502

KNOWLEDGE - BASED RESOLUTION SYSTEM

Dissertation Submitted in Partial Fulfilment
of the Requirements for the Degree of
MASTER OF PHILOSOPHY

1982

by

VIJAY KUMAR SINGHAL

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067

CERTIFICATE

The research work entitled "KNOWLEDGE-BASED RESOLUTION SYSTEM" and embodied in this dissertation has been carried out in the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi - 110 067.

The work is original and has not been submitted in part or full for any other degree or diploma of any University.

R. Sadananda
(DR. R. SADANANDA)
Supervisor

(VIJAY KUMAR SINGHAL)
Student

(PROF. DILIP K. BANERJI)
Dean

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067

DEDICATED

To my late grand-parents

'AMMA' and 'BABA'

whose blessings are always with me.

ACKNOWLEDGEMENTS

I am deeply grateful to my supervisor Dr. R. Sadananda, Associate Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi, whose frank co-operation and efficient guidance enabled me to complete this work successfully. Throughout my stay in this School, I enjoyed his valuable suggestions and affectionate nature.

I sincerely express my gratitude to Prof. Dilip K. Banerji, Dean and Prof. N.P. Mukherjee, Ex-Dean of the School, for providing me with all the facilities during my study and project work.

I take this rare opportunity to thank all of my friends and class-fellows for their help and encouragement at various stages of my study. I also thank whole of the faculty and the staff, especially Mr. C.A. Thakur, Section Officer of the School, for their co-operation.

Next, I express my deep respects to my parents and elder brothers, Dr. Ram Murti Singhal and Shri Govind Swaroop Singhal, for their love, assistance and support given to me throughout my student-life.

My special thanks are also due to Mr. S.K. Sapra for typing this dissertation with care and patience.

VIJAY KUMAR SINGHAL

CONTENTS

CHAPTERS			Page No.
1.	INTRODUCTION		1
2.	FIRST - ORDER PREDICATE CALCULUS		4
	2.1	Syntax	5
	2.2	Semantics	6
	2.3	Variables and Quantifiers	7
	2.4	Validity and Satisfiability	8
3.	THE RESOLUTION AND ITS REFINEMENTS		11
	3.1	Basic Resolution System	11
		3.1.1 Substitution and Unification	11
		3.1.2 Resolvents	13
		3.1.3 The Resolution Principle	14
		3.1.4 Soundness and Completeness of Resolution	16
	3.2	Refinements of Resolution	16
		3.2.1 Simplification Strategies	17
		3.2.2 Refinement Strategies	19
		3.2.3 Ordering Strategies	22
4.	THE KNOWLEDGE - BASE		24
	4.1	The Knowledge Level	25
	4.2	The Nature of Knowledge	26
	4.3	Representation of Knowledge	29
5.	DESIGNING THE KNOWLEDGE - BASE		32
	5.1	An Overview	32
	5.2	Design of the Knowledge-Base	37
		5.2.1 Thumb Rules	37
		5.2.2 The Data-Base	40
		5.2.3 Global Declarations	43

CHAPTERS		Page No.

6.	CONCEPTFERENCE	44
	6.1 Coding and Attributes	44
	6.2 Concept Formation	46
	6.3 Inference Phase	47
	6.4 An Example	48
7.	THE KNOWLEDGE - BASED RESOLUTION SYSTEM	51
	7.1 The Algorithm	51
	7.2 Explanation	52
	7.3 Examples	54
8.	CONCLUSIONS	61
	REFERENCES	63

CHAPTER - 1

INTRODUCTION

In human beings, the ability of performing certain tasks, such as solving puzzles, proving theorems, playing games, writing programs or even driving a car, is called 'intelligence'. In other words, such a type of tasks require intelligence. 'Artificial Intelligence' is an area of study in Computer Science which concerns with realising intelligence in this sense by machines.

Here we are concerned with 'problem solving'. Much of the Artificial Intelligence research has concentrated on it. In its broadest sense, problem solving encompasses all of Computer Science because any computation task can be regarded as a problem to be solved [14]. A way to learn about intelligence is to study the problem solving behaviour of a human being.

Mechanical theorem proving is an important subject in Artificial Intelligence. It has long been man's ambition to find general decision procedures to prove theorems. Mechanical theorem proving techniques have been applied to many areas, such as program analysis, program synthesis, deductive question-answering systems, problem-solving systems and Robot technology. These techniques are finding increasing number of applications in real life situations [5].

A major breakthrough in mechanical theorem proving was made by Robinson [17] in 1965. He developed a single inference rule, called the 'resolution principle', which

was shown to be highly efficient and easily implementable on Computers. Since then many improvements of the resolution principle have been made.

The application of Data Base concepts in the field of Artificial Intelligence is increasing rapidly in recent years. These data-bases are conceptually different and therefore they are knowledge-bases,as they are popularly known. This work is an effort to develop a resolution system based on the application of the knowledge-base to make the implementation of the resolution principle easier, more efficient, less-expensive and time-saving.

Formal and logical models are becoming important tools of Computer Science. In particular, there seems to be quite a few concepts and models in symbolic logic which can be used for modelling problems in and around Computer Science. In our work, we have used the first-order predicate calculus due to its simplicity and power of expression. We have tried to give a brief introduction of first-order logic in Chapter-2.

In Chapter-3, we have reproduced the resolution principle and some search strategies, which are also being implemented in our knowledge-based resolution system.

We have given a brief description of what we mean by the 'knowledge-base', in Chapter-4. In Chapter-5, we make a brief survey of knowledge-bases so far implemented by different programs, and give the design of the knowledge-base of our system.

As a particular domain of problems, upon which we wish to apply the system developed by us, we have chosen a system, called 'Conceptference', which has been developed by Sadananda and Mahabala [18]. In Chapter-6, we give a brief introduction of Conceptference.

We have given the text of the algorithm of the knowledge-based resolution system, alongwith its explanation and two examples of its application, in Chapter-7. In the last chapter, we have made our comments and conclusions on the work done and also discussed about the further possibilities arising out of our efforts.

CHAPTER-2

FIRST-ORDER PREDICATE CALCULUS

Solutions to many problems might require logical analysis. For this purpose, we need some kind of formal language, in which we can state premises and make valid logical deductions. The 'first-order logic' or 'first-order predicate calculus' is a system of logic in which it is possible to express much of mathematics and many statements of everyday language.

Consider the following examples of deduction of statements :

1) Ram is a man.
2) Every man is mortal.
3) Therefore, Ram is mortal.

Let us denote 'x is a man' by MAN(x) and 'x is mortal' by MORTAL(x). Then, we can represent the above statements as :

1) MAN(Ram)
2) $\forall x$, MAN(x) $\Rightarrow$ MORTAL(x)
3) MORTAL(Ram)

where symbol '$\forall$' means 'for all' and '$\Rightarrow$' means 'implies'.

In this example, 'MAN' and 'MORTAL' are 'predicates', which can have values 'True' or 'False'; 'x' is a 'variable', which can have any value in its range; and 'Ram' is a 'constant', which denotes a particular value of variable x.

This system has a number of rules of inference that allow us to make valid logical deductions of new

statements from a set of given statements. Because of its generality and logical power, the predicate calculus is very suitable for a wide range of applications. Now we give the syntax and semantics of the first-order predicate calculus.

2.1 SYNTAX

The basic alphabet of first-order predicate calculus consists of the following set of symbols :

(1) Punctuation marks : ',', '(', ')'

(2) Logical Symbols : '$\sim$'(not), '$\Rightarrow$'(implies), '$\wedge$'(and), '$\vee$'(or)

(3) Variables : x, y, z, ...

(4) Constants : a, b, c, ...

(5) Function letters : f, g, h, ...

(6) Predicate letters : P, Q, R, ...

Using these symbols we can construct many expressions. Some classes of such expressions are defined as follows :

1. Terms :

(a) A constant is a term.

(b) A variable is a term.

(c) If f is a function symbol and $t_1, \ldots, t_n, n \geq 1$, are terms, then $f(t_1, \ldots, t_n)$ is a term.

(d) No other expressions are terms.

2. Atomic formulas (or Atoms) :

(a) A predicate symbol is an atomic formula.

(b) If $t_1, \ldots, t_n, n \geqslant 1$, are terms and P is a predicate symbol, then the expression $P(t_1, \ldots, t_n)$ is an atomic formula.

(c) No other expressions are atomic formulas.

3. Well-formed formulas (wffs) :

(a) An atomic formula is a wff.

(b) If A is a wff, so is $\sim A$.

(c) If A and B are wffs, so is $A \Rightarrow B$ $\{$ and hence, also $A \wedge B$ and $A \vee B$, because $A \wedge B = \sim(A \Rightarrow \sim B)$ and $A \vee B = (\sim A) \Rightarrow B \}$.

2.2 SEMANTICS

To make well-formed formulas meaningful, we have to interpret them in terms of 'domain' and 'assignments' of truth values to constants, function symbols and predicate symbols, occuring in a formula in the first-order logic. The well-formed formulas are said to have the value 'T' or 'F' depending on whether the assertions are true or false over the domain.

Definition : An 'interpretation' or a 'model' of a well-formed formula W in the first-order logic consists of a non-empty domain D and an assignment of values to each constant, function symbol and predicate symbol occuring in W as follows :

(a) For every constant symbol, we assign an element in D.

(b) For every function symbol, we assign a mapping from D^n to D, where $D^n = D \times D \times \ldots \times D$, n times.

(c) For every predicate symbol, we assign a mapping from D^n to $\{T, F\}$.

The value of a non-atomic well-formed formula can be computed recursively from the values of its component formulas. In this computation, we use the following definitions :

(1) If X is any wff, then $\sim X$ (read 'not X') has value T, if X has value F and $\sim X$ has value F, when X has value T.

(2) If X_1 and X_2 are any wffs, then the values of $X_1 \vee X_2$, $X_1 \wedge X_2$ and $X_1 \Rightarrow X_2$ are given by the following 'truth table' :

X_1	X_2	$X_1 \vee X_2$	$X_1 \wedge X_2$	$X_1 \Rightarrow X_2$
T	T	T	T	T
T	F	T	F	F
F	T	T	F	T
F	F	F	F	T

2.3 VARIABLES AND QUANTIFIERS

Sometimes to emphasize the domain D, we speak of an interpretation of the well-formed formula over D. When we evaluate the truth value of a formula in an interpretation over the domain D, $(\forall x)$ will be interpreted as 'for all elements x in D' and $(\exists x)$ as 'there exists an element x in D'. The sign '$\forall$' is called the 'universal quantifier' and the corresponding variable, here x, is called a

'universally quantified variable'. The sign '$\exists$' is called 'existential quantifier' and the corresponding variable is called an 'existentially quantified variable'.

For interpretation of a well-formed formula containing quantifiers over a domain D, in addition to above mentioned rules, we use the following two rules to evaluate the well-formed formula :

(1) $(\forall x)$ W is evaluated to T if the truth value of W is evaluated to T for every x in D, otherwise it is evaluated to F.

(2) $(\exists x)$ W is evaluated to T if the truth value of W is T for at least one x in D, otherwise it is evaluated to F.

By truth table method we can show that $\sim(\forall x)$ $P(x)$ always has the same truth value as does $(\exists x)(\sim P(x))$. Similarly, $\sim(\exists x)\ P(x)$ and $(\forall x)\ (\sim P(x))$ are equivalent.

2.4 VALIDITY AND SATISFIABILITY

<u>Definition</u> : A well-formed formula is called 'valid' if it has the value T for its all interpretations.

Thus the well-formed formula $P(a) \Rightarrow (P(a) \vee P(b))$ is valid because it always has the value T regardless of its interpretations, as it may be shown by the truth table. By truth table method we can always determine the validity of a well-formed formula that does not contain any quantifier. When quantifiers occur we can not always determine the validity of that well-formed formula, because there does not

exist any general method to evaluate the value of all of the infinite interpretations represented by the quantifiers.

The validity of certain kinds of formulas containing quantifiers can be determined. It has been shown that if a well-formed formula is infact valid then a procedure exists for verifying its validity.

Definition : A well-formed formula W is said to be 'satisfiable' or 'consistent', if there exists an interpretation I of W such that W has a value T under this interpretation. The interpretation I is said to be satisfying W.

Definition : A well-formed formula W is said to be 'unsatisfiable' or 'inconsistent' if there exists no interpretation satisfying W.

If the same interpretation satisfies each wff in a set of wffs, then we say that this interpretation satisfies the set of well-formed formulas.

Definition : A well-formed formula W 'logically follows' from a set S of well-formed formulas, if every interpretation satisfying S also satisfies W.

A 'proof', that a well-formed formula W is a logical concequence of a given set S of wffs, is a demonstration that W logically follows from S. Given an arbitrary wff W and an arbitrary set S of wffs, unfortunately there can exist no effective procedure which may always decide whether W logically follows from S or not. If W does follow from S, then there are procedures that will always

detect it, but if W does not follow from S, these procedures will not always be able to find this fact.

<u>Theorem</u> : A well-formed formula W logically follows from a set S of well-formed formulas, if, and only if, the set $S \vee (\sim W)$ is unsatisfiable.

The above theorem is very useful to determine whether or not a well-formed formula W follows from a set S of wffs. To show that W logically follows from S, we will show that $S \vee (\sim W)$ is unsatisfiable. In order to show that a set S of well-formed formulas is unsatisfiable, we must show that there exists no interpretation for which each of the wffs in S has truth value T.

Fortunately, some very powerful procedures do exist to perform this task. These procedures demand that well-formed formulas in a set be put in a special form called 'clause form'. For the procedure to convert a well-formed formula into clause form readers are referred to [10, 14].

CHAPTER-3

THE RESOLUTION AND ITS REFINEMENTS

3.1 BASIC RESOLUTION SYSTEM

Some of the purposes of programming a computer to prove theorems concern Artificial Intelligence and deduction. Writing a theorem proving program, which uses mathematical logic allows us to study deductions in its purest form. Deduction is important because it plays a major role in solving many kinds of problems and not only in Mathematics. For this purpose the programmer may develop powerful, natural, intuitive inference rules to which heuristics can be added easily [20].

The 'resolution principle' is such a powerful q and natural rule of inference, which was developed by Robinson [17]. Roughly speaking, the resolution principle draws the most general possible conclusion from two given statements. The conclusion and the two statements generally contain variables. The resolution principle is more natural, more intuitive and easier for people to use than are the inference rules used by earlier programs. Furthermore, it is easier to think of heuristics to add to the resolution principle [20]. Before discussing the resolution principle, we discuss below the unification and resolvents.

3.1.1 <u>Substitution and Unification</u> :

The process called 'unification' is a basic part of the formal manipulations performed in obtaining resolvents. We shall refer as 'literal' an atomic formula or its negative.

The terms of a literal can be variable letters, constant letters or expressions consisting of function letters and terms. A 'substitution instance' of a literal is obtained by substituting terms for variables in the literal. For example, a substitution instance of the literal $P(x,f(y),b)$ could be $P(a,f(g(z)),b)$ in which variable x has been substituted by a constant 'a' and y has been substituted by a term $g(z)$.

<u>Definition</u> : A 'substitution' θ is a finite set of ordered pairs $\{(t_1,v_1),\ldots,(t_n,v_n)\}$, where the pair (t_i,v_i) means that the variable v_i is throughly substituted by the term t_i and no two variables are the same, i.e. $i \neq j \Rightarrow v_i \neq v_j$.

For example, the substitution used in above example to obtain an instance of $P(x,f(y),b)$ is $\theta = \{(a,x),(g(z),y)\}$.

<u>Definition</u> : The 'composition' of two substitutions α and β, denoted by $\alpha\beta$, is that substitution obtained by applying β to the terms of α and then adding those pairs of β, whose variables do not occur among the variables of α.

For example the composition of $\alpha = \{(g(x,y),w)\}$ and $\beta = \{(a,x),(b,y),(c,w)\}$ is $\alpha\beta = \{(g(a,b),w),(a,x),(b,y)\}$. If a substitution θ is applied to every member of a set $\{L_i\}$ of literals, we denote the set of substitution instances by $\{L_i\}_\theta$.

<u>Definition</u> : A set of literals $\{L_i\}$ is said to be 'unifiable' if there exists a substitution θ such that $L_{1\theta} = L_{2\theta} = \ldots = L_{n\theta}$. In such a case, the substitution θ is said to be a 'unifier' of $\{L_i\}$.

For example, $\theta = \{(a,x),(b,y)\}$ unifies $\{P(x,f(y),c),P(x,f(b),c)\}$ to yield $\{P(a,f(b),c)\}$.

<u>Definition</u> : A unifier λ of a set of literals $\{L_i\}$ is said to be the 'simplest' or 'most-general unifier' of $\{L_i\}$ if, for every unifier θ of $\{L_i\}$, there exists a substitution δ such that $\{L_i\}_{\lambda\delta} = \{L_i\}_\theta$.

The common instance produced by a most-general unifier is unique except for alphabetic variants. For example, the substitution $\lambda = \{(b,y)\}$ is the most-general unifier for the set of literals in the above mentioned example.

There is an algorithm, called the 'unification algorithm' that produces the most-general unifier λ for a unifiable set $\{L_i\}$ of literals and reports failure when the set is not unifiable [3,17].

<u>Definition</u> : If a subset of the literals in a clause $\{L_i\}$ is unifiable by the most-general unifier λ , then we call the clause $\{L_i\}_\lambda$ a factor of $\{L_i\}$. A clause may have more than one factor but only finitely many.

3.1.2 <u>Resolvents</u> :

We can now define the process by which we can sometimes infer a new clause, called a 'resolvent', from two given clauses, called the 'parent clauses'.

<u>Definition</u> : Let the parent clauses be $\{L_i\}$ and $\{M_j\}$ and no variable is common in the two clauses. Suppose that $\{l_i\} \subseteq \{L_i\}$ and $\{m_j\} \subseteq \{M_j\}$ be two subsets of $\{L_i\}$ and $\{M_j\}$ respectively, such that a most-general unifier λ exists

for the set $\{l_i\} \cup \{\sim m_j\}$. Then we say that the clauses $\{L_i\}$ and $\{M_j\}$ 'resolve' and that the new clause

$$[\{L_i\}-\{l_i\}]_\lambda \cup [\{M_j\}-\{m_j\}]_\lambda$$

is a 'resolvent' of the two clauses.

The resolvent is an 'inferred clause' and the process of forming a resolvent from two parent clauses is called 'resolution'. If two clauses resolve they may have more than one resolvents because there may be more than one ways to choose $\{l_i\}$ and $\{m_j\}$. In any case, they can have at most a finite number of resolvents.

As an example, consider the two clauses,

$$\{L_i\} = P(x,f(a)) \vee P(x,f(y)) \vee Q(y)$$

$$\text{and } \{M_j\} = \sim P(z,f(a)) \vee \sim Q(z)$$

There are four different resolvents of these two clauses, as following :

(1) $P(z,f(y)) \vee Q(y) \vee \sim Q(z)$

(2) $P(z,f(a)) \vee Q(a) \vee \sim Q(z)$

(3) $Q(a) \vee \sim Q(z)$

(4) $P(x,f(a)) \vee P(x,f(z)) \vee \sim P(z,f(a))$

If we resolve $\sim P(a) \vee Q(a)$ and $P(a)$, we get $Q(a)$, which is the same thing as inferred by $P(a) \Rightarrow Q(a)$ (which is equivalent to $\sim P(a) \vee Q(a)$) and $P(a)$. Thus we see that the resolution is nothing but a general rule of inference.

3.1.3 <u>The Resolution Principle</u> :

As indicated in section 2.4, we desire to be able to find a proof that a well-formed formula W in the predicate calculus logically follows from a set S of wffs. For this,

it is sufficient to show that the set $S \cup \{\sim W\}$ is unsatisfiable. The process showing the unsatisfiability of a set of clauses is called 'refutation process'.

It has been shown that if we add in an unsatisfiable set S of clauses more clauses generated by the resolution between the pairs of S, then the new set will still be unsatisfiable and if we continue to perform resolutions on the sets thus obtained, then we will eventually generate the empty clause, denoted by 'NIL', showing the unsatisfiability [3,10,14,17].

Let us denote by R(S) the union of S with the set of all resolvents obtainable between the pairs of clauses in S, and by $R^2(S)$ we mean R(R(S)) etc. Then, if S is unsatisfiable, we are guaranteed for some finite n that the empty clause will be contained in $R^n(S)$.

The graph showing the process of resolution is called the 'refutation graph'. The refutation graph for the unsatisfiable set $\{B(x), \sim B(x) \vee C(x), \sim C(a) \vee D(b), \sim C(c) \vee E(d), \sim D(x) \vee \sim E(y)\}$ is shown in fig.-1.

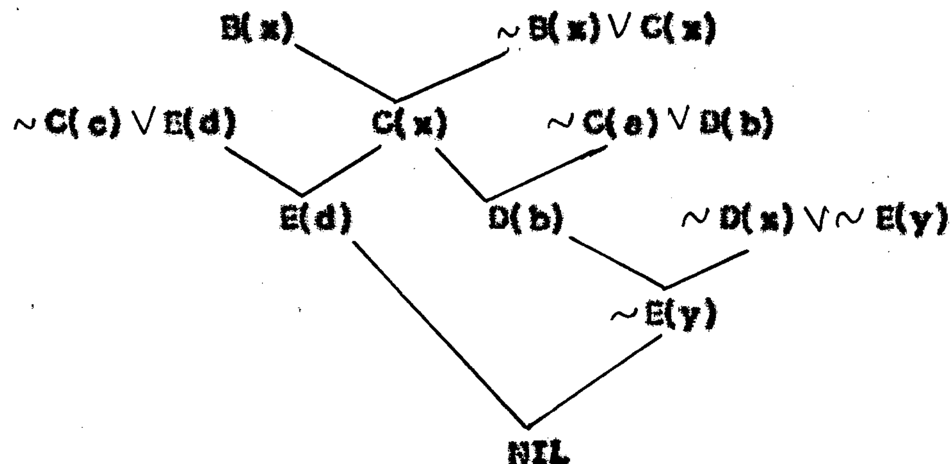


fig.-1

3.1.4 Soundness and Completeness of Resolution :

Soundness and completeness of the resolution principle was established by Robinson [17]. The resolution principle is sound, because a clause logically implies each of its factors and two clauses, taken together, imply each of their resolvents [20]. It is also called complete, because if the original set of clauses is unsatisfiable, then the empty clause will eventually be produced in a finite number of applications of the resolution principle and if the empty clause is ever produced then the original set of clauses must have been unsatisfiable. Below we state the completeness theorem without proof.

Theorem (Completeness of the resolution principle) : A set S of clauses is unsatisfiable if, and only if, there is a deduction of the empty clause from S.

The proof of the theorem is available in the literature [3,10,14,17]. The theorem implies that the generation of empty clause from a set of clauses by resolution is a necessary and sufficient condition for the set being unsatisfiable. Therefore, to establish the unsatisfiability of a set of clauses we have to perform only a finite number of applications of the resolution.

3.2 REFINEMENTS OF RESOLUTION

In section 3.1.3, we have seen that our search for a reputation starts with a set S of clauses and adding to this set all of the resolvents from the pairs of clauses in S to produce the set R(S). Next, all of the resolvents

between the pairs of clauses in R(S) will be added to produce the set $R(R(S))=R^2(S)$, and so on, until we get an empty clause. This type of search is usually impractical because the sets $R(S), R^2(S), \ldots$ grow too rapidly. At this place, we will consider some practical ways in which the basic procedure can be altered keeping soundness and completeness preserved.

First, we will define some terminology. By a 'restriction' of procedure P we mean a procedure which generates a subset of deductions of P for all given sets and a proper subset of deductions for at least one given set. A 'refinement' of procedure P is either a restriction of P or a procedure that simply reorders the development order of deductions associated with P. Sometimes, we refer to a re-ordering refinement as a 'strategy'.

Practical proof procedures depend on search strategies to speed up a search. These strategies are of three types: simplification strategies, refinement strategies and ordering strategies. Here we present a brief discussion of those strategies which are easily applicable in the knowledge-based resolution system described in this dissertation.

3.2.1 Simplification Strategies :

Sometimes a set of clauses can be simplified just by elimination of certain clauses or literals in the clauses. Resolution is the most eligible procedure for such elimination rules. These simplification are

such that the simplified set of clauses is unsatisfiable if, and only if, the original set is unsatisfiable. We consider them here briefly.

(a) Pure Literal Elimination : A literal L is called 'pure' in a given set of clauses if it has no instance which is complementary to an instance of another literal in the set. Clauses containing pure literals can safely be removed at any time in the process and this removal preserves the unsatisfiability.

(b) Tautology Elimination : A clause is called to be a 'tautology', if it contains a literal and its complement both. Such clauses may be eliminated in general resolution, however not always [10], without hitting the unsatisfiability of the set of clauses. The clause like $P(a) \vee Q(b) \vee \sim Q(y)$ may at once be removed from the set.

(C) Subsumed Clause Elimination :

Definition : A clause $\{L_i\}$ 'subsumes' a clause $\{M_j\}$ if there exists a substitution θ such that $\{L_i\}_\theta \subseteq \{M_j\}$. In such a case clause $\{M_j\}$ is called a 'subsumed clause'.

For example, $P(x)$ subsumes $P(y) \vee Q(z)$.

A clause in S subsumed by another clause in S can be eliminated without affecting the unsatisfiability of the rest of the set. Eliminating clauses subsumed by others frequently leads to substantial reductions in the number of resolutions to be performed for finding a proof.

The subsumption algorithm, i.e. an algorithm that tests whether or not a clause L subsumes another clause

M, is given in [5]. In general, tautologies may be eliminated as soon as they are generated during a search, but subsumed clauses should be eliminated only after each 'level' has been completed [14].

(d) Truth Evaluation Elimination : Some authors introduce elimination rules based on evaluation of literal in some interpretations, because it is sometimes possible to easily evaluate the truth value of literals. The rule is stated as : One can eliminate the derived clause which contains a literal that is evaluated to 'True' in the interpretation and eliminate a literal in a derived clause if it is 'false in the interpretation.

3.2.2. Refinement Strategies :

Refinement strategies state that not all of the possible resolutions need to be performed in order to find a refutation. In other words, only resolutions between clauses meeting certain criterion need to be performed. We shall denote by $R_c(S)$ the union of S with the set of all resolvents between the pairs of S meeting criterion C. We note that $R_c(S) \subseteq R(S)$. Such a strategy is called 'resolution relative to C' and we compute $R_c^2(S)$ etc., until for some n, $R_c^n(S)$ contains the empty clause.

The potential value of a refinement strategy is that fewer resolutions need be performed at each level. A refinement strategy is useful only if its use reduces the total search effort including the effort needed to test against the criterion C [14]. Here, we consider one major

refinement strategy that is very much useful for the knowledge-based resolution system of this dissertation and is also easily adoptable on computers.

<u>Ancestry-Filtered Form Proofs</u> :

A 'resolution proof graph' or 'refutation graph' is a structure of nodes, each node being a clause. Nodes in the graph having no ancesters are called 'tip nodes', which in fact correspond to clauses in S, called 'base-clauses'. The node in the graph having no descendants is called the 'root node' and it corresponds to the clause that is proved by the graph, possibly the empty clause.

<u>Definition</u> : A refutation graph is called to be in 'Ancestry-Filtered' (AF) form if each node in the graph corresponds to either of the following :

(1) a base-clause,

(2) an immediate descendant of a base-clause,

(3) an immediate descendant of two non-base clauses A and B such that A is an ancestor of B.

A base clause C in an AF-form graph is called a 'top node' if every other node in the tree is either a base-clause or a descendant of C. The theorem given below claims that an AF-form refutation graph always exists for any unsatisfiable set of clauses. Therefore, a refinement strategy based on searching for AF-form refutation graph is complete.

<u>Theorem</u> : Let G(NIL) be some refutation graph for an unsatisfiable set S of clauses and let C be a clause in S

occuring in G(NIL). Then a refutation graph G'(NIL) in AR-form exists for S with C as a top node of G'(NIL).

Proof of the above theorem can be found in [22] as stated in [14]. Trusting on the above theorem we can restrict our search of proof for the search of AR-form of the refutation graph. We note that the above theorem also gives us the freedom of selecting a top node for the AR-form refutation graph. We can utilize the above theorem as follows.

The top node must be one that occurs in some refutation graph. We select it from some subset $K \subseteq S$ that is certain to contain only clauses that occur in some refutation graph. For example, K might be those clauses originating from the negation of the inference to be proved. Thus to resolve a pair (A,B) of clauses, they must satisfy the following criterion :

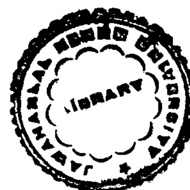
One member of the pair belongs to S and the other is a descendant of the top clause,

OR

One member of the pair is an ancestor of the other.

For example, the AR-form of the refutation graph of fig.-1 is given in fig.-2, where the clause B(x) is taken as the top clause.

If we denote by $R_{AR}(S)$ the union of S with the set of all resolvents between pairs of S allowed by AR-form strategy, then by the above theorem we are guaranteed that,



TH-1217

if S is unsatisfiable, then there will exist some n such that empty clause belongs to $R^n_{AP}(S)$.

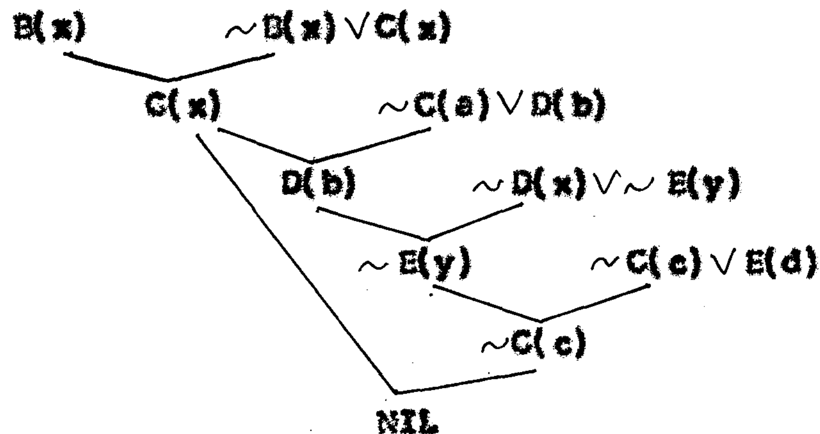


fig.-2

3.2.3 Ordering Strategies :

Within the range of resolution choices provided by the various refinement strategies, it is sometimes possible to search for a refutation by carefully ordering the resolutions to be performed. Ordering strategies do not prohibit any particular types of resolutions, but merely provide guidance about which one should be performed first.

Two very efficient ordering strategies are the 'unit-preference strategy' and the 'fewest-components strategy'. The unit-preference strategy makes use of unit, (or one-literal) clauses, and unit factors which are unit clauses. One performs resolution operations where at least one parent clause is a unit clause, i.e. 'unit resolutions', before it is normally be performed under the

given search plan. It is an obvious strategy, since the object of generating resolvents is to produce the empty clause.

The fewest-component strategy orders resolutions according to the lengths of the resolvents produced. Thus, those two clauses that will produce the shortest resolvent are resolved first. This strategy is some what expensive to apply because of its lengthy computation. In our knowledge-based resolution system, we will use this strategy in a slightly changed manner, i.e. among all possible resolvents of a selected pair of clauses, we will consider that one first which is of the shortest length. So, we call this strategy as the 'shortest-length strategy'.

The above two ordering strategies are made use in our system in addition to simplification strategies and AF-form proof strategy.

CHAPTER-4

THE KNOWLEDGE - BASE

The word 'knowledge' is a key to much of modern theorem proving. Some-how, we want to use the latest available knowledge accumulated by the humans to help direct search for the proof. The use of knowledge and built-in procedures partially eliminates the need for long list of axioms, which tend to slow up proofs and excessive amount of memory [4]. Such knowledge must be organised in a way that is easy to use and change.

We store information and knowledge in a special type of data base, called 'knowledge-base', and process that information to obtain other informations and interrogate the knowledge-base when necessary to answer questions. The central idea behind this is that facts are stored about objects, i.e. terms arising in a proof, rather than predicates. Also knowledge is stored about concepts in form of procedures or lists or other data-structures.

According to Davis [6], all theorem proving programs have at least two components: the 'Inference Engine' and the 'Knowledge-Base', as shown in fig.-3. The knowledge-base is the programs store of task specific knowledge that makes possible high performance, and the inference engine is an interpreter that uses the knowledge-base to

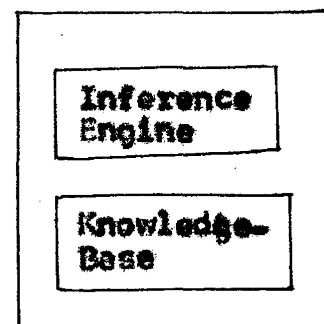


fig.-3

solve problem at hand repeatedly selecting knowledge-sources from the knowledge-base and applying them.

The knowledge-base is theoretically, if not physically, a different concept from that of data-base. While in data-base we store raw data and information, in knowledge-base we store what we call 'knowledge', in general sense.

It is not always possible to give a formal definition of knowledge. In first sight, we can say that 'knowledge-base' consists of the facts, rules, actions and anything that is not just a data but is used in decision-making. To understand the knowledge completely, we will have to turn to a paper by Newell [13], in which he has tried to explain the concept of 'knowledge' with reference to what he calls the 'knowledge level'.

4.1 THE KNOWLEDGE LEVEL

The system at the knowledge level is called the agent, which is a composition of a set of actions, a set of goals and a body. The medium at the knowledge level is the knowledge. The agent processes its knowledge to determine the actions to be taken and actions are selected to attain the agent's goals. When we say that a system is at the knowledge level, we want to say that the system has some knowledge and some goals and we can rely that the system will do whatever is within its powers of knowledge to obtain the goals.

The behavioural law that governs an agent and permits of it behaviour is the 'rationality principle', which is formulated as follows :

Principle of Rationality : If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action.

This principle is silent about what happens when the principle applies to more than one actions for a given goal or to more than one goal in a given situation. These short-comings can be covered by adding two auxiliary principles as following :

Equipotence of acceptable actions : For given knowledge, if action A_1 and action A_2 both lead to a goal G, then both actions are selected.

Preference to joint satisfaction : For given knowledge, if goal G_1 has the set of selected actions $\{A_{1i}\}$ and goal G_2 has the set of selected actions $\{A_{2j}\}$, then the effective set of selected actions is the intersection of $\{A_{1i}\}$ and $\{A_{2j}\}$.

4.2 THE NATURE OF KNOWLEDGE

Knowledge is defined to be the medium at the knowledge level. This can be put into a complete definition given by Newell[13] as follows :

Knowledge : Whatever can be ascribed to an agent, such that its behaviour can be computed according to the principle of rationality.

Accordingly, knowledge is to be characterized entirely functionally, i.e. in terms of what it does, and not physically, i.e. in terms of physical objects, with particular properties and relations, because knowledge is not just a collection of symbolic expressions plus some static organisation, it requires both processes and data-structures.

<u>How it works</u> : Fig.-4, shows the situation which involves an observer and an agent. The observer treats the agent as a system at the knowledge level, having knowledge K and goals G with possible actions A, so that he can predict the agent's actions using the principle of rationality. What the agent really has is a symbol system S, that permits it to carry out the calculations of what action it will take, because it has knowledge K and goals G with actions A in the environment. Thus the agent has the knowledge by virtue of a system that provides the ability to act as if it had the knowledge. The total system runs without being any physical structure that is the knowledge.

Rieger [16] has defined the knowledge to be of two types : 'static' and 'dynamic'. For this, he gives example of a bicycle (fig.-5). The static knowledge about bicycle is : hight, shape, relative position of components, characteristics of components, price, colour etc., and the dynamic knowledge is : function of the whole

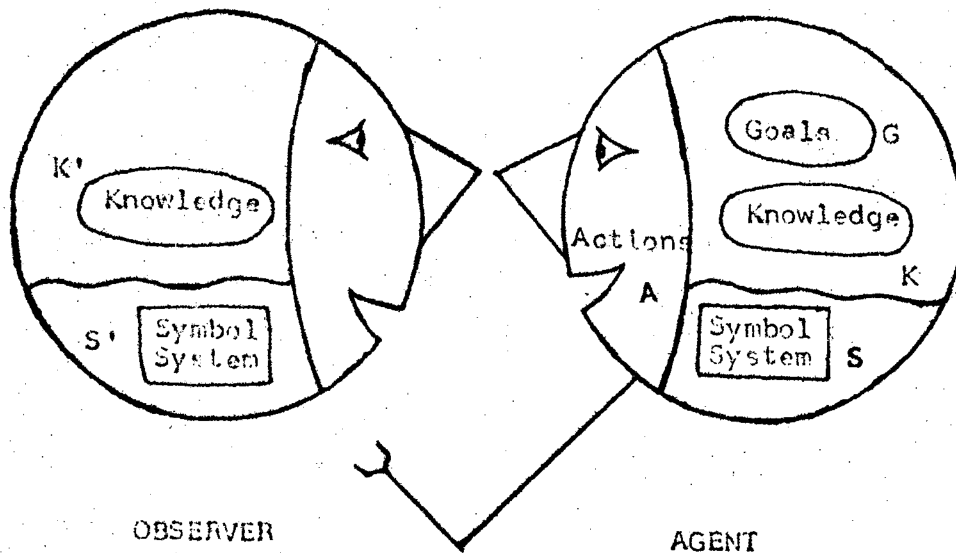


fig.-4

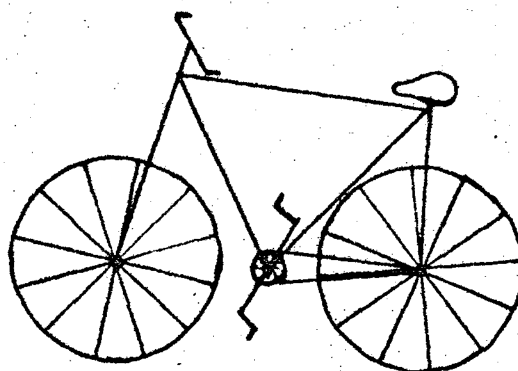


fig.-5

device, i.e. to translate the up-down pumping of legs into the forward motion, and function of each component. Both types of knowledge, he says, are not independent, but they are distinguishable.

In a broader sense, we can define the knowledge to be of two types : 'concrete' and 'abstract'. The concrete knowledge concerns with the individual phenomena, entities or relationships in the model of the reality, and the abstract knowledge concerns with the interpretations of concrete informations and by which we could draw inferences and conclusions about other facts.

4.3 REPRESENTATION OF KNOWLEDGE

There are myraid issues concerning the design, construction, maintenance and refinement of expert knowledge-bases. The hardest ones are the design of the representation appropriate to the processing that will be carried out and the acquisition of this knowledge[21]. Designing a uniform representation facilitates maintaining an evolving knowledge-base and permits checks of redundancy and consistency. Feigenbaum[8] discusses some of the social factors that influence the design of a knowledge representation. In fact, each of us is attacking the problem of representing a knowledge of how things work in the world, then using that knowledge to understand perceptions[16].

Recalling Newell[13], the principle of rationality provides a general functional equation for knowledge. The problem of agent is to find systems and symbol level that are solutions to this functional equation and hence can serve as representation of knowledge. Logics provide solutions to this problem. We can find many situations in which agent's knowledge can be characterized by an expression in a logic and from which we can derive actions to take.

A logic is not the knowledge itself, it is just a representation of the knowledge, that is a structure at the symbol level. If we are given logical expressions $\{L_i\}$, say, then by saying that the agent knows $\{L_i\}$, we mean to say that the agent knows all that he can infer from the conjunction of $\{L_i\}$.

The theory of knowledge level provides a definition of representation as 'a symbol system that encodes the body of knowledge'. Without providing a theory of representation it suggests that a useful way of thinking about representation is the idea that 'the representation is the knowledge plus its access'. In other words, the representation consists of a system for providing access to a body of knowledge.

Concrete knowledge can easily be represented using some kind of data structures, for example, lists, tables etc. For abstract knowledge, two approaches exist.

First, the declarative approach, which is based on formules using some mathematical notation, for example, predicate calculus, and second, the productive approach, in which the knowledge is built into procedures which deal with modifying the knowledge-base and answering queries.

Artificial Intelligence contains many systems that are not logics, but can be used as representation of the knowledge. Similarly, many concepts of science subjects bring some representational structures very close to the needs of Artificial Intelligence. Good examples are algebraic mathematics and chemical notations [13].

CHAPTER - 5

DESIGNING THE KNOWLEDGE - BASE

5.1 AN OVERVIEW

In Chapter-4, we have given an introduction of knowledge, its representation and knowledge-base. Here we wish to discuss briefly some approaches of the representation of knowledge used by different programs.

Schank and Abelson [19] have been investigating techniques to represent the type of larger-stereo typed patterns of 'how-to-do-it' knowledge, called 'scripts', to be used for understanding multisentence short stories.

Minsky [11] introduces 'frame' as a data-structure to represent a stereo-typed situation, like being in a certain kind of living room or going to a child's birth-day party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame, some is about what one can expect to happen next, some is about what to do if these expectations are not confirmed. Collections of related frames are linked together into 'frame systems'. The effect of important actions are mirrored by trans-formations between the frams of a system.

Automatic theorem provers of Ballantyne and Bledsoe [1] use the knowledge-base in their programs and implement the procedural representation of knowledge, in which they state rules in form of tables.

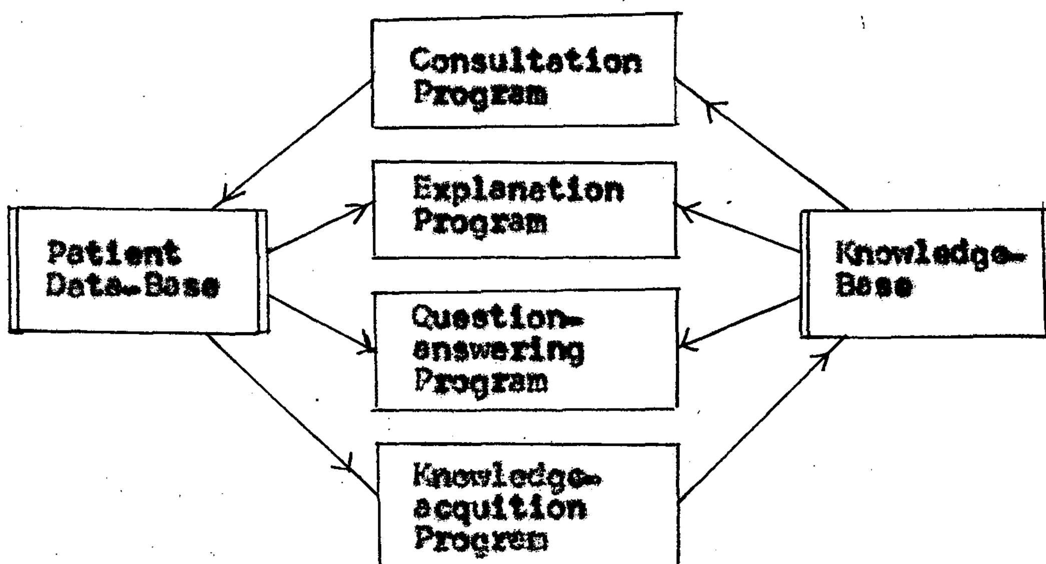


fig.-6 : MYCIN System

Davis and others[7] have developed a knowledge-based application program, called the MYCIN system, and used 'production rules' as a knowledge representation. As shown in fig.-6, the system has six components : four programs, the knowledge-base and the patient data-base. All of the system's knowledge of infectious desease is contained within the knowledge-base. Data about a specific patient collected during a consultation is stored in the patient data-base. Arrows indicate the direction of the information flow.

The MYCIN system was developed originally to provide consultative advice on diagnosis and therapy of infectious deseases, in particular bacterial infections in the blood. They say that production rules offer a knowledge-representation that greatly facilitates the

accomplishment of these goals. Such rules are straight forward enough to make feasible many interesting features beyond performance, yet powerful enough to supply significant problem solving capabilities [7].

Each of MYCIN's rules is a simple conditional statement. The premises is constrained to be a Boolean expression, the action containsone or more conclusions and each is completely modular and independent of the others. It also has some short-comings, because it is not always easy to map a sequence of desired actions or tasks into a set of production rules, whose goal directed involcation will prove that sequence.

Barstow[2] in his paper on automatic programming uses the knowledge-base and represents the knowledge in form of rules in his system, called PECOS. His experimental technique was to select a particular programming domain, elementary symbolic programming and a particular programming task, the implication of abstract algorithm, and to try to codify the knowledge needed for the domain and the task. The form used to express the knowledge was a set of rules, each intended to embody one small fact about elementary symbolic programming. A computer system, i.e. PECOS, was then built for applying such rules to task of implimenting abstract algorithm.

The resulting knowledge-base consists of about four hundred rules dealing with a variety of symbolic

programming concepts. The most distract concepts are collections and mapping alongwith the appropriate operations, for example, testing the membership in a collection, computing the inverse image of an object under a mapping etc., and control structures, for example, enumerating the objects in a collection. The implementation techniques covered by the rules include the representation of collection as linked lists, arrays (both ordered and unordered) and Boolean mappings, the representation of mapping as tables, sets of pairs, property list markings and inverted mappings. PECOS write the resulting program in LISP.

Detailed explanation of PECOS and representation rules is given in Barstow[3].

Davis[6] rises a question : how can we insure that the knowledge imbedded in a program is applied effectively? Traditionally, the answer to this question has been sought in different problem solving paradigms and in different approaches to encoding and indexing knowledge. Paradigms explored have included means-end analysis, resolution, heuristic search, problem reduction etc., while indexing and retrieval have been based on name, effect and context. All of these being useful, however, share a common short-coming, that is, they become in ineffective in the face of a sufficiently large knowledge-base. The problem now is : how can we make it possible for a system to continue functioning in the face of a very large number of plausibly useful chunks of knowledge?

In reference to the above question, Davis [6] proposes a frame-work for viewing issues of knowledge indexing and retrieval, that includes what appears to be a useful perspective on the concept of a 'strategy'. He views strategies as a means of controlling invocation in situations where traditional selection mechanism becomes ineffective. He uses 'meta-rules' as a means to specify strategies. The information in meta-rules is the advice about the likely utility of object level rules. It can help guide program's performance, but it is different in kind from the information carried in an object level rules. Adding meta-rules to the system required only a minor addition to the control structure of the system.

A paper by Fikes [9] describes a demonstration system, called 'Odyssey', which was designed to show how task domain knowledge could be used to help a user to prepare for a business trip. The domain knowledge in Odyssey deals with the structure of the trips, the steps involved in trip preparation, properties of dates and times, cities, airports etc. The project that built the system was focussed on the problem of representing knowledge and using it effectively. He has developed a 'frame oriented' style of programming that combines the features of frame structured knowledge representation as in [11], and object oriented programming, as in [19].

5.2 DESIGN OF THE KNOWLEDGE-BASE

In this section, we give the design of the knowledge-base implemented in the resolution system of this dissertation. Our knowledge-base consists of three parts : thumb rules, data-base and global declarations. In the first part, we give some guiding rules to govern our search for a refutation graph. These rules include all the search strategies, which we want to implement in our system and which have been discussed in section 3.2. We have stated these guiding rules as simple conditional statements of every-day English. By doing so we come much nearer to 'production-rules' of Davis, Buchanan and Shortliffe[7], which have been discussed in section 5.1.

The second part, that is data-base, contains the data and information about clauses of the problem in hand, and they help the system in deciding which rule is applicable where. The third, and last, part contains some global declarations, axioms or assumptions, which are used in semantic sense of the clauses, or in other words, in truth evaluation of the literals of clauses.

5.2.1 Thumb Rules :

Here we present all the rules and their explanations :

Rule-1 (Pure-literal Elimination Rule) : If a clause C contains a literal L such that there are no instances of L which is complementary to an instance of any literal in other clauses, then remove clause C.

This rule allows us to remove those clauses, which contain any 'pure-literal'. The following four rules are related to other simplification strategies :

<u>Rule-2</u> (Tautology Elimination Rule) : If a clause C contains literals L_1 and L_2 such that an instance of L_1 is complementary to an instance of L_2, then remove the clause C.

<u>Rule-3</u> (Subsumed-clause Elimination Rule) : If a clause C_1 subsumes another clause C_2, then remove the clause C_2.

<u>Rule-4</u> (Truth-evaluation Elimination Rule-I) : If a clause C contains a literal L, whose truth value is evaluated to 'true', then remove the clause C.

<u>Rule-5</u> (Truth-evaluation Elimination Rule-II) : If a clause C contains a literal L whose truth value is evaluated to 'false', then remove the literal L from the clause C.

As stated earlier, above five rules are nothing but the simplification strategies discussed in section 3.2.1. First the given set of clauses is tested against these five rules and if a rule is found to be applicable to some clause, the system takes action according to that rule. Applicability of all these rules is tested with help of the other two parts of the knowledge-base, as we will discuss later.

Now, we give the fundamental rule of the resolution process, which decides the resolvability of two given clauses.

Rule-6 (Resolvability Rule) : If there exist clauses C_1 and C_2 such that C_1 contains a literal L_1 and C_2 contains a literal L_2 such that an instance of L_1 is complementary to an instance of L_2, then C_1 may be resolved with C_2.

This should be noted that this rule gives the necessary and sufficient condition of two clauses being resolvable. The following rule is a refinement strategy, called the 'ancestry-filtered form strategy', discussed in section 3.2.2, which tests whether two resolvable clauses may be resolved according to the criterion of AF-form refutation graph.

Rule-7 (Ancestry-Filtered Form Strategy) : Two resolvable clauses C_1 and C_2 may be resolved, if any one of the two clauses is a base clause and the other is a descendant of the top clause ; or, if one of the two is an ancester of the other.

The following two rules represent the ordering strategies, described in section 3.2.3, which just define the preference among more than one choices :

Rule-8 (Unit-Preference Strategy) : Between two clauses C_1 and C_2, which are otherwise equivalent in properties, prefer C_1 over C_2, if it has the lesser number of literals than C_2.

Rule-9 (Shortest-Length Strategy) : Between two resolvents R_1 and R_2 of two resolvable clauses C_1 and C_2 prefer R_1 over R_2, if R_1 has lesser number of literals than R_2.

The rules stated above are the most important part of our knowledge-base. They are, however, helpless without the second part, i.e. the data-base, which contains facts and informations about the clauses. In next section, we present the data-base, which helps the system in deciding the applicability of these rules during a search for the proof.

5.2.2 The Data-Base :

We store two data files on the data-base. The first one is the 'CLAUSE-FILE', that keeps all necessary informations about the clauses having one record for every clause. The key to the CLAUSE-FILE is the serial number of the clause. A record of the CLAUSE-FILE is shown and its data-items are explained below :

CLAUSE-FILE

CLAUSE-NO	CLAUSE-STATUS	PARENT-CLAUSE-1	PARENT-CLAUSE-2	NO-OF-LITERALS	...

...	NO-OF-PREDICATES	POSITIVE-PREDICATES	NEGATIVE-PREDICATES	LITERALS

Explanation :

CLAUSE-NO : Key to the CLAUSE-FILE ('1' for the Top-clause and successive natural numbers for others).

CLAUSE-STATUS : 0, if the clause has not been removed. (such a clause is called a 'live-clause')
1, if the clause has been removed from the set.

PARENT-CLAUSE-1 PARENT-CLAUSE-2	:	CLAUSE-NOs of the parent clauses, if the clause is not a base-clause; 0's if the clause is a base-clause.
NO-OF-LITERALS	:	Total number of literals occuring in the clause.
NO-OF-PREDICATES	:	Total number of predicates occuring in the clause.
POSITIVE-PREDICATES	:	The list of PREDICATE-NOs of those predicates which appear in the clause in affirmative form.
NEGATIVE-PREDICATES	:	The list of PREDICATE-NOs of those predicates, which appear in the clause in negative form.
LITERALS	:	The list of all literals appearing in the clause.

A record is kept for each clause in the given set and as soon as a new clause is generated during the search, a new record is created for that clause in the CLAUSE-FILE, and the process continues. Thus this file is of indefinite length. If during the search a clause is being removed from the set, then the corresponding CLAUSE-STATUS is set to '1'. Removed clauses are ignored in further search.

The CLAUSE-FILE is used when testing a situation against any of the following rules : Rule-2, Rule-3, Rule-4, Rule-5, Rule-7 and Rule-8. The most important use of this file is made when testing against the Rule-7, that is, when we decide whether a pair of clauses can be

resolved under the AR-form strategy. This file helps the system to check whether a clause is a base-clause, a descendant of the top-clause or an ancester of the other clause.

The other file is the PREDICATE-FILE, which consists of one record for each predicate, which appear in the given set of clauses. A record of the PREDICATE-FILE is shown and its data-items are explained below :

PREDICATE-FILE

PREDICATE-NO	CLAUSES-WITH-POSITIVE-LITERALS	CLAUSES-WITH-NEGATIVE-LITERALS

Explanation :

PREDICATE-NO : Serial number of the predicate and the key to the PREDICATE-FILE.

CLAUSES-WITH-POSITIVE-LITERALS : The list of CLAUSE-NOs of those clauses which contain a literal having this predicate in affirmative form.

CLAUSE-WITH-NEGATIVE-LITERALS : The list of CLAUSE-NOs of these clauses which contain a literal having this predicate in negative form.

This file is utilised in deciding whether or not Rule-1, Rule-2 and Rule-6 are applicable in a situation arising in the search.

The PREDICATE-FILE is of the fixed length for a given problem, while the CLAUSE-FILE is of varying length. Whenever a new clause is added in the set of clauses, or a

clause is removed from it, both files are updated accordingly to keep the latest information available to the system.

5.2.3 Global Declarations :

The third part of our knowledge-base consists of some global declarations, axioms and rules of predicate calculus, which may be useful for the system to function effectively. Global declarations are some semantic informations about the predicates of a particular domain of problems. For example, here our problem domain is 'Conceptference' of [18], which will be described in Chapter-6. In connection to this domain, we can make some rules about the predicates like LEFT, ABOVE, INSIDE, SQUARE etc. Some examples of such possible rules are given below :

(1) TRIANGLE, SQUARE, RECTANGLE are unary.

(2) LEFT, ABOVE, INSIDE are binary.

(3) LEFT, ABOVE are not transitive.

(4) INSIDE is transitive.

This part of the knowledge-base is useful in deciding about the applicability of Rule-4 and Rule-5, that is, when we try to evaluate a literal in a clause.

CHAPTER - 6

CONCEPTFERENCE

In order to select a domain of problems to apply upon which the knowledge-based resolution system given in this dissertation, we have chosen the system 'Conceptference' which was developed by Sadananda and Mahabala [18].

The system 'Conceptference' works in two phases. In the first phase, called 'training'phase', Conceptference is exposed to a domain of pictures identified by their pre-assigned names (see fig.-7) and it develops 'concepts' of these pictures in terms of their spartial attributes and relationships and generates internal representation for them. These concepts are stored by the system in the form of well-formed formulas of the first-order predicate calculus.

In the second phase, called 'inference phase', the system employs the resolution principle to draw inferences about existence, identification and description of the pictorial objects of an arbitrary scene.

6.1 CODING AND ATTRIBUTES

The system is designed in LISP. The input of the system is hand coded. A typical rectangle (fig.-8) is coded as below :

```
(RECTANGLE ((a(1 0)(b d))
    (b(2 0)(a c))(c(2 1)(b d))
        (d(1 1)(a c))))
```

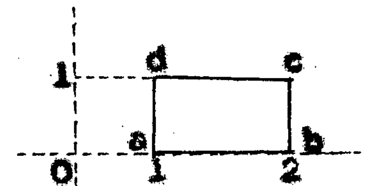
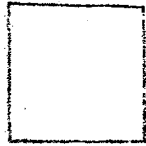
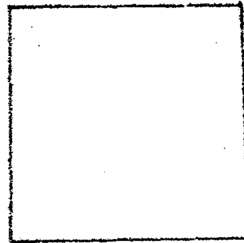
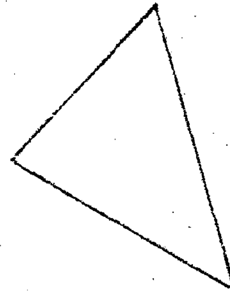
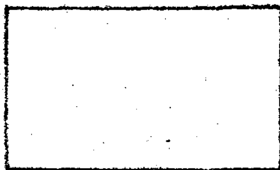


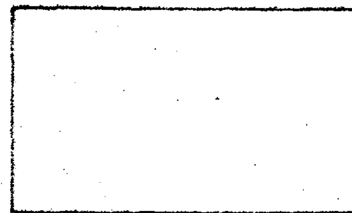
fig.-8

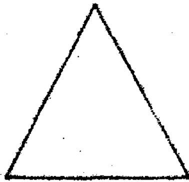

SQUARE

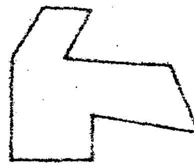

SQUARE

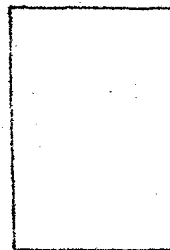

TRIANGLE


RECTANGLE

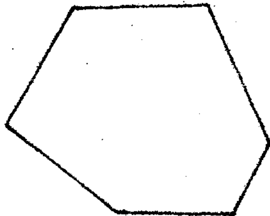

RECTANGLE

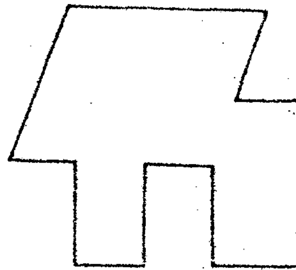

TRIANGLE

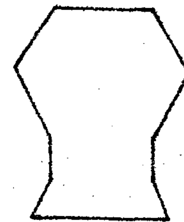

POLYGON

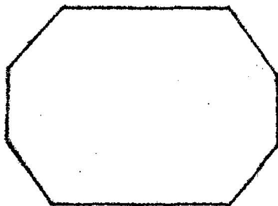

RECTANGLE

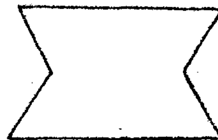
SQUARE


HEXAGON


POLYGON


DECAGON


OCTOGON


HEXAGON

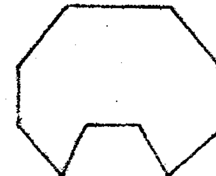

DECAGON

fig.-7

The first member represents the name or designation and the second is a list of lists, each of whose first member is an ordered sequence followed by its cartesian coordinates followed by an unordered list of labels to which the first has a direct connection.

Conceptference is equipped with a sequence of geometric attribute detectors, whose values are the various attributes whenever a picture input is available.

The set of values for the adhoc detectors provided in the present implementation consists of (1) number of sides, (2) ratio of all sides mutually taken two at a time, (3) no. of angles, (4) ratio of various angles taken two at a time, (5) the greatest angle, (6) the least angle, (7) the greatest side, (8) the least side, (9) the ratio of the greatest angle to the least angle, (10) the ratio of the greatest side to the least side and (11) the constant, $(P/\sqrt{A})$, where P is the perimeter of a simply closed curve and A is the enclosed area.

6.2 CONCEPT FORMATION

A set of attribute detectors feed the LISP function ABSTRACT1, which does the 'abstraction'. The arguments of ABSTRACT1 could be either lists generated by the attribute detectors or they could be the result of some previous stage of abstraction. ABSTRACT2 is a LISP function designed to generalize ABSTRACT1. ABSTRACT2 accepts arbitrary lists of attribute lists carrying on successively two at a time calling ABSTRACT1 repeatedly.

To initialize the 'concept' of a triangle, for instance, conceptference works as follows : In the initial phase, various triangles are coded as described and the attribute detectors together with ABSTRACT2 obtain the 'abstraction' of the concept of a triangle. When the system is exposed to a set of three triangles, the intermediate result after the attribute lists are formed may look like :

(TRIANGLE(3(1.0 0.7 1.4)3(0.5 2.0 1.0) 90 45 1.4 1.0 2.0 1.4 4.9))

(TRIANGLE(3(1.7 0.5 1.2) 3(3.3 1.5 2.0) 90 30 2.0 1.0 3.0 2.0 5.2))

(TRIANGLE(3(1 1 1) 3(1 1 1) 60 60 1 1 1 1 4.6))

They yield the 'concept' of a triangle as :

(TRIANGLE(3(X1 X2 X3) 3(X4 X5 X6)Y1 Y2 Y3 Y4 Y5 Y6 Y7))

An informal interpretation of the concept of a triangle could be 'A triangle is one with three sides, three angles, the ratio of sides being at variance from triangle to triangle as are the ratio of angles, the length of the greatest and least sides, the ratio of the perimeter to the square-root of the enclosed area'.

6.3 INFERENCE PHASE

In its second phase, conceptference operates in the domain of a first-order theory. The entire scene is expressed in terms of well-formed formules of the first-order predicate calculus. The names of the concepts formed in the first phase are indeed the predicate letters constituting the well-formed formules generated in the second phase.

Besides, the system generates binary relations such as LEFT, ABOVE, INSIDE etc., between every pair of neighbouring pictures expressed in the form of well-formed formulas. To decide the neighbourhood we choose an arbitrary minimum distance d. The distance is calculated as the least Euclidean distance between labeled co-ordinates of the two pictures.

Conceptference accepts questions in the form of assertions expressed as well-formed formulas. The inference mechanism utilizes the resolution principle after converting the set of well-formed formulas, into the clause form. The deductive strategy takes into account the nature of the assertion to be established.

6.4 AN EXAMPLE

A description of fig.-9, is generated in terms of a set of well-formed formulas, some of whose members are :

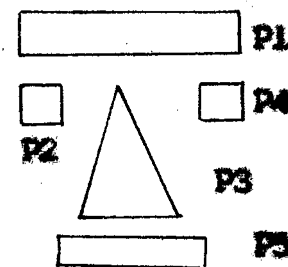


fig.-9

(1) RECTANGLE (P1)
(2) SQUARE (P2)
(3) TRIANGLE (P3)
(4) SQUARE (P4)
(5). RECTANGLE (P5)

In addition, the well-formed formulas expressing the binary relations between the neighbours are generated, some of which are :

(6) LEFT (P2 P3)
(7) LEFT (P3 P4)
(8) ABOVE (P3 P5)

Now, one interrogates conceptference whether there exists an object whose shape and whose relations with other objects leads one to identify it as a 'nose'. He defines the nose as : 'A nose is something triangular in shape, surrounded by a mouth, rectangular in shape; and a pair of eyes, squarish in shape. The mouth is below the nose and there is one eye on the left and the nose is to the left of another eye'. Or in terms of a well-formed formula :

$$\mathrm{TRIANGLE}(X) \wedge \mathrm{SQUARE}(Y) \wedge \mathrm{SQUARE}(Z) \wedge \mathrm{RECTANGLE}(W) \wedge \mathrm{LEFT}(Y\ X) \wedge \mathrm{LEFT}(X\ Z) \wedge \mathrm{ABOVE}(X\ W) \Rightarrow \mathrm{NOSE}(X)$$

The negation of this implification expressed in the clause form is :

$$\sim\mathrm{TRIANGLE}(X) \vee \sim\mathrm{SQUARE}(Y) \vee \sim\mathrm{SQUARE}(Z) \vee \sim\mathrm{RECTANGLE}(W) \vee \sim\mathrm{LEFT}(Y\ X) \vee \sim\mathrm{LEFT}(X\ Z) \vee \sim\mathrm{ABOVE}(X\ W) \vee \mathrm{NOSE}(X)$$

Now the resolution principle is used to establish the existence of nose, which is identified as P3. The refutation graph of this proof is given in fig.-10.

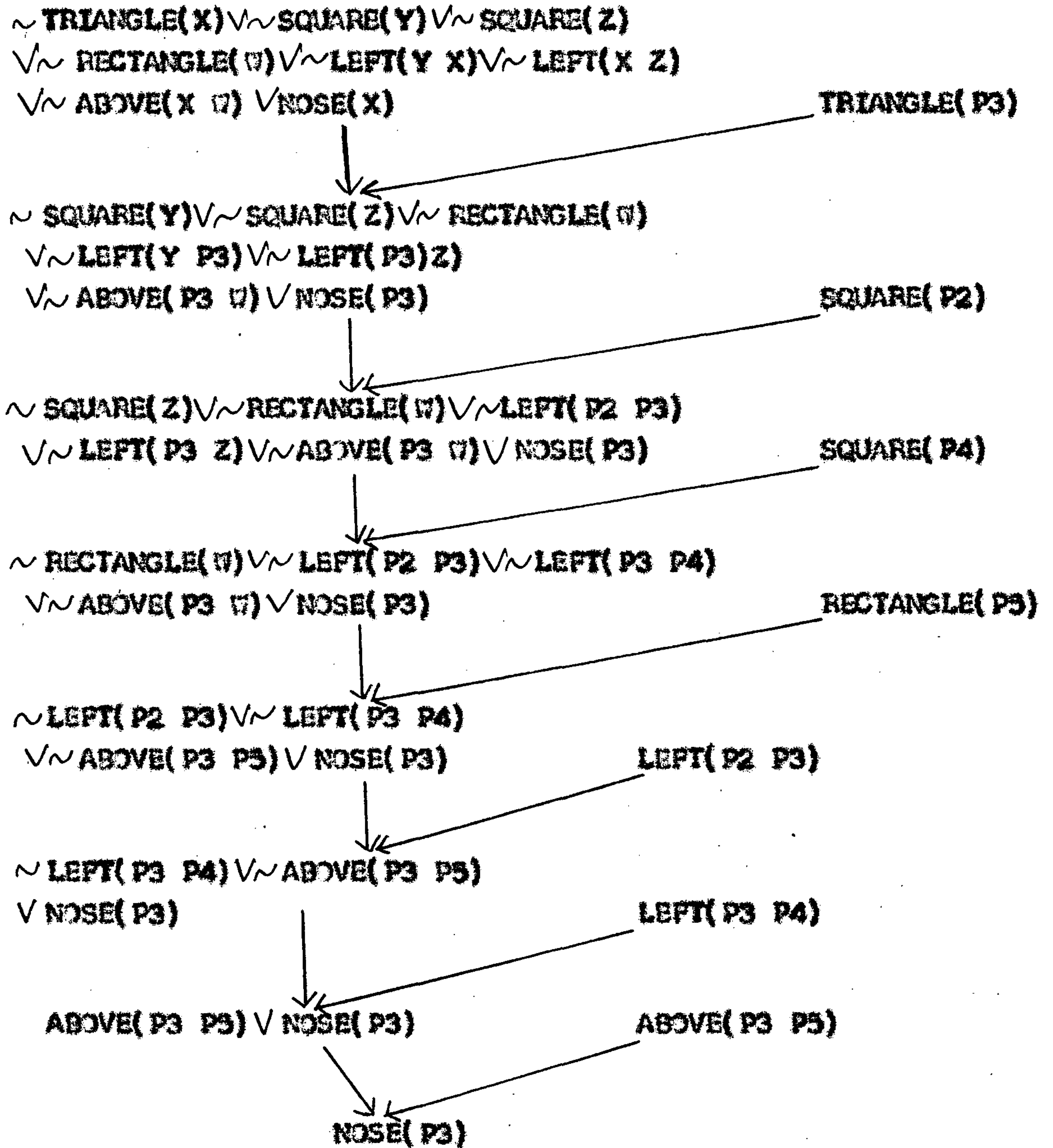


fig.-10

CHAPTER – 7

THE KNOWLEDGE – BASED RESOLUTION SYSTEM

7.1 THE ALGORITHM

We present here the algorithm of our knowledge-based resolution system. This algorithm is based on backtracking and works in co-operation with the thumb rules stored in the first part of our knowledge-base and listed in section 5.2.1. Following is the text of the algorithm :

Step-1 : Select the top clause C. $C_1 \leftarrow C$. Stack $\leftarrow$ empty.

Step-2 : Find a live-clause C_2 resolvable with C_1 and satisfying Rule-7 such that C_2 has not yet been resolved with C_1. If such a clause exists, then go to Step-5; else, remove C_1.

Step-3 : Modify the data-base according to the deletion.

Step-4 : (Backtracking) If stack is empty, then terminate : algorithm fails; else, $C_1 \leftarrow$ stack and go to Step-2.

Step-5 : Find the set S' of all resolvents of C_1 and C_2.

Step-6 : Remove all tautologies and subsumed clauses from S'. Apply Rule-4 and Rule-5, if applicable, on the remaining set.

Step-7 : If S' is empty, go to Step-2.
Else, if S' contains the empty clause, then terminate : the set is unsatisfiable;
else, select a resolvent C_3 from S' satisfying the Rule-9.

Step-8 : Store C_3. Stack $\leftarrow C_1$. $C_1 \leftarrow C_3$.

Step-9 : If C_1 subsumes any live-clause, remove it.

Step-10 : Modify the data-base according to addition and deletions (if any) and go to Step-2.

In the following section we explain the above algorithm in details.

7.2 EXPLANATION

Our search for a refutation graph begins with modifying the given set of clauses, which also includes the negation of the set of inferred clauses, according to Rule-1, Rule-2 and Rule-3 and storing CLAUSE-FILE and PREDICATE-FILE on the data-base.

Then, our first step in the search is to select a top clause C, preferring one of the inferred clauses for this purpose. Taking the top clause as the current clause C_1, we find a clause C_2 resolvable with C_1, which has not already been resolved with it and which also satisfies the criterion of ancestry-filtered form strategy (Rule-7).

If no such clause exists, we note that the clause C_1 is a redundant clause in the set, so we remove it from the set by setting the CLAUSE-STATUS of this clause to '1' and go to stack for the next current clause. At this point, if the stack is empty, the algorithm fails indicating that either the set is not unsatisfiable or the top-clause needs to be replaced. We may replace the top-clause with next clause in the set and run the system again, if we have doubt about the satisfiability of the given set.

But, if C_2 exists, then at Step-5, we find the set S' of all possible resolvents of C_1 and C_2.

At Step-6, we remove all tautologies and subsumed clauses, if any, from set S' and then try to evaluate the literals of the remaining set. If we succeed in some case, we take actions according to Rule-4 and Rule-5, that is, we remove those resolvents which contain any literal which is evaluated to 'true' and those literals which are evaluated to 'false'.

If the remaining set S' is empty, then we go back to Step-2 for some other choice of C_2. Else, if S' contains the empty clause, then we terminate our search with the conclusion that the given set of clauses, including the negation of the inferred clause, is unsatisfiable, or in other words, the inferred clause is a logical consequence of the given set of clauses.

If S' is neither empty nor contains the empty clause, then we select a resolvent C_3 from S' according to Rule-9 and store it as a new clause. The current clause C_1 now goes to the stack and C_3 becomes the new current clause, i.e. C_1.

At Step-9, we check whether C_1 subsumes any live-clause in the graph. If so, we delete such clauses from the set not to consider them in onward search.

At the last step, we modify our data-bases according to new additions and deletions, if any, and go to Step-2 to carry on the search.

Examples, given in the next section, illustrate the application of this algorithm.

7.3 EXAMPLES

Example-1 : Consider the example of fig.-11, taken from Sadananda and Mahabala [18], in which we have the following set of clauses :

(1) ~TRIANGLE(X)V~ SQUARE(Y)V~SQUARE(Z) V~RECTANGLE(W)V~ LEFT(Y X)V~LEFT(X Z) V~ ABOVE(X W)V NOSE(X)

(2) RECTANGLE(P1)

(3) SQUARE(P2)

(4) TRIANGLE(P3)

(5) SQUARE(P4)

(6) RECTANGLE(P5)

(7) LEFT(P2 P3)

(8) LEFT(P3 P4)

(9) ABOVE(P3 P5)

(10) ~NOSE(P3)

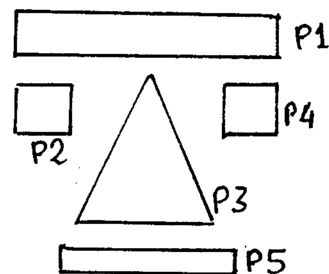


fig.-11

The last clause is the negation of the inferred clause NOSE(P3). We select the negation of inferred clause as the top-clause. So, the current clause C_1 is the top-clause, i.e. ~NOSE(P3).

Now, we see that clause (1) is the only clause that has the predicate NOSE, so it can be resolved with C_1.

The only resolvent of these two clauses is :

$$\sim TRIANGLE(P3) \vee \sim SQUARE(Y) \vee \sim SQUARE(Z) \vee \sim RECTANGLE(W) \vee \sim LEFT(Y\ P3) \vee \sim LEFT(P3\ Z) \vee \sim ABOVE(P3\ W)$$

Here, we note that the literal ~TRIANGLE(P3) is evaluated to 'false' in light of clause (4), that is TRIANGLE(P3). So, we remove this literal from the resolvent and the remaining resolvent is :

$$\sim SQUARE(Y) \vee \sim SQUARE(Z) \vee \sim RECTANGLE(W) \vee \sim LEFT(Y\ P3) \vee \sim LEFT(P3\ Z) \vee \sim ABOVE(P3\ W)$$

which is stored as clause (11) on the data-base. It may be noted that removing the literal ~TRIANGLE(P3) from the initial resolvent is equivalent to resolving it with the clause (4).

Now the current clause C_1 goes to the stack, and the clause (11) becomes new current clause. The search for a resolvable clause with C_1 takes us to clause (3), which can be resolved with C_1 with respect to the predicate SQUARE. The resolution of these two clauses produces three possible resolvents as follows :

(a) $\sim RECTANGLE(W) \vee \sim LEFT(P2\ P3) \vee \sim LEFT(P3\ P2) \vee \sim ABOVE(P3\ W)$

(b) $\sim SQUARE(Z) \vee \sim RECTANGLE(W) \vee \sim LEFT(P2\ P3) \vee \sim LEFT(P3\ Z) \vee \sim ABOVE(P3\ W)$

(c) $\sim SQUARE(Y) \vee \sim RECTANGLE(W) \vee \sim LEFT(Y\ P3) \vee \sim LEFT(P3\ P2) \vee \sim ABOVE(P3\ W)$

The resolvent (a) looks like a tautology, but we can not be sure, because LEFT(X Y) is not equivalent

to ~LEFT(Y X). We can, however, evaluate literals ~LEFT(P2 P3) and ~LEFT(P3 P2) with help of clause (7), i.e. LEFT(P2 P3). We see that the former is false while the later is true, so we remove this resolvent from the set according to Rule-4. Similarly, the literal ~LEFT(P3 P2) is true in resolvent (c), so we delete this resolvent, too, from the set.

Now, the only resolvent left in the set is the resolvent (b), whose literal ~LEFT(P2 P3) is evaluated to 'false', so we delete this literal from the resolvent according to Rule-5. Thus the remained resolvent is :

~ SQUARE(Z) V~ RECTANGLE(W) V~ LEFT(P3 Z) V~ ABOVE(P3 W)

which is stored as clause (12) on the data-base and the data-base is modified accordingly.

Next, this new clause becomes our new current clause and two resolvable clauses, now, are;

(i) SQUARE(P2)

and (ii) SQUARE(P4)

which are clauses (3) and (5) respectively of our data-base. The respective resolvents of these clauses with the current clausesare :

(i) ~RECTANGLE(W) V~ LEFT(P3 P2) V~ ABOVE(P3 W)

and (ii) ~RECTANGLE(W) V~ LEFT(P3 P4) V~ ABOVE(P3 W)

We see that the literal ~LEFT(P3 P2) of resolvent (i) is evaluated to be true with clause (7), so we ignore this resolvent. Now, the remaining resolvent has a literal

~LEFT(P3 P4), which can be evaluated as 'false' in light of clause (8), so, we delete this literal from the resolvent and get the following :

~RECTANGLE(W)V~ABOVE(P3 W)

which becomes clause (13) and new current clause. Now, two clauses which satisfy Rule-7 and are resolvable with this clause are :

(1) RECTANGLE(P1)

(2) RECTANGLE(P5)

First, we resolve our current clause with (1) and get the resolvent as :

~ABOVE(P3 P1)

which, though does not contain any variable, can not be evaluated under given information. So, we store this resolvent as clause (14) and it becomes the new current clause, former current clause going to stack.

At this point, we find that there is no clause in our set which could be resolved with this. So, we remove our current clause from the set and backtracking turn to the stack for new current clause. Thus, clause (13) again becomes the current clause.

This time it is resolved with clause (6) and gives the resolvent C_3 as

~ABOVE(P3 P5)

which is evaluated to be false in light of the clause (9), i.e. ABOVE(P3 P5). So, applying Rule-5, the resolvent now

becomes an empty clause, and hence, we terminate the search with the conclusion that the set of clauses is unsatisfiable. In other words, the inferred clause NOSE(P3) is a logical consequence of the given set of clauses, which means that the object P3 in the given figure, i.e. fig.-11, is recognised as a 'nose'.

Example-2 : Consider the following set of clauses :

(1) $\sim B(x) \vee \sim C(x)$
(2) $\sim G(x) \vee D(x)$
(3) $\sim F(x) \vee C(x)$
(4) $\sim D(x) \vee \sim B(x)$
(5) $\sim F(x) \vee B(x)$
(6) $\sim A(x) \vee F(x) \vee G(f(x))$
(7) $\sim G(x) \vee B(x)$
(8) $A(g(x)) \vee F(h(x))$

We take the clause (1) as the top-clause, which is also the initial current clause. Two clauses (5) and (7) are resolvable with it. First, we resolve it with clause (5), i.e. $\sim F(x) \vee B(x)$, w.r.t. the predicate B, and get the resolvent as

$$\sim C(x) \vee \sim F(x)$$

which becomes clause (9) and new current clause. Next, it is resolved with clause (3) w.r.t. predicate C and we get the resolvent as

$$\sim F(x)$$

which is stored as clause (10) and becomes new current clause. Now, it is resolved with clause (6) and we get the resolvent as

$$\sim A(x) \vee G(f(x))$$

Proceeding in this way and after some back-tracking we get the AR-form refutation graph as shown in fig.-12.

It is to be noted that before each step of resolution, we have to consult our knowledge-base to find whether a pair of clauses is resolable and whether it could be resolved under the AR-form strategy.

After examining the examples given above, we note the following facts : First, number of resolutions has been reduced due to eliminating some literals after evaluation. Secondly, a lot of calculations and comparisions have also been saved, because the knowledge-base atonce provides necessary information, which may take a lot of time to find when every clause is examined again and again. A resolution takes more time than evaluating a literal or retrieving some information from the knowledge-base, because our knowledge-base is neither much big nor it is complex, and number of clauses and number of literals in clauses is large.

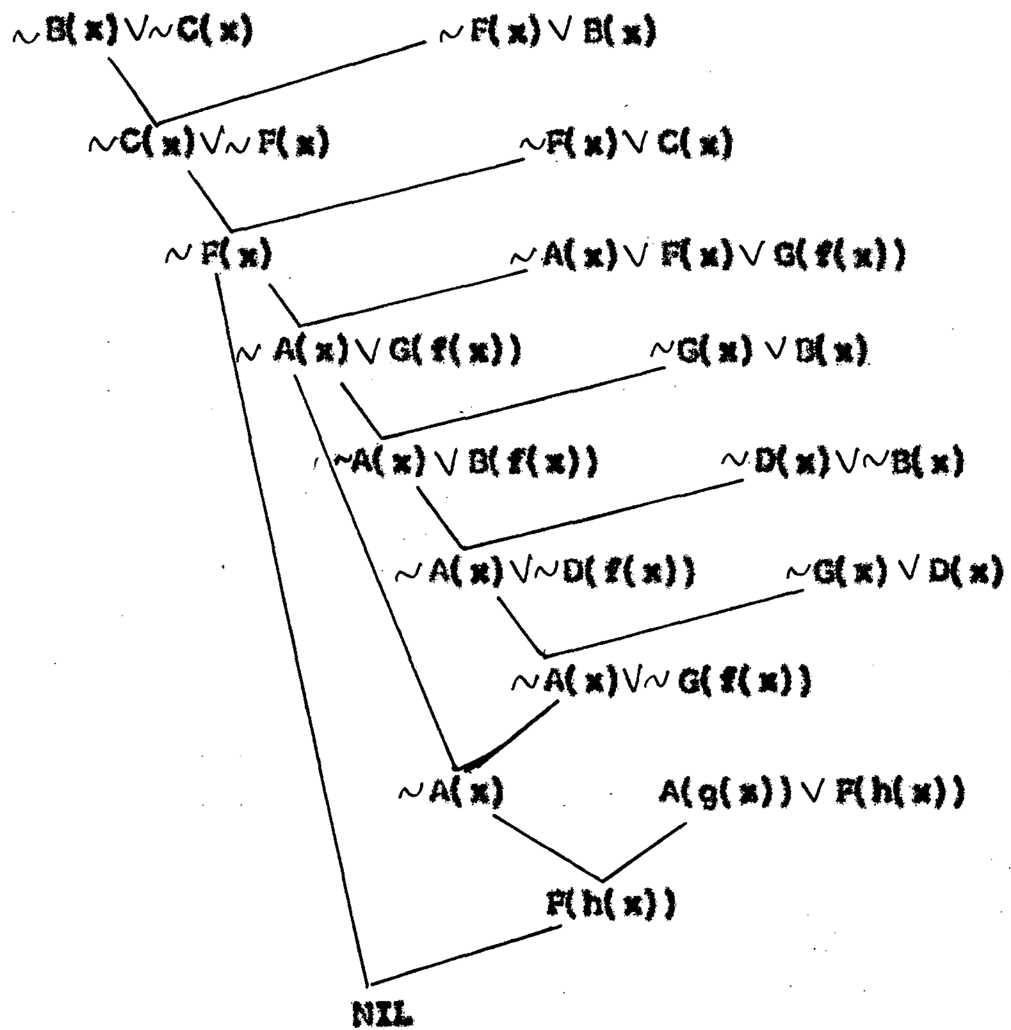


fig.-12

CHAPTER - 8

CONCLUSIONS

The example-1 given in the left chapter shows that the knowledge-based resolution system described in this dissertation is very effective and efficient for the given domain of problems. It can be, however, seen that it is also very effective for other types of problems, because it is a general proof procedure for theorem proving, which utilizes the resolution principlo as well as knowledge-base to make the search for the proof less-expensive and time-saving.

A lot of work, of course, remains to be done, because the presented system is based only on basic resolution alongwith some refinement strategies. Some other proof procedures based on the resolution principle have been developed recently, which are somehow more effective and efficient, though more difficult to apply on the Computer. For example, Plaisted [15] has come with a particular kind of analogy, which is applied to resolution theorem proving in the first-order predicate calculus.

He defines a class of mappings, called 'abstract mappings', which satisfy certain properties. These mappings convert a set A of clauses into a simpler set B of clauses, such that proofs from A correspond to proofs from B having a similar structure. He has given some new inference rules

related to resolutions introducing multi-clauses (m-clauses), which are multi-sets of literals, i.e. with each literal in the m-clause, a count is kept of 'how many times it occurs' in the m-clause. A version of resolution, called m-resolution, is defined for m-clauses and in addition m-abstractions are defined. These map a set A of m-clauses into a simpler set B, such that m-resolution proofs from A map onto m-resolution proofs in B having the same shape.

The advantage of m-abstractions is that they preserve much more of the structure of a proof than do ordinary abstractions. The use of abstractions and related methods of analogy gives a way to use semantic information and specialized knowledge.

In constrast of the method of abstraction of Plaisted [15], which require statements to be in clause form, Murray [12] has developed a proof procedure, called 'NC-resolution', which considers a version of the first-order predicate calculus, in which the statements are not required to be in the clause form as required by standard resolution theorem provers. It requires the well-formed formulas to be quantifier-free and to have distinct variables, in which all variables are universally quantified.

The present system can be extended by applying the above mentioned proof procedures, which may lead us to build more effective and efficient theorem provers.

REFERENCES

1. Ballantyne, A.M. and W.W. Bledsoe : "Automatic Proofs of Theorems in Analysis Using Non-standard Techniques", Journal of ACM, Vol. 24, No. 3, pp. 353-374, 1977.

2. Barstow, D.R. : "An Experiment in Knowledge-Based Automatic Programming", Artificial Intelligence, Vol. 12, pp. 73-119, 1979.

3. Barstow, D.R. : "Knowledge-Based Program Construction", North-Holland Publishing Co., 1979.

4. Bledsoe, W.W. : "Non-Resolution Theorem Proving", Artificial Intelligence, Vol. 9, pp. 1-35, 1977.

5. Cheng, C.L. and R.C. Lee : "Symbolic Logic and Mechanical Theorem Proving", Academic Press, 1973.

6. Davis, R. : "Meta Rules : Reasoning About Control", Artificial Intelligence, Vol. 15, pp. 179-222, 1980.

7. Davis, R., B. Buchanan and E. Shortliffe : "Production Rules as a Representation for a Knowledge-Based Consultation Program", Artificial Intelligence, Vol. 8, pp. 15-45, 1977.

8. Feigenbaum, E.A. : "The Art of Artificial Intelligence" in 'Theories and Case Studies of Knowledge Engineering', Proceedings of the 5th International Joint Conference on Artificial Intelligence, pp. 1014-1079, 1977.

9. Fikes, R.E. : "Odyssey : A Knowledge-Based Assistant", Artificial Intelligence, Vol. 16, No. 3, pp. 331-361, 1981.

10. Loveland, D.W. : "Automated Theorem Proving : A Logical Basis", North-Holland Publishing Co., 1973.

11. Minsky, M.L. : "A Framework for Representing Knowledge" in 'The Psychology of Computer Vision', Ed. Patrick H. Winston, pp. 211-277, 1975.

12. Murrey, N.V. : "Completely Non-Clausal Theorem Proving", Artificial Intelligence, Vol. 18, pp. 67-86, 1982.

13. Newell, A. : "The Knowledge Level", Artificial Intelligence, Vol. 18, pp. 87-217, 1982.

14. Nilsson, N.J. : "Problem Solving Methods in Artificial Intelligence", McGraw-Hill, 1971.

15. Plaisted, D.A. : "Theorem Proving with Abstraction", Artificial Intelligence, Vol. 16, pp. 47-108, 1981.

16. Rieger, C. : "On Organisation of Knowledge for Problem Solving and Language Comprehension", Artificial Intelligence, Vol. 7, pp. 89-128, 1970.

17. Robinson, J.A. : "A Machine Oriented Logic Based on the Resolution Principle", Journal of ACM, Vol. 12, 1965.

18. Sadananda, R. and H.N. Mahabala : "Conceptference : A System that Develops Concepts and Infers on Pictorial Data", Proceedings of the IEEE, Transactions on Systems, Man and Cybernetics, Vol. SMC-8, pp. 232-236, 1978.

19. Schank, R. and R. Abelson : "Scripts, Plans and Knowledge", Proceedings of 4th International Joint Conference on Artificial Intelligence, 1975.

20. Slagle, J.R. : "Artificial Intelligence : The Heuristic Programming Approach", McGraw-Hill, 1971.

21. Sridharan, N.S. : "Guest Editorial", Artificial Intelligence, Vol. 11, pp. 1-4, 1978.

22. Vates, R., B. Raphael and T. Hart : "Resolution Graph", Artificial Intelligence, Vol. 1, pp.257-289, 1970.