generating hypotheses. It is therefore closely related to the analysis of backward chaining and implication.

\[
\begin{array}{c|c|c}
   p \rightarrow q & \text{a rule; assumed TRUE} \\
   q & \text{an observation; assumed TRUE} \\
   p & \text{an explanation; possibly but not necessarily TRUE} \\
\end{array}
\]

**Example:**

Rule: If traffic density is high, the road is concrete. (3.1)
Observation: Crescent Lane is concrete. (3.2)
Possible explanation: The traffic density on Crescent Lane must be high (3.3)

(Abduction): 
Here, the inference of high traffic density is only a *possible explanation.* 3.1 only explains 3.2: it is not necessarily true.

A more rigorous logical formulation of explanation is provided by **modus tollens**, defined as \(((p \rightarrow q) \land \neg q) \rightarrow \neg p\) (Fourth row of Table 3.1). Modus tollens provides a logically sound explanation which is necessarily TRUE.

When two facts are always concomitant, by induction we can affirm the rule that expresses the relationship between them. Knowing that when \(p\) is TRUE, \(r\) is also TRUE, we can derive (induce) a rule, expressed as \((p, r) \rightarrow (p \rightarrow r)\).
Reasoning by transitivity is as follows: \((p \rightarrow q, q \rightarrow r) \rightarrow (p \rightarrow r)\).

While rule-based systems are based on the four types of reasoning, viz., modus ponens, modus tollens, induction, and transitivity, the more commonly used are sets of deductions, also called forward chaining, or sets of abductions, also called backward chaining, by transitivity. Forward chaining is the way to test the consequences of a set of starting assertions (facts).

Consider a series of assertions or if-then statements. Variables \(p_i\) and \(q_i\) are used to represent the truth values of the antecedents and consequents, respectively, in the following implications.

**Implication 1**
- \(p_1\) A township is planned near lake L.
- \(q_1\) There are no houses near lake L, at present.

**Implication 2**
- \(p_2\) There are no houses near lake L, at present.
- \(q_2\) Lake L is clean.

**Fact**
- \(p_3\) Townships near lakes pollute the lakes.

**Conjecture**
- \(q_3\) Lake L will be polluted when houses are built near it. \((3-4)\)

The consequents of some implications in \((3-4)\) are the same as the antecedents of others. The set of following assertions, each of which we assume to be TRUE, form a rule base, or knowledge base:

\[
\begin{align*}
p_1 & \rightarrow q_1 \\
p_2 & \rightarrow q_2 \\
p_2 & \rightarrow q_1 \\
p_3 & \\
\end{align*}
\]

\((3-5)\)

This knowledge base may be rewritten using variable substitution and simplifies to:

\[
\begin{align*}
p_1 & \rightarrow q_1 \\
q_1 & \rightarrow q_2 \\
p_3 & \\
\end{align*}
\]

\((3-6)\)
While it is assumed that the assertions in (3-6) are all true, the same cannot be said about the components of those statements. To determine the truth of statement \( q_3 \), given that \( p_1 \) is TRUE, forward chaining can be used to "chain" production of new TRUE facts from a starting one, and arrive at a truth value for \( q_3 \). The logical sequence is as follows:

\[
\begin{align*}
(p_1 \rightarrow q_1) \text{ and } (p_1 \text{ being TRUE}) & \rightarrow (q_1 \text{ is TRUE}) \\
(q_1 \text{ is TRUE}) \text{ and } ((q_1 \rightarrow p_2) \text{ being TRUE}) & \rightarrow (p_2 \text{ is TRUE}) \\
(p_2 \rightarrow q_2) \text{ and } (p_2 \text{ being TRUE}) & \rightarrow (q_2 \text{ is TRUE}) \\
(q_2 \text{ is TRUE}) \text{ and } (p_3 \text{ being TRUE}) & \rightarrow (q_3 \text{ is TRUE})
\end{align*}
\]

The sequence of productions employed in deducing "the lake L will be polluted when houses are built near it" from "A township is planned near the lake" is shown in Figure 3.2. This type of diagram is referred to as an inference net. Inference nets are seldom as simple as this; there could be hundreds of rules (instead of five in the present case) in a typical rule-base. The action indicated by following an arrow from the antecedent of a rule to its consequent (and thus producing a statement whose truth value is known to be TRUE) is shown in the Figure. This is referred to as firing the rule. The process of deciding which rule to fire is fundamental to the inference control strategy.

\[\text{Figure 3.2. Inference Net for Three-Rule Deduction Example.} \quad (\text{Given } p_1 \text{ is TRUE})\]
3. 1. 3. Inference Rules

The semantics of predicate calculus provide a basis for formal theory of logical inference. The ability to infer new correct expressions from a set of true assertions is an important feature of the predicate calculus. These new expressions are correct in that they are consistent with all previous interpretations of the original set of expressions.

Modus ponens is a sound inference rule. Given an expression of the form \( p \rightarrow q \) and another expression of the form \( p \) such that both are TRUE under an interpretation \( I \), then modus ponens allows us to infer that \( q \) is also TRUE for that interpretation, and \( q \) will be TRUE for all interpretations for which \( p \) and \( p \rightarrow q \) are TRUE.

3. 1. 3. A. Resolution

The resolution approach yields new clauses from an initial set. Resolution is a powerful and indirect method of inference. The utility of resolution in inference application is subtle. Often, resolution is used to show that a set of clauses is inconsistent, in the sense that the resolution process produces a logical contradiction. A clause is proven to be true, in the context of a set of clauses known to be true, by appending the logical NOT of this clause to the set and seeking a contradiction. If the resolution process fails to find a contradiction, the negative of what we seek to prove is logically consistent with the database, and thus the clause cannot be TRUE.
Resolution proceeds by augmenting the database with the negation of the desired hypothesis. Then clauses are resolved pairwise in the augmented database until a contradiction is found. If none is generated, we conclude that the situation is consistent, and therefore the hypothesis is FALSE. First, all implications and equalities are to be removed from the well-formed formulae.

\[(p \rightarrow q) \text{ is replaced with } ((\neg p) \cup q)\]
\[(p \equiv q) \text{ is replaced with } ((\neg p) \cup q) \cap (p \cup (\neg q))\]

Pairwise resolution of clauses uses the strategy of verifying that
\[(a \cup b) \cap (\neg a \cup c) \rightarrow (b \cup c)\]

Example: A database has its initial condition (D1) is consistent and its contents are given below. All statements are assumed to be TRUE.

1. \(p_1\)
2. \(p_1 \rightarrow q_1\)
3. \(q_1 \rightarrow q_2\)

The goal is to prove, using D1 that \(q_2\) is TRUE. Therefore, \(\neg q_2\) is added to D1. Clauses are simplified to a suitable form, and pairwise resolution proceeds. The modified database D2 is:

1. \(p_1\)
2. \(\neg p_1 \cup q_1\)
3. \(\neg q_1 \cup q_2\)
4. \(\neg q_2\)

Resolution of (2) with (3) yields \((\neg p_1 \cup q_2)\) which then becomes the 5th element of the database (D3). Resolution of (4) with (5) yields \(\neg p_1\), and the new database (D4) has the following configuration

1. \(p_1\)
2. \(\neg p_1 \cup q_1\)
3. \(\neg q_1 \cup q_2\)
4. \(\neg q_2\)
5. \(\neg p_1 \cup q_2\)
6. \(\neg p_1\)

Resolution of (1) with (6) produces a contradiction. Therefore, \(\neg q_2\) is inconsistent with D1. Hence, by refutation, \(q_2\) is TRUE.

More details on Resolution can be found in Section 11.2 of [Luger & Stubblefield 1989].
3. 1. 3. B. Unification

In order to apply inference rules such as modus ponens, an inference system must be able to determine when two expressions are the same or match. In propositional calculus, this is trivial: two expressions match if and only if they are syntactically identical. In predicate calculus, the process of matching two sentences is complicated by the existence of variables in the expressions. Unification is a systematic procedure for instantiation of variables in wffs. Since the truth value of predicates is a function of the values assumed by their arguments, the controlled instantiation of values provides a means of validating the truth values of compound statements containing predicates. The basis of unification is substitution.

**Unification** is the process of making two expressions identical (that is, unify them) by finding appropriate substitutions or bindings of variables in these expressions.

A substitution is a set of assignments of terms to variables, with no variable being assigned more than one term.

A set of expressions is unifiable if and only if there exists one or more (unifying) substitutions that make the expressions identical.

In a simplified manner, substitution can be illustrated as follows: If a has a property $P$ and all objects that have property $P$ also have property $Q$, we conclude that $a$ has property $Q$ [Schalkoff 1990].

\[
\begin{align*}
P(a) \\
\forall x P(x) \rightarrow Q(x) \\
\therefore Q(a)
\end{align*}
\]

In concluding $Q(a)$, $x$ was substituted with $a$. This was made possible by the implication $P(x) \rightarrow Q(x)$ that was TRUE for all $x$, and in particular for $x = a$. 
Substitutions permit simplifications or reduction of expressions through the cancellation of complementary literals.

One of the most important and logically sound inferencing techniques involves using modus ponens with statements (especially rules) containing variables. A single substitution may be represented by a pair of the form \((t, R)\), denoting a substitution of the variable \(R\) with the constant value \(t\), in a statement. Application of a set of substitutions \(\theta = (t_i, R_i), i = 1, 2, \ldots, n\), to a statement \(P\) as \(P\theta\), where every occurrence of variable \(R_i\) in \(P\) is replaced by \(t_i\). Two statements, \(P\) and \(P_1\) are unifiable, that is, they have a unifying substitution, \(\theta\), if \(P\theta = P_1\theta\). In other words, a set of substitutions, \(\theta\), exists that makes \(P\) and \(P_1\) identical.

If statement \(P_1\) contains no variables, then \(P_1 = P\theta\) represents a unifying substitution. For example, the two statements

\[
P_1 = (\text{block}_1 \ \text{left_of} \ \text{block}_2)
P = (A \ \text{left_of} \ B)
\]

wherein \(A\) and \(B\) are variables, unify with

\[
\theta = \{(\text{block}_1 / A), (\text{block}_2 / B)\}
\]

Assume the following axioms in the initial database:

\[
\begin{align*}
(A \ \text{left_of} \ B) & \rightarrow (B \ \text{right_of} \ A) \quad (3-8) \\
(\text{block}_1 \ \text{left_of} \ \text{block}_2) & \quad (3-9)
\end{align*}
\]

The following inference procedure is logically valid:

1. Unify antecedent of implication statement (3-8) with fact (3-9).

These unify with \(\theta = \{(\text{block}_1 / A), (\text{block}_2 / B)\}\).

Therefore, (3-8) and modus ponens allow deduction of the new fact

\[
(B \ \text{left_of} \ A) \ \theta
\]

and, with substitution

\[
(\text{block}_2 \ \text{right_of} \ \text{block}_1) \quad (3-10)
\]
3. 2. PRODUCTION SYSTEMS

PRODUCTION systems represent symbolically tractable, solutions to many AI applications. A production system consists of

1. An unordered set of production rules (Production Memory) that modify the existing database of facts and/or production memory and whose applicability is conditioned on the current database of facts (or production memory).
2. A database of facts (Data Memory), maintained separately from the production memory.
3. A control mechanism or a rule interpreter, also known as the Inference Engine, that determines the applicability of the rules, the selection of appropriate rules, and the resolution of conflicts that may arise when more than one production become applicable at the same time.

![Diagram of a Rule-Based Production System](image)

**Figure 3.3. The Structure of a Rule-Based Production System**

Production systems are a subset of **pattern directed systems**—systems whose production applications are driven by input (or initial) data patterns. Specifying the production conditions in the form of if statements and actions via then, leads to a rule-based system paradigm. The term **expert system**, is used to indicate a subset of production systems that are restricted to specific task domains. Figure 3.3 shows the structure of a rule-based production system.
3.2.1. The Inference Engine (IE)

A rule in the Rule Base is said to be triggered when all of the antecedents of the implication are satisfied — that is, when these antecedents are present (or asserted) in memory. The inference engine can be treated as a finite state machine (FSM) with a cycle consisting of the action states:

1. Match rules.
2. Select rules.
3. Execute rules.
4. Check stopping condition (goal satisfaction).

The IE somewhat exercises control over the production system operation in the sense that it either regulates the production of new databases (forward chaining) or the verification of hypothetical information (backward chaining). Fundamental to the operation of the IE is the process of matching. Since more knowledge may be encoded with rules using variables, this involves matching with variables and thus, a suitable unification algorithm.

3.2.2. Rule Selection

Another critical aspect of IE operation is in the selection of rules. One type of specialised production system is the commutative system. A rule is defined to be “applicable in the context of a database, denoted by $\mathcal{D}$”, if the conjunction of the antecedents for that rule are satisfied by $\mathcal{D}$, that is, the rule is eligible for firing. A commutative production system has three important properties [Nilsson 1980]:

1. Any rule applicable to $\mathcal{D}$ is also applicable to any database derived from $\mathcal{D}$ by successive applications of applicable rules.
2. If a goal is satisfied by $\mathcal{D}$, then it is also satisfied by any database produced by applying applicable rules to $\mathcal{D}$.
3. The database generated by firing of a sequence of rules applicable to $\mathcal{P}$ is invariant to permutations in the firing sequence. (This does not imply that if we find a sequence of rules that change the initial state of the system to the goal state, then we can arbitrarily reorder this sequence and still find that the reordered sequence will achieve the goal.)

A production system having the commutative property enables the IE to avoid many of the solution paths that differ only in the order in which initially applicable rules are applied. For large rulebases, this may be a significant number. This allows the IE to select and apply applicable rules without provision for reconsideration at a later point in the inference sequence. This irrevocable control strategy is in contrast with the tentative control strategy which selects and applies rules, but has a provision for reconsideration at a later point in the inference process. A control strategy that allows backtracking, PROLOG's inference mechanism, for example, adopts the tentative strategy. A practical or generally useful control strategy is not likely to be a simple matter of choice.

The best example of a non-commutative production system is a planning system. In planning systems, the order in which productions are invoked is usually critical to the outcome of the system. Achievements of a plan that meets the prescribed goal is highly dependent upon the chosen sequence of operations that make up the plan.

3.2.3. Conflict Resolution

Conflict resolution is a selection process for determining “good” rules, from among all applicable rules. There are several approaches to conflict resolution. Some of them are the following:
1. In choosing "rule a" versus "rule b", examine the antecedents of both rules. If the antecedents of "rule a" are a superset of those of "rule b" then the former is more specialised than the latter, because more constraints are applicable to the former. "rule a" As per the strategy, known as specificity-ordered conflict resolution, the rule with the stricter precondition is chosen.

2. Examine the conflict set, and choose a rule whose firing takes the system closer to the goal, as compared to other rules. This implies that the goal has already been specified.

3. Choose the rule that has the largest number of consequents, assuming that more the additional information available, closer the system would be to the goal.

4. Rank rules according to an a priori firing desirability.

If rules contain variables, it is possible for a rule to be used repeatedly. This makes room for possible additional conflict resolution strategies, as listed below:

5. Choose the most recently used rule.
6. Choose the least recently used rule.
7. Choose the rule with the least (or most) number of variables.

### 3. 2. 4. Inference Strategies

When a rule is said to have fired, as a result of the conflict resolution process, the rule is removed from consideration, and the cycle of considering all currently triggered rules, including any triggered by the result of the rule just fired, continues.

This approach, called refraction, is to avoid looping on the same rule. Rules removed after being fired are re-incorporated into the production memory when conditions relating to their antecedents have changed. This iterative process is referred to as recognize-act-cycle, and is represented graphically in Figure 3.4.

![Figure 3.4. The Recognize-Act Cycle](image-url)
The computational difference between the two inferencing strategies, namely, the forward chaining approach and the backward chaining approach, merits brief discussion. In the forward chaining approach, the inference process proceeds exhaustively from the starting set of facts to a set of new facts. Thus, all new facts are generated, unless a control or stopping mechanism is embedded inside the process. A consequence of this is the checking and firing of a substantial number of rules; some of these rule firings may contribute very little to advance the system towards a goal.

The backward chaining paradigm, on the other hand, seeks only to prove the validity of a chosen fact or expression whose truth value is not known a priori. Computationally, it is perhaps more efficient than forward chaining, since it represents a goal-directed strategy that may eliminate checking of many superfluous paths. If more than one hypothesis is involved, backward chaining attempts to verify each one independently.

Forward chaining is appropriate when

There exist many equally acceptable goal states, a narrow body of relevant information, by way of facts and rules, and a single initial state.
All or most of the required facts are in the initial database.
It is difficult, initially, to form a goal or hypothesis to be verified.

Backward chaining is appropriate when

There exists a single goal state (like in diagnosis, where we desire to confirm the occurrence of a single event or malady) and a large amount of potentially relevant initial information [Brownston et al. 1985].
Relevant data must be acquired as a part of the inference process, for example, "asking questions".
Large numbers of applicable rules exist. This potentially leads to the production of many extraneous facts.
3. 2. 5. The OPS5 Production System

An important family of AI languages comes directly out of the production system language research at Carnegie Mellon University (CMU), USA [Brownston et al. 1985]. These are the OPS languages; OPS stands for Official Production System. These languages have proved highly effective for programming and designing production systems, expert systems, and other AI applications. OPS5 is the most widely used interpreter, released by CMU in 1981.

From a programming viewpoint, an OPS5 implementation consists of a set of suitably coded facts in working memory (WM) and a set of rules in production memory. These are shown in Figure 3.4.

A production is designated by a list whose car is the symbol p. The nth element of the list is the rule name find-segment (as shown below), and the antecedents are a series of lists ending with the symbol →. In the simplest form, the LHS of a rule is a set of condition elements that specify patterns to be matched against facts in working memory. The RHS of the rule (indicated after the arrow, as shown below) is a sequence of actions. A simple rule, for example, for finding in WM a line segment whose endpoints are the values of the variables (<x0>, <y0>, <x1>, <y1>) in OPS5 is
Further details on OPS5 programme structure is given in Appendix B.

It was stated in Section 3.2.1 (Page 97) that the inference engine of a production system can be treated as a FSM with a cycle consisting of three action states, namely, Match, Select, and Execute. In the first state Match, the machine finds those rules which are triggered. The triggered rules are all potential candidates for execution, and they are collectively referred to as the conflict set. The conflict set is passed along to the second state Select, which applies the selection strategy (determined by the specific production system strategy) to determine which rules are suitable for execution.

3. 2. 5. A. The Rete Match Algorithm

In the OPS5 interpreter, the recognize-act cycle (vide Page 99) has been changed to

1. Conflict resolution. Input is the conflict set. Select one production with a satisfied LHS. If there are no productions with satisfying LHS, return control to the user.
2. Execute. Perform the actions specified on the RHS of the selected production.
3. Match. Evaluate the LHSs of the productions to determine which are satisfied given the current contents of working-memory. Output is the conflict set.

If Halt, return control to the user or go to step 1 [Digital 1988].
The Rete Match [Forgy 1982] is an algorithm for computing the conflict set. That is, it is an algorithm to compare a set of LHSs to a set of elements to discover all the instantiations. The cycle is more convenient to the user because when it ends, the conflict set is consistent with the current contents of WM. The algorithm exploits two properties common to all production systems in reducing the effort of performing the match. First, it exploits the fact that only a small fraction of working-memory changes in each cycle. Therefore, information about the LHS matches and partial matches in one cycle is saved and used in the next cycle, updating the information as necessary to reflect the changes made to working-memory on each cycle. Thus the amount of effort expended by the matcher depends primarily on the rate of change of working-memory rather than the absolute size of working-memory.

Second, it exploits the commonality between condition-elements of productions, to reduce the number of tests performed. Since the rules have to work together, they must be able to access the same working-memory elements. Hence their LHS must contain many similar conditions and terms if their patterns are to contend for access. Exploiting this requirement, Rete processes the patterns before the system is interpreted. The productions are processed by a rule compiler that locates these common terms and eliminates as many of them as possible. This allows many operations for the entire rule set to be done just once rather than doing them over and over again. These patterns are compiled into a special data-flow graph called a Rete\textsuperscript{1} graph. The nodes in the graph designate computations to be performed by the matcher. The edges in the graph indicate the way the data is to flow from node to node through the graph. Redundant computations are eliminated by constructing only

\textsuperscript{1} Rete means 'net'
a single node to perform a given a computation, linking the node to all the places where its result is needed.

3.2.6. Major Advantages of Production Systems

The production system offers a general framework for implementing search. The major advantages of production systems include:

Separation of Knowledge and Control. The production system is an elegant model of separation of knowledge and control in a computer programme. Control is provided by the recognize-act cycle of the production system loop, and the knowledge is encoded in the rules themselves. This renders the knowledge base easier to modify without disturbing the program control code and, conversely, the program code can be modified without affecting the rule set.

A Natural Mapping onto State Space Search. The successive states of working memory form the nodes of a state space graph. The production rules represent the set of possible transitions between states, while conflict resolution managing the selection of a branch in the state space. These factors simplify the implementation, debugging, and documentation of search algorithms.

Modularity of Production Rules. There is lack of any syntactic interactions among production rules. rules may only effect the firing of other rules by changing the pattern in working memory. A rule does not invoke another one directly as if the latter were a subprogramme; nor a rule can set the value of variables in other rules. The scope of the variables of these rules is confined to the individual rules. This syntactic independence supports incremental development of applications by successively adding, deleting, or modifying the knowledge (rules) base.

Tracing Facility and Explanation. The modularity of rules and the iterative nature of their execution make it easier to trace execution of a production system, by forcing each stage of the recognize-act cycle to display the selected rule. Moreover, since each rule represent a fragment of the problem-solving knowledge, each rule becomes self-explanatory regarding its part in the system as a whole.

3.2.7. Production Systems for GIS Design

Based on investigations to-date, two distinct classes of GIS have become perceptible. One class uses map-based data, especially in vector-format, and finds applications in engineering, town planning, decision support system, boundary analysis,
and thematic representations. The second class uses image-based data, especially in raster format, and finds applications in image analysis and remote sensing.

The work reported in this thesis was an attempt towards designing a GIS of the first category, that is, a GIS based on data extracted from an authoritative map. As already stated, the characteristic feature of geographic data is that they are spatially indexed. Two basic alternatives are available in the construction of a data model that incorporates spatial addressing:

- Objects may be represented with each object having spatial location as an essential property.
- Locations may be represented with each location being characterised by a set of object-properties.

These alternatives complement each other, and have resulted in the vector and tessellation models, respectively [Smith et al. 1987].

The spatial data model adopted for the present work is the vector model. The basic logical unit in a vector model is the line, used to encode the locational description of an object, and represented as a string of coordinates of points along the line. Closed areas, modelled as polygons, are represented by the set of lines that constitute their boundaries.

The vector model of geographic space can be classified as unlinked or topological. In the unlinked model, also termed the spaghetti model, each map entity is encoded separately in vector form without referencing any of its neighbouring entities. Spatial relationships are not encoded, rendering spatial analysis of data cumbersome. This approach, nevertheless, is adequate for routine analysis of
geographic data, involving the distribution of objects and their display [Smith et al. 1987].

To recapitulate, the present work is aimed at

1. Designing a GIS based on data extracted from a planar map.
2. Representing each identifiable geographic primitive as a distinct object with its spatial properties defined.
3. Formulating composite spatial objects made up from primitives, based on rules defining such compositions.

The facts and rules are then organised into a production system, OPS5 to be specific, that can be used to produce thematic, or other mapping. It should also be capable of answering queries about locations of some class of spatial objects within a given spatial window, or about the identities of objects found within a given spatial window, or about the measurements such as, point-to-point distance, or perimeters, et cetera.

Even though GIS problems are well-defined, data in a GIS application are ill-structured. The use of production systems, in which control is exercised by choosing the most applicable rule in a knowledge base of rules, is a natural method for modelling such diverse tasks as the search of complex data bases of maps, reasoning about the data, and producing outputs in the desired format. The adopted scheme derives all the advantages of OPS5, pointed out in Section 3.2.6, like ease of modification, clarity of design (since the properties of objects are encoded directly into fact / rule), self-documenting code, et cetera. The benchmark studies conducted on several expert system tools, indicate that OPS5 is the fastest [Gevarter 1987], [Mettrey 1991].
3. 3. SUMMARY

THE COMPLEXITY of spatial objects suggests the applicability of AI techniques for representing the spatial objects, for developing procedures for reasoning about spatial objects, and for answering various queries about the objects. There are several ways of solving problems and representing knowledge in AI applications. Among them, production systems render the representation of knowledge systematic and modular. The implementation of control mechanisms is much more easier than in a procedural method.

The chapter has discussed the logical foundations of production systems in general, with particular reference to OPS5. Even though GIS problems are well-defined, data in a GIS application are ill-structured. The use of production systems, in which control is exercised by choosing the most applicable rule in a knowledge base of rules, is a natural method for modelling such diverse tasks as the search of complex data bases of maps, reasoning about the data, and producing outputs in the desired format.