LFA+: A Fast Chaining Algorithm for Rule-Based Systems

Xindong Wu and Guang Fang
Department of Software Development, Monash University
900 Dandenong Road, Melbourne, VIC 3145, Australia
AND
Matjaž Gams
Jozef Stefan Institute
Jamova 39, 1000 Ljubljana, Slovenia
xindong@insect.sd.monash.edu.au; matjaz.gams@ijs.si

Keywords: expert systems, rule-based systems, fast chaining, conflict resolution

Edited by: Rudi Murn
Received: May 20, 1997 **Revised:** April 8, 1998 **Accepted:** July 14, 1998

A significant weakness of rule-based production systems is large computational requirement for performing matching. Time complexity of algorithms is generally still NP-hard (non-polynomial) to the number of rules in a rule base. LFA is a linear-chaining algorithm for rule-based systems which does not require a specific conflict resolution step for chaining. However, its applications are still restricted, e.g., it cannot process first-order rules efficiently.
This paper reviews the design of chaining algorithms for rule-based systems, and analyses some well-known chaining algorithms such as RETE and LFA. The central contribution is the design of a robust LFA algorithm, LFA+, which can processes first-order logic rules.

1 Introduction

1.1 Rule-based systems

Rule-based systems (RBSs) are an important type of pattern-directed inference systems. They consist of three basic components as follows:

1. A set of rules, which can be activated or fired by patterns in data.
2. One or more data structures (data bases), which can be examined and modified.
3. An interpreter or inference engine that controls selection and activation of the rules.

A rule includes a left-hand side, LHS, which is responsible for examining items in the data structures, and a right-hand side, RHS, which is responsible for modifying data structures. Data examination consists of comparing patterns associated with the LHSs with elements in the data structures. The patterns may be defined in many ways, such as simple strings, complex graphs, semantic networks, tree structures, or even arbitrary segments of code which are capable of inspecting data elements. Data modification can involve firing actions to modify data, rules, or even the environment. Information in the data can be in the form of lists, trees, nets, rules, or any other useful representation.

The organisation of rule-based systems is modular, and the characteristics of them are as follows [Waterman & Hayes-Roth 78]:

- RBS modules[1] separate permanent knowledge (rules in the rule base) from temporary knowledge (data in the working memory).
- RBS modules are structurally independent. They facilitate incremental expansion of the system and massive code understanding (Modules can be dealt with one by one).
- RBS modules facilitate functional independence. It is generally useful to distribute different functions to different modules.
- RBS modules may be processed by using a variety of control schemes, i.e. different modules may have different control structures.
- RBSs separate data examination from data modification because of the separation of LHSs and RHSs of rules.
- RBSs use rules with a high degree of structure, and are a natural knowledge representation (the natural "IF ··· THEN ···" structure).

In the light of problem-solving methods, rule-based systems can be divided into two classes, namely forward-chaining systems and backward-chaining systems. Forward-chaining systems are antecedent-driven, while backward-chaining systems

[1] A RBS module is a bundle of mechanisms for examining and modifying one or more data structures.

are consequent-driven. Forward-chaining systems are commonly known as rule-based systems.

Rule-based systems are a well-known type of system in which the control structure can be mapped into a relatively simple recognise-act paradigm. A typical interpreter of a rule-based system performs the following operations in each 'recognise-act' cycle:

1. Match

   Find out the rule set in the rule base whose LHSs are satisfied by the existing contents of the working memory.

2. Conflict resolution

   Select one rule with a satisfied LHS; if no rule has a satisfied LHS then stop.

3. Act

   Perform actions in the RHS of the selected rule and go to step 1.

By using suitable interpretations of each of the above actions, the operation of a chaining-based inference engine can be readily described as iteration of such actions. A forward-chaining engine regulates the rules of new databases, while a backward-chaining engine controls the verification of hypothetical information. Another view of the inference engine is that it generates one or more inference nets linking the initial system state to a goal state [Schalkoff 90].

The fundamental operation of the inference engine is the process of matching. Partial matches or complete matches often involve matching with variables for which a suitable unification algorithm which ensures that variable bindings are consistent is necessary. This procedure may require many tests and comparisons. So it is usually difficult to design a fast-chaining algorithm for a large rule-based system.

There are three basic approaches to the problem of conflict resolution in a rule-based system as follows [Rich & Knight 91]:

- Assign preference based on matched rules in the rule base.
- Assign preference based on matched objects in the working memory.
- Assign preference based on actions that matched rules would perform.

1.2 Problems with the 'Recognize-Act' Paradigm

For naive rule-based systems, all but the smallest systems are computationally intractable because of the complexity of matching in the 3-phase cycles. The successful match of a rule in the rule base with the working memory does not always mean that the rule will be fired. A rule may fail to match with the working memory in an overall problem-solving process, but it probably needs to be tested in each 3-phase cycle when the working memory is changed. Meanwhile, some other rule may be successful in matching with the working memory from the very beginning of a problem-solving process, but may fail to receive enough priority to fire in each conflict resolution phase. When there are changes in the working memory, the rule needs to be tested again and again. It has been observed that some systems spend more than nine-tenths of their total run time performing pattern matching in large rule-based systems [Forgy 82]. As a result of these problems, efficiency is a major issue in large rule-based systems.

Since rule-based systems may be expected to exhibit a high standard performance in interactive domains or in real-time domains, many researchers have worked towards improving the efficiency of such systems. As yet, the most significant results have been the RETE algorithm (See Section 4.4) and other RETE-like algorithms such as TREAT (See Section 4.5). These algorithms are match algorithms which avoid matching all rules with the working memory in order to find appropriate rules on each 3-phase cycle so that efficiency can be improved. However, the following two problems still exist in all known rule-based systems except KEshell [Wu 93a]:

1. All complete chaining algorithms are exponential in time complexity. Non-worst-case sub-exponential algorithms are not possible for general cases.
2. Chaining in rule-based systems is a much more complicated process than testing the satisfiability of individual propositional formulae. It is not possible to know in advance precisely how many 3-phase "match — conflict resolution — act" cycles are needed for each problem solving task.

In KEshell, a new algorithm called LFA (See Section 4.6) has been designed. LFA is a linear forward-chaining algorithm for rule-based systems. The most significant advantages of LFA are that its time complexity is $O(n)$ where n is the number of rules in the rule base, and that it does not need an independent conflict resolution step. By using a two-level "rule schema + rule body" structure (See Section 3.2.2), knowledge representation in KEshell can explicitly express numeric computation and inexact calculus in the same way as inference rules in rule bodies. As long as knowledge representation has an applicable extension, and processing measures show further improvements, LFA should achieve a wider range of applications. However, its knowledge representation cannot represent first-order logic rules efficiently. This is a significant restriction for applications.

The research objective of this paper is to relax the above mentioned limitation of LFA so that it can effi-

ciently process first-order logic rules. We will present the design of a robust LFA algorithm, LFA+, based on LFA [Wu 93a], which has the following components:

- Extended knowledge representation for first-order logic rules, which includes specific representations of recursive rules and rules with negative condition elements.
- Sorting measures, for ordering the knowledge in a knowledge base.
- Linear forward chaining.

The paper is organised as follows. Section 2 introduces expert system principles and some concepts of the first-order logic language, and explains one definition for describing the LFA+ algorithm. In Section 3, knowledge representation issues are addressed, and two languages for rule-based systems — OPS5 and *rule schema + rule body* are described and compared. Section 4 discusses algorithm design issues and techniques, and analyses the RETE, TREAT and LFA algorithms. In Section 5, knowledge representation measures, sorting strategies, the chaining procedure and analyses of the LFA+ algorithm are presented in detail. Finally, Section 6 outlines conclusions and future research. Definitions are listed in teh Appendix.

2 Background in Expert Systems and First Order Logic

2.1 Expert systems

An "expert system" is a computer program which uses knowledge and inference procedures to solve problems that are difficult enough to require human expertise for their solutions [Raeth 90]. Expert system technology aims at improving qualitative factors and can provide expert-level performance to complex problems. A typical expert system consists mainly of the following parts:

- A working memory/data base, which stores the evidence and intermediate results of problems during the chaining process.
- A knowledge base (KB) or knowledge source.
- An inference/chaining engine for solving users' problems by applying the knowledge encoded in the knowledge base.
- An explanation engine or tracing engine for telling the users how the solutions were obtained.
- A knowledge acquisition engine for acquiring knowledge or modifying the knowledge base when necessary.
- A knowledge base management subsystem that detects inconsistencies in the KB.

Their relationships are shown in Figure 1.

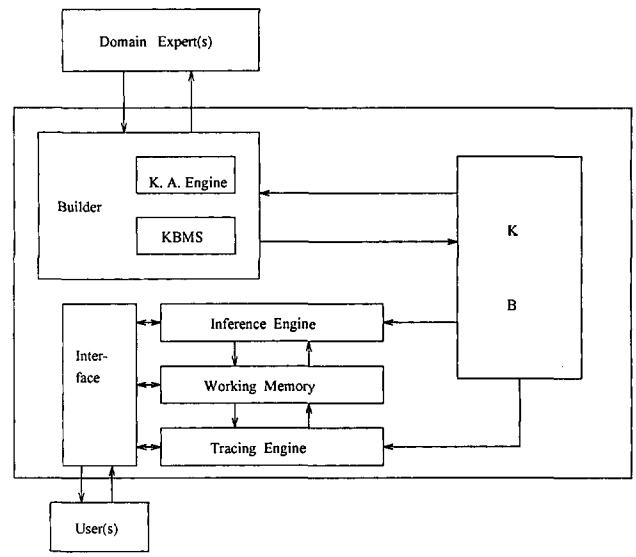

Figure 1: An expert system strucrure

Conventional software programs are designed to control computers algorithmically and tell the computers exactly what to do in problem-solving. These programs are usually procedural. Once a program system has been encoded, it is difficult to change the system design. On the other hand, expert systems excel at encoding knowledge declaratively, and they can be modified flexibly because of the separations of knowledge from expert system shells and knowledge from data, and their modular structures. Furthermore, expert systems have the following features [Pedersen 89]:

- They use symbols to encode the world which can be used in varied ways.
- Most expert systems support uncertainty representation.
- Expert systems can handle unknown cases of a problem by applying the knowledge in the knowledge base.
- They can explain their reasoning.
- They can make multiple conclusions.
- They can tailor conclusions.

2.2 Language of first-order logic

A first-order language is identified by a triple <V,F,P>:

- V is a set of variables.

- F is a set of functors, each of which has an arity.
- P is a set of predicate symbols, each of which has an arity.

The *terms* (See the definition below) of the language are built from variables and functions (Constants are viewed as functors of arity 0), while *predicates* are built from terms and predicate symbols (Propositions are viewed as predicate symbols of arity 0).

Definition: A *term* is defined inductively as follows:

- A variable is a term.
- A constant is a term.
- If f is an n-ary function symbol and $t_1, \cdots, t_n$ are terms, then f(t_1,$\cdots$,t_n) is a term.

Interpretation

Truth value interpretation of a first-order logic language is a triple <D,F,R>:

- D is the domain.
- F is a mapping from functions of domain elements to the domain.
- R is a mapping from predicates of domain elements to truth values.

Horn clauses are a subset of first-order logic languages, but the subset is powerful enough to encode Turing machines. A Horn clause has the following form:

p(t) :- $q_1(t_1)$, $q_2(t_2)$, $\cdots$, $q_n(t_n)$.

where p and q_1, q_2, $\cdots$, q_n are predicate letters, n $\geq$ 0, and all variables which occur in the terms t, t_1, t_2, $\cdots$, t_n are universally quantified at the front of the clause (implicitly). If n is 0 then the clause is referred to as a *fact*, otherwise, it is called a *rule*.

The atom p(t) is referred to as the head of the clause, and $q_1(t_1)$, $q_2(t_2)$, $\cdots$, $q_n(t_n)$ as the body of the clause. The terms t, t_1, t_2, $\cdots$, t_n may be arbitrary terms, and hence may contain variables and/or functions.

A logic program is a set of Horn clauses. However, it is often useful to consider sub-classes of this class of programs in rule-based systems. One type of these programs is a Datalog program, in which terms are only allowed to be either variables or constants.

2.3 Unification and Match

Unification

Unification is the basis of the uses of logical inference in artificial intelligence. It is a method of finding such variable bindings for two predicates or terms that they can be identical [Sterling et al. 86].

Match

Two terms *match* if [Bratko 90]:

- they are identical, or
- the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.

The following is an extended definition, *partial match*, for describing LFA+ (See Section 5.2).

Definition — Partial match

Given a premise factor, *p-factor*, of one rule schema and a conclusion factor, *c-factor*, of another rule schema, partial match of *p-factor* with *c-factor*, written as partial-match(p-factor,c-factor), has the following meanings:

- If *p-factor* is a variable then *c-factor* is the same variable.
- If *p-factor* is a proposition *p* or *not(p)* then *c-factor* is *p* or *not(p)*.
- If *p-factor* is a predicate $p(\cdots)$ or $not(p(\cdots))$ then *c-factor* is $p(\cdots)$ or $not(p(\cdots))$.

3 Knowledge representation

3.1 Introduction

In order to solve complex problems encountered in AI, a considerable amount of knowledge, as well as some mechanisms for manipulating knowledge, are necessary. Barr and Feigenbaum identify four types of knowledge as follows [Miranker 87]:

- Objects, i.e. nouns and adjectives that describe them.
- Events: Object interaction.
- Performance: How to do something, also known as procedural knowledge.
- Meta-knowledge: Knowledge about knowledge.

Knowledge plays two roles in AI programs as follows:

- It may define the search space and the criteria for determining a solution to a problem.
- It may improve the efficiency of a reasoning procedure by informing an inference procedure of the best places to look for a solution.

Knowledge representation occurs at two levels [Rich & Knight 91]:

- Data level, at which facts are described.
- Symbol level, in which representations of objects at the data level are defined in terms of symbols that can be manipulated by programs, such as PROLOG rules.

Knowledge representation and search are the two main themes of AI problem solving but they are not independent issues. If a particular search method is applied, and a method of knowledge representation may represent the problem more easily and it more efficiently supports the operations required by the search strategy, a particular problem may be more easily solved. For a particular problem, different combinations of knowledge representation methods and search may yield more or less effective means for solving the problem. The next section will focus on representing knowledge by using rules.

3.2 Knowledge representations for rule-based systems

The use of rules for encoding knowledge is a particularly important issue because rule-based reasoning systems have played a very important role in AI evolution from a purely laboratory science into a commercially significant one. This section outlines two representation methods namely OPS5 and "rule schema + rule body", which have been applied in some rule-based systems.

3.2.1 OPS5

OPS5 [Forgy 82] is a rule-based system language. An OPS5 rule comprises the following:

1. Symbol P.
2. Rule name.
3. Left-hand side (LHS).
4. Symbol →
5. Right-hand side (RHS).

All of these are enclosed in parentheses.

A typical LHS structure is as follows: { <object> (computer ↑Name <name> ↑price <cost>)}. This structure is used to describe computer objects; it includes computer name and price.

RHS structure is similar to LHS structure, but contains an action, e.g. 'modify' before <money> (See below) .

The ↑ is the OPS5 operator that distinguishes attributes from values.

A variable is a symbol beginning with the character '<' and ending with the character '>', e.g. <object>.

The predicates in OPS5 include =, <>, <, >, <=, >=. A predicate is placed between an attribute and a value.

The following is a typical rule from [Brownston et al. 86]:

```
(P have-enough-money-to-buy-computer
    { <object> (computer ↑name <name>
      ↑price <cost>)}
    { <money> (saving-account ↑balance
      {<balance> ≥ <cost> })}
⟶
    (modify <money> ↑balance
      (computer <balance> - <cost>)))
```

where the meanings are apparent.

3.2.2 Rule schema + rule body

"Rule schema + rule body"[Wu 93a] is a 2-level method of knowledge representation. A rule schema is used to describe the hierarchy among factors or nodes in a reasoning network. A rule body consists of computing rules and/or inference rules and is used to express specific evaluation methods for factors and/or certainty factors in corresponding rule schemata.

A rule schema has the general form:
IF E1,E2,..,En THEN A,
where E1, E2, ..., En is a conjunction (AND) of all premise factors and A is a predicate or variable called a conclusion factor.

Each rule schema has a corresponding rule body. In a rule body, there are one or more inference rules such as production rules and/or computing rules for computation.

A rule schema with its corresponding rule body is called a *rule set*. A rule set is an independent knowledge unit in the "rule schema + rule body" representation and can be described in *Backus Naur* form as follows:

```
<rule set> := <rule set number><rule
  schema><rule body>
<rule set number> := <integer>
<rule schema> := 'IF' <premise
  factors> 'THEN' <conclusion factor>
<premise factors> := <premise factor>
  {','<premise factors>}
<premise factor> := <factor>
<conclusion factor> := <factor>
<factor> := <logic
  assertion>|<variable name>
<logic assertion> := <predicate(object)>
<rule body> := (<C-rule>|<I-rule>)
  {<rule body>}
<C-rule> :=(<factor>|
  (CF'('<factor>')'))'='
  <assignment expression>
<assignment expression> :=
  <value>|<algebraic expression>
<I-rule> := 'IF' <antecedents> 'THEN'
  <conclusion>
<antecedents> := <antecedent>{'and'
  <antecedents>}
<antecedent> := (<factor>|
```

```
  CF'('<factor>')'))
  <relational symbol>
  <assignment expression>
<relational symbol> :=
  '>'|'<'|'='|'<>'|
  '>='|'<=''
<conclusion> := <C-rule>
<value> := <integer>|
  <real>|<symbolic value>|
  <probability>|<fuzzy value>
```

The terms <variable>, <predicate(object)>, <algebraic expression> and different kinds of values above have the standard interpretations.

3.3 Comparison between OPS5 rules and "rule schema + rule body" representation rules

There are a number of advantages with the "rule schema + rule body" representation [Wu 93a]. Firstly, rule schemata in a knowledge base provide a way of describing meta-knowledge about concrete rules in rule bodies which facilitate sorting rule sets in a rule base. Secondly, it expresses computing rule sets in the same form as inference rule sets. Further, it provides natural "IF–THEN" expertise expression in two-level structures, and also provides flexible processing of inexact reasoning in rule bodies, and so forth.

However, the negative aspect of these advantages is that "rule schema + rule body" representation is not so powerful as OPS5 rules. For example, it cannot efficiently represent first-order logic rules.

4 Design of rule-based system algorithms

Usually an algorithm refers to a method of solving a well-specified computational problem for a system. With the development of rule-based systems, efficiency has been a major consideration up to this stage. A rule-based algorithm is considered more efficient than others if its cost for per working memory change is lower. This section addresses some measures for improving the efficiency of rule-based system algorithms and analyses some good algorithms [Fang & Wu 94].

4.1 The knowledge

For the purpose of obtaining efficiency, three types of knowledge or state information may be incorporated into a rule-based system algorithm as follows [McDermott et al. 78]:

1. *Condition membership*, which provides knowledge about the possible satisfaction of each individual condition element. An algorithm that uses condition membership can ignore further processing of those rules which are not active, i.e. those rules that one or more positive condition elements are not partially satisfied.

2. *Memory support*, which provides knowledge about which working memory elements individually partially satisfy each individual condition element. Associated with each condition element in rule-based systems is a memory which indicates precisely which subset of working memory elements partially match the condition element.

3. *Condition relationship*, which provides knowledge about the interaction of condition elements within a rule, and partial satisfaction of rules. The process for condition relationship is similar to that of maintaining the results of intermediate joins in database systems.

Two further types of knowledge can be identified as follows:

4. *Conflict set support*, which provides knowledge about which rule has consistent variable bindings between its condition elements. The conflict set is retained across each 3-phase cycle, and the contents of the conflict set are used to limit search in a rule-based system during the chaining process [Miranker 87].

5. *Premise-conclusion relationship*, which provides knowledge about which premise factor of a rule schema is the conclusion factor of another rule schema or which premise factor of a rule schema partially matches (See the definition in Section 2.3) with the conclusion factor of another rule schema. This knowledge is used to arrange rule sets in the rule base into order [Wu 93a].

4.2 Matching techniques

In each 3-phase cycle, matching is a crucial step. This section introduces four techniques for matching, namely indexing, filtering, and decision tree and decision table methods.

4.2.1 Indexing

In the matching process, the current state can be used as an index for immediate selection of matching rules, provided that the rule preconditions are stated as exact descriptions of the current state [Rich & Knight 91]. The simplest form of indexing for rule-based systems is that the interpreter begins the match process by extracting one or more features from each working memory element and uses these features to hash into the rule collection. This obtains a set of rules that might have satisfied LHSs. A more efficient form of indexing adds memory to the process. For example, one scheme involves storing a count with

each condition element. The counts are all set to zero when the system starts the execution. When a data element enters the working memory, all condition elements matching the data element have their counts increased by one. When a data element leaves the working memory, all condition elements matching the data element have their counts decreased by one. The interpreter deals with those LHSs that have non-zero counts for all their positive condition elements. This scheme has been combined into a few algorithms with other efficient measures [Forgy 82].

4.2.2 Filtering

Filtering is a method which uses a filter namely a body of code that uses the knowledge sources (KSs) introduced in the last section to reduce the number of rules tested by a rule-based system. If a filter contains enough information, a significant number of rules can be excluded from consideration. A filter admits to further testing of any subset of rules that may be unsatisfied by its KSs.

Filters usually are in the form of discrimination nets of which the famous RETE and TREAT algorithms are the best examples. RETE incorporates memory-support and condition-relationship; whereas TREAT takes one more knowledge source into account— conflict set support (See Section 4.4). RETE and TREAT will be further analysed in the following sections.

4.2.3 The decision tree method

The decision tree method compiles the set of condition elements which are defined in the form of lists into a near-optimal decision tree [Malik 81]. Firstly, it restricts a segment variable (e.g. @), which represents list fragments of an undefined length, to the tail of a sublist, and assumes that any datum in the working memory necessarily matches some condition elements of rules. And it treats two kinds of element features: those which compute the length of a list or a sublist, and those which extract an atom from some specified position. These tasks are done by the internal nodes of the tree. A leaf of the tree is a pointer that points to a stack which contains all data that match the corresponding condition elements. However, sometimes there exist overlapping cases in which some leaves contain more than one condition element. These considerations lead to an algorithm. It starts by selecting an 'efficient' feature which is defined everywhere as root of the tree from the discriminating feature table (built from condition element features). In the table '-' and '@' denote undefined values and variable feature values respectively. And then it recursively lets each branch have a label which corresponds to a subtable of the identification table. The recursion stops if some subtable has an empty set. An example from [Malik 81] is as follows:

Assume five rule antecedents are given:

```
R1: (A(B C x)) (D x) (x F y) (E F y) --> ...
R2: (A(B C x)) ~(D x) (x F y) (B F y)
      ~(C B @) --> ...
R3: (A(x F y)) (H x) (x F z) --> ...
R4: (A(B F @)) (B F y) (Z C B)} --> ...
R5: (A(x B)) (x C B)} --> ...
```

where x, y and z are variables, and capital letters represent propositions. An identification table which plots these feature values v.s. the set of condition elements is constructed in Figure 2.

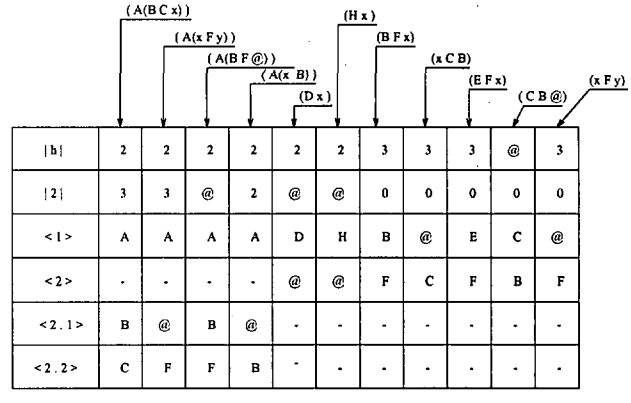

Figure 2: An Identification Table

After non-discriminating features (e.g. |2| with 0 values) are deleted from the table, a discriminating decision tree can be built in Figure 3.

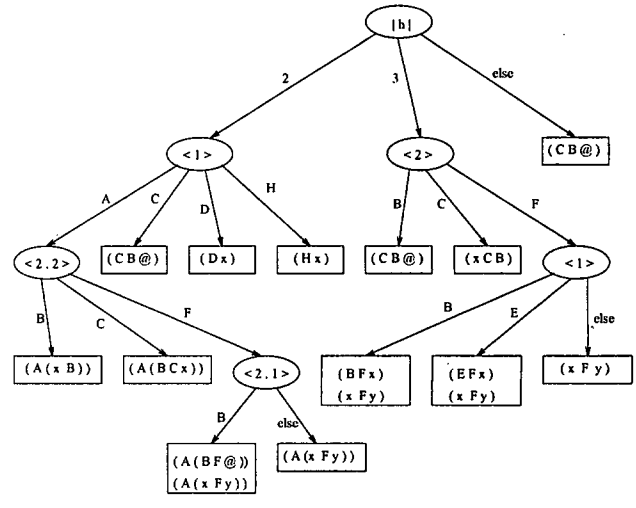

Figure 3: A Discriminating Decision Tree

At the beginning of each "match – conflict resolution – act " cycle, the interpreter traverses the tree with each modified datum in the working memory, computing a feature value at each tree node and selecting the branch corresponding to the value. So a leaf containing the condition element can be reached. Characteristic of the decision tree method is that it avoids the redundant computations.

This method is suited to rule-based systems where condition elements of rules are represented in the form of lists. Taking into account the tree optimisation possibilities, the potential performance of the method seems to be promising.

4.2.4 The decision table method

This method is based on the table knowledge representation in [Colomb 89], and it is mainly suited to propositional rule-based systems. The transformation of the rule set in a rule base to a table representation involves a transformation algorithm which eliminates all rows in the table which are inconsistent and all rows which are subsumed by other rows. Its chaining procedure can be described as in Figure 4.

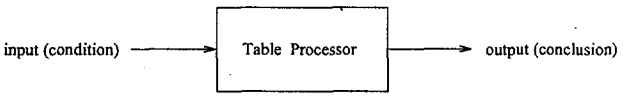

Figure 4: The chaining procedure of the decision table method

A table consists of rows which can be viewed as assignments of values to corresponding variables. A simple example of a table is as follows:

```
it-has-two-legs, fly, it-is-a-bird
it-has-four-legs, it-is-an-animal
```

When the input is *it-has-two-legs* and *fly*, the first row in the table will fire, and the output of the table processor (i.e. conclusion) will be *it-is-a-bird*; similarly, when the input is *it-has-four-legs*, the second row will fire, and the output of the table processor will be *it-is-an-animal*. Usually, rows in the decision table have interpretations as consequents such as the case of Garvan ES1 system.

Because of the replacement of intermediate assertions with expressions which imply them and the negation processing in the transformation procedure, the size of a knowledge table may be explosively large. In order to solve this problem, a few algorithms have been designed for reducing ambiguity and redundancy in [Colomb & Chung 90]. These algorithms can greatly reduce the response time of a system. In addition, an unambiguous table can be further transformed into a decision tree by ID3-like algorithms [Quinlan 86, Wu 93b]. A decision tree executes a number of nodes logarithmic in the number of rows in the decision table. Therefore, the decision table method can process large propositional rule-based systems efficiently.

4.3 Conflict resolution

The output from the matching process, and the input to the conflict resolution, is a set referred to as conflict set. All rules in which LHSs have been satisfied by working memory elements can be identified by conflict set elements, which are termed instantiations. An instantiation is an ordered pair of a rule name and a list of working memory elements matching the condition elements of the rule. It is the job of conflict resolution to find an instantiation which will be executed in the act phase of a cycle.

A conflict-resolution strategy is a coordinated set of principles for making selections among competing instantiations. A rule-based system's performance depends on its conflict-resolution strategy for both sensitivity and stability [Brownston et al. 86]. Sensitivity is the fast degree by which a system responds to the dynamically changing demands of its environment, while stability is its continuity of behavour. The following principles can be applied for any conflicrt resoluiton strategy [Brownston et al. 86]:

- **Refraction**

  Refraction prevents rules from firing on the same data more than once. The intention is to avoid the trivial form of infinite looping which might occur when a rule does not change the working memory contents.

- **Data ordering**

  Data ordering which orders data by recency or activation is a basic principle of conflict resolution and a powerful way of adding sensitivity to a conflict-resolution strategy. It gives preference to rules that match those elements most recently added to working memory or that are strongly related to recently-added data. This principle is usually combined with other principles to narrow down the selection of one instantiation to fire next.

- **Specificity ordering**

  The specificity principle gives preference to rules that are more specific according to some standard which can be measured in a variety of ways. For example, one specificity principle depends on a specificity function that is correlated with the complexity degree of rule condition elements.

- **Rule ordering**

  Rule ordering (which tends to be less sensitive) provides static ordering of a rule set independent of the way in which rules are instantiated by data. The ordering may be computed by using some rule feature/features. Either total or partial ordering can be given by a relation on rules. If total rule ordering has been provided, the rules can be stored in the order and scanned linearly until a matching one is found. LFA [Wu 93a] is a successful example of using the rule ordering strategy.

- **Arbitrary choice and parallel selection**

  None of the above principles can guarantee that only a single instantiation will remain in the conflict set. If single firing is required for each cycle, an arbitrary decision referred to as arbitrary choice ordering can be made after all conflict-resolution principles have been applied; however, in some systems especially parallel systems, all the remained instantiations can be fired in one cycle, which is called parallelism in firing.

The following are two alternative conflict-resolution strategies for OPS5 systems — LEX and MEA [Forgy 81]:

LEX

The LEX conflict-resolution strategy includes four steps which are applied in order to find an instantiation:

1. Discard from the conflict set those instantiations that have already fired. If there are no instantiations that have not fired, conflict resolution fails and no instantiation is selected.

2. This step partially orders the remaining instantiations in the conflict set on the basis of recency of working memory elements by using the following algorithm to compare pairs of instantiations: Compare the most recent elements from two instantiations. If one element is more recent than the other, the instantiation containing that element dominates. If the two elements are equally recent, compare the second most recent element from the instantiations. Continue in this way until either one element of one instantiation is found to more recent than the corresponding element in the other instantiation, or no element remains for one instantiation. If one instantiation is exhausted before the other, the other dominates. If the two instantiations are exhausted at the same time, neither dominates.

3. If no one instantiation in particular dominates all others under the previous step, this principle is necessary for comparing the dominant instantiations on the basis of the specificity of the LHSs of the rules. Count the number of tests (for constants and variables) that have to be made in finding an instantiation for the LHS. The LHSs that require more tests dominate.

4. If no single instantiation dominates after the previous step, make an arbitrary selection of one instantiation as the dominant instantiation.

MEA

The MEA strategy differs from that of LEX in that another step has been added after the first step in LEX. It places extra emphasis on the recency of a working memory element matching the first condition element of a rule. If no single instantiation dominates, then the remaining set is passed through the same sequence of orderings as in LEX.

4.4 RETE

The RETE match algorithm [Forgy 82] is an algorithm for computing the conflict set. It improves matching efficiency by incorporating memory-support and condition-relationship to avoid iterating computations over the working memory and the rule base; its concrete measure is using a sorting network (Subparts of which can be shared) which is compiled from the condition element patterns of rules to test features of data elements, and to store information. The structural form of the sorting network is given in Figure 5.

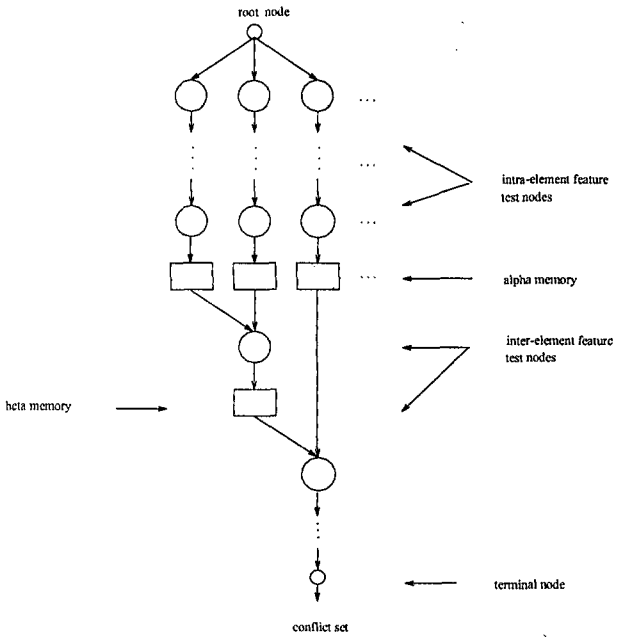

Figure 5: The Structural Form of RETE's Sorting Networks

RETE deals with two types of element features i.e. intra-element features and inter-element features. Intra-element features are the features that involve one working memory element. For example, the class of an element must be 'Expression'; the value of an 'OP' attribute must be '+'. However, inter-element features result from having a variable occur in more than one pattern. For instance, the value of an attribute of an element must be equal to the value of an attribute of another element.

As shown in Figure 5, when the pattern compiler processes an LHS, it builds a chain of intra-element feature test nodes, which are one-input, for each condition element pattern of the LHS based on the intra-element features which are required by the condition

element pattern. And then it builds inter-element feature test nodes for testing the inter-element features of the LHS. The inter-element feature test nodes are two-input and left-associative [Ho & Marshall 92]. Finally it builds a terminal node to represent the production rule.

The match procedure is as follows: The root node receives a token, which is state information (The tags '+' and '–' in a token indicate how the state information is to be changed), and then passes the copy of the token to all its successors i.e. the intra-element feature test nodes. A '+' token that has satisfied the intra-element feature tests is added to the alpha memory. A '–' token that has satisfied the intra-element feature tests has a corresponding '+' token that already presents in the alpha memory. The corresponding '+' token is removed. Once a token updates an alpha memory, it continues to go through the network and the next node is an inter-element feature test node. Inter-element feature test nodes store the first token and wait until the second one arrives, and then compare them. If they find that the variables between the two tokens are bound consistently, they join the two tokens into a bigger one. The bigger token is stored in the beta memory, and then sent to another inter-element feature test node for other consistent tests of variable bindings (if possible). If all the variable bindings are consistent for an LHS, the final token is sent to the terminal node. The terminal node receives the token, and adds the rule instantiation (Which is an ordered pair of the form <rule, list of data elements matched by the LHS of the rule>) of the LHS to the conflict set.

Over the naive matching algorithms, advantages of the RETE match algorithm can be summarised as follows:

- It does not need the interpretive step by using the sorting network.
- Sufficient state is maintained so that it can avoid many iterated computations.
- The subparts of the network for similar condition element patterns can be shared.

There are also some disadvantages inherent to the RETE match algorithm:

- It is just a matching algorithm.
- Time complexity of RETE is NP-hard to the number of rules in the rule base.
- The removal of data elements performs such operations for adding the data elements that the deletions of working memory elements are expensive.
- It is inefficient when most of the data changes in each cycle, because in that case RETE needs to maintain its state between cycles i.e. it cannot efficiently process non-redundant rule-based systems.

4.5 TREAT

The TREAT match algorithm [Miranker 87] is a RETE-like algorithm. It not only makes use of condition membership and memory support knowledge sources, but also combines them with a new source of information, *conflict set support.* Its significant features are that (1) in some cases it performs much better than the RETE algorithm, and (2) it can be used in parallel systems.

The TREAT algorithm constructs a sorting network from the condition patterns of the rule set. But no subpart of the network can be shared by more than one condition pattern. Furthermore, it adopts the following measures:

- *Conflict Set Support*

TREAT retains the conflict set across system cycles and uses its contexts to reduce the number of comparisons required to find consistent variable bindings [Miranker 87]. As a result, it reduces the computations between beta memories that the RETE algorithm needs.

- *Handling Negated Condition Elements*

When a data element which partially matches a positive condition element is added into the working memory, the conflict set remains the same, except that the addition of the working element element results in new instantiations. If a rule is active (See *Condition support*) and the new instantiations contain the new working memory element, then the instantiations are added into the conflict set.

When a working memory element which partially matches a negative condition element is deleted, no new rules will be instantiated. In that case, the instantiations that contain the removed working memory element will be invalidated and are removed from the conflict set.

When a rule firing adds a working memory element that partially matches a negated condition element, there may be some rule instantiations that are invalidated and will have to be removed from the conflict set. In this case, the invalidated instantiations will not contain the working memory element. To find the instantiations which must be removed from the conflict set, the negated condition element which is partially matched is temporarily transformed to be positive to form a new rule. The working memory element is used as a seed to build instantiations of this new rule. Then the new instantiations are compared with the conflict set. If any instantiation exists in the conflict set, then remove them.

When a working memory element is removed and it partially matches a negated condition element, if there is no other similar data element whose variable bindings are consistent with those of the removed one in the working memory, it may cause some rule instantiations to enter the conflict set.

– *Memory Support*

The alpha memories forming the memory support part of the TREAT match algorithm are the same as those of the RETE match algorithm. Information related to each condition element is stored in arrays indexed by CE-num's (condition element numbers). All the condition elements in a rule-based system are numbered with CE-num's. The alpha memories are partitioned into *old-mem, new-delete-mem and new-add-mem* as three separate vectors. The addition and deletion of a working memory element are different from those of the RETE match algorithm. This can be seen from the algorithm illustration below.

– *Condition Support*

Associated with each rule is a rule-active property. A rule is active if each of its positive condition elements is partially matched by some working memory elements. The rule-active property of a rule is affected by updating the contents of the *old-mem* for the rule.

The advantages of the TREAT algorithm are as follows:

- The deletion of elements is simpler than in RETE.
- Inactive rules are ignored.
- It can handle both temporally redundant and non-redundant rule-based systems.
- It can be easily implemented in parallel rule-based systems.

However, TREAT also has some disadvantages:

- No computing results can be shared by condition patterns or rules.
- The time complexity for matching is still NP-hard as RETE.
- It is also a matching algorithm.

4.6 LFA

Unlike RETE-like algorithms, the LFA algorithm [Wu 93a] is a linear forward-chaining algorithm. It adopts a 2-level "rule schema + rule body" knowledge representation outlined in Section 3.2.2. The major features of the LFA algorithm are that chaining is carried out in 2-phase "match-act" cycles instead of the 3-phase "match – conflict resolution – act" cycles, and it can choose one rule set in each cycle without any specific conflict resolution.

The following outlines the concrete measures that the LFA algorithm has adopted:

– *"Rule Schema + Rule Body"*

"Rule schema + rule body" represents knowledge in two levels. Rule schemata describe the hierarchy among factors (include premise factors and conclusion factors) or nodes in a reasoning network. Rule bodies, which consist of computing rules and/or inference rules, are used to express specific computing methods for the factors and/or certainty factors in their corresponding rule schemas. This 2-level structure facilitates sorting the knowledge base and avoids matching all the rules in a knowledge base with the working memory when some piece of data is not available.

– *Sorting the Knowledge Base* [Wu 93a]

At the end of knowledge acquisition or knowledge modification, the knowledge base is sorted or compiled into a partial order: If rule schema N is *if factor-1, factor-2, ..., factor-n, then factor*, then all the schemas with *factor-1, factor-2, ..., factor-n* as their conclusion factors have rule-set numbers smaller than N.

Other sorting measures are as follows:

1. Processing dead cycles.

   A cycle like *if A then B, if B then C,* and *if C then A* in a domain reasoning network is called a dead cycle if none of A, B, and C is a leaf node in the domain reasoning network and there is no other rule schema whose conclusion factor is one of them. A dead cycle cannot be numbered and has to be changed to a live cycle or removed. Figure 6 is an example of dead cycles.

2. Renumbering schemata.

   Renumber all rule schemata which have all of their premise factors being leaf nodes in the domain reasoning network. For any factor F, if all schemata with it as their conclusion factor have been renumbered, it is treated as a leaf node for further renumbering. If all rule schemata in a knowledge base have been renumbered then stop.

3. Resolving live cycles.

   A cycle like *if A then B, if B then A,* and *if C then A* is called a live cycle and A is called a live node in the live cycle if C is not involved in any dead cycle (Figure 7 shows a live cycle). A live cycle can be resolved by treating one of its live nodes as a leaf node for further renumbering. Resolve all live cycles and goto 2.

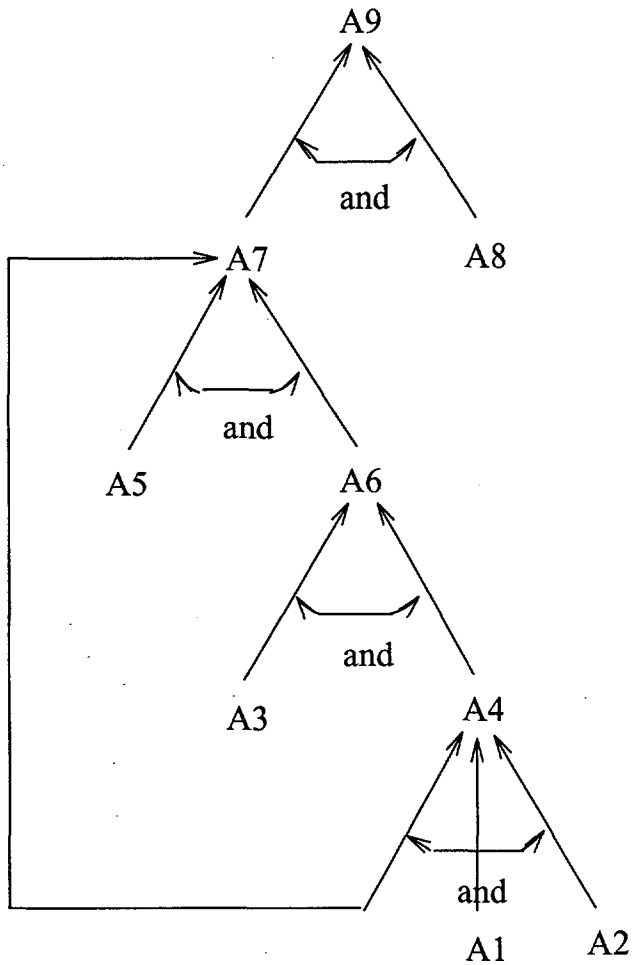

Figure 6: A dead cycle

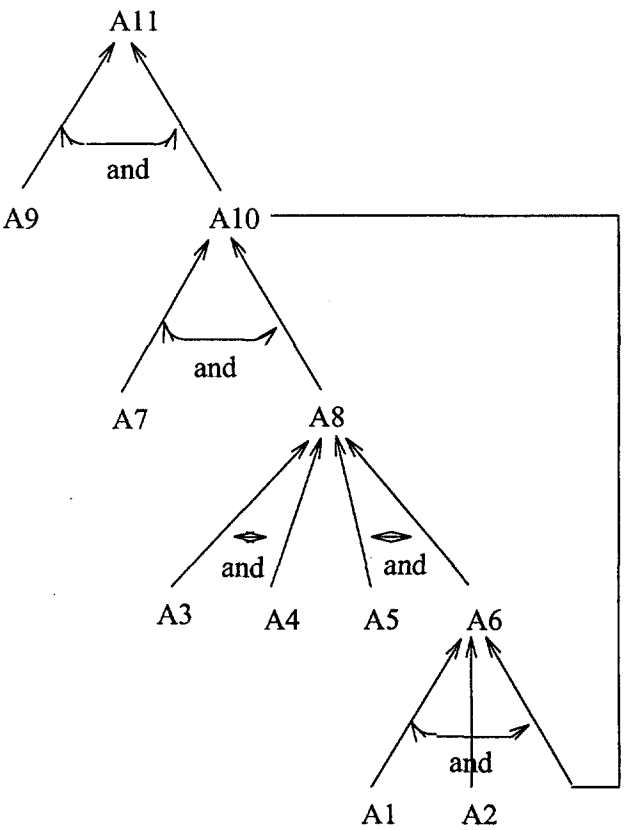

Figure 7: A live cycle

– Linear Forward Chaining

After knowledge compilation, chaining is performed as follows:

```
FOR the first TO the last renumbered
      schema in the knowledge base
 DO
   IF there exist data in the working
     memory for each of the
     condition factors of the schema
   THEN fire the corresponding
        rule body of the schema.
ENDFOR.
```

Advantages of the LFA algorithm:

– Time complexity is $O(n)$ where n is the number of rules in a knowledge base.

– It is a complete forward-chaining algorithm, in which the conflict resolution step is unnecessary.

– Natural knowledge representation.

– Computational knowledge and uncertainty calculus are integrated with logic inference in the 2-level knowledge representation.

– Intermediate computing results can be shared.

Disadvantages of the LFA algorithm:

- First-order rules can not be efficiently processed.
- Its chaining is in a fixed order, and thus all problem evidence needs to be provided at the beginning of chaining in order to ensure that the inference can be accomplished.

5 LFA+: A robust LFA algorithm

For rule-based systems, LFA is the best forward-chaining algorithm to date in terms of time complexity. However, as with other algorithms, it also has its own limitations. The significant drawback which greatly restricts application is that LFA cannot efficiently deal with first-order logic rules. Based on the original LFA algorithm, this section describes a robust LFA algorithm — LFA+, which mainly tackles this problem and efficiently deals with recursive rules and negation.

The idea to transform first-order logic rules into simpler domains was used before in inductive logic programming [Lavrac & Dzeroski] and in empirical learning in order to transform more complex expressive mechanisms into e.g. attribute-value descriptions [Gams et al. 91].

5.1 Knowledge representation

Knowledge representation adopts the basic "Rule Schema + Rule body" structure (See Section 3.2.2), but expressiveness is extended to include first-order logic rules. In addition, some other measures are presented for the following purposes:

1. It facilitates avoiding matching all the rules in a rule base with the working memory at rum time when some piece of data is not available.
2. It supports ordering the knowledge.

– Predicate representation

A predicate can be represented in the following form:

<predicate symbol>(object-list).

In the object-list, an object is a constant or a variable.

– Rule schemata representation

A rule schema takes the general form:

IF *factor1, factor2,* $\cdots$*, factorn* THEN *factor*

where *factor1, factor2,* $\cdots$*, factorn, factor* may be variables for computing rules or logic assertions for inference rules. But any variable in logic assertions is replaced by the '_' notation which means "don't care" in rule schemata. The *factor* can also be an algebraic function or an action defined to modify the working memory.

– Negation representation

Negation representation takes the form: not(p), where p represents a predicate. Its meaning is defined as that of [Bratko 90] — If p cannot be proven to be TRUE, then not(p) is TRUE.

Two constraints are given to this representation:

1. 'not(p)' and 'p' cannot appear in two different rules respectively within a rule set. For example, for

   p(X,Y) :- s(X,Y), not(r(X)).
   p(X,Y) :- s(X,Y), r(Y).

   there are two rule sets:

   Rule schema: IF s(_,_), not(r(_)) THEN p(_,_)
   Rule body: IF s(X,Y) and not(r(X)) THEN p(X,Y)

   and

   Rule schema: IF s(_,_), r(_) THEN p(_,_)
   Rule body: IF s(X,Y) and r(Y) THEN p(X,Y)

   Thereby, the confused representation that not(r(_)) and r(_) appear in the same schema can be avoided, and when data do not exist for r(_), the matching procedure for the rule body in the second rule set is unnecessary at run time.

2. 'not(p)' and 'p' may appear in the same inference rule in a rule set, but only the 'p' is included in the corresponding premise factors of the rule schema. For instance, the rule set for

   p(X,Y,Z) :- s(X,Y,Z), not(r(X,Y)), r(Y,Z).

   is

   Rule schema: IF s(_,_,_), r(_,_) THEN p(_,_,_)
   Rule body: IF s(X,Y,Z) and not(r(X,Y)) and r(Y,Z) THEN p(X,Y,Z)

This knowledge representation can be described in extended *Backus Naur form* as follows:

```
<rule set> := <rule set number><rule
  schema><rule body>
<rule set number> := <integer>
<rule schema> := 'IF' <premise
  factors> 'THEN' <conclusion factor>
<premise factors> := <premise
  factor> {','<premise factors>}
<premise factor> := <factor>
<conclusion factor> := <factor>
<factor> := <logic
  assertion>|<variable>|< algebraic
  function>|<action>
<logic assertion> :=
```

```
  <predicate>|(not'('<predicate>')')
<predicate> := <predicate
  symbol>{'('<object-list>')'}
<object-list> := <object>{','
  <object-list>}
<object> := <constant>|<variable>|'_'
<algebraic function> :=
  (functor'('<variable-list>')')
<variable-list> := <variable>
  {','<variable-list>}
<rule body> := (<C-rule>|<I-rule>)
  {<rule body>}
<C-rule> := (<factor>|
  (CF'('<factor> ')'))'='
<value>|<algebraic expression>
<I-rule> := 'IF' <antecedents> 'THEN'
  <conclusion>
<antecedents> := <antecedent>{'and'
  <antecedents>}
<antecedent> :=<logic
  assertion>|<relation expression>
<relation expression> :=
  (<factor>|(CF'('<factor>')')|
  <value>|
  <algebraic expression>)
  <relation symbol>
  (<factor>|(CF'('<factor>')')|<value>|
  <algebraic expression>)
<value> := <integer>
  |<real> |
  <symbolic value> |
  <probability>|<fuzzy value>
<relation symbol> :=
  '>'|'<'|'='|'<>'| '>='|'<=''
<conclusion> :=
<logic assertion>|<action)>|<C-rule>
```

– Recursive rule representation

In the above form, the further specific representation for recursive rules may be described as follows:

```
<I-rule> := <I-rule-1><I-rule-2>
<I-rule-1> := 'IF' <antecedents 1> 'THEN'
<conclusion 1> {<I-rule-1>}
<I-rule-2> := 'IF' <antecedents 2> 'and'
<antecedent 2> 'THEN'
  <conclusion 2>{<I-rule-2>}
```

Here, <conclusion 1>, <antecedent 2> and <conclusion 2> have the same predicate symbol.

In rule schemata, if <antecedents 1> and <antecedents 2> become the same, and so do <conclusion 1>, <antecedent 2> and <conclusion 2>, and there is only one rule for <I-rule-1>, then <I-rule-1> and <I-rule-2> can be put into one rule set, and the rule schema is as follows:

```
IF <antecedents 1> THEN <conclusion 1>
```

Otherwise they have to be broken into different rule sets, but these rule sets as a whole take part in the numbering process. Their schemata together are referred to as a recursive schema set.

For example, the rule set for

```
ancestor(X,Y) :- father(X,Y).
ancestor(X,Y) :- father(X,Z),
  ancestor(Z,Y).
ancestor(X,Y) :- father(Z,Y),
  ancestor(X,Z).
```

is as follows:

```
rule schema:
IF father(_,_) THEN ancestor(_,_)
rule body:
IF father(X,Y) THEN ancestor(X,Y)
IF father(X,Z) and ancestor(Z,Y) THEN
ancestor(X,Y)
IF father(Z,Y) and ancestor(X,Z) THEN
ancestor(X,Y)
```

However, the rule sets for

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- sister(Z,Y), parent(X,Z).
parent(X,Y) :- brother(Z,y),
  parent(X,Z).
```

are

```
rule schema:
IF mother(_,_) THEN parent(_,_)
rule body:
IF mother(X,Y) THEN parent(X,Y)
```

```
rule schema:
IF father(_,_) THEN parent(_,_)
rule body :
IF father(X,Y) THEN parent(X,Y)
```

```
rule schema:
IF sister(_,_), parent(_,_) THEN
  parent(_,_)
rule body:
IF sister(Z,Y) and parent(X,Z) THEN
  parent(X,Y)
```

and

```
rule schema:
IF brother(_,_), parent(_,_) THEN
  parent(_,_)
rule body:
IF brother(Z,Y) and parent(X,Z) THEN
  parent(X,Y)
```

As long as the <object> is defined as follows:

```
<object> := <object
  function>|<variable>|'_'
<object function> := <object functor>
  {'('<variable list>')'}
```

clearly, this knowledge representation can represent any Horn clause rules.

5.2 The LFA+ algorithm

This algorithm consists of two procedures namely sorting the knowledge in a knowledge base and linear-forward chaining.

- **Sorting knowledge in a knowledge base**

This procedure aims at placing rule sets in the rule base in order. The order of the rule sets is the order of the rule schemata. All measures adopted are described as follows:

– Processing of negations

Some schemata may have been separated due to the constraints of negation representation. These schemata together are called a schema set. In this case, the schema set as a whole takes part in the numbering process (See next step). Inside the schema set, all schemata take the order of the rule-body rules in situations that it is unnecessary to distinguish not(p) and p and in the order all rule-body rules can be put into a rule set.

– Schema numbering

If rule schema #N is

IF *factor1, factor2,* $\cdots$, *factorn* THEN *factor*

then all schemata whose conclusion factors partially match (See Section 2.3) with any of the *factor1, factor2,* $\cdots$, *factorn* have rule set numbers smaller than N. This process puts the schemata into partial order.

– Processing of live cycles

A cycle such as "IF a1 THEN b1, IF b2 THEN a2, and IF c THEN a3", in which a1, a2 and a3 partially match with each other, and b1 partially matches with b2, is referred to as a live cycle and a1, a2 and a3 become a live node in the live cycle if c is not involved in any dead cycles (e.g. a live cycle in Figure 8). Schemata in a live cycle can be numbered by starting from one of the live nodes of the live cycle.

– Processing of dead cycles

A cycle such as "IF a1 THEN b1, IF b2 THEN c1, and IF c2 THEN a2", in which a2 partially matches with a1, b1 partially matches with b2 and c1 partially matches with c2, is referred to as a dead cycle if none of a1, a2, b1, b2, c1 and c2 can be instantiated (Such as in Figure 9). A dead cycle cannot be numbered — it has to be changed into a live cycle or removed.

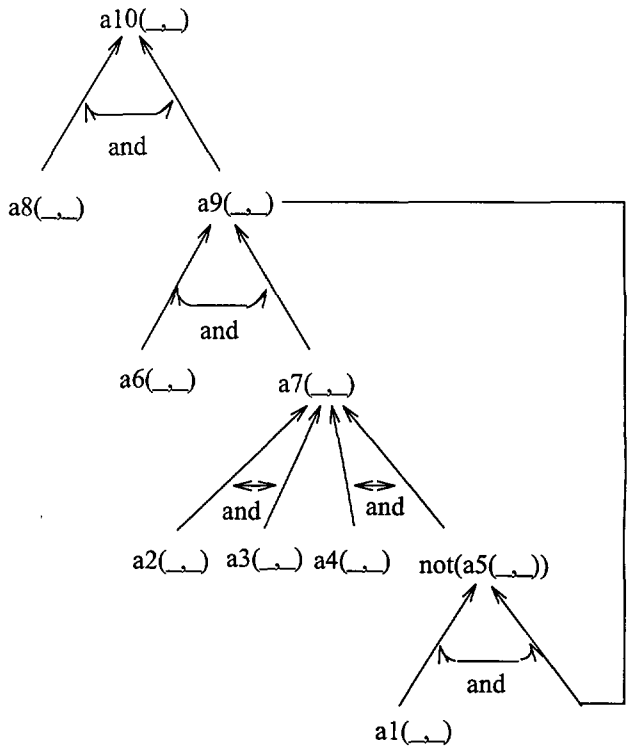

Figure 8: A live cycle

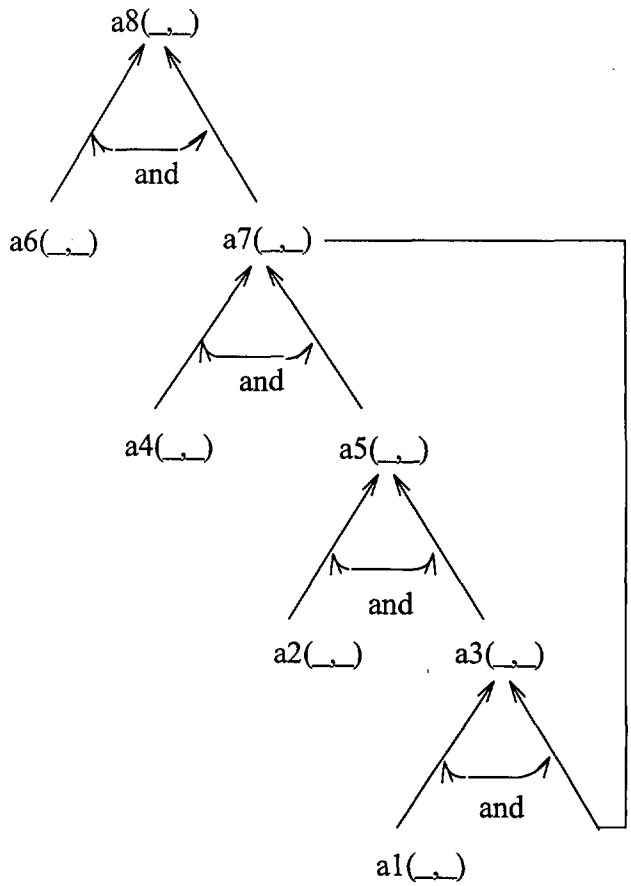

Figure 9: A dead cycle

– **Processing of parallel schema set**

If some schemata have the same conclusion factor (A negation schema set is treated as one schema) and are not involved in any dead cycles or any live cycles, then all these schemata together are referred to as a parallel schema set. A parallel schema set as a whole takes part in the numbering process, but inside the parallel schema set, the order of the schemata is arbitrary.

– **Rule schema renumbering**

Renumbering the schemata in a knowledge base until they are all in order. If they are already in order then stop.

• **Linear-forward-chaining**

After sorting knowledge in the knowledge base, the LFA+ algorithm performs the following process:

```
Loop:  from the first to the last schema do
If data exist in the working memory for each
of the premise factors (But for not(p), p
may or may not exist) in the schema, then
fire the corresponding rule body of the
schema
Endloop
```

5.3 Advantages and disadvantages

The time complexity of the LFA+ algorithm is also $O(n)$, where n is the number of rules in the rule base, as the original LFA algorithm. Yet, with the original LFA algorithm, the robust LFA+ algorithm has the following advantages:

– It can deal with first-order logic rules.

– Recursive inference rules can be efficiently processed.

– It facilitates coping with rules that include negative condition elements.

However, the LFA+ algorithm also has some disadvantages in common with other algorithms:

– It cannot meet the dynamic requirements to the knowledge base during inference.

– Its chaining is in a fixed order and thus all problem evidence needs to be provided at the beginning of chaining in order to ensure that the inference can be accomplished.

5.4 An example

Suppose the following Prolog rules are given:

```
sibling(X,Y) :- brother(X,Y).   (1)
sibling(X,Y) :- sister(X,Y).    (2)
sibling(X,Y) :- brother(Y,X).   (3)
sibling(X,Y) :- sister(Y,X).    (4)
parent(X,Y) :- father(X,Y).     (5)
parent(X,Y) :- mother(X,Y).     (6)
ancestor(X,Y) :- parent(X,Y).   (7)
parent(X,Y) :-
    bling(Z,Y), parent(X,Z).    (8)
ancestor(X,Y) :-
    parent(Z,Y),ancestor(X,Z). (9)
sibling(X,Y) :-
    brother(Z,Y),sibling(X,Z),
    X \== Y.                    (10)
sibling(X,Y) :-
    sister(Z,Y),sibling(X,Z),
    X \== Y.                    (11)
sibling(X,Y) :-
    brother(Y,Z),sibling(X,Z),
    X \== Y.                    (12)
sibling(X,Y) :-
    sister(Y,Z),sibling(X,Z)
    X \==Y.                     (13)
```

Based on the knowledge representation method of LFA+, their corresponding rule schemata can be represented as follows:

```
IF brother(_,_) THEN sibling(_,_) (1')
IF sister(_,_) THEN sibling(_,_)  (2')
IF brother(_,_) THEN sibling(_,_) (3')
IF sister(_,_) THEN sibling(_,_)  (4')
IF father(_,_) THEN parent(_,_)   (5')
IF mother(_,_) THEN parent(_,_)   (6')
IF parent(_,_) THEN ancestor(_,_) (7')
IF sibling(_,_),parent(_,_)
   THEN parent(_,_)               (8')
IF parent(_,_),ancestor(_,_)
   THEN ancestor(_,_)             (9')
IF brother(_,_),sibling(_,_)
   THEN sibling(_,_)              (10')
IF sister(_,_),sibling(_,_)
   THEN sibling(_,_)              (11')
IF brother(_,_),sibling(_,_)
   THEN sibling(_,_)              (12')
IF sister(_,_),sibling(_,_)
   THEN sibling(_,_)              (13')
```

As shown in (10'), (11'), (12'), and (13'), variables in relational expressions of variables (X \== Y) are not represented in schemata, because they are determined by the same variables in corresponding predicates. Since (1') and (3'), (2') and (4'), (10') and (12'), and (11') and (13') are the same respectively, they can be combined. So they become the following:

```
IF brother(_,_) THEN sibling(_,_) (1")
IF sister(_,_) THEN sibling(_,_)  (2")
```

```
IF father(_,_) THEN parent(_,_)    (3")
IF mother(_,_) THEN parent(_,_)    (4")
IF parent(_,_) THEN ancestor(_,_)  (5")
IF sibling(_,_),parent(_,_)
   THEN parent(_,_)                (6")
IF parent(_,_),ancestor(_,_)
   THEN ancestor(_,_)              (7")
IF brother(_,_),sibling(_,_)
   THEN sibling(_,_)               (8")
IF sister(_,_),sibling(_,_)
   THEN sibling(_,_)               (9")
```

Apparently, (1") and (2") are a parallel schema set, so are (3") and (4"). (1"), (2"), (8") and (9") form a kind of recursive schema set, so do (3"), (4") and (6"); (5") and (7") form another kind of recursive schema set, they can be combined. Based on the schema numbering measures of LFA+, these schemata can be put into the following order:

```
1 IF brother(_,_) THEN sibling(_,_)
2 IF sister(_,_) THEN sibling(_,_)
3 IF brother(_,_),sibling(_,_)
       THEN sibling(_,_)
4 IF sister(_,_),sibling(_,_)
       THEN sibling(_,_)
5 IF IF father(_,_) THEN parent(_,_)
6 IF mother(_,_) THEN parent(_,_)
7 IF sibling(_,_),parent(_,_)
      THEN parent(_,_)
8 IF parent(_,_) THEN ancestor(_,_)
```

where No. 8 is the combination of (5") and (7").

Finally, the ordered rule sets are as follows:

```
1 IF brother(_,_) THEN sibling(_,_)
  IF brother(X,Y) THEN sibling(X,Y)
  IF brother(X,Y) THEN sibling(Y,X)
2 IF sister(_,_) THEN sibling(_,_)
  IF sister(X,Y) THEN sibling(X,Y)
  IF sister(X,Y) THEN sibling(Y,X)
3 IF brother(_,_),sibling(_,_)
     THEN sibling(_,_)
  IF brother(Z,Y) and sibling(X,Z) and
     X \== Y THEN sibling(X,Y)
  IF brother(Y,Z) and sibling(X,Z)
     and X \== Y THEN sibling(X,Y)
4 IF sister(_,_),sibling(_,_)
     THEN sibling(_,_)
  IF sister(Z,Y) and sibling(X,Z)
     THEN sibling(X,Y)
  IF sister(Y,Z) and sibling(X,Z)
     THEN sibling(X,Y)
5 IF father(_,_) THEN parent(_,_)
  IF father(X,Y) THEN parent(X,Y)
6 IF mother(_,_) THEN parent(_,_)
  IF mother(X,Y) THEN parent(X,Y)
7 IF sibling(_,_),parent(_,_)
     THEN parent(_,_)
  IF sibling(Z,Y) and parent(X,Z)
     THEN parent(X,Y)
8 IF parent(_,_) THEN ancestor(_,_)
  IF parent(X,Y) THEN ancestor(X,Y)
  IF parent(Z,Y) and ancestor(X,Z)
     THEN ancestor(X,Z)
```

5.5 A chaining implementation of LFA+

The implementation is carried out with Sicstus Prolog on DEC stations. It consists of two parts namely receiving_data and chaining. Main program 'LFA+' has three parameters: Interp_f, Leaf_file and Kb_file.

Interp_f is an interpretation file, which has the following contents:

1. One or more domain value lists. For example, in domain([a,b,...]), the list is a value range of object variables; whereas in domain('~Y',[1,2,...]), the list is the value range of the variable '~Y'.
2. A truth value interpretation list. E.g. r_map([map(ancestor(adam,john,true)),...]).

Leaf_file includes the predicates which need to be provided by the users, and Kb_file is a rule file which contains all the rules for a problem solving.

The following is an implementation example. For convenience, we keep the Prolog rules as rule bodies since the order of rule sets is the order of rule schemata. So the contents of the Kb_file are as follows:

```
1.
'IF'. brother(_,_). 'THEN'. sibling(_,_).
 sibling(X,Y) :- brother(X,Y).
 sibling(X,Y) :- brother(Y,X).
2.
'IF'. sister(_,_). 'THEN'. sibling(_,_).
 sibling(X,Y) :- sister(X,Y).
 sibling(X,Y) :- sister(Y,X).
3.
'IF'. brother(_,_). ','. sibling(_,_).
     'THEN'. sibling(_,_).
 sibling(X,Y) :- brother(Z,Y),
     sibling(X,Z),X \== Y.
 sibling(X,Y) :- brother(Y,Z),
     sibling(X,Z), X \== Y.
4.
'IF'. sister(_,_). ','. sibling(_,_).
      'THEN'. sibling(_,_).
 sibling(X,Y) :- sister(Z,Y),
      sibling(X,Z),X \== Y.
 sibling(X,Y) :- sister(Y,Z),
      sibling(X,Z),X \== Y.
5.
'IF'. father(_,_). 'THEN'. parent(_,_).
 parent(X,Y) :- father(X,Y).
```

```
6.
'IF'. mother(_,_). 'THEN'. parent(_,_).
 parent(X,Y) :- mother(X,Y).
7.
'IF'. sibling(_,_). ','. parent(_,_).
      'THEN'. parent(_,_).
 parent(X,Y) :- sibling(Z,Y),parent(X,Z).
8.
'IF'. parent(_,_). 'THEN'. ancestor(_,_).
 ancestor(X,Y) :- parent(X,Y).
 ancestor(X,Y) :- parent(Z,Y),
       ancestor(X,Z).
```

Suppose the contents of the Leaf_file are the following:

```
brother('~X1','~Y1').
sister('~X2','~Y2').
father('~X3','~Y3').
mother('~X4','~Y4').
```

and the contents of the interpretation file are as follows:

```
domain([adam,eve,david,doris,john,mary,
   edgar,fred,lucy,margaret,violet,
   patrick]).
r_map([map(sister(doris,john),true),
   map(sister(margaret,fred),true),
   map(sister(lucy,edgar),true),
   map(sister(margaret,violet),true),
   map(sister(margaret,patrick),true),
   map(sister(margaret,fred),true),
   map(sister(violet,fred),true),
   map(sister(violet,margaret),true),
   map(sister(violet,patrick),true),
   map(brother(fred,violet),true),
   map(brother(fred,patrick),true),
   map(brother(patrick,fred),true),
   map(brother(patrick,margaret),true),
   map(brother(patrick,violet),true),
   map(brother(edgar,lucy),true),
   map(brother(fred,margaret),true),
   map(brother(john,doris),true),
   map(father(adam,doris),true),
   map(father(adam,john),true),
   map(father(david,edgar),true),
   map(father(david,lucy),true),
   map(father(john,fred),true),
   map(father(john,margaret),true),
   map(mother(eve,john),true),
   map(mother(eve,doris),true),
   map(mother(doris,edgar),true),
   map(mother(doris,lucy),true),
   map(mother(mary,fred),true),
   map(mother(mary,margaret),true)]).
```

The following records the example run.

```
>sicstus
SICStus 2.1 #9: Thu Apr 21 09:39:25 +1000
| ?-
| ?-consult('datalog.tex').
{consulting /fang/project/datalog.tex...}
{Undefined predicates will just fail}
yes
| ?- 'LFA+'(interp_file,leaf_file,kb_file).
{consulting /fang/project/interp_file...}
{/fang/project/interp_file consulted,
67 msec 1632 bytes}
Interpretation interp_file loaded.
Can you provide a value for ~X1 (y/n/q)?
y.
input:john.
Can you provide a value for ~Y1 (y/n/q)?
y.
input:diris.
Wrong value!
Can you provide a value for ~Y1 (y/n/q)?
y.
input:doris.
More values for prev. variables (y/n/q)?
y.
New value for ~X1 (y/n)?
y.
input:fred.
New value for ~Y1 (y/n)?n.
More values for prev. variables (y/n/q)?
n.
Can you provide a value for ~X2 (y/n/q)?
y.
input:doris.
Can you provide a value for ~Y2 (y/n/q)?
y.
input:john.
More values for prev. variables (y/n/q)?
n.
Can you provide a value for ~X3 (y/n/q)?
y.
input:adam.
Can you provide a value for ~Y3 (y/n/q)?
y.
input:john.
More values for prev. variables (y/n/q)?
q.
chaining ... 1: ->sibling(john,doris)->
    sibling(doris,john) 2:
  3: ->F->F 4: 5: ->parent(adam,john)
  6:  7: ->parent(adam,doris)
  8: ->ancestor(adam,john)->
    ancestor(adam,doris) !.
```

As shown above, there are not data for rule sets 2, 4 and 6; rules in rule set 3 are false, in which there are two rules in the rule body.

6 Conclusions and future research

With the development of rule-based expert systems, efficiency has been a major consideration for chaining algorithms [Fang & Wu 94]. The naive approaches are to combine indexing with direct interpretation of the LHSs in the rule base. They are inefficient in dealing with large knowledge bases.

The RETE match algorithm has made some significant improvements. It compares a set of LHSs of rules with a set of data elements in the working memory to compute the conflict set, and does not need the interpretive step. The indexing function is represented as a network of simple feature recognisers. This algorithm can efficiently process the conflict set, since it does not iterate over the working memory and the rule base. However, the RETE match algorithm only incorporates memory support and condition relationship knowledge sources. It still has significant disadvantages. For example, deletion of working memory elements is expensive, and time complexity is NP-hard (non-polynomial) to the number of rules in a knowledge base.

The TREAT algorithm is a RETE-like algorithm. It makes use of condition membership, memory support, and conflict set support knowledge sources. The obvious improvement is that it can be easily adopted in parallel systems. However, in some cases its performance is worse than that by using RETE.

LFA is the best chaining algorithm up to date in terms of theoretical time complexity. By adopting rule ordering method, its time complexity of chaining can be $O(n)$ where n is the number of rules in a knowledge base. This advantage results from its knowledge representation method namely the 2-level "rule schema + rule body" knowledge represntation (See Section 3.2.2), and using premise-conclusion knowledge to compile the rule set in the knowledge base. However, it is difficult to deal with first-order logic rules by using LFA.

LFA+ is a robust forward-chaining algorithm. It can process first-order logic rules efficiently. This mainly benefits from its knowledge representation. LFA+ inherits LFA's knowledge representation method, namely represents knowledge in a 2-level "rule schema + rule body" structure. But it has been extended to cover first-order logic rules (See Subsection 5.1). Based on this representation, LFA+'s sorting and chaining procedures for first-order logic rules can be the same ones of LFA for processing propositional logic rules. Therefore, LFA+ is a powerful linear-chaining algorithm for rule-based expert systems.

However, due to its static rule ordering method, its chaining is in a fixed order. So all problem evidence must be provided at the beginning of chaining in order to ensure that the inference can be accomplished for problem solving. This is a restriction to its application in data sensitive rule-based systems which give preferences to those rules that match the most recent data elements added to the working memory.

LFA+ is well suitable to be implemented by using logic programming language tools. When it is implemented by using imperative language tools, it has not provided memory support for avoiding iterating computations for matching working memory elements with condition elements when data exist for the corresponding rule schemata, so processing efficiency will decrease.

For future research, it is important to maintain LFA+'s linear performance. In order to extend its application, the following two directions may be considered:

- Combining other sorting measures with the 2-level "rule schema + rule body" knowledge representation method or modifying the "rule schema + rule body" structure if necessary in order that it can meet the requirements of data sensitive systems.

- Introducing the memory support knowledge source to LFA+. This involves building a mechanism related to how to organise those memories and how to efficiently locate the memories.

Also, meeting the dynamic demands of rule-based systems (i.e. the knowledge can be changed at run time) is a challenge. This is a common problem in all known chaining algorithms for rule-based systems. For example, RETE-like algorithms do not allow knowledge modification at run time either.

References

[Avron and Edward 81] Avron Barr and Edward A. Feigenbaum, *The Handbook of Artificial Intelligence, Vol. I*, Heuris Tech Press, 1981

[Bratko 90] Ivan Bratko, *PROLOG — Programming for Artificial Intelligence*, Second Edition, Addison-Wesley Publishing Company, 1990

[Brownston et al. 86] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin, *Programming Expert Systems in OPS5 — An Introduction to Rule-based Programming*, Addison-Wesley Publishing Company, Inc, 1986

[Colomb 89] R. M. Colomb *Representation of Propositional Expert Systems as Decision Tables*, Proceedings of the 3rd Australian Joint Conference on Artificial Intelligence, 1989

[Colomb & Chung 90] R. M. Colomb and C. Y. C. Chung *Ambiguity and Redundancy Analysis of a Propositional Expert System*, Proceedings of the

4th Australian Joint Conference on Artificial Intelligence, 1990

[Fang & Wu 94] Guang Fang and Xindong Wu, *Chaining in Rule-based Systems*, Proceedings of the 7th Australian Joint Conference on Artificial Intelligence, 1994, 575–582

[Forgy 81] C.L. Forgy, *OPS5 User's Manual*, Department of Computer Science, Carnegie-Mellon University, 1981

[Forgy 82] C.L. Forgy, *A fast algorithm for the many pattern/many object pattern match problem,* Artificial Intelligence, 19(1982), 17-27

[Gams et al. 91] Matjaz Gams, Matija Drobnic and Marko Petkovsek, *International Journal of Man-Machine Studies*, **34**(1991),49-68

[Ho & Marshall 92] Ho Soo Lee and Marshall I. Schor, *Match Algorithms for Generalised RETE Networks*, Artificial Intelligence, 54(1992), 249-274

[Kacsuk 90] Peter Kacsuk, *Execution Models of PROLOG for Parallel Computers*, Pitman, London, 1990

[Lavrac & Dzeroski] Nada Lavrac and Saso Dzeroski, *Inductive Logic Programming: Techniques and Applications*, Ellis Howood, (1994).

[Malik 81] Malik Ghallab, *Decision Trees for Optimising Pattern-Matching Algorithms in Production Systems*, Proceedings of the 7th Inter. Joint Conf. on AI, 1981, 310-312

[McDermott & Forgy 78] J. McDemott and C. Forgy, *Production System Conflict Resolution Strategies*, Academic Press, Inc., 1978

[McDermott et al. 78] J. Mcdermott, A. Newell and J. Moore, *The Efficiency of Certain Production System Implementations*, Academic Press, Inc., 1978

[Miranker 87] D.P. Miranker, TREAT:*A New and Efficient Match Algorithm for AI Production Systems*, PhD Thesis, Columbia University, 1987

[Pedersen 89] Ken Pedersen,*Expert Systems Programming — Practical Techniques for Rule-based Systems*, John Wiley & Sons, Inc, 1989

[Quinlan 86] J. R. Quinlan, *Induction of Decision Trees*, Machine Learning Vol 1, No. 1, 1986

[Raeth 90] Peter G. Raeth, *Expert Systems —A Software Methodology for Modern Applications*, IEEE Computer Society Press, 1990

[Rich & Knight 91] Elaine Rich and Kevin Knight, *Artificial Intelligence*(Inter. Edition(2)), McGraw-Hill Inc., 1991

[Schalkoff 90] Robert J. Schalkoff, *Artificial Intelligence* — An Engineering Approach, McGraw-Hill Inc. 1990

[Sterling et al. 86] Leon Sterling and Ehud Shapiro, *The Art of PROLOG — Advanced Programming Techniques*, The MIT Press, 1990

[Steven 90] Steven L. Tanimoto, *The Elements of Artificial Intelligence* — Using Common Lisp, Computer Science Press, 1990

[Waterman & Hayes-Roth 78] D. A. Waterman and Frederick Hayes-Roth, *An Overview of Pattern-directed Inferences Systems*, Academic Press Inc., 1978

[Wu 93a] Xindong Wu, *LFA: A Linear Forward-chaining Algorithm for AI Production Systems*, Expert System: The Int. J. of Knowledge Engineering, **10**(1993), 4: 237–242

[Wu 93b] Xindong Wu, Inductive Learning: Algorithms and Frontiers, *Artificial Intelligence Review*, **7**(1993), 2: 93-108

Appendix – Definitions

Backus-Naur form

This expression refers to a formal language for context-free grammars. A grammar consists of a set of rewrite rules, each of which has a *left-hand side* and a *right-hand side*, separated by the metalanguage symbol ::=. The left-hand side of each rule is a nonterminal symbol of the grammar, while the right-hand side is a sequence of nonterminal and terminal symbols. Nonterminal symbols are usually surrounded by angle brackets < and >.

Binding

This term refers to the association between a variable and a value for the variable that holds within some scope, such as the scope of a rule, function call, or procedure invocation.

Bound

A variable that has been assigned a value by the process of binding is said to be bound to that value.

Certainty factors (CFs)

Certainty factors are properties associated with attribute/value pairs and rules, commonly used to represent uncertainty, or likelihood. Certainty factors are usually automatically maintained by expert system shells.

Condition elements

The *left-hand side* of a rule in a rule-based system is sometimes expressed as a set of patterns (or templates) which are to be matched against the contents of the working memory; each such pattern is called a condition element. When a rule is instantiated, each condition element is found to match one element of the working memory.

Conflict set

A conflict set is a set of all instantiations generated by the match process during a recognise-act cycle. The process of conflict resolution selects one instantiation from the conflict set and fires it.

Cycle

A cycle is a single iteration of a loop. In production systems, an execution consists of iterated recognise-act cycles.

Domain reasoning network

A domain reasoning network (e.g. Figure 8 in Section 5.2) is an AND/OR tree associated with a knowledge base in *rule schema + rule body* by the following analogies:

1. Nodes in the tree correspond to factors in the knowledge base.
2. A rule schema such as *IF E1, ···, En THEN A* in the knowledge base corresponds to the arcs, which indicate the hierarchy among factors in the tree (shown in Figure 10).

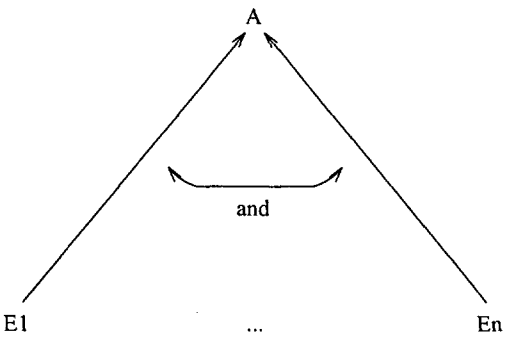

Figure 10: A rule schema

Filtering

The exclusion of either data (data filtering) or rules (rule filtering) from the match process for the sake of efficiency is termed filtering.

Fire

This term means to execute the set of actions specified in the right-hand side of an instantiation of a rule.

Forward chaining

Forward-chaining is a problem-solving method that starts with initial evidence of a problem and applies inference rules to generate new evidence until either one of the inferences satisfies a goal or no further inferences can be made. In forward-chaining production systems, the applicability of a rule is determined by matching the conditions specified on the left-hand side against the evidence currently stored in working memory.

Instantiation

Instantiation refers to a pattern or formula in which variables have been replaced by constants. In a production system, an instantiation is the result of successfully matching a rule against the working memory contents. It can be represented as an ordered pair of which the first member identifies the rule that has been satisfied, while the second member is a list of working memory elements that match the condition elements of the rule.

Interpreter

The interpreter is a part of a production system that executes the rules.

Match

In a production system the match process compares a set of patterns from the left-hand sides of rules against the data in data memory to find all possible ways in which the rules can be satisfied with consistent bindings (i.e. instantiations).

Mixed chaining

A search strategy which uses both backward and forward chaining during a single processing of a knowledge base is known as mixed chaining.

Object

An object is an entity in a programming system that is used to represent declarative knowledge and possible procedural knowledge about a physical object, concept, or problem-solving strategy.

One-input node

This term refers to a node in the RETE match algorithm network associated with a test of a single attribute of a condition element. It passes a token if and only if the attribute test is satisfied.

Pattern

A pattern is an abstract description of a datum that places some constraints on the value(s) it may assume, but needs not specify it in complete detail.

Temporal redundancy

This term refers to the tendency of rule systems to make relatively few changes to the working memory, and hence to the conflict set, from one recognise-act cycle to the next. The RETE algorithm exploits temporal redundancy so as to avoid unnecessary recomputations of all matches.

Two-input node

Two-input nodes are nodes in the RETE algorithm network that merge matches for a condition element with matches for all preceding condition elements.