INDUCE.3: A PROGRAM FOR LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

by

W. Hoff
R. S. Michalski
R. Stepp

Reports of the Intelligent Systems Group, ISG 86-9, UIUCDCS-F-86-960, Department of Computer Science, University of Illinois, Urbana, 1986.

INDUCE 3: A Program for Learning Structural Descriptions from Examples

BY

William A. Hoff
Ryszard S. Michalski
Robert E. Stepp

File No. UIUCDCS-F-86-960

ISG 86-9

FINAL DRAFT REPORT

INDUCE 3: A Program for Learning Structural Descriptions from Examples

William A. Hoff
Ryszard S. Michalski
Robert E. Stepp

Department of Computer Science
University of Illinois
at Urbana-Champaign

ABSTRACT

The program INDUCE 3 is a general-purpose inductive learning program that transforms symbolic descriptions of real world events into more general or more useful descriptions of those events. The program produces such descriptions by performing various generalizing and simplifying tranformations on the input descriptions, under the guidance of criteria specified by the user. Many of these transformations are built into the program, others are supplied by the user.

Table of Contents

1. Introduction

This paper is a description of, and a user's guide to the program INDUCE 3. The predecessors of this program include INDUCE 2 (containing modifications made primarily by William A. Hoff) and INDUCE 1 (the work of James B. Larson (1977) which is described in his Ph.D. thesis [Larson 1977]). Prior to version INDUCE 2, several enhancements were made to INDUCE 1 by Thomas Dietterich in 1978 [Dietterich 1978] and Mihran Tuceryan in 1980. The major differences between INDUCE 2 and INDUCE 3 are (1) the addition of a subcommand for performing conceptual clustering, and (2) large-scale changes in the internal structure of the program to simplify data structures and control flow. For the sake of completeness, this paper includes appropriately modified and extended sections from the two papers cited above.

1.1. Purpose of the program

INDUCE 3 manipulates symbolic descriptions of real-world events in order to obtain more general or more useful descriptions of those events. The program produces such descriptions by performing various generalizing tranformations on the input descriptions. Many of these transformations are built into the program, others are supplied by the user. The user also supplies criteria for what types of output descriptions he wants (i.e., the user must tell the program what he considers to be a "useful" description).

Descriptions are represented as VL (Variable-valued Logic) decision rules, in the form

$$\text{CONDITION} \Rightarrow \text{DECISION}$$

where CONDITION describes some set of situations and DECISION describes some new situation or action which is indicated if a given situation satisfies the CONDITION. If the situation does not satisfy the CONDITION, the rule makes a NULL decision for that situation. The descriptions in CONDITION and DECISION are represented in the VL_2 logic system (described in section 2 and in [Michalski 1972, 1980]). This system is an extension of first order predicate calculus with a rich set of operators, and the facility for allowing the user to define the domain size and structure for each variable and function appropriate for the problem at hand. The approach taken here is to apply inference rules (generalization and reformulation) to initial decision rules in order to form new generalized and optimized decision rules which retain the decision making capabilities of the original rules.

1.2. The Induction Problem

The induction problem being investigated is as follows:
Given (1) a set of input VL decision rules:

$$\begin{array}{llll} C_{1,1} \Rightarrow D_1; & C_{1,2} \Rightarrow D_1; & \ldots & C_{1,r1} \Rightarrow D_1 \\ C_{2,1} \Rightarrow D_2; & C_{2,2} \Rightarrow D_2; & \ldots & C_{2,r2} \Rightarrow D_1 \\ \vdots & \vdots & \vdots & \vdots \\ C_{m,1} \Rightarrow D_m; & C_{m,2} \Rightarrow D_m; & \ldots & C_{m,rm} \Rightarrow D_m \end{array} \qquad (1.1)$$

where $C_{i,j}$ and D_i are expressions in the VL_2 system representing the CONDITION and DECISION parts of decision rules respectively, and (2) problem-oriented background knowledge (including the preference

criterion for evaluating the hypotheses), then find a set of output VL decision rules (hypotheses):

$$\begin{array}{llll} C'_{1,1} \Rightarrow D_1; & C'_{1,2} \Rightarrow D_1; & \ldots & C'_{1,t1} \Rightarrow D_1 \\ C'_{2,1} \Rightarrow D_2; & C'_{2,2} \Rightarrow D_2; & \ldots & C'_{2,t2} \Rightarrow D_1 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ C'_{m,1} \Rightarrow D_m; & C'_{m,2} \Rightarrow D_m; & \ldots & C'_{m,tm} \Rightarrow D_m \end{array} \qquad (1.2)$$

where $t_i \leq r_i$ for $1 \leq i \leq m$, and that are:

1) consistent and complete with the input rules (1.1)
2) augmented by the background knowledge, and
3) optimized according to the preference criterion.

The new rules are *consistent* with the input rules if for any situation for which the output rules assign a decision (a non-NULL decision), the initial rules assign the same decision or a NULL decision. They are *complete* if for any situation for which the input rules assign a decision, the new rules assign a decision. From any given input rules it is usually possible to derive many sets of rules which are consistent and complete. Therefore, a preference criterion (assembled by the user from a few simple functions) is used to select one or a few alternatives which are most desirable in the context of the specific induction problem. The algorithm is restricted to work with sets of rules which make only one decision for a given situation.

1.3. Suitable Applications

INDUCE 3 is a general-purpose inductive learning program, designed to aid a user in solving a wide range of practical inductive problems. In order to use it, the user (usually an expert in the given domain) must determine the initial set of descriptors, formulate the input rules, suggest rules for generating new descriptors, define the preference criterion and various parameters, evaluate the output rules, and repeat the process if desired.

The problems for which INDUCE 3 is applicable should have the following characteristics:

(1) Distinct objects or events exemplify a collection of different concepts (the concepts may represent distinct categories, objects, or decision classes assigned to objects, etc).

(2) A set of measurements, called descriptors (variables or relations) is available for characterizing each object or situation. At least some of the descriptors should be relevant to the inductive learning problem. It is also desirable that the user have some idea of what kind of transformations of the initial descriptors might produce new (derived) descriptors that would be more relevant to the problem. These suggestions are provided to the program as a part of the program's *background knowledge*.

(3) The input formulas asserting concept classes for each object are assumed to be true. Moreover, the descriptions of examples (ie., symbolic representations of the objects or situations) are assumed to be precise, consistent, and complete.

(4) For each descriptor, a domain is defined by specifying the set of values this descriptor may assume. The program distinguishes between nominal (or categorical) descriptors (whose domain is an unordered set), linear descriptors (whose domain is a totally ordered set), and structured descriptors (whose domain is a hierarchical tree of values). In the case of non-discrete linear variables (ie., continuous variables), their domain must be quantized into a reasonably small number of discrete values (on the order of 10-100, depending on the setting of the program constants).

(5) A criterion is defined for determining the most preferable hypothesis among all the hypotheses that consistently and completely characterize the known facts. (The program provides the user with a set of elementary criteria from which the user assembles a single general criterion that is most

appropriate to the given problem.)

INDUCE 3 is designed to be applied to problems that cannot be adequately represented with traditional quantitative variables. It has the power to accept and produce structural descriptions, in addition to attribute values. Such structural descriptions involve not only quantitative and qualitative attributes of individual objects, but also properties of object parts and relations among the parts.

1.4. Previous Work

Inductive inference is viewed here as a process of rewriting and/or generalizing available descriptions in order to derive new descriptions that are computationally more economical and conceptually simpler. Statistical methods have probably been the most widely used form of inductive inference. These methods require the availability of a large set of data, assume the independence of variables, and require an understanding of the type of underlying distribution of the data [Croft 1971]. In addition, results may be difficult to conceptualize and interpret [Larson 1976].

An early approach to automated inductive inference using a logic-based approach was developed by Hunt [1966]. He described a number of different schemes for generating decision trees which can be used to distinguish between sets of letter sequences. Although a decision tree produces a procedure which can be easily executed on a computer, it lacks the flexibility necessary to represent more general concepts. A recent implementation of a decision tree-based system is Quinlan's ID3 program [Quinlan 1979].

The Heuristic-Dendral program [Buchanan et.al. 1969] provides a model appropriate for representing the structure of chemical compounds and some transformations representing possible chemical reactions which can be applied to the compound representations under certain known physical constraints. The program finds a set of possible structures of a compound knowing its empirical formula and mass spectrometer data by suggesting various structures for the compound and applying transformations to the structures under the guidence of a set of heuristics based on the mass spectrometer data. The Meta-Dendral program [Buchanan et.al. 1972] finds a general mechanism or theory which explains the transformations which take place, relying on the knowledge of those transformations which are plausible and those which are forbidden.

Computer aided medical diagnosis is another area in which logical inductive inference methods have been suggested. Pople [Pople et.al. 1972] has suggested a graph structure representation of biomedical facts and an approach to forming theories by finding common subgraphs using user-supplied suggestions. Of particular note is work in the area of computer aided medical diagnosis and plant pathology [Michalski 1973,1974, Chilausky et.al 1976, Larson 1976]. That work, based on VL_1 (a variable valued logic system which is an extension of propositional calculus) and the program AQ11, was used to infer descriptions of classes of liver diseases and soybean diseases [Michalski and Chilausky 1980].

Winston [1970] demonstrates specific procedures which discover descriptions from examples in the toy blocks world. A description which is discriminant (i.e. can be used to distinguish one object or set of objects from other sets) is formed by matching the similar parts of the object under consideration with another object (near-miss) and then isolating the structures which are different between the two objects. This differs from the approach taken in the following chapters in that matching is only done here in order to adequately describe some specific feature which distinguishes between two objects. (e.g. to describe the second part from the top in an object, one may have to include some predicates which define the second from the top in terms of other descriptors. If the distinguishing feature involves this part, then the definition of the part used in the description must be common to both objects.) Winston also uses modifiers such as 'must', 'may', 'must not' in descriptions. A form of these modifiers is inherent in the VL_2 approach (e.g., discriminant descriptions involve the 'must' modifier, characteristic descriptions involving only one set of objects yield descriptions involving a type of 'may' modifier).

The program ARITHMETIC [Bongard 1970] finds an algebraic rule which explains sample relationships. The program is given sets of tables, each table containing 3-tuples of a ternary relation. A set of 33 predicates are used in the program (although not explicitly given in the reference) where a predicate may be: e.g. if the quotient of the first two elements of the 3-tuple is positive, then the

predicate is true. For each row of a table, a set of features is generated by finding Boolean combinations of the predicates applied to the row. A feature describing the table is generated for each Boolean combination by finding the product of Boolean combinations for all rows. The set of features which appear to be most useful in distinguishing one table from the others is selected as the description of each table.

Another type of problem is that of inducing sequence extrapolation rules. The program ELEUSIS [Dietterich 1980] is an expert on the card game ELEUSIS, in which players attempt to infer a secret rule invented by the dealer. The secret rule tells which cards can be played at the next position in the sequence.

Lenat [1976] describes the program AM which infers concepts in number theory, starting from elementary concepts in set theory and applying a large body of heuristic rules. This program generalizes from initial descriptions, and can propose new concepts and test them on examples.

A summary of more general systems follows. These systems have emphasis in two types of approaches: 1) generation of descriptions of sets of objects represented in a logic system of some kind, and 2) creation of new concepts in a sequential manner by generating and modifying hypotheses. A summary of several types of learning systems can be found in [Banerji 1975]. Of particular interest here is the work of Morgan [Morgan 1972] in which a formal system based on the first order predicate calculus with falsehood preserving transformations is presented. Briefly, the idea that inductive inference can be described as backwards reasoning is apparently not sufficent for a practical system. For example, if E_2 were derived from the assertions $\neg E_1$ and $E_1 \vee E_2$, then backward reasoning would somehow have to generate the two assertions above given only E_2. There are far too many expressions E_1 which can be applied to a situation such as this to yield a practical system. Instead, Morgan defines a falsehood preserving transformation of a deductive inference rule into an inductive inference rule (in Morgan's notation):

$$E_1 \vdash_F E_2$$

where E_2 is false for every interpretation in which E_1 is false. A set of transformations (F-rules) can be created to convert a deductive logic system into an inductive logic system (e.g. if E_1 and E_2 are atomic expressions, then

$$E_1 \vee E_2 \quad \vdash_F \quad E_1E_2$$

i.e., from the disjunction $E_1 \vee E_2$, one may infer the conjunction E_1E_2 while preserving falsehood). With this system, theorem proving techniques using falsehood preserving rules may be applied to inductive problems. (In later chapters, the symbol $|\!<$ is used to denote such a generalizing transformation.)

A number of authors have presented systems which use a graph structure representation of expression in a type of logic system. The 'parameterized structure representation' of Hayes-Roth [1976] is used in inductive tasks which learn descriptions of sets of objects and transformations from one set of objects to another set of objects from examples. The problems addressed are closely akin to the work in the following sections of this paper, one of the objectives being to find the common properties of all examples of one class which are available to the system using a graph-theoretical representation. The number of possible alternatives in Hayes-Roth's method is limited by a fixed utility function which evaluates intermediate results and discards hypotheses of low utility. The work differs from that presented in the following sections in that only independent descriptions are sought using a fixed utility criterion (these are called characteristic descriptions in this paper). The structure used for representation does not take into account specific domain structures which are inherent in the VL system and there is no facility for generating new descriptors within the system.

A formal approach using the predicate logic system [Vere 1975] produces the largest common set of descriptors of a set of examples by representing examples using a graph structure and finding the largest common subgraph of these structures. Such methods suffer from the NP-complete nature of graph isomorphism algorithms (this problem is addressed in the following sections by finding the smallest useful

subgraphs of graphs of examples instead of the largest subgraph.). Neither Vere nor Hayes-Roth use negative examples adequately in their implementations.

Hedrick [1974] uses a semantic net to represent examples and to build and modify hypotheses as new examples are given to the program. The semantic net supports only binary relations but otherwise looks similar to the graph structure of Vere and Hayes-Roth.

Kochen [1974] presents a different type of system with a set of initial events containing state variables, actions and relations between the actions, a learning program which applies certain transformations to events at various time steps. At each time step, weighted hypotheses are formed which reduce the set of states stored in memory (i.e., those states which are explained by the hypotheses).

Sammut [1981] describes a concept-learning program MARVIN, which uses a form of predicate calculus to represent inductive assertions. This program constructs examples and asks the user whether they are instances of the concept to be learned. It uses the answers to refine its representation of the concept until no more generalizations can be made.

Production systems provide a rich tool for the introduction of many inference techniques (recall that VL decision rules are similar to production rules). Briefly, a production system architecture contains a working memory, a set of productions which modify working memory, and a recognize-act cycle with conflict resolution to dictate the order in which productions are applied to memory and to add new productions when necessary.

Waterman [1970, 1974, 1975] uses these to solve several problems. A program which plays poker has been developed [Waterman 1970] which designs betting strategies in terms of production rules. More recently [Waterman 1975], the approach has been applied to recognizing letter sequences with success. Rychener [1976] has applied production systems to chess end games and natural language input of a toy blocks world. These two authors use distinct production system architectures which differ in the ordering of working memory, ordering of productions and in the way in which new productions are added to an existing set of productions to correct errors made by the system.

2. Representing Decisions in the VL_2 System

Much of the information in this section is found in [Larson et.al. 1977, Michalski 1974b, 1983]. It is included here to give the reader a familiarity with the VL_2 system. (In previous papers, the system VL_2 was called VL_{21}.)

The variable-valued logic system VL_2 is a language for describing situations (e.g. objects, classes of objects) and expressing decision and inference rules. The language provides for a compact expression of descriptions, which is both easily readable and sufficiently precise to facilitate formal manipulation.

There are three major differences between VL_2 and the first order predicate calculus:

(1) Instead of predicates, the language uses selectors which can be viewed as tests for membership of values of predicates or functions in a certain set.

(2) An expression in VL_2 can have a truth status: TRUE, FALSE, or UNKNOWN. The truth status UNKNOWN provides an interpretation of a VL_2 description in a situation when outcomes of certain measurements are not known.

(3) Each descriptor (variable, predicate and function symbol) is assigned a domain (or value set) together with a characterization of the structure of the domain. (This feature facilitates the process of rule generalization and allows for the application of different generalization transformations according to the structure of the domain.)

There are three types of domains currently distinguished:

(1) Nominal (Unordered)
Elements of the domain are considered to be independent entities; no structure is assumed to relate them. A variable or function symbol with this domain is called *nominal* (e.g. blood type, names of objects, etc.).

(2) Interval (Linearly Ordered)
The domain is a linearly ordered set. A variable or function symbol with this domain is called *interval* (e.g. military rank, temperature, size).

(3) Structured (Tree Ordered)
Elements of the domain are ordered into a tree structure. A parent node in the tree represents a concept which is more general than the concepts represented by the child nodes (e.g., the parent of the nodes 'triangle', 'rectangle', 'pentagon' may be a 'polygon'). A variable or function symbol with such a domain is called *structured*.

2.1. VL System Structure

The VL_2 system is defined by a 5-tuple (V,F,S,R,I) where:

V - is a set of variable symbols. Each variable symbol is associated with a domain $D(x_i)$. A group of variables which have the same domain are labelled with the same variable symbol but a different subscript (e.g. $x_1,x_2,..,x_k$, $y_1,y_2,...,y_l$ are specifications of variables in two variable groups which assume values from two domains denoted D(x) and D(y) or alternatively, $D(x_i)$ and $D(y_i)$.

F - is a set of n-ary functions and predicate symbols. Each n-ary function symbol represents a mapping from an argument space into the domain of the function. For a function $f(x_1,x_2,...,x_n)$, this is a mapping:

$$D(x_1) \times D(x_2) \times \ldots \times D(x_n) \rightarrow D(f)$$

where $D(x_1)$, $D(x_2)$, ... $D(x_k)$, D(f) represent the domains of the variables x_1, x_2, ... x_k and the domain of the function f, respectively. A predicate is a function whose domain is the set [TRUE,FALSE].

Included in the domains of all function and variable symbols is the value NA (not applicable).

S - is a set of symbols including:

$$(\ )\ [\ ]\ =\ <\ >\ \neq\ \geq\ \leq\ \rightarrow\ \leftrightarrow\ \Rightarrow\ \&\ \mathrm{v}\ \exists\ \exists.\ \forall\ \forall.\ ,\ .\ \surd$$

where the symbol & denotes conjunction, and the symbol v denotes disjunction.

R - is a set of formation rules described in section 2.3

I - is a set of interpretation rules described in section 2.4

2.2. Selector Formation and Interpretation Rules

A well formed VL_2 formula (wff) is composed of quantifier forms, selectors, and logical connective symbols.

A *selector* is a form:

$$[L\ \#\ R] \text{ or } [L]$$

where

L - called the *referee,* is an atomic form. An *atomic form* is a variable symbol or a function or predicate symbol followed optionally by a list of atomic forms enclosed in parentheses. In the first form above, L must contain a function symbol (that function is called an *atomic function*), and in the second form L must contain a predicate symbol.

R - the *reference* is a set of values in the domain of the atomic function of L. R may be in several forms:

Reference Example	Description
a	a constant in the domain of the atomic function of L
a v b	a list of values in the domain of L separated by v
a..b	a pair of values in the domain of L separated by (..)
*	the symbol (*) representing all values in the domain of L (except NA)
NA	the value NA (not applicable)

- is one of the following relational symbols

$$=\quad \leq\quad \geq\quad <\quad >\quad \neq$$

If R is a set of values, then L is related to R by # if

when # is = or $\neq$	L has a value (does not have a value) in the set R
when # is $\leq$ $\geq$ < >	L has a value related to every value of R by #.

The selector is interpreted as a unit of information about a situation with value or truth-status TRUE if the relation R # L holds or FALSE if the relation does not hold, or UNKNOWN in which case the selector is interpreted as a question about the situation which must be answered in order to determine

if the selector is satisfied. If some variables in the atomic form of the selector are quantified, these quantifiers must be considered when determining the truth-status of a selector.

If R is *, then L is related to R for any value of L except NA (in this case, # is always =). Below are some examples of a selector:

Selector	Interpretation: (TRUE if)
[color(wall$_1$) = white]	The color of the wall represented by wall$_1$ is white.
[length(box$_1$) $\geq$ 1]	The length of the box represented by box$_1$ is greater than or equal to 1.
[weight$_1$ = 2..5]	The variable weight$_1$ may have a value between 2 and 5 inclusive. The selector restricts the range of values of the variable weight$_1$ to the values 2 through 5.
[ontop(x_1,x_2)]	The part represented by x_1 is on top of the part represented by x_2.

2.3. VL Formation Rules

Formulas in the VL_2 logic system are used to describe situations, and also to express decision rules and inference rules. The VL formulas are defined by the following formation rules:

(1) A selector is a VL formula (wff).

(2) If V, V_1 and V_2 are wff, then so are:

(V)	a formula in parentheses
$\neg$V	inverse
V_1 & V_2 or V_1V_2	conjunction (the symbol & is used to represent conjunction)
V_1 v V_2	disjunction
V_1 $\checkmark$ V_2	exception
$V_1 \rightarrow V_2$	V_1 implies V_2
$V_1 \leftrightarrow V_2$	V_1 is equivalent to V_2
$\exists\, x_1,x_2,\ldots,x_k(V)$	existentially quantified formula
$\exists\!\cdot\, x_1,x_2,\ldots,x_k(V)$	distinctly existentially quantified formula (explained below)
$\forall\, x_1,x_2,\ldots,x_k(V)$	universally quantified formula
$\forall\cdot\, x_1,x_d,\ldots,x_k(V)$	distinctly universally quantified formula (explained below)

Not all of these forms are considered in the following sections. A graph structure representation is presented which includes all of these forms, but the types of formulas actually included in the algorithm and the implementation involve only conjunction and distinct existential quantification.

2.4. Interpretation Rules

The interpretation of VL formulas is done in the context of each situation. That is, each situation is treated as a domain over which the formulas are evaluated: the value sets of the quantified variables and the interpretation of the functions and predicates is done individually for each situation. A VL formula may have truth-status TRUE, FALSE, or UNKNOWN, when applied to a given situation. The connectives (¬ v &) are interpreted in the normal manner:

VL formula	Interpretation
¬V	FALSE if V is TRUE, TRUE if V is FALSE, UNKNOWN if V is UNKNOWN.
V_1 v V_2	TRUE if either V_1 or V_2 is TRUE, UNKNOWN if both V_1 and V_2 are UNKNOWN or one is UNKNOWN and the other FALSE, FALSE otherwise.
V_1 & V_2	UNKNOWN if both V_1 and V_2 are UNKNOWN or one is true and the other is UNKNOWN, TRUE if both V_1 and V_2 are TRUE, FALSE otherwise.

The remaining connectives may be rewritten in equivalent forms:

VL Formula	Equivalent form
$V_1 \rightarrow V_2$	$\neg V_1$ v V_2
$V_1 \leftrightarrow V_2$	$(V_1 \rightarrow V_2)$ & $(V_2 \rightarrow V_1)$
$V_1 \surd V_2$	V_1 & $\neg V_2$ v $\neg V_1$ & V_2

A VL_2 system is used to describe a set of situations. In order to effectively apply a formula to a set of situations, the VL_2 system should contain variables, functions, and predicates which adequately characterize the situations. To determine the truth-status of a formula with regard to a specific situation, an event is created (an event may be viewed as an interpretation of a situation in the VL_2 system). An event is a sequence of assignments to variables, functions and predicates in the system which characterize a specific situation. Quantified variables may be assigned a set of values. One function assignment may be made to a given set of values of arguments if the value of the function is known. If a function does not have an assignment for a given set of values, then the value NA (not applicable) is assumed.

A selector [L # R] (or [L']) is satisfied by an event if there is a set of assignments to variables and functions (or predicates) in L (or L') such that L is related to R by # (or L' has the value TRUE). A VL_2 formula is satisfied by an event if it has truth status TRUE when applied to the event.

The quantified formulas are interpreted as follows:

$\exists\ x_1,x_2,...,x_n$ (V)	is TRUE (or FALSE) in a given situation if there exists (or does not exist) values for $x_1,x_2,...,x_n$ in the event assignments which makes the truth-status of the formula V equal to TRUE ? - if it is not known whether there exist values ...
$\exists.\ x_1,x_2,...,x_n$ (V)	is TRUE (or FALSE) in a given situation if there exists (or does not exist) *distinct* (different) values for $x_1,x_2,...,x_n$ in the event assignments which makes the truth-status of the formula equal to TRUE. This obviates the need for extra predicates in an expression like $x_1 \neq x_2$, $x_2 \neq x_3$, $x_1 \neq x_3$, etc. ? - if it is not known whether there exist values ...
$\forall\ x_1,x_2,...,x_n$ (V)	is TRUE (or FALSE) in a given situation if for all assignments to the variables $x_1,x_2,...,x_n$, the formula V has truth-status equal to TRUE.
$\forall.\ x_1,x_2,...,x_n$ (V)	is TRUE (or FALSE) in a given situation if for all *distinct* assignments to the variables $x_1,x_2,...,x_n$, the formula V has truth-status equal to TRUE.

2.5. VL Decision Rules

If V_1 and V_2 are VL_2 formulas, a general form of a VL decision rule is

$$V_1 \Rightarrow V_2 \qquad (2.5.1)$$

The formula V_1 is called the *condition* part and V_2 is the *decision* part. A restricted form of the VL_2 decision rule will be used in the following sections.

In the computer implementation, the formula V_2 is assumed to be a product of selectors which contain variables (0-ary functions) as the referee. The variable which appears in the decision part of a formula is called the *decision variable*.

A decision rule in the form 2.5.1 may be applied to a set of situations as follows: If the condition part of the decision rule (V_1) is given truth-status TRUE, then the decision part of the decision rule (V_2) also assumes the truth-status TRUE. For each event (assignment e:=L) which satisfies V_1, a new set of assignments are made to the event using the decision part of the rule to form the set of all events which satifsfy the conjunction V_1 & V_2. For example, given the decision rule:

$$\exists.\text{part}_1,\text{part}_2\ [\text{touching}(\text{part}_1,\text{part}_2)] \Rightarrow [d=1] \qquad (2.5.2)$$

and the events

$$\begin{array}{ll} e_1: & \text{touching}(p_1,p_1)=\text{TRUE} \\ e_2: & \text{touching}(p_2,p_1)=\text{TRUE} \\ e_3: & \text{touching}(p_2,p_2)=\text{TRUE} \end{array} \qquad (2.5.3)$$

with the following defined domains: $D(\text{part}_1)=D(\text{part}_2)=[p_1,p_2]$, D(touching)=D(ontop)=[TRUE,FALSE],

D(d)=[0,1]. Rule 2.5.2 is applied to each event to give:

$$\begin{array}{ll} e_1: & \text{touching}(p_1,p_1)=\text{TRUE} \\ e_2: & \text{touching}(p_2,p_1)=\text{TRUE; d=1} \\ e_3: & \text{touching}(p_2,p_2)=\text{TRUE} \end{array} \qquad (2.5.4)$$

A decision rule:

$$\exists.\text{part}_1,\text{part}_2\ [\text{touching}(\text{part}_1,\text{part}_2)] \Rightarrow \exists.\text{part}_1,\text{part}_2\ [\text{ontop}(\text{part}_1,\text{part}_2)] \qquad (2.5.5)$$

applied to each event from 2.5.3 gives the following events:

$$\begin{array}{ll} e_1: & \text{touching}(p_1,p_1)=\text{TRUE} \\ e_2: & \text{touching}(p_2,p_1)=\text{TRUE; (ontop}(p_1,p_2) \vee \text{ontop}(p_2,p_1)) \\ e_3: & \text{touching}(p_2,p_2)=\text{TRUE} \end{array} \qquad (2.5.6)$$

Note that $\text{ontop}(p_1,p_1)$ and $\text{ontop}(p_2,p_2)$ are not given status TRUE since the quantifier $\exists.$ insists that the two variables part_1 and part_2 have different values.

Given a set of decision rules each in the form 2.5.1, the set may be applied to a set of events. If an event satisfies the condition part of a decision rule, new assignments are made to the event as indicated by the decision part of the satisfied rule.

In the remainder of the paper, events are only used as a formal basis for defining certain concepts. Since the number of events necessary to completely describe a situation is quite large, only the VL_2 formulas themselves are manipulated by the algorithms.

2.6. Generalization Rules

From one set of decision rules (1.1), a new set of decision rules (1.2) is obtained by applying certain transformation rules (t-rules). In this paper, we will restrict our attention to rules which generalize the condition part of a rule. A more detailed description of transformation rules can be found in [Michalski 1980] and [Michalski 1983]. Given two rules:

$$R_1: \quad V_1 \Rightarrow D \qquad\qquad R_2: \quad V_2 \Rightarrow D$$

V_1 is more *general* than V_2 if every event satisfying V_2 also satisfies V_1. If the converse is also true, then V_1 is equivalent to V_2. This generalizing transformation is denoted $R_1 \prec R_2$. Generalizing transformations usually produce not only a more general decision rule from a set of decision rules but also a 'simpler' one than the original.

2.6.1. Dropping a Selector

This generalization rule is defined as:

$$V\ [L = R] \Rightarrow D \quad \prec \quad V \Rightarrow D$$

where V denotes any VL formula. Although this rule is interesting in a formal sense, it should be applied with care since the number of generalizations possible with successive applications of this rule is very large. In the INDUCE algorithm, this rule is applied in reverse. First, very general decision rules are created and then they are specialized by adding selectors to them. The INDUCE algorithm is discussed in more detail in section 4.

2.6.2. Extending the Reference

This generalization rule is used in the INDUCE program by the AQ procedure. The rule has several forms depending on the domain type of the descriptor. If the descriptor L has a nominal domain type:

$$\begin{array}{l} V\ [L = a] \Rightarrow D \\ V\ [L = b] \Rightarrow D \end{array} \quad \mid\!\prec \quad V\ [L = a \vee b] \Rightarrow D$$

If the descriptor L has an interval domain type:

$$\begin{array}{l} V\ [L = a] \Rightarrow D \\ V\ [L = b] \Rightarrow D \end{array} \quad \mid\!\prec \quad V\ [L = a..b] \Rightarrow D$$

If the descriptor L has a tree structured domain:

$$\begin{array}{l} V\ [L = a] \Rightarrow D \\ V\ [L = b] \Rightarrow D \end{array} \quad \mid\!\prec \quad V\ [L = c] \Rightarrow D$$

where c is a predecessor of both a and b in the generalization structure of the domain of L.

2.6.3. Extension Against

This generalization rule is used by the AQ procedure. If the descriptor L has a nominal domain:

$$\begin{array}{l} V_1\ [L = R_1] \Rightarrow D \\ V_2\ [L = R_2] \Rightarrow \neg D \end{array} \quad \mid\!\prec \quad V_1\ [L \neq R_2] \Rightarrow D$$

assuming $R_1 \cap R_2 = \text{null}$.

If the descriptor L has an interval domain type:

$$\begin{array}{l} V_1\ [L = a..b] \Rightarrow D \\ V_2\ [L = c..d] \Rightarrow \neg D \end{array} \quad \mid\!\prec \quad V_1\ [L = e..f] \Rightarrow D$$

assuming $[a..b] \cap [c..d] = \text{null}$ and the following conditions:

(1) if $b < c$ then $e = 0$, $f = c\text{-}1$

(2) if $a > d$ then $e = d+1$, $f = \underline{h}$ (0 and $\underline{h}$ are the minimum and maximum elements in the domain of L)

If the descripor L has a tree structured domain:

$$\begin{array}{l} V_1\ [L = a] \Rightarrow D \\ V_2\ [L = b] \Rightarrow \neg D \end{array} \quad \mid\!\prec \quad V_1\ [L = c] \Rightarrow D$$

assuming the following conditions:

(1) $a \cap b = \text{null}$

(2) the constant (c) is the most general parent of (a) which is not a parent of (b) (c may be equal to a).

2.6.4. The Counting Rule

This generalization rule is a constructive rule, because it creates new descriptors that count the number of parts with a certain property. It is used by the INDUCE procedures that introduce meta-descriptors.

$$V\,[x_1(P_1) = A] \ldots [x_1(P_k) = A] \Rightarrow K \quad |< \quad V\,[\#PS(x_1{=}A) = k] \Rightarrow K$$

where

$P_1, P_2, \ldots, P_k$	are variables denoting parts of an object
x_1	stands for a certain attribute of the P_i's, e.g., color, size, texture, etc.
$\#PS(x_1{=}A)$	denotes a new descriptor interpreted as "the number of P_i's (e.g., parts) with attribute x_1 equal to A".

2.6.5. Generating Chain Properties Rule

If the arguments of different occurrences of the same relation in an event are linearly ordered by the relation (e.g., are objects ordered linearly by a relation *above, left-of, next-to, contains,* etc.), that is form a *chain,* the rule generates descriptors which characterize various objects in the chain; for example,

LST-object:	the "least object", i.e., the object at the beginning of the chain (e.g., the bottom object in the case of relation *above)*
MST-object:	the "most object", i.e., the object at the end of the chain
MDL-object:	the "middle" object
Ith-object:	the ith object in the chain

or characterize the chain itself, for example the *chain-length.*

3. Syntax of Input Rules

This section describes the syntax of the input to the program. A summary of the input commands is given in the user's manual (Appendix 1). The program accepts as input: 1) a set of decision rules, 2) a problem environment description including a set of background knowledge rules, domain definitions, variable costs etc., and 3) a set of parameters which control certain aspects of the program operation. Decision rules, background knowledge rules, and domain structures are entered as VL_2 type formulas in the formats described below.

3.1. Decision Rules

Decision rules must satisfy the following grammar:

```
<n-vlrule>    ::=   <number> <vlrule>
                |   <vlrule>
<vlrule>      ::=   <condition> "==>" <selector>
<condition>   ::=   <selector> <condition>
                |   <selector>
<selector>    ::=   "[" <fn-sym> "(" <alist> ")=" <reflist> "]"
                |   "[" <fn-sym> "(" <alist> ")]"
                |   "[" <variable> "=" <reflist> "]"
                |   "[" <fn-sym> "=" <reflist> "]"
<alist>       ::=   <variable> "," <alist>
                |   <variable> "." <alist>
                |   <variable>

<reflist>     ::=   <snumber> v <reflist>        {see Note below}
                |   <snumber> .. <snumber>
                |   <snumber>
                |   *
<fn-sym>      ::=   string of letters
<variable>    ::=   <fn-sym> <number>
<snumber>     ::=   <symbol> | <number>
<symbol>      ::=   string of letters
<number>      ::=   string of digits
```

Note: In the current implementation, "or" is denoted by "," rather than by "v".

Examples of decision rules:

Example 1:

$$[f(x_1,x_2) = 2 \text{ v } 3][g(x_1) = 1..4] \Rightarrow [D = 1].$$

In this expression, the structure of the domain of the function g is set to interval by the program (see note 4 below). The domain of f is nominal. All variables (e.g. x_1 and x_2) are quantified by the operator $\exists$. (see note 3 below).

Example 2:

$$[p(x_1,x_2)][x_2 = 2 \text{ v } 4] \Rightarrow [D = 2].$$

In this example, the function p is assumed to have two values (i.e., it is a predicate, see note 6 below). The selector containing p is satisfied if it has the value 1. The second selector restricts the possible values of x_2 to the set of values [2,4].

Notes about the application of the VL_2 grammar:

1. The condition part is a single product and the decision part involves only one variable. It is

assumed that one decision variable has been selected to be studied by the user. Also, if the condition part of a formula contains more than one conjunct, it must be split up to create a set of decision rules with condition parts which are single products, via this equivalence rule:

$$V_1 \text{ v } V_2 \Rightarrow D \quad \models \quad V_1 \Rightarrow D,\ V_2 \Rightarrow D$$

2. Each atomic form is a function symbol with a list of single variable arguments. It is assumed that the user has converted forms such as $[g(f(x_i))]$ into a form $[g(y_j)][p(y_j,x_i)]$ by introducing a new predicate $p(y_j,x_i)$for each function symbol f which is in an argument list and a variable y_j for each occurrence of the function $f(x_i)$ in an argument list. The predicate p is assumed to have the value TRUE if $y_j = f(x_i)$ and FALSE otherwise. For example:

$$[\text{shape}(\text{part}(x_1)) = \text{triangle}]$$

is assumed to be transformed into e.g. an expression

$$[\text{shape}(p_1) = \text{triangle}][\text{contains}(x_1,p_1)]$$

3. All variables (arguments) are assumed to be existentially quantified. Variables with the same function symbol part are assumed to have the same domain. Furthermore, variables with values from the same domain are assumed to take on distinct values from the domain. Variables may be restricted to a subrange of values by using the third option in the selector definition. Using this method, a constant may be specified as an argument to a function.

4. If a reference of the second form is specified at least once in the input (e.g. $[f(x_1)=2..2]$ or $[f(x_1)=2..5]$), a domain of type interval is assumed; otherwise, the domain of a variable or function is assumed to be of nominal type or tree-structured if such a structure is specified.

5. A selector such as [p=*] means that the selector is satisfied for any value of p other than the value NA (not applicable). If a selector is omitted from the decision rule entirely, the program assumes that the function has the value NA. Such a domain value has no generalization. Although NA is not specified in an input rule, it can be a valid value of of a variable.

6. If the second form of the selector is used (e.g. $[p(x_1,x_2)]$), the program assumes that the function symbol has the value 1. This may be used to specify a TRUE value for a predicate (predicates are treated as functions with domain [0,1]). In general, to simplify the expressions, only positive values of predicates are specified. The program then uses only positive instances of the predicate in the generalizations. If negative values of a predicate are desired in the generalizations, these relations should be included in the initial decision rule specification. (e.g. $[\text{ontop}(p_1,p_2)]$ specifies that p_1 is on top of p_2; $[\text{ontop}(p_1,p_2)=0]$ specifies that p_1 is not on top of p_2.)

7. If the fourth form of the selector is used (e.g. $[p(x_1.x_2)]$, then the order of the arguments is assumed to be irrelevant.

3.2. Background Knowledge Rules

Background knowledge rules are used to define background knowledge and are entered in along with the events. There are two types of background knowledge rules. The first type (A-rules) generate new descriptors as arithmetic expressions of the initially provided descriptors. The second type (L-rules) define new descriptors as a logical combination of other descriptors.

3.2.1. A-rules

A-rules define an arithmetic derived descriptor. The grammar for A-rules is:

```
<A-rule>     ::=   <fn-sym> "(" <elist> ")=" <expr>
<expr>       ::=   <term> <adop> <expr>
                 |  <term>
<term>       ::=   <factor> <mulop> <term>
                 |  <factor>
<factor>     ::=   "(" <expr> ")"
                 |  <number>
                 |  "-" <number>
                 |  <fn-sym> "(" <elist> ")"
                 |  "-" <fn-sym> "(" <alist> ")"
<elist>      ::=   <variable> "," <elist>
                 |  <variable>
<adop>       ::=   "+" | "-"
<mulop>      ::=   "*" | "/" | "%"
```

<expr> is an arithmetic expression written in standard algebraic form. The variables of the function on the left hand side must appear on the right hand side. All functions and predicates must have linear domains. The operators which may be used are: + (addition), - (subtraction), - (unary minus), * (multiplication), / (integer division - remainder discarded), and % (integer modulus). The operators *, /, and % are evaluated before - and + . Example:

$$\text{girth}(x1) = \text{length}(x1) + \text{width}(x1).$$

3.2.2. L-rules

L-rules are useful for introducing new descriptors which describe groups of parts, expressing the symmetry of a descriptor, expressing the transitivity of a descriptor, etc. The L-rules must satisfy this grammar:

```
<L-rule>          ::=   <condition> "==>" <consequence>
<consequence>     ::=   <selector> <consequence>
                    |   <selector>
```

where <condition> and <selector> are the same as in the decision rule grammar. Notice that the right hand side of an L-rule can be a conjunction of one or more selectors, unlike the decision rule syntax, in which the right hand side was restricted to a single 0-ary function selector. There are further semantic restrictions on A-rules:

(1) The condition must form a connected graph structure (see section 4 - briefly, this means that the selectors that comprise the condition cannot be divided into two sets, such that the variables in one set are completely disjoint from the variables in the other set).

(2) The consequence need not form a connected graph structure by itself, but each connected subgraph of the consequence must be connected to the condition (ie, share some variable with the condition).

The L-rules are applied to the events as a group. For every occurrence of the condition part in the event, the consequence of the L-rule is added to the event, if it is not already there. All the L-rules in the group are applied to all the events, until no further consequences can be added. L-rules are useful for introducing new descriptors, and these new descriptors may describe groups of parts. For example:

$$[\text{man}(x_1)][\text{woman}(x_2)][\text{married}(x_1,x_2)] \Rightarrow [\text{family}(x_3)][\text{consists}(x_3,x_1,x_2)].$$

The group of L-rules to be applied is selected from the data base of all L-rules at the time the induction process is performed. An explanation of how this is done is given in section 4.2.7, as well as a detailed description of how L-rules are applied to events.

3.3. Domain Structure Specification

The domain of a function or variable may consist of a structured set of values. Currently, tree-structured value sets are handled by the program. The structure is described by T-rules according to the grammar

$$\text{<T-rule>} \ ::= \ [\ \text{<fn-sym>} = \text{<ref>}\ ] \Rightarrow [\ \text{<fn-sym>} = \text{<ref>}\ ].$$

where the two function symbols are the same and fn-sym and ref are as in the decision rule grammar. For example:

$$[s = 1,3,4] \Rightarrow [s = 6].$$
$$[s = 0,2] \Rightarrow [s = 7].$$
$$[s = 4] \Rightarrow [s = 8].$$
$$[s = 6,8] \Rightarrow [s = 9].$$

Entering a tree structure automatically sets the type of the domain for the function symbol to the structured type. Each element in the ref of the condition part of this rule must either be a leaf of the corresponding tree or have been previously defined with another T-rule specification.

3.4. Preference Criterion

The preference criterion is defined as a *Lexicographical Evaluation Functional (LEF).* The LEF is defined by a sequence of "criterion-tolerance" pairs (c_1,τ_1), (c_2,τ_2), ..., where c_i is an elementary criterion selected from the list below, and τ_i is a "tolerance threshold" ($\tau \in [0...100\%]$). In the first step, all hypotheses are evaluated on the first criterion, c_1, and those that score best are retained. A parameter set by the user, VLMAXSTAR, is the number of hypotheses that are retained. If any of the remaining hypotheses are equivalent to the worst hypothesis within the range defined by the threshold τ_1, they are also retained. Next, the retained hypotheses are evaluated on criterion c_2 with threshold τ_2, similarly to the above. This process continues until the number of retained hypotheses is reduced to VLMAXSTAR or fewer, or the sequence of criterion-tolerance pairs is exhausted. In the latter case, the retained hypotheses have equivalent quality with respect to the given LEF, and any one may be chosen arbitrarily.

The user specifies the order in which the optimality criteria will be applied, and the tolerance associated with each one. The default optimality criteria (or optimality cost functions) in the order of application are the following:

(1) Minimize the inconsistencies of a rule with tolerance 0.30, i.e., the number of negative events covered by the rule (this is cost function number 3 in section 4.2.3). This allows the program to produce consistent generalizations quickly. The high tolerance removes highly inconsistent rules while leaving selection of nearly consistent rules to the remaining cost functions.

(2) Minimize the number of products in the complete generalization with tolerance 0.00 (this is cost function number 1 in section 4.2.3).

(3) Minimize the number of selectors in each product produced by the program (function 2 in section 4.2.3).

(4) Minimize the cost of functions in each product (function 4 in section 4.2.3). If costs are specified, this criterion may be moved forward. If the user wishes certain functions to appear in the resulting products, the costs for these functions may be specified (given negative cost). Similarly, functions which are very difficult or costly to measure may be given appropriate positive costs.

(5) Maximize the intersection of resulting rules (cost function 5 in section 4.2.3). In real situations, the separate products produced by the program may represent a large number of common input decision rules along with some peculiarities of specific situations which arise. Use of this cost function will favor the selection of a more representative result as opposed to one which describes only a particular set of situations. Section 4.2.6 contains a more detailed discussion of a similar cost function in the VL_1 system.

4. Computer Implementation

This section gives a general description of the computer implementation of INDUCE, including the logical representation of VL_2 rules, and the overall algorithm. A very general description of the algorithm is this:

(1) Select class.
After the user specifies which set of events is to be covered, the program applies A-rules and L-rules to the input rules to add selectors to them. It also adds selectors containing meta-functions, equivalence predicates, and extremities predicates.

(2) Focus attention.
Select a positive event (the focus of attention) and generate a *star* for this event (i.e., set of alternative consistent generalizations which cover this event). This is done in two phases:

(i) Find a consistent generalization of the focus-of-attention event by locating the most promising selectors of the event and adding new selectors to each of these selectors until a set of consistent generalizations of this event is obtained.

(ii) The VL_2 rules obtained from (i) are transformed into VL_1 rules in order to use the AQ procedure to extend the references of the functions in the consistent rules. They are then transformed back into VL_2 rules, with extended references.

(3) Select best generalization.
Select the best generalization from this set and remove rules from the set produced in step 1 for which this is a generalization. Repeat steps 3-4 until no more rules remain which involve the decision variable and value of step 2.

(4) Continue by selecting another value of the current decision variable or selecting another decision variable until all decisions have been considered.

The primary purpose of the two phase approach is computation efficiency. It is much faster to compare VL_1 rules (represented as bit strings) than VL_2 rules (represented as graphs). In a recent test, the procedures that test to see if one VL_2 covers another took up over 77% of the total running time. The procedures which perform the AQ algorithm took up less than 4% of the total running time.

A more detailed description of the algorithm is given below, following an explanation of the logical representation of the decision rules.

4.1. Graph Representation of VL Decision Rules

Some of the information in this section appears in [Larson et.al. 1977] and is given here as a background for the description of the computer implementation (INDUCE 3 implements only a subset of the representations which follow). A VL decision rule can be represented as a graph with labelled nodes and directed labelled edges. The labels on the nodes can be: a) a selector containing k-ary descriptors without argument lists, b) a k-ary descriptor without arguments, c) a quantified variable with an optional subrange of values, d) a logical operator. (From here on, a node is referred to by its label, e.g., a selector node means a node with a selector label.) The edges are labelled with integers from 0,1,... . Edges not labelled 0 refer to the position of an argument in the label at the head of the edge. (Edges have non-zero labels only if the position in the argument list of the head node is important. Labels of 0 may be dropped for convenience.)

Several different types of relations may be represented by edges. The type of relation is determined by the label on the node at each end of the edge. The types of relations are:

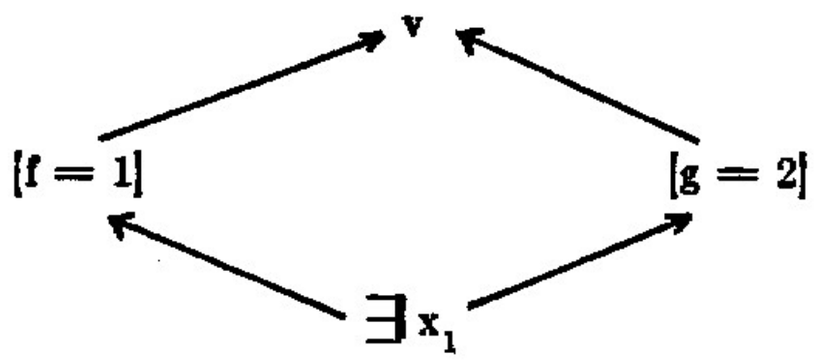
(1) Function Dependence - The label of the head node of the edge has a k-ary descriptor. The value represented by the edge is the value of the atomic form in the tail if the tail is a selector node, a descriptor value if the tail is a descriptor node, or one or all of a set of descriptor values if the tail is a quantified variable. The edge label specifies which argument of the head node assumes this value (Figure 4.1).

$$\exists x_1 \longrightarrow [g = 1..2] \longrightarrow [f = 1] \longleftarrow \exists x_2$$

Functional Dependence: $\exists x_1,x_2([g(x_1)=1..2][f(g(x_1),x_2) = 1])$

Figure 4.1

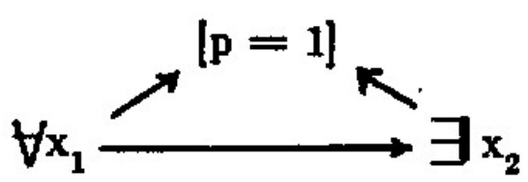
(2) Logical Dependence - The head node is a logical operator (e.g. v, &, $\Rightarrow$) and the tail node is a selector node, or a logical operator node. If the tail node is a selector, then the value represented by the edge is the truth value of the selector at the tail (Figure 4.2)



Logical Dependence : $\exists x_1([f(x_1) = 1] \text{ v } [g(x_1) = 2])$

Figure 4.2

(3) Implicit Variable Dependence - The labels of the head and tail nodes are quantified variables. This type of dependence represents the implicit function (which can be represented by a Skolem function) (Figure 4.3).

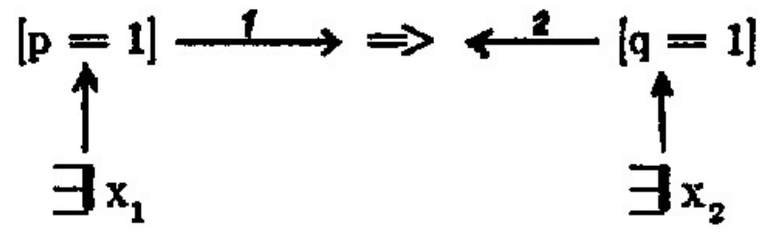


Implicit Variable Dependence: $\forall x_1 \exists x_2[p(x_1,x_2) = 1]$

Figure 4.3

(4) Scope of Variables - The head node is a logical operator and the tail is a quantified variable. This type of dependence may be necessary for certain binary logical operators such as ($\rightarrow$, $\leftrightarrow$). For the

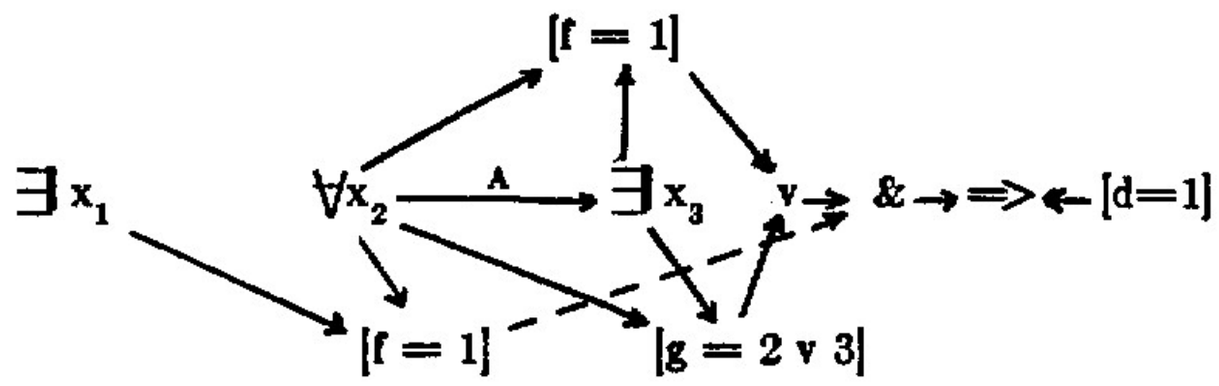
functions v and &, this type of dependence is implicit in the functional dependence of the arguments. (Figure 4.4)



Scope of Variables: $\exists x_1,x_2([p(x_1) = 1] \Rightarrow [g(x_2) = 1])$

Figure 4.4

The graph of a more complex decision rule is given in Figure 4.5. The value of x_3 is dependent in an unspecified way on the value of x_2 (the edge labelled A). The disjunction (v) depends on the values of x_2 and x_3, but this is clearly specified by the functional dependence of f and g on x_2 and x_3. Finally, observe that the decision operator (⇒) does not explicitly depend on the specific values of x_1, x_2, or x_3, but instead depends on the truth value of the entire premise using some set of value assignments for x_1,x_2, and x_3.



Graph Structure Example:

$$\exists x_1 \forall x_2 \exists x_3 (([f(x_2,x_3) = 1] \text{ v } [g(x_2,x_3) = 2 \text{ v } 3]) \;\&\; [f(x_1,x_2) = 1] \Rightarrow [d = 1]$$

Figure 4.5

4.2. Algorithm

This section describes the algorithm in more detail. The program is written in Pascal, and consists of about 6000 lines of comments and code, organized into about 80 procedures. Some major tasks which are performed by some groups of procedures are described below.

4.2.1. Formation of a Complete Generalization

A generalization is found of a set of decision rules containing a specified value I in the decision part. For discriminant generalization, two sets of products are generated: a set F1 which contains all products in the CONDITION parts of rules with a decision value of I, and a set F0 which contains all other products. Each product is called a c-formula (conjunctive-formula).

One c-formula E1 of F1 is selected at random and a conjunctive-formula (c-formula) is generated which is a generalization of E1, consistent with respect to the set F0, and near optimal with respect to a user defined criterion. A c-formula is *connected* if its graph structure representation is weakly connected by functional dependence relations. A c-formula is *consistent* with respect to a set of c-formulas F0 if it does not intersect with any element of the set F0 (i.e., there is no event which satisfies both c-formulas). INDUCE 3 tries to form generalizations which are both consistent and connected. If the input rules cannot be represented as connected graphs, the resulting generalizations may not be connected, but will (if possible) be consistent.

Once a generalization of E1 is found, it is saved in a set CQ and all elements of F1 which are covered by this generalization are removed from F1. One c-formula E1 *covers* another c-formula E0 if E1 is a generalization of E0. Another element of the new set F1 is selected and the procedure repeated. When there are no more elements in F1, the complete, consistent generalization of the set of c-formulas F1 is the disjunction of all c-formulas in CQ.

Characteristic generalization is performed in a similar manner, except that there is no set of events F0. In this case, there is only one decision class, and so a rule (or rules) is found which is as specific as possible and covers all the events.

4.2.2. Determine Cover and Intersection of 2 Formulas

Two similar procedures are decribed here. The test to determine whether a c-formula E covers a c-formula E' is used when E' is an element of the set F1. The test to determine whether E intersects with E' is used when E' is an element of F0 (i.e., to determine if E and E' are consistent). The procedure uses the graph structure representations of E and E' (G and G' with nodes and edges V,E,V',E' respectively). Unlike previous versions (e.g., INDUCE 2), the graph G is not assumed to be weakly connected. E covers E' if there is a *specializing isomorphism* (s-isomorphism) from G to a subgraph of G'. The reverse mapping (from a subgraph of G' to G) is called a *generalizing isomorphism* (g-isomorphism). E intersects with E' if there is an *intersecting isomorphism* (i-isomorphism) between G and a subgraph of G'. Each isomorphism from G to a subgraph of G' is a 1-to-1 correspondence between nodes and edges of G and a subset of nodes and edges of G' where the correspondence (or matching) of nodes and edges is defined as follows:

A node n of G *matches* a node n' of G' if each of the following conditions is true:

(1) They are both selector nodes or both quantified variable nodes.

(2) If they are selector nodes, then the function symbols in both nodes are the same. If they are variable nodes, they are of the same group of variables.

(3) With an s-isomorphism or g-isomorphism, the set of values associated with n is a generalization of the set of values associated with n'. (The sets of values may be equal.) With an i-isomorphism, the sets of values intersect. In the case of selector nodes, these values are the elements in the reference of the selector. In the case of quantified variable nodes, these values are the subranges of the variables.

An edge of G matches an edge of G' if these two conditions hold:

(1) They have the same label

(2) The respective head nodes match and the tail nodes match.

To speed rejection, a quick scan through the nodes is made to see if there is a correspondence between nodes of G and a subset of nodes of G' (ignoring links between nodes). If there is a possible correspondence, a procedure is invoked which locates a subgraph of G' which is isomorphic to G and assigns each node of G to a corresponding node of G'. The procedure is as follows:

(1) Select an un-visited starting node ($n_{0,G}$) of G which contains the most labelled and ordered incoming edges. (This is the selector node with the largest number of non-order-irrelevant arguments.) Selecting a node of this type insures that there is a minimum of backtracking through the starting node.

(2) From the starting node $n_{0,G}$ perform a depth-first traversal of G. The ordered list of nodes visited in the traversal is saved in the *node-list* and becomes the order in which nodes of G' will be matched. If the traversal does not visit every node of G, then G must consist of more than one graph (i.e., G is not connected). In that case, steps (1) and (2) are repeated until all nodes of G have been visited and appear in the list. The final list of nodes in graph G is the concatenation of all traversals made over different parts of G.

(3) Matching nodes in G' are found for each node of G, ordered by the node-list. Let $n_{i,G}$ and $n_{i,G'}$ denote the ith matching nodes, where $n_{i,G}$ is the ith node in the node-list. In order for a node $n_{i,G'}$ of G' to match a node $n_{i,G}$ of G, both must involve the same descriptor symbol, have the same value, and node $n_{i,G'}$ must lie at the end of an edge connecting it to node $n_{i-1,G'}$ which has already been matched to node $n_{i-1,g}$. (This last condition is dropped if node $n_{i,G}$ is a starting node.) As matching nodes are found, they are *assigned* to each other and a record of the matching nodes and edges is kept for each assignment in a backtrack list. To establish a 1-to-1 correspondence, nodes of one graph which are previously assigned can only match corresponding assigned nodes of the other graph.

(4) If there is no node of G' which matches a node of G, the procedure backtracks to the previous node on the backtrack list, erasing the last nodes on the backtrack list and the assignments associated with them. If no node of G' can be found at this point, then the procedure again backtracks until a new match is found or the backtrack list is exhausted.

(5) If the processing of the node-list G is complete, then G covers (intersects) G'. If the backtrack list is exhausted, then G does not cover (intersect) G'.

A feature is included which finds all subgraphs of G' which are isomorphic to G. This feature is used in extending the references of a consistent c-formula (section 4.2.5) and in adding background knowledge to c-formulas.

(6) If the traversal of G is complete, then the current set of assignments is the desired mapping. To find the next isomorphism, the procedure returns to step 4 assuming that the last node on the backtrack list did not match.

If all subgraphs of G' that are isomorphic to G are desired, the algorithm may be called repeatedly to find additional isomorphisms. The algorithm saves the state of each successful search so that the search many be continued to find additional isomorphisms.

4.2.3. Trimming a Set of c-formulas

Trimming is the process of selecting the MAXSTAR best elements of a set of c-formulas with regard to a user defined criterion. The user specifies the cost functions which are to be used, the order in which they should be applied, and the tolerance associated with each cost function. Implemented cost functions are:

(1) The number of events of the current set F1 which are covered by a c-formula. (The negative of this quantity is used to obtain a cost.) This function minimizes the number of c-formulas in CQ.

(2) The number of selectors in a c-formula. The function minimizes the number of selectors in the c-formula.

(3) The number of events of F0 which intersect with a c-formula. This function leads more rapidly to consistent c-formulas.

(4) The total cost of all functions contained in a c-formula.

(5) The number of events of the original set F1 which are covered by a c-formula. (The negative of this quantity is used to obtain a cost.) This function finds the most representative c-formulas.

A set of c-formulas is trimmed using n cost functions (cf_1,cf_2,...cf_n) and relative tolerance for each cost function (tol_1,tol_2,...tol_n). The costs are applied in the order specified by the user (cf_1 first, cf_2 second, etc). For each cost function cf_i, the MAXSTAR best c-formulas along with all c-formulas equivalent in cost to the MAXSTAR best c-formulas are passed to the evaluation using the next cost function cf_{i+1}. Other c-formulas in the set of c-formulas are discarded. With the last specified cost function (cf_n), only the MAXSTAR best c-formulas are retained.

For each cost function cf_i, i=1,2,...,n, equivalence of two c-formulas in cost is defined using an absolute tolerance (AT_i). Suppose the set of c-formulas P is composed of a list p_1,p_2,...p_m. After values for cost function cf_i have been evaluated $cf_i(p_j)$ for each c-formula p_j, the maximum and minimum cost function values are determined $cf_i(p_{max})$ and $cf_i(p_{min})$. An absolute tolerance (AT_i) is calculated using the user specified tolerance tol_i as follows:

$$AT_i = tol_i * (cf_i(p_{max}) - cf_i(p_{min}))$$

The MAXSTAR c-formulas of least cost are determined and the list reordered (p_1,p_2,...,$p_{MAXSTAR}$,...,p_m). If i<n then c-formulas which are not equivalent in cost to $p_{MAXSTAR}$ are discarded.

$$P = P - [p_j : cf_i(p_j) - cf_i(p_{MAXSTAR}) > AT_i]$$

If i=n, (the last cost function) then only the MAXSTAR best c-formulas are retained.

$$P = P - [p_j : j > MAXSTAR].$$

The set of c-formulas which remains is the desired trimmed set of c-formulas.

4.2.4. Formation of a Set of Consistent Generalizations

A *star* (denoted by MQ) is formed which covers E1. (The star is a set of consistent c-formulas which cover E1.) The procedure begins by forming a partial star (P) which contains a set of c-formulas each consisting of one selector of E1. (The *partial star* may contain c-formulas which are not consistent with respect to F0.) This partial star is trimmed according to the user supplied optimality criterion. The conjunction in each c-formula which remains after trimming is multiplied by each selector of E1 which is directly connected to it (ie, shares one or more variables with it) to form a new partial star. Consistent c-formulas are placed in MQ. The partial star is again trimmed and new selectors added to each product until the desired set MQ of c-formulas is obtained. Several paramenters control the sizes of sets in this procedure:

MAXSTAR - the number of c-formulas in a partial star after trimming

NCONSIST - the minimum number of consistent c-formulas which must be in MQ

ALTER - the maximum number of new alternatives which may be formed by adding selectors to an element of a partial star.

(1) A partial star P is formed which contains all selectors of E1 with unary functions.

(2) P is trimmed to contain only the best MAXSTAR c-formulas. Consistent c-formulas are placed into MQ. If fewer than NCONSIST elements are in MQ, then step 3 is executed. Otherwise, the AQ procedure is applied to the elements of MQ (as described in section 4.2.5).

(3) A new partial star P' is formed from the old one (P). For each element p_i in P, a list of all variables found in p_i (i.e., arguments of selectors of p_i) is formed. All arguments of equivalence type selectors which occur in the corresponding selector of E1 are also included in the list.

(4) For each element p_i in P, a list of all selectors of E1 which are not already in p_i and which have at least one argument in the variable list (found in step 3) is created. If there are more than ALTER elements in this list, the best ALTER selectors are retained (using as a criterion the cost of functions in the selectors).

(5) For each of these selectors, a new c-formula is formed which contains the original selectors in p_i and the new selector. If the new c-formula contains an equivalence selector with only one argument, then the new c-formula is discarded; otherwise, it is placed in P'.

(6) If no new c-formulae are formed, a search is made for any selector to add. It is in his way that disconnected parts of the graph are added.

(7) Steps 2 through 5 are repeated, setting P = P' in step 2 until NCONSIST elements are in MQ or until no new elements are in the new partial star P'.

The algorithm for characteristic generalization is similar to the above. The partial star is formed the same way - starting with all the unary selectors of E1, and the rules are grown by adding selectors to them as described above. The difference is that a rule is placed into MQ not when it is consistent (this has no meaning in this case), but when it fails to cover a certain percentage of the events. This percentage is specified by the parameter MINCOVER. NCONSIST such MQ rules must be found before the growing algorithm terminates. If MINCOVER=100, for example, fairly trivial rules will usually be found. If MINCOVER=50, more interesting rules will be found, but these rules may not cover all of the events.

4.2.5. Extending the References of a Consistent c-formula

Each consistent c-formula of MQ (obtained in section 4.2.4 step 2) contains an alternative, near optimal conjunction of selectors of E1 which distinguishes E1 from any c-formula of F0. Using some methods developed for the program AQ7, the reference of each of these selectors may be generalized to obtain a consistent c-formula which will possibly cover
more c-formulas of F1.

Given a graph G of a consistent c-formula mq in MQ, a *c-structure* is created (G*) by replacing all references of nodes of G with * (the complete set of values for the function in the selector). The nodes of G* are enumerated n^*_i (i=1,2,...,m) and a VL_1 system is created with each VL_1 variable x_i related to a node n^*_i of G*. The domain (denoted D_i) of variable x_i is the same as the domain of the function or variable in the node n^*_i.

The VL_1 event space may be defined:

$$E = D_1 \times D_2 \times \ldots \times D_n.$$

Two sets of VL_1 complexes L and L' are formed from the events of the current set F1 and the set F0 respectively. Individual complexes in these sets are denoted l_i and l'_i. Each complex covers a set of points in the space E. For each element of F1 and F0 all isomorphisms from the c-structure G* to the graph representation G of a c-formula in F1 or F0 are determined. For each isomorphism obtained from F1 and F0, a VL_1 complex is created (l_i or l'_i). Denoting the value sets of the nodes in a subgraph of G which is isomorphic to G* as $R_1, R_2, \ldots, R_m$, the corresponding VL_1 complex may be written:

$$[x_1=R_1][x_2=R_2] \ldots [x_m=R_m].$$

This complex covers the VL_1 events:

$$R_1 \times R_2 \times \ldots \times R_m$$

in the event space E.

First, the complex l_1 which results from extracting the values from the nodes of the graph of mq is generated. Then all other isomorphisms from G* to c-formulas of F1 are determined and a complex added to L for each isomorphism which results in a new complex that is not already in L. The set L' is created in a similar manner by generating all distinct complexes resulting from isomorphisms from G* to c-formulas of F0.

Since the c-formula mq is consistent with regard to F0, the complex l_1 is disjoint from all complexes in L'. (That is, there is no point in the VL_1 event space E which is in both l_1 and a complex of L'.) However, other complexes l_i in L may not be disjoint from L'. A near optimal extension of l_1 against L' in E may be calculated using a version of the AQ7 program. The best complex in this extension (l_q) is calculated according to a user defined criterion. The l_q complex is converted to a c-formula by replacing the value set of each node n^*_i of G* by the reference of the selector with variable x_i in l_q. This c-formula is consistent with respect to the set F0. (This is evident since, if there were a VL_1 event which satisfied both the c-formula from l_q and a c-formula of F0, then one could find a VL_1 complex -- using an isomorphism between G* and the c-formula of F0 -- which intersected with l_1 and L'. But since l_1 and L' are disjoint, this can not happen.)

The cost function for the AQ procedure computes the cost of a complex. Cost functions may be selected from the following:

(1) The number of elements in L which are covered by a complex but not covered by any previous l_q. This is the AQ counterpart to the cost function 1 in section 4.2.3) (Use the negative of this value to

get a cost.)

(2) The number of selectors in a complex (the AQ counterpart to function 2 in section 4.2.3).

(3) The number of elements of L covered by a complex which are associated with different events of F1.

(4) The total cost of variables which appear in a complex (i.e., the cost of functions or variables in associated nodes of G*). This is the counterpart of the function 4 in section 4.2.3.

(5) The total number of events in L covered by a complex.

(6) The number of events in L' covered by a complex (the AQ counterpart to function 3 in 4.2.3).

Trimming is done in the manner described above for c-formulas (sec 4.2.3). A MAXSTAR parameter is specified for this procedure and the l_q is selected from the extension using a MAXSTAR value of 1. The next section gives a more complete description of the AQ procedure.

4.2.6. The AQ Procedure

This section contains a very brief description of the AQ procedure implemented in the program INDUCE 3. The reader is referred to [Larson et.al. 1975] for further details.

Let $L=\{l_1,l_2,..,l_m\}$ and $L'=\{l'_1,l'_2,...,l'_{m'}\}$ be the sets of complexes described in the previous section. (These two sets are denoted F1 and F0 respectively in other references; the above notation was selected to avoid confusion with sets of c-formulas.) The program selects one element (e_0) from L and forms a star about this element (a *star* about e_0 is a set of complexes each of which contains e_0 but does not intersect with any element of L' and is nearly maximal under inclusion). One element (l_q) is selected from the star using the optimality criterion and placed in a set of output complexes; all other elements of L which are covered by this l_q are removed from L and the process is repeated with a new e_0 until the set L is exhausted.

A star is generated by forming a sequence of elementary stars and partial stars, one for each element of L'. An elementary star ($ES(e_0,l'_i)$ or ES_i) is a set of complexes which covers e_0 does not intersect with l'_i, and is maximal under inclusion and under domain structure constraints. A *partial star* ($P_i(e_0)$ or P_i) is a set of complexes which contain e_0 but do not intersect with any l'_j, $j<=i$ (note that $P_{m'}$ is in fact a star). To generate an elementary star ES_i, the extension against rule is applied to each selector of e_0 in the context of l'_i. The result is a set of selectors, each one corresponding to one selector of e_0. To form a partial star P_{i+1} from a partial star P_i, each element of P_i is multiplied by each element of ES_{i+1} (i.e., the set of complexes in P_i and ES_i may be viewed as a logical sum of products of selectors; the multiplication is then the normal result of expanding a product of sums in the VL_1 system).

The partial star P_0 is initialized to be the entire event space E. As each partial star P_i is generated, absorption laws are applied to discard any complex of P_i which is contained in another complex of P_i. The partial star is then trimmed to AQMAXSTAR number of elements using a procedure identical to that in section 4.2.3. The final partial star ($P_{m'}$) is trimmed with an AQMAXSTAR value of 1 to produce l_q. If a parameter LQST is set (has the value TRUE), then l_q is stripped down to an expression with the following properties:

(1) the stripped l_q contains the same variables as the original expression,

(2) the stripped l_q covers the same elements of L as the original,

(3) the reference of each selector of l_q contains the fewest elements of all complexes satisfying 1 and 2

above under domain structure constraints (i.e., interval variables must have a range of values in the reference).

If the set L' is null, then (1) above is replaced with:

(1) the stripped l_q contains all variables $x_1, x_2, \ldots, x_n$.

The latter condition occurs when generating descriptive descriptions covering only one set of complexes (see section 4.2.4). Stripping is done by finding the disjunction of all complexes in L covered by l_q and then adjusting the reference of each selector in the sum to conform to the domain structure (i.e., for interval domains, this involves filling the gaps to while form an interval, while for tree structured domains, this involves finding the lowest level generalization of all elements in the reference).

A simple example may clarify the procedure. Given 3 variables with domains:

Variable	Structure	Values
x_1	nominal	[0:2]
x_2	interval	[0:3]
x_3	tree-struc	[0:6]
	0,1,2=>5	
	3,4=>6	

and input complexes and parameters:

L: l_1: $[x_1=0]$ $[x_2=1]$ $[x_3=2]$
l_2: $[x_1=1]$ $[x_2=1]$ $[x_3=0]$
l_3: $[x_1=1]$ $[x_2=2]$ $[x_3=0]$

AQMAXSTAR = 2
LQST = TRUE

L': l'_1: $[x_1=1]$ $[x_2=2]$ $[x_3=3]$
l'_2: $[x_1=0]$ $[x_2=3]$ $[x_3=1]$
l'_3: $[x_1=3]$ $[x_2=2]$ $[x_3=4]$

cost functions: -1 (maximize number of complexes covered)
2 (minimize number of selectors)
tolerance = 0 for both functions.

Let $e_0=l_1$, the elementary star ES_1 and the partial star P_1 contain:

ES_1: $[x_1=0 \vee 2]$, $[x_2=0..1]$, $[x_3=5]$

since multiplication by the entire space E (P_0) leaves each element unchanged. The trimmed P_1 is

P_1: $[x_2=0..1]$, $[x_3=5]$

since each of these selectors cover at least 2 elements of L while the selector $[x_1=0,2]$ covers only 1 element. The elementary star ES_2 is

ES_2: $[x_2=0..2]$, $[x_3=2]$

since the references of the variable x_1 intersect in l_1 and l'_2. The resulting partial star P_2 is

P_2: $[x_2=0..1]$, $[x_2=0..1][x_3=2]$, $[x_2=0..2][x_3=5]$, $[x_3=2]$

which may be reduced by absorption to

P_2: $[x_2=0..1]$, $[x_2=0..2][x_3=5]$, $[x_3=2]$.

Since the third element of this partial star covers only one element of L, the trimmed partial star leaves

P_2: $[x_2=0..1]$, $[x_2=0..2][x_3=5]$

Now considering l'_3, ES_3 is

ES_3: $[x_1=0 \vee 1]$, $[x_2=0..1]$, $[x_3=5]$

and P_3 after absorption becomes

P_3: $[x_2=0..1]$, $[x_2=0..2][x_3=5]$

The l_q selected from the star P_3 is

l_q: $[x_2=0..2][x_3=5]$

since this complex covers all three complexes in L. Stripping reduces the complex to

l_q: $[x_2=1..2][x_3=5]$

If the set L' is ignored, a descriptive generalization of L can be formed

l_q: $[x_1=0,1][x_2=1..2][x_3=5]$.

4.2.7. Background Knowledge Rules

Background knowledge rules are entered along with the events, and are kept in a list. There are two types of background knowledge rules: A-rules, which add arithmetically derived descriptors, and L-rules, which add logically derived descriptors. This section discusses the application and use of L-rules.

Each L-rule is represented as a single graph structure, but the program treats the condition of the L-rule as a connected subgraph by itself. This is necessary because the program matches only the condition of the L-rule against an event.

Application of an L-rule to an event

The program determines if the condition of the L-rule covers the event by trying to find a specializing isomorphism from the connected subgraph that represents the condition, to the graph that represents the event. If it does, it adds the nodes in the consequence to the event, if they are not already in the the event. The test to see if a node is already in the event is done as follows: The node is already in the event if it has the correct function symbol and value set, and its neighbors are in the event. This recursive test bottoms out because the nodes in the condition of the L-rule are already in the event, by definition.

For example, figure 4.6 shows the application of an L-rule to an event. The L-rule is

$$[P(x_1)] \Rightarrow [Q(x_1,x_2)][R(x_2)]$$

and the event is

$$[M(y_1)][P(y_2)][Q(y_2,y_1)].$$

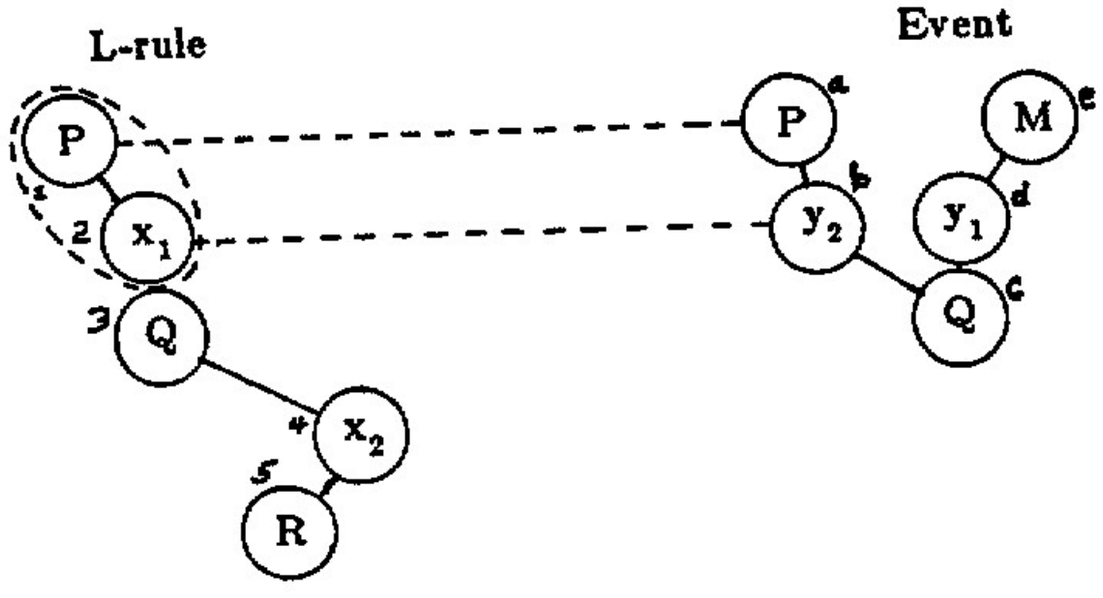


Figure 4.6

Nodes 1 and 2 consititute the condition of the L-rule, and an isomorphism has been found between the condition and nodes a and b of the event. (This means that nodes 1 and 2 are in the event, and a list of assignments records this, as in the procedure described in section 4.2.2.) Now node 3 potentially matches node c and node 4 potentially matches node d, but node 5 does not match node e (different function symbol). (If node 5 had been a function symbol M, they all would have matched.) Thus, none of the nodes {3,4,5} match, and they are all added to the event. Figure 4.7 shows the event after the new nodes have been added. The above procedure is done for every isomorphism that is found between the condition of the L-rule and the event.

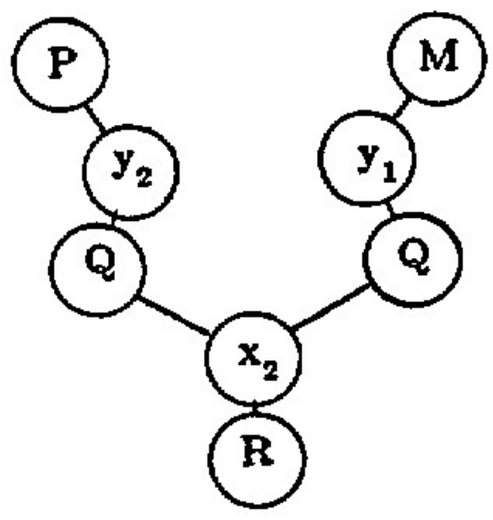


Figure 4.7

Viewing L-rules as logical implication

The above procedure and its effects can be more clearly understood by viewing L-rules and events as sentences in first order predicate calculus. For example, the above event can be represented in FOPC:

$$\exists .y_1,y_2\ P(y_2)\ M(y_1)\ Q(y_2,y_1) \rightarrow D$$

(where D is a predicate stating the decision class membership) and the L-rule as

$$\forall x_1\ (\ P(x_1) \rightarrow \exists .x_2\ Q(x_1,x_2)\ R(x_2)\ )$$

Using the rules of logic, the event can be transformed into

$$\exists .y_1,y_2\ \ P(y_2)\ M(y_1)\ Q(y_2,y_1)\ \&\ \exists .x_2\ Q(y_2,x_2)\ R(x_2)\ \rightarrow\ D$$

This cannot be further simplified. However, if R had been an M, then the event

$$\exists .y_1,y_2\ \ P(y_2)\ M(y_1)\ Q(y_2,y_1)\ \&\ \exists .x_2\ Q(y_2,x_2)\ M(x_2)\ \rightarrow\ D$$

could have been simplified to

$$\exists y_1,y_2 \; P(y_2)\, M(y_1)\, Q(y_2,y_1) \rightarrow D$$

which is the same as not adding the consequence of the L-rule to the event.

Application of a group of L-rules

A group of L-rules is applied to the events by applying each L-rule to each event in turn, and repeating the process until no further modifications can be made to the events. By using such a loop, the application of one L-rule can make possible the application of another. For example, the application of the L-rule:

$$[\text{block}(x_1)][\text{block}(x_2)][\text{ontop}(x_1,x_2)] \Rightarrow [\text{stack}(x_3)][\text{top\_part}(x_3,x_1)][\text{bottom\_part}(x_3,x_2)]$$

can make possible the application of

$$[\text{stack}(x_1)][\text{top\_part}(x_1,x_2)][\text{ontop}(x_3,x_2)] \Rightarrow [\text{ontop}(x_3,x_1)]$$

because the predicates "stack" and "top_part" are introduced by the first L-rule and are used by the condition of the second L-rule.

Selecting the L-rules to apply

A facility has been included in the program which selects a group of L-rules from the entire list of L-rules. The L-rules selected are the "best" ones, ie. the ones that will most likely lead to lower cost rules produced by the generalization algorithm. The reason for doing this is that the L-rules increase the size of the events, which increases the memory and time requirements. It is desirable to keep this growth to a minumum.

The group of L-rules is chosen when the user specifies which event set is to be covered, because the choice of L-rules depends on the event set. However, the L-rules have been previously divided into *connected sets*. A connected set of L-rules is one in which the application of any one of its members affects the applicability of only the L-rules in this same set. In other words, for all B_i in S, the application of B_i affects the application of B_{i1}, B_{i2}, ... and the B_{ij} are all in S. The procedure described here chooses a group of connected sets, not a group of individual L-rules, since it does not make sense to split up the L-rules in a connected set.

The group of connected sets of L-rules is chosen by trimming the entire list of all connected sets, according to user defined cost criteria. This procedure is almost identical to that used for trimming a set of c-formulas, described in section 4.2.3. The cost functions used are as follows (listed in default order):

(1) The number of events of the current set F1 which are covered by the connected set (the negative of this number is used to obtain a cost). A connected set of L-rules covers an event if any of its members do.

(2) The number of events of F0 which are covered by the connected set.

(3) The number of selectors in the condition parts of the L-rules in the set (the negative of this number is used to obtain a cost). L-rules with large conditions are desirable because they potentially lead to great reductions in the size of the final generalization, i.e. when the selectors in the condition of the L-rule are dropped in favor of the selectors in the consequence.

(4) The number of selectors in the consequence parts of the L-rules in the set. L-rules with large consequences are undesirable because they greatly increase the size of the events.

The connected sets of L-rules are trimmed according to the parameters specified by the user: the order of cost functions, their tolerances, and the value of MAXL, which is the maximum number of connected sets to be applied.

4.2.8. Adding New Functions and Predicates to c-formulas

The program currently has the capability of automatically adding three types of new attributes to existing c-formulas: 1) global descriptors (*meta functions*) which count the frequency of occurrence of selectors with unary functions; 2) equivalence type predicates of the form [f(x_1,x_2)=same] i.e., the value of f is the same for x_1 and x_2; 3) extremity type predicates ([lst-f(x_1)] or [mst-f(x_1)]) which indicate that the argument x_1 is at one end of a sequence of binary predicates. These functions are added at the user's discretion.

Meta Selectors

There are two types of meta functions currently calculated and added to a c-formula as a meta selector: One type (#PT(f=a) where f is an atomic function and a is a value in D(f)), counts the number of times a particular selector ([f(..)=a]) appears in a c-formula. The second type (FORALL(f=a)) is a predicate which is true if a function assumes only one value in a c-formula and false otherwise.

For each atomic function-reference pair which appears in any c-formula, a meta selector is added to the c-formula which has a meta function in the referee. For example, a c-formula

$$[tx(x_1)=1][tx(x_2)=1][sh(x_1)=1][sh(x_2)=0]$$

generates the four meta selectors:

[#xs(tx=1)=2] - the number of parts (x's) with tx=1 is 2
[#xs(sh=0)=1] - the number of parts (x's) with sh=0 is 1
[#xs(sh=1)=1] - the number of parts (x's) with sh=1 is 1
[forall-xs(tx=1)] - all parts (x's) have tx=1.

Since the number of such selectors may be quite large, the list of meta functions is trimmed to a small set. The size of the set is determined by a parameter METATRIM supplied by the user using as criteria the degree to which a value of the new function will separate the sets F1 and F0. For each meta function-value combination (meta selector) which is generated, the number of c-formulas of F1 and F0 which satisfy the selector is calculated. Associated with the meta function (mf) are two numbers: F1COV and F0COV. F1COV is the maximum number of c-formulas of F1 covered by a meta selector arising from the meta function mf. F0COV is the number of c-formulas of F0 which are covered by the meta selector which gave the highest F1COV value.

The list of possible meta functions is trimmed to METATRIM remaining meta functions by sorting in descending order, the list of meta functions according to the primary field F1COV and the secondary field F0COV and selecting the first METATRIM functions from this list. The meta selectors which result from applying each of the selected meta functions to each c-formula are automatically appended to the c-formulas and carried with each c-formula during the generalization process. Setting METATRIM to 0 bypasses the entire meta selector generation process.

Equivalence Predicates

These predicates may have arbitrarily many arguments whose order is irrelevant. They are calculated by scanning all selectors in each c-formula for sets of 2 or more selectors with unary functions which have the same atomic function and reference. Such a set of selectors is said to be equivalent and a new predicate is created which contains an argument from each of the atomic forms of all equivalent

selectors. For example, a c-formula of the form:

$$[f(x_1)=1][f(x_2)=1][f(x_3)=1][f(x_4)=2][f(x_5)=2]$$

leads to the creation of predicates:

$$[samef(x_1.x_2.x_3)][samef(x_4.x_5)]$$

Extremeties Predicates

These are unary predicates which represent the ends of a sequence of selectors with the same binary predicate:

$$[p(x_1,x_2)][p(x_2,x_3)] \ldots [p(x_{i-1},x_i)]$$

where, in the sequence, if i>2 then the first argument of the j-th selector ($1 \leq j < i$) is the same as the second argument of the j+ 1-st selector. Also, the variables x_1 (and x_i) do not appear as the second (or first) argument of another selector with atomic function p. The new predicates formed for the arguments x_1 and x_i of such a sequence are written:

$$[MST\text{-}p(x_1)] \text{ and } [LST\text{-}p(x_i)].$$

For example, the situation in figure 4.8 is described by the c-formula:

$$[ontop(x_1,x_2)][ontop(x_1,x_3][ontop(x_2,x_4)]$$

Adding extremeties predicates would give rise to the new predicates:

$$[MST\text{-}ontop(x_1)][LST\text{-}ontop(x_3)][LST\text{-}ontop(x_4)]$$

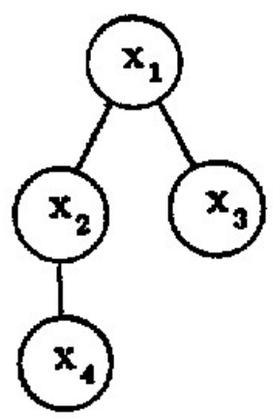


Figure 4.8

5. Examples of Decision Rule Generation using INDUCE 3

This section presents examples of runs using INDUCE 3. For each example, a sample of the input decision rules is listed along with tables describing function domains and program parameters. The resulting generalizations are given as decision rules in disjunctive normal form. The set of objects associated with a description below is given as the value of the decision part of the rule (e.g., for a set O_1, the associated decision part is [d=1]). With some examples, several alternatives were produced by the program. Each alternative which was produced is given as a separate decision rule. The various symbols contained in these tables are defined below:

Program Parameters:

Parameter	Description
ALTER	the number of alternatives which are retained in a partial star (see section 4.2.4).
AQCRIT	the list of cost functions used in the AQ procedure in the order of application (the minus sign indicates that the quantity which is calculated by the function is negated to obtain the cost of the complex) (see section 4.2.5).
AQMAXSTAR	the MAXSTAR parameter for the AQ procedure. This parameter specifies the number of alternative complexes retained in each partial star in the AQ procedure (see section 4.2.6).
AQTOLERANCE	the tolerance associated with each cost function. If this figure is less than 1, then it is a relative cost, otherwise it is an absolute tolerance (see section 4.2.3).
EXTMTY	the predicates of the type [mst-f(x_i)] and [lst-f(x_i)] were added to input decision rules.
EQUIV	the predicates of the type [f(x_1.x_2)=same] were added to the input decision rules.
LQST	the LQST parameter in the AQ procedure. If this parameter is set, then each l_q which is generated in the AQ procedure is stripped so that the reference in the selector for each variable in the l_q complex contains the minimum number of elements under the constraints of the domain structure while covering the same number of events of L (see section 4.2.6).
LRCRIT	the list of cost functions used to trim the number of L-rule sets down to MAXL. Values are entered in the same way as in VLCRIT. A description of available cost functions is in section 4.2.7.
LRTOLERANCE	the tolerance associated with each cost function. Values are entered in the same way as in VLTOLERANCE.
MAXL	the maximum number of L-rule sets that will be applied to the input events.
METATRIM	the number of meta-selectors of the type "forall" and #ps(number of parts) which are added to each formula.
NCONSIST	the number of consistent generalizations which are created in the procedure which forms consistent generalizations (i.e., the minimum number of elements

in the star MQ) (see section 4.2.4).

REGENSTAR	a parameter controlling the processing of rules containing selector reference values of "*" (entire domain) resulting from the application of the AQ procedure. REGENSTAR=1 (the default) causes selectors containing "*" to remain as they were (non-*) before AQ was applied. REGENSTAR>1 causes the "*"-valued selectors to be removed from the rules and rules assembled into a partial vl-star which will again have additional selectors added to it until NCONSIST new rules are produced. ON each such iteration, REGENSTAR is effectively reduced until it reaches 1. At that time the operation for REGSTAR=1 is performed and iteration stops. REGENSTAR=0 causes "*"-valued selectors to remain with "*" as their reference list.
VLCRIT	the list of cost functions used to trim a partial star of c-formulas in the order of application (a minus sign indicates that the quantity which is caluculated by the function is negated to obtain the cost of the c-formula) (see section 4.2.3).
VLTOLERANCE	the tolerance associated with each cost function. A value which is less than 1 is assumed to be a relative tolerance; otherwise, it is an absolute tolerance (see section 4.2.3).

Domain descriptions:

Column	Description
NAME	the name of the function
NARG	the number of arguments of the function
TYPE	the domain structure: NOMinal, LINear (interval), or STRuctured. (Quantified variables have type "vbl"; symbolic values have type "val".)
COST	the cost of the function
MIN	the minimum value in the domain of the function (or value of "val"-type symbolic value)
MAX	the maximum value in the domain of the function
STRUCTURE	the allowable generalizations in tree structure domains

Meta Selectors:

Column	Description
MS	the meta-selector number associated with this function
TYPE	either "#ps" (number of parts) or "forall-ps"
FUNCTION	the associated function and value (i.e., the associated selector ignoring arguments; e.g., "forall-parts(texture=0)" is a predicate which has the value

1 if all parts have a texture of 0 and 0 otherwise)

F1COV — the number of c-formulas in F1 covered by one value of the meta-function.

F0COV — the number of c-formulas in F0 which are covered by the meta-selector associated with the value F1COV above (see section 4.2.8).

5.1. Figures Example

The objects in figure 5.1 (from [Michalski 1977]) are examples of three classes of situations: O_1, O_2, and O_3, each class containing 3 examples. The figures are expressed as VL_2 formulas using 6 descriptors below. The descriptors and their domains are as follows:

Descriptor	Structure	Domain
ontop	nominal	0-false, 1-true
inside	nominal	0-false, 1-true
next	nominal	0-false, 1-true
size	interval	small, medium, large
texture	nominal	clear, shaded
shape	tree-struc	triangle, circle, ellipse,rectangle,diamond, u-shape,square, polygon,curved circle, ellipse ⇒ curved triangle, rectangle,square, diamond ⇒ polygon

The input decision rule for the first object in O_1 has the following form:

[ontop(p_1,p_2)] [ontop(p_2,p_3)] [size(p_1)=medium] [size(p_2)=medium] [size(p_3)=large]
[texture(p_1)=clear] [tx(p_2)=shaded] [tx(p_3)=clear] [shape(p_1)=diamond] [shape(p_2)=circle]
[shape(p_3)=ushape] [inside(p_2,p_3)]⇒[d=1].

with domain structure: (note "," denotes internal disjunction)

[shape=circle,ellipse] ⇒ [shape=curved].

[shape=triangle,diamond,square,rectangle] ⇒ [shape=polygon].

The resulting domain table is:

	name	narg	type	cost	min	max	structure
1	forall	0	nom	1	1	1	
2	#pt	0	lin	1	0	0	
3	shape	1	str	1	0	6	0 1 ⇒ 30; 2 3 4 5 ⇒ 29;
4	circle	0	val	1	0	0	
5	ellipse	0	val	1	1	1	
6	curved	0	val	1	30	30	
7	triangle	0	val	1	2	2	
8	square	0	val	1	3	3	
9	diamond	0	val	1	4	4	
10	rectangle	0	val	1	5	5	

11	polygon	0	val	1	29	29
12	size	1	lin	1	0	2
13	p	-	vbl	1	-	-
14	p1	-	vbl	1	-	-
15	small	0	val	1	0	0
16	medium	0	val	1	1	1
17	p2	-	vbl	1	-	-
18	large	0	val	1	2	2
19	d	0	nom	1	1	3
20	ontop	2	nom	1	1	1
21	p3	-	vbl	1	-	-
22	texture	1	nom	1	0	1
23	clear	0	val	1	0	0
24	shaded	0	val	1	1	1
25	ushape	0	val	1	6	6
26	inside	2	nom	1	1	1
27	p4	-	vbl	1	-	-
28	next	2	nom	1	1	1

5.1.1. Generalized decision rules using no new functions

The program was run with the above decision rules several times with different parameters for each run. The first run through the program used modest values for the parameters AQMAXSTAR, ALTER, VLMAXSTAR and NCONSIST and default values and ordering of criteria with no new functions added. The parameter values and resulting generalizations are given below:

Parameters:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
variable name   vtype   vcost      .    trace:
   #pt           lin       1      .    stops:
   size          lin       1      .    print rules: true
   shape         str       1      .
                                  .
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
parameters: general          vl              aq              l-rule
        regenstar    1    vlmaxstar  5    aqmaxstar  4    maxl      5
        metatrim     0    nconsist   5    aqcutfl    20
        desctype  disc    alter      5    lqst       false
        extmty    false   mincover   10
        equiv     false
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            vlnf=3                aqnf=2                lrnf=4
criterion: vlcrit     vltol     aqcrit     aqtol     lrcrit     lrtol
             3         30%        -1       0.00       -1        0.00
            -1        0.00         2       0.00        2        0.00
             2        0.00                             -3        0.00
                                                        4        0.00
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

For the set O_1, the program discovered several alternative generalizations which describe one or two objects in O_1, but could not find any which cover all objects of O_1. The generalization is therefore a set of rules, which taken together completely cover the objects in O_1. The first rule in the set (which covers

the first and second objects in O_1) is the following decision rule:

[inside(p_2,p_1)][shape(p_2)=circle] => [d=1].
(If there is a circle-shaped part inside another part, then make decision d=1.)

The other rule which covers objects one and three in O_1 is:

[ontop(p_1,p_2)][size(p_2)=large][texture(p_2)=clear] => [d=1].
(There is a part on top of a large clear part.)

The description of the set O_2 produced several alternative generalizations which describe one or two parts of O_2 but only one generalization which covered all 3 parts in O_2:

[ontop(p_1,p_2)][shape(p_2)#ushape][shape(p_1)#polygon] => [d=2].
(There is a non-polygon part on top of a non-ushaped part.)

The following generalization covers the three examples O_3:

[ontop(p_1,p_2)][texture(p_1)=shaded][texture(p_2)=shaded] => [d=3]
(There is one shaded part on top of another shaded part.)

5.1.2. Generalization with EXTMTY type predicates

In the second application of the program to the example in figure 5.1, the EXTMTY type predicates were added to each input decision rule. The only difference in parameters is that these new predicates are added to the input decision rules. The results are the following:

For the set O_1, the best generalization was:

[size(p_1)#small][texture(p_1)=clear][mst-ontop(p_1)] => [d=1]
(The top part is a clear, not small, polygon.)

One product was found to cover the set O_2:

[shape(p_1)#polygon][mst-ontop(p_1)] => [d=2]
(The top part is not a polygon.)

The objects in O_3 are covered by the generalization:

[shape(p_1)#curved][texture(p_1)=shaded][mst-ontop(p_1)] => [d=3]
(The top part is shaded but not curved shape.)

5.1.3. Generalizations with meta-selectors

The next pass of the program included meta-selectors which were determined by the program to be the best for describing each set of objects (using as a criteria the values of F1COV and F0COV). The results for O_1 along with the interpretation of the three selected meta-selectors are:

[#ps(texture=clear)=2] => [d=1]
(There are 2 clear parts.)

The selected meta-selectors are:

MS	TYPE	FUNCTION	F1COV	F0COV
1	#ps	texture = clear	3	0
2	#ps	size = medium	3	3
3	#ps	texture = shaded	2	1

The three meta-selectors count the number of parts with each texture and the number of small sized parts.

The set O_2 was covered by the product:

[shape(p_1)=circle][#ps(shape=square)=0][#ps(shape=diamond)=0][#ps(texture=clear)#0] => [d]=
(There is a circle and some clear parts but no squares or diamonds.)

The selected meta-selectors are:

MS	TYPE	FUNCTION	F1COV	F0COV
1	#ps	shape = square	3	4
2	#ps	shape = diamond	3	5
3	#ps	texture = clear	2	0

The most interesting simplification is with the set O_3. Many alternatives were discovered, and the one selected was:

[#ps(texture=clear)=0] => [d=3]
(No parts have clear texture.)

The selected meta-selectors are:

MS	TYPE	FUNCTION	F1COV	F0COV
1	#ps	texture = clear	3	0
2	forall	texture = shaded	3	0
3	#ps	shape = diamond	3	5

5.1.4. Descriptive generalizations

If the program is given the description of only one set of objects at a time, then a descriptive generalization of sorts may be found by the algorithm discussed in section 4.2.4. The significant difference of this algorithm is that the rules are grown (ie, made more specific) until they fail to cover a certain percentage of the set of objects. This percentage is specified in the parameter MINCOVER. By setting MINCOVER = 100, it is guaranteed that a single product will be found which will cover all objects. If MINCOVER < 100, the rules will usually be more specific, but a single product may not be found. In this example, a background knowledge rule (L-rule) was added for the predicate 'ontop' to implement the transitive closure for this predicate:

$$[\text{ontop}(p_1,p_2)][\text{ontop}(p_2,p_3)] \Rightarrow [\text{ontop}(p_1,p_3)].$$

In the trials below, MINCOVER = 100. The resulting descriptions cover all the objects in the given set and contain as many selectors as possible. Of course, since the descriptions of the other two sets of objects are not included, the descriptions are not discriminant but they are complete. The parameters used were:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
variable name    vtype    vcost      .     trace:
   #pt             lin        1     .     stops:
   size            lin        1     .     print rules: true
   shape           str        1     .
                                    .
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
parameters: general              vl                  aq                l-rule
        regenstar     1     vlmaxstar  2     aqmaxstar  4     maxl       5
        metatrim      3     nconsist   2     aqcutfl   20
        desctype   char     alter      2     lqst   false
        extmty     true     mincover 100
        equiv      true
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
              vlnf=3                 aqnf=2                 lrnf=4
criterion:  vlcrit      vltol     aqcrit      aqtol      lrcrit      lrtol
               3          30%       -1         0.00        -1         0.00
              -1         0.00        2         0.00         2         0.00
               2         0.00                             -3         0.00
                                                           4         0.00
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

For the set O_1, the following characteristic description was found:

[#ps(texture=clear)=2][#ps(size=medium)=2][#ps(texture=shaded)=1,2]
& [ontop(p1,p2)][size(p1)=medium][size(p2)#small][texture(p1)=clear]
& [texture(p2)=clear][shape(p1)=polygon][shape(p2)#triangle][lst-ontop(p2)] => [d=1]

(There is a medium-size clear polygon (directly or indirectly) ontop of the bottom part which is a non-small-size clear non-triangle, and there are two clear parts, two medium-size parts, and one or two shaded parts.)

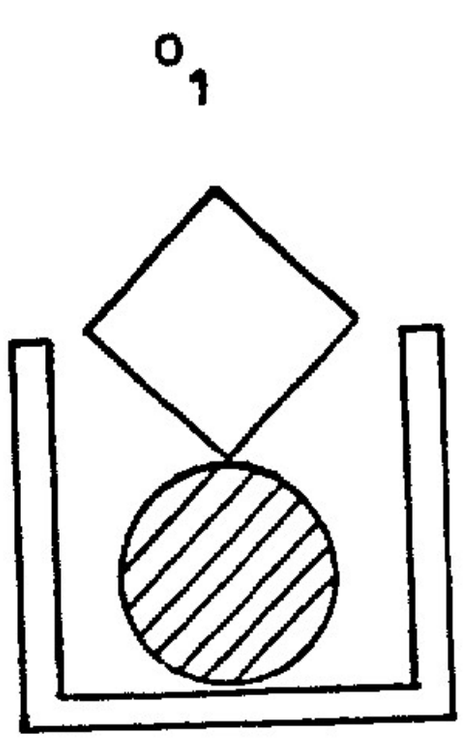
The following product was found for the set O_2:

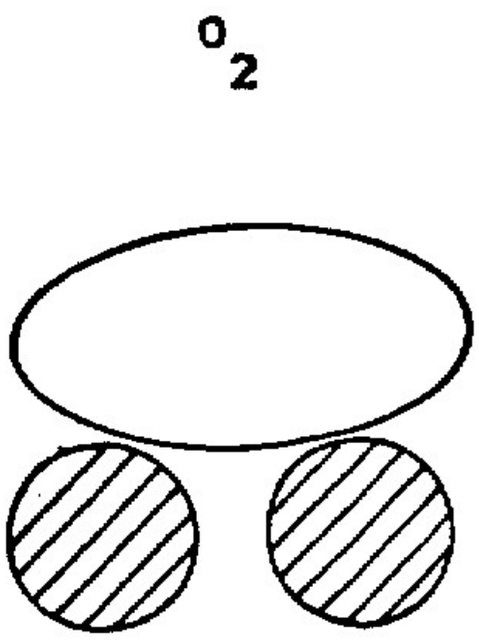
[#ps(shape=square)=0][#ps(shape=diamond)=0][#ps(texture=clear)#0]
& [ontop(p1,p2)][ontop(p1,p3)][size(p1)#small][size(p2)=*]
& [size(p3)#large][texture(p1)=*][texture(p2)=*][texture(p3)=*]
& [shape(p1)=curved][shape(p2)#polygon][shape(p3)#ellipse][lst-ontop(p2)] => [d=2]

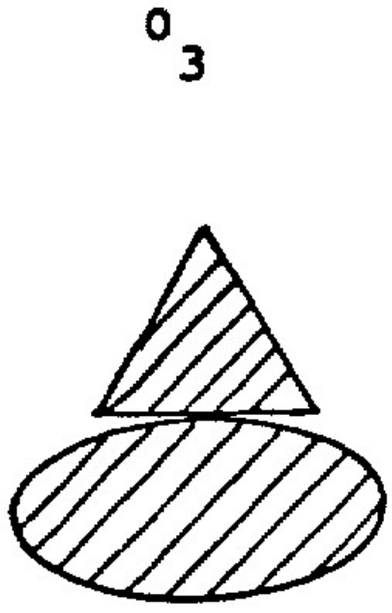
(The is a non-small any-texture curved part (directly or indirectly) ontop of both a bottom part and a non-large any-texture non-ellipse. The bottom part is a non-polygon of any size or texture. There are no square or diamond shaped parts, but some are clear.)

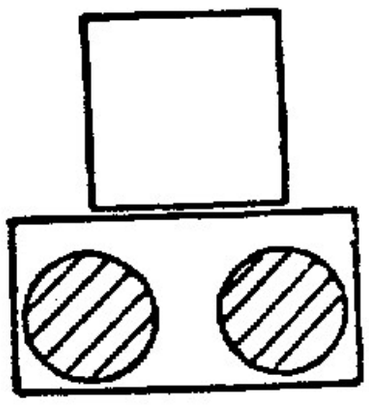
The following product was found for the set O_3:

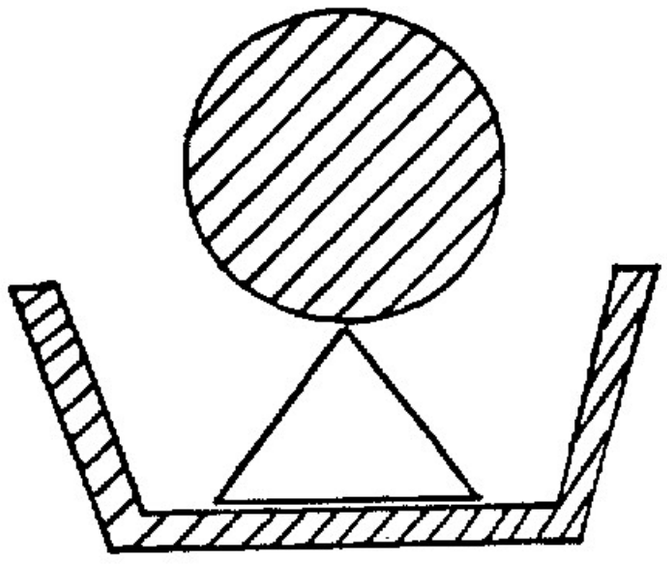
[#ps(texture=clear)=0][forall-ps(texture=shaded)][#ps(shape=diamond)=0]
& [texture(p1)=shaded][shape(p1)#polygon][size(p1)#small][lst-ontop(p1)] => [d=3]
(The bottom part is a non-small-size shaded non-polygon, and there are no clear parts or diamond-shaped parts, and all parts are shaded.)

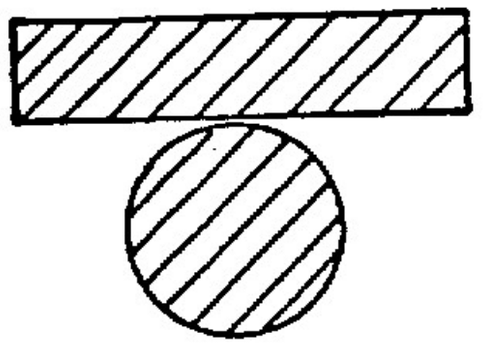


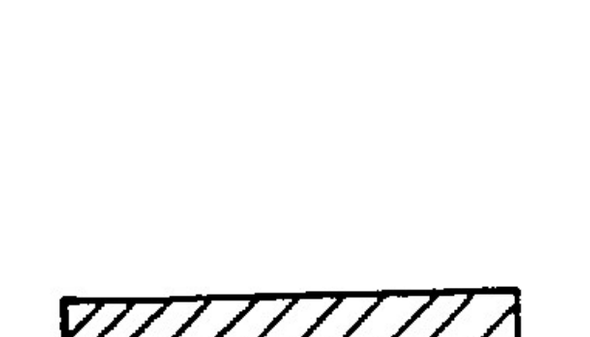












Figures Example
Figure 5.1

Blank Page

5.2. Chemical Example

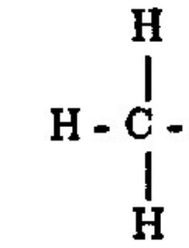
In this example, the program was to learn rules to identify classes of organic molecules. There are three examples of molecules for each class, which are shown in figure 5.2. A molecule is represented by its structural formula - the types of atoms and the bonds between the atoms. The descriptors used were:

t(a1) - a function indicating the type of atom a1, has possible values "c" (carbon), "o" (oxygen), and "h" (hydrogen).

sb(a1.a2) - a predicate which is true if atom a1 has a single bond to atom a2. The period between the arguments indicates that the order of the arguments is irrelevant.

db(a1.a2) - a predicate which is true if atom a1 has a double bond to atom a2.

In the molecules shown in figure 5.2, a single line represents a single bond, and a double line indicates a double bond. For example, the first molecule in the class "alcohols" is pictured below:

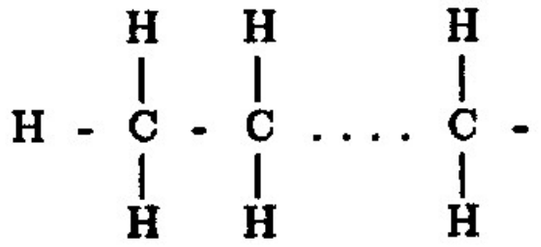


and is represented by the following input decision rule:

[t(a1)=c] [t(a2)=h] [t(a3)=h] [t(a4)=h] [sb(a1.a2)] [sb(a1.a3)]
& [sb(a1.a4)] [sb(a1.a5)] [t(a5)=o] [t(a6)=h] [sb(a5.a6)] => [d=alcohol].

The classes chosen were: alkenes, ethers, carboxylic acids, phenyl ethers, alcohols, and phenols. The characteristics of these classes are given below. The information for this experiment came from *Introduction to Organic Chemistry* by Streitwieser and Heathcock, Macmillan Publishing Co., Inc., N.Y., 1976.

Alkenes Molecules in this class consist of a carbon-carbon double bond, with alkyl groups attached to the carbons, ie. R_2-C=C-$R_{2'}$ where the R's are alkyl groups. An alkyl group is a hydrogen atom or a string of carbons and hydrogens, all single bonded, ie.



Alkyl Group

Alcohols Alcohols are compounds in which an alkyl group replaces one of the hydrogens of water. They are organic compounds of the form R-OH, where R is an alkyl group. For example, the first alcohol example in figure 5.2 is methyl alcohol, CH_3–OH, which has the alkyl group -CH_3 (methyl).

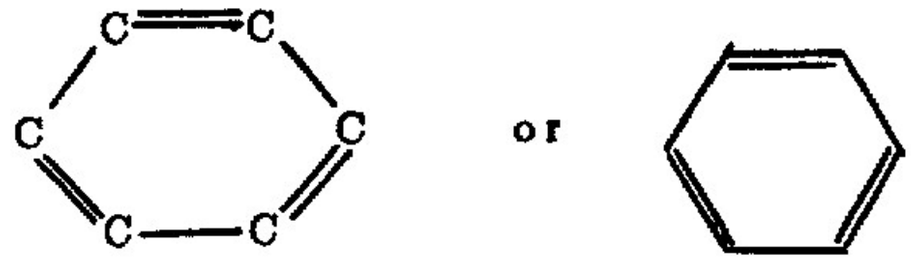
Ethers Ethers are analogs of water in which both hydrogens are replaced by alkyl groups, ie., have the form R-O-R, where the R's are alkyl groups.

Carboxylic Acids

Carboxylic acids are distinguished by the functional group CO_2H. which is called a carboxy group. A hydrogen or an organic group may be attached to the carboxy group, ie. HCO_2H or RCO_2H.

Phenyl Ethers

A phenyl ether is an ether which contains an aromatic (or benzene) ring; ie. Ar-O-R or Ar-O-Ar. The aromatic ring "Ar" is a cyclic group of 6 carbons, and is symbolized:



Phenols

Phenols are a class of compounds that have a hydroxy (-OH) group attached to an aromatic ring.

In addition to the input events shown in figure 5.2, the program was provided with a set of background knowledge rules (L-rules), which introduce new descriptors that name groups of atoms. The new descriptors were:

g(a1) - a function which describes the type of group of group a1. Possible values are "methyl", "chh", "oh", "ar", and "ogrp". These values are explained below.

con(a1,a2) - a predicate which is true if group a1 contains atom a2.

attchd(a1.a2) - a predicate which is true if group a1 is attached to group a2. The groups are attached if an atom in one group is bound to an atom in the other group.

The L-rules which define these descriptors are given below as they actually were given to the program. Percent signs enclose comments.

```
% define the inference rule: if two atoms in different groups are
   bonded together, then the two groups are bonded together %
l
[con(a1,a2)][sb(a2.a3)][con(a4,a3)] => [attchd(a1.a4)].

% define a methyl group: a carbon bound to 3 hydrogens %
l
[t(a1)=c][t(a2)=h][t(a3)=h][t(a4)=h][sb(a1.a2)][sb(a1.a3)][sb(a1.a4)]
=> [g(a5)=methyl][con(a5,a1)][con(a5,a2)][con(a5,a3)][con(a5,a4)].

% define a CH2 group: a carbon bound to 2 hydrogens %
l
[t(a1)=c][t(a2)=h][t(a3)=h][t(a4)=c][t(a5)=c v o]
[sb(a1.a2)][sb(a1.a3)][sb(a1.a4)][sb(a1.a5)]
=> [g(a6)=chh][con(a6,a1)][con(a6,a2)][con(a6,a3)].
```

In the last rule, it was necessary to specify what atoms were on each side of the CH_2 group, although those atoms are not included in the CH_2 group. This is the reason for the selectors [t(a4)=c] and [t(a5)=c v o]. If this were not done, then the L-rule would fire in too many places. In particular, it would fire when matched against a CH_3, or methyl group, since the methyl group does contain a carbon bound to two hydrogens. It is desirable to have an atom belong to at most one group.

```
% define an OH group: an oxygen attached to a hydrogen %
l
[t(a1)=o][t(a2)=h][sb(a1.a2)] => [g(a3)=oh][con(a3,a1)][con(a3,a2)].

% define an aromatic ring group: a ring of 6 carbon atoms %
```

```
]
[t(a1)=c][t(a2)=c][t(a3)=c][t(a4)=c][t(a5)=c][t(a6)=c]
[db(a1.a2)][sb(a2.a3)][db(a3.a4)][sb(a4.a5)][db(a5.a6)][sb(a6.a1)]
=> [g(a7)=ar][con(a7,a1)][con(a7,a2)][con(a7,a3)][con(a7,a4)][con(a7,a5)][con(a7,a6)].

% this rule says that an O bonded to two diff groups is a group itself %
]
[t(a1)=c][con(a2,a1)][sb(a1.a3)][t(a3)=o][sb(a3.a4)][t(a4)=c][con(a5,a4)]
=> [g(a6)=ogrp][con(a6,a3)].
```

The last rule considers the case of an oxygen atom bound to two carbon atoms, each of which are in different groups. It defines the oxygen atom to be a group unto itself. This was done because it is desirable to be able to consider the molecule solely at the level of groups, rather than at a level of a mixture of atoms and groups.

Each of the classes: alcohol, ether, alkene, carboxylic acid, phenyl ether, and phenol, was covered against all the other classes. Since there is only one set of L-rules (they are all relevant to each other), they were all used (no trimming was done). The parameters used are given below:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
variable name   vtype   vcost      .    trace:
   #pt            lin      1      .    stops:
                                  .    print rules: true
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
parameters: general          vl              aq              l-rule
        regenstar    1    vlmaxstar  8    aqmaxstar  2    maxl     5
        metatrim     0    nconsist   4    aqcutfl   20
        desctype  disc    alter      8    lqst   false
        extmty   false    mincover 100
        equiv    false    maxback    0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
           vlnf=3              aqnf=2              lrnf=4
criterion: vlcrit    vltol    aqcrit    aqtol    lrcrit    lrtol
             3         30%       -1      0.00       -1      0.00
            -1        0.00        2      0.00        2      0.00
             2        0.00                           -3      0.00
                                                     4      0.00
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Meta functions were not used in these examples because it is the structure of the molecules that is the most interesting aspect, and the meta functions would tend to dominate the generalizations and not allow the structure to come out in the rules.

The L-rules described above are applied to the events and the new descriptors may be added. Listed below is the input rule for the first alcohol molecule, after the L-rules have been applied. The L-rules have added the selectors describing a methyl group and an OH group:

```
[t(a1)= c][t(a2)= h][t(a3)= h][t(a4)= h]
   [sb(a1,a2)][sb(a1,a3)][sb(a1,a4)][sb(a1,a5)]
   [t(a5)= o][t(a6)= h][sb(a5,a6)][con(a7,a5)]
   [g(a7)= oh][con(a7,a6)][con(a8,a1)][g(a8)= methyl]
   [con(a8,a2)][con(a8,a3)][con(a8,a4)][attchd(a7,a8)]
   => [d=alcohol].
```

The program found a number of generalizations describing the first class of molecules, alcohols. The one it chose was:

[sb(a1.a2)][sb(a2.a3)][sb(a2.a4)][con(a5,a4)][g(a5)=oh] => [d=alcohol].
(An atom in an OH group is singly bound to an atom which has two other single bonds.)

This rule is shown in graphical form in figure 5.3. The diagrams represent the necessary substructures of the molecules required by the rules. An asterisk in place of an atom means that any atom is allowed at that place. Membership in a group is indicated by placing the group name in parentheses above the atom which is contained in that group. An asterisk in parentheses means that any group is allowed at that place. Finally, a single line between two atoms indicates a single bond between them; a single line between two groups indicates that one group is attached to the other.

The rule found by the program is more general than the definition of "alcohol" given earlier, which was "an alkyl group bound to an OH group". The reason is that the chemist's definition given earlier was a characteristic description (i.e., one that distinguishes alcohols from all other possible classes), whereas the rule found by the program was a discriminant description (i.e., one that distinguishes alcohols from only the other classes provided).

Also note that this rule does not cover any of the molecules in the class "phenol" even though they also have OH groups. The reason is that the rule specifies that the OH group is bound to an atom that has two *single* bonds. The molecules in the phenol class contain double bonds.

The class "ether" was covered by the following product:

[g(a1)=methyl,chh][g(a2)=methyl,chh][g(a3)=ogrp][attchd(a1.a3)]
[attchd(a2.a3)] => [d=ether]
(A methyl or chh (CH_2) group is attached to an ogrp (a single oxygen atom), which is attached to another methyl or chh group.)

This rule is also shown in figure 5.3 (as are the rules for all the classes). It agrees very closely to the established definition of "ether."

The following description was found for the class "alkene":

[t(a2)=h][t(a3)=h][sb(a1.a2)][sb(a1.a3)][db(a1.a4)] => [d=alkene]
(There is an atom which has single bonds to 2 hydrogens, and a double bond to some other atom.)

This rule is more general than, but not as clear as, the chemist's definition. The rule distinguishes alkenes because they are the only class which has a carbon double bond, and where the carbon is singly bonded to two hydrogens.

The rule for carboxylic acid is very simple:

[t(a2)#c][db(a1.a2)] => [d=carbacid]
(There is an atom which is double-bonded to a non-carbon atom.)

Although very simple and general, this rule discriminates the class of carboxylic acids because this class was the only one (of the classes provided to the program) that had any double-bonded oxygens.

The rule for phenyl ether:

[g(a1)=ar,ogrp][g(a2)=ar,ogrp] => [d=phether]
(There are two groups that are aromatic rings or ogrp's.)

This rule captures the structure of the class "phenyl ether" but is more general than the definition of the class.

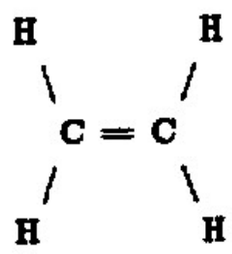
Finally, the rule for phenols:

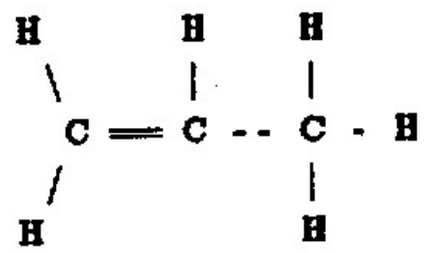
[g(a1)=ar][g(a2)#ogrp][attchd(a1.a2)] => [d=phenol]
(An aromatic ring is attached to a group other than an "ogrp".)

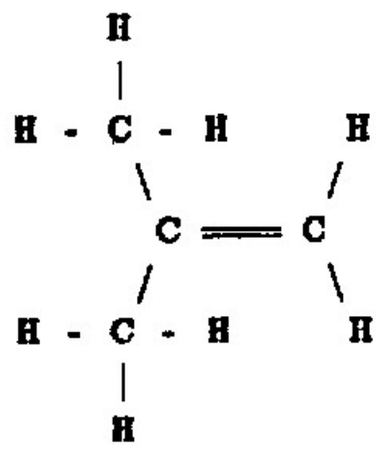
This rule is more general than the definition because the definition requires the aromatic ring to be attached to an OH group, whereas the program's rule requires it to be attached to anything except a single oxygen atom (ogrp).

The last two rules illustrate how the L-rules have shortened the output decision rules. If there were no L-rule defining the aromatic ring, the decision rule would probably include a large number of selectors describing its structure. This would increase the length of the rule considerably and make it much more difficult to understand.

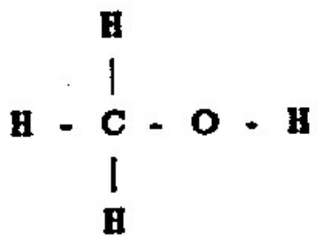
Alkenes

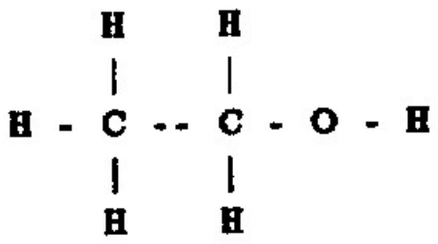


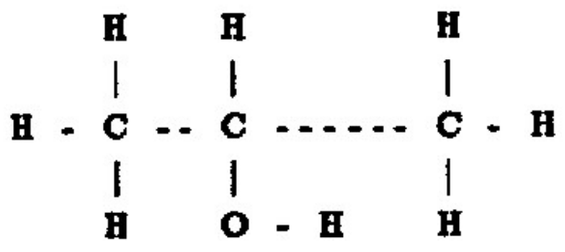




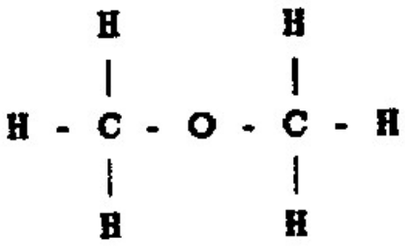
Alcohols

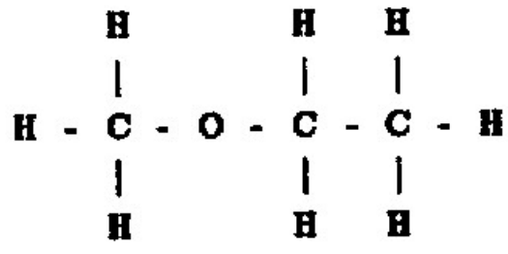


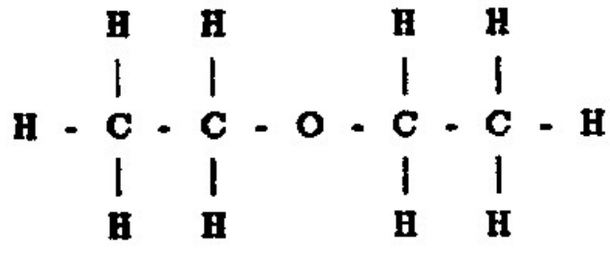




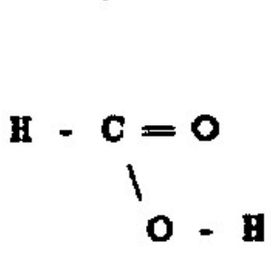
Ethers

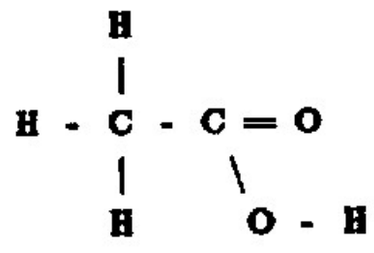


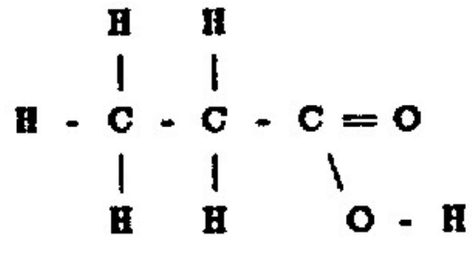




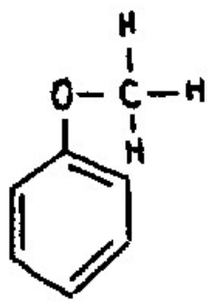
Carboxylic Acids

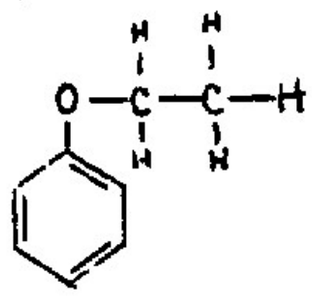


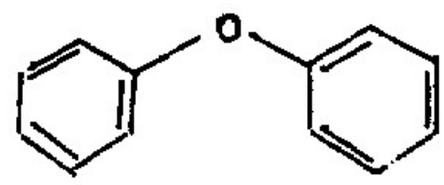




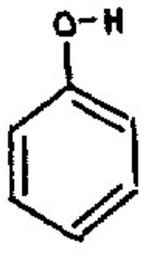
Phenyl Ethers







Phenols



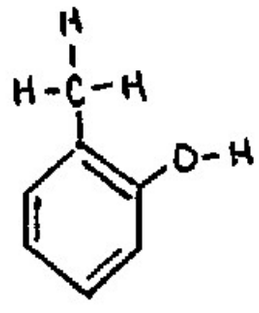
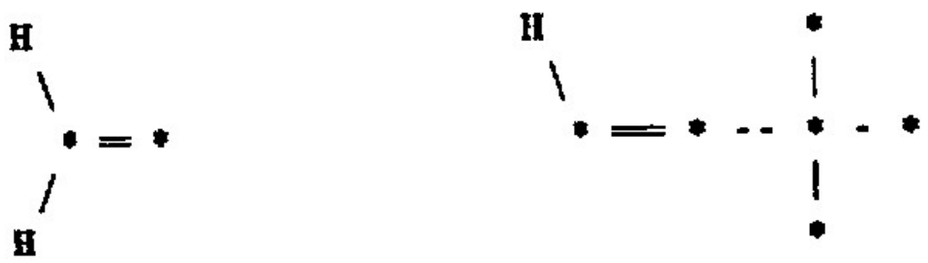
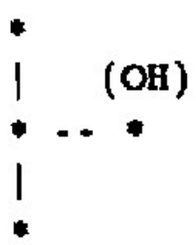


Figure 5.2 - Chemical Example (Input Examples)

Alkenes



Alcohols



Ethers

(methyl) -- (*) -- (methyl)

(CH_3) -- (*) -- (CH_2 or methyl)

Carboxylic Acids

* = O

Phenyl Ethers

(ar) -- (*) -- (*)

Phenols

(ar) -- (methyl or OH)

Figure 5.3 - Chemical Example (Output Rules)

6. Limitations and Disadvantages

The current program has a number of limitations and disadvantages. These are of three types: (1) implementational and technical problems, which could be corrected by suitable reprogramming, (2) conceptual problems which could be solved by extensions to the program, and (3) fundamental problems whose solution would require a different approach. This section discusses the first two types of problems. Problems of the third type are discussed in the next section, "Summary", along with the important features and advantages of the program.

6.1. Implementational and Technical Problems

Implementational and technical problems include:

(1) The program is relatively large (about 6000 lines of Pascal code) and difficult to modify because it is not well organized for a program of this size.

(2) It requires a relatively high user-sophistication and knowledge of the program. Specifically, to take full advantage of the capabilities of the program, the user must know the meaning of a large number of parameters and how changes in those parameters affect the results. It should be mentioned, however, that for problems of small complexity (where the number of events, descriptors, and background knowledge rules is small) the default settings of the parameters are adequate.

(3) The program is slow when there are many events and many selectors in each event. The most time-consuming part of the program is the procedure which determines if one VL_2 expression covers another (equivalent to determining if one graph is a subgraph of another). There is no way to speed up this process (it is NP-complete), but there is a simple way to cut down on the number of times this procedure is performed. This can be done by tracking the events that are covered by each hypothesis. In the "rule-growing" process, rules are specialized by the addition of selectors. The current program tests each new hypothesis against all the positive and all the negative events. However, it is only necessary to check if the new hypothesis covers the events that were covered by the old hypothesis. This is because adding a selector can never cause a hypothesis to cover more events, only fewer.

(4) Currently, the only relations (denoted by #) allowed in a selector [L # R] are equality and inequality. The ability to use other relational operators such as "less than" or "greater than" is desired.

(5) A more general facility for doing constructive induction could be be implemented by an extension of L-rules. One could allow L-rules to introduce new L-rules to the rule base. Also, L-rules could contain free variables which would be bound to some values when the L-rule was matched to an event. This would allow descriptors which describe relationships between events to be introduced. For example, if each event contains two descriptors, one specifying the time and the other specifying height, then L-rules could introduce a descriptor specifying how much the height changed from the previous time. This formalism and examples of its use in constructive induction is given by Vaidya [1983].

(6) A better generalization algorithm could be implemented, possibly using a best-first tree search. The process of adding selectors to build rules can be viewed as a tree search for a goal node. Each node is a rule - a conjunction of selectors. A goal node is a consistent rule. The idea is to find a goal node with the lowest cost. Cost is usually measured by: (i) the number of events covered that shouldn't be, (ii) the number of events covered that should be (the negative of this), and (iii) the length of the rule. Starting with only single selector nodes, always expand the node with the lowest cost, until a goal node is reached. Expand a rule by adding to it a selector which is weakly connected - shares some variable with the rule.

This method would eliminate the need for the parameters ALTER and VLMAXSTAR, and would produce lower cost rules. It would not guarantee the finding of the lowest cost rule, because the estimate of the cost of a rule is not always less than the actual distance to the goal (as required by the A* algorithm). This method may require more rule evaluations, although this was not the case for an example involving identifying organic molecules worked out by hand.

(7) The present algorithm for characteristic generalization requires the user to specify the value of parameter MINCOVER (the percentage of the events that the rule must cover. The program specializes the rule (by adding selectors) until it no longer covers at least MINCOVER percent of the events. This approach often fails to find the best characteristic description, because it is dependent on the order in which selectors are added. Adding a selector which is not shared by all events may cause the rule to fall below the MINCOVER threshold, whereas if a different selector had been added, the rule might still cover all the events and be more specific. A better algorithm for characteristic generalization is proposed by Dietterich and Michalski [1981]. This approach uses a two phase method for computational reasons. The program first searches for descriptors in the space defined by the structure-specifying descriptors (non-unary descriptors). Then, and after a number of plausible candidates have been found, it completes the search in the space defined by the attribute descriptors (unary descriptors). This approach may be better because it generalizes an event in all possible ways, instead of specializing it in only a few ways.

6.2. Conceptual Problems

The problems of the second type, which require research on a more conceptual level, include:

(1) The current program cannot perform incremental learning. Such learning takes the form of (i) modifying old hypotheses in the light of new events and (ii) creating hypotheses for new classes by covering the events in the new class against the updated hypotheses for the old classes. To do this would require a change in the way the program determines if two formulas intersect. At present, two formulas intersect only if they share common descriptors. However, during generalization some descriptors are dropped, so that two c-formulas may intersect even though they do not share common descriptors. For example, the program may have produced the following two decision rules, which do not share common descriptors:

$$[P(x1)] \Rightarrow [d=1] \qquad [Q(x1)] \Rightarrow [d=2]$$

However, an event such as $[P(x1)][Q(x1)] \Rightarrow [d=3]$ which belongs to a new class, satisfies both formulas. As another example, the program may have generalized the event $[P(x1)][Q(x1)] \Rightarrow [d=1]$ to produce the decision rule $[P(x1)] \Rightarrow [d=1]$ for class 1. However, the program may now be given an event from a new class, d=2:

$$[Q(x1)][R(x1)] \Rightarrow [d=2]$$

If this event is generalized against the decision rule for class 1, the resulting decision rule for class 2 may be $[Q(x1)] \Rightarrow [d=2]$. This rule is consistent with the decision rule for class 1, but is inconsistent with the event in class 1.

(2) The program does not generate intermediate decision rules. Such rules would define subconcepts (relationships among subsets of parts of an object) and would be useful in simplifying the output hypotheses. For example, one would like the program to generate the description of an arch in order to describe a sequence of arches.

(3) The program is not able to interactively query the user for advice and guidance. For example, it would be helpful if it could generate its own examples to show the user, as done by Sammut [1981].

7. Summary

This paper has presented a comprehensive description of the inductive learning program INDUCE 3 — its purpose, the theory behind it, details of implementation, and examples of its use. INDUCE 3 differs from previous work in its formal basis, flexible optimality criteria and domain definitions, ability to add new descriptors to existing descriptions, and apparent extensibility. The most important features of the program are:

(1) It is capable of learning structural descriptions of objects.

(2) It can produce either discriminant and characteristic descriptions.

(3) It is general in the sense that it is not restricted to any particular domain.

(4) It uses the variable-valued logic VL_2 calculus (an extension of the predicate logic calculus) as a description language for examples and generated hypotheses. The language uses a few additional syntactic forms and typed descriptors, distinguishing between nominal, linear, and structured descriptor types. Instead of predicates, it uses selectors, which are generally more expressive. The formalism provides a simple linguistic interpretation of descriptions without losing the precision of the conventional predicate calculus.

(5) It allows the user to provide background knowledge for the given problem. The background knowledge includes (i) the preference criterion for output hypotheses, (ii) rules for generating new descriptors as arithmetic expressions of the initially provided descriptors (A-rules), and (iii) rules (called L-rules) for defining any background concepts, constraints and facts relevant to the problem (the rules are expressed as implicative statements linking any two VL_2 conjunctive expressions).

(6) It is written in Pascal, a popular language, and is thus easily transportable.

The current program has many limitations. Problems of a technical nature and problems of a deeper conceptual nature are discussed in the previous section. Discussed below are more fundamental problems, whose solutions are not well understood, and which may require a different approach than the work presented in this paper. These are possible directions for future research:

(1) The program can only accept and produce declarative, rather than procedural, descriptions, due to the nature of the representation language.

(2) The program assumes that the presented descriptions are totally correct. It cannot handle incorrect descriptions, descriptions with some degree of uncertainty, or descriptions of different examples that intersect.

(3) The hypotheses produced by the program are restricted to one form, a conjunctive VL_2 expression with distinct existential quantifiers. The program should be able to produce expressions containing connectives such as exception or implication, as well as the other quantifiers (normal existential and universal).

(4) A major limitation is that the semantics of the predicates and functions are not known to the program. The program only knows the domain and type of each descriptor. An "annotation" is needed which gives background knowledge for each descriptor. Some ideas for the annotation of predicates are given in [Michalski 1983]. The annotation should contain information to assist the program in generalizing and creating intermediate decision rules. For example, the program should be able to locate similar descriptors, more general descriptors, or more specific descriptors within some descriptor hierarchy.

(5) Descriptions and rules for manipulating these descriptions should be represented within the same formalism. This would make it possible for the program to modify its own algorithm according to the problem at hand, and create its own heuristic rules. Also, the internal state of the program should be represented in the same formalism, so that the program could keep track of its own behavior.

LIST OF REFERENCES

Banerji, R. 1975, "Learning with Structural Descriptions," working paper Temple University, Philadelphia, Penn.

Bongard, M. 1970, *Pattern Recognition,* translated by Theodore Chevon, Spartan books, New York, N. Y.

Buchanan, B.G., Sutherland, G.L., and Feigenbaum, E.A. 1969, "Heuristic DENDRAL: a Program for Generating Explanatory Hypotheses in Organic Chemistry", in *Machine Intelligence 4,* (B. Meltzer and D. Michie eds), Edinburgh University Press.

Buchanan, B.G., Sutherland, G.L., and Feigenbaum, E.A. 1972, "Heuristic Theory Formation, Data Interpretation, and Rule Formation," *Machine Intelligence 7,* (B. Meltzer and D. Michie eds), John Wiley & Sons.

Chilausky R., Jacobsen B., Michalski R.S. 1976, "An application of Variable Valued Logic to Inductive Learning of Plant Disease Diagnostic Rules", *Proceedings of the Sixth Annual Symposium on Multiple Valued Logic,* Logan, Utah.

Croft, J.A. 1971, "A comparative Study of Mathematical Methods for Diagnosing Disease," Ph.D dissertation, Northwestern University Evanston, Illinois.

Dietterich, T.G. 1978, "Description of Inductive Program INDUCE 1.1", Internal Report, Department of Computer Science, University of Illinois, Urbana, Illinois.

Dietterich, T.G. 1980, "The Methodology of Knowledge Layers for Inducing Descriptions of Sequentially Ordered Events," M.S. Thesis and Report No. 1024, Department of Computer Science, University of Illinois, Urbana, Illinois.

Dietterich, T.G. and Michalski, R.S. 1981, "Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods," *Artificial Intelligence Journal* Vol. 16, No. 3, July 1981, pp. 257-294.

Hayes-Roth, F. and McDermott, J. 1976, "Knowledge Acquisition from Structural Descriptions," Department of Computer Science, Carnegie Mellon University.

Hedrick, C.L. 1974, "A Computer Program to Learn Production Systems Using a Semantic Net," Department of Computer Science, Carnegie Mellon University, Pittsburgh, Penn.

Hunt, E.B. 1966, *Experiments in Induction,* Academic Press.

Kochen, M. 1974, "Representation and Algorithms for Cognitive Learning," *Artificial Intelligence 5.*

Larson, J., Michalski, R.S., "AQVAL/1-AQ7 User's Guide and Program Description," Department of Computer Science Technical Report No. 731, University of Illinois.

Larson, J. 1976, "A Multi-Step Formation of Variable Valued Logic Hypotheses," *Proceedings of the Sixth International Symposium on Multiple Valued Logic,* Logan, Utah.

Larson, J. 1977, "Inductive Inference in the Variable-Valued Predicate Logic Systems VL_{21}: Methodology and Computer Implementation," Ph.D. Thesis, Report No. 869, Department of Computer Science, University of Illinois, Urbana, Illinois.

Larson, J., Michalski, R.S. 1977, "Inductive Inference of VL Decision Rules," Workshop on Pattern Directed Inference Systems, Hawaii.

Lenat, D.B. 1976, "AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search," Ph.D. Thesis, Report No. STAN-CS-76-570, Computer Science Department, Stanford University.

Michalski, R.S. 1972, "A Variable-Valued Logic System as Applied to Picture Description and Recognition," *Graphic Languages, proceedings of the IFIIP Working Conference on Graphic Languages,* Vancouver Canada.

Michalski, R.S. 1973, "AQVAL/1 AQ7 Computer Implementation of a Variable Valued Logic System and the Application to Pattern Recognition," *Proceedings of the First International Joint Conference on Pattern Recognition,* Washington, D.C.

Michalski, R.S. 1974a, "Variable-Valued Logic: System VL_1," *Proceedings of Fourth International Symposium on Mutiple-Valued Logic,* West Virginia University, Morgantown, West Virginia.

Michalski, R.S. 1974b, "Learning by Inductive Inference," NATO Advanced Study Institute on Computer Oriented Learning Process," France.

Michalski, R.S. 1974c, "Problems of Designing an Inferential Medical Consulting System," First Illinois Conference on Medical Information Systems University of Illinois Urbana, Illinois.

Michalski, R.S. 1977, "Toward Computer-aided Induction: a brief review of currently implemented AQVAL programs", 5th International Conference on Artificial Intelligence, Cambridge, Mass.

Michalski, R.S. and Chilausky, R.L. 1980, "Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *International Journal of Policy Analysis and Information Systems,* Vol. 4, No. 2.

Michalski, R.S. 1980, "Pattern Recognition as Rule-Guided Inductive Inference," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. PAMI-2, No.4, July 1980, pp. 349-361.

Michalski, R.S. 1983, "A Theory and Methodology of Inductive Learning," chapter in *Machine Learning,* R.S. Michalski, J. Carbonell and T. Mitchell (Editors), Tioga Publishing Company.

Morgan, C.G. 1972, "Inductive Resolution." Master's thesis, Department of Computer Science, Edmonton, Alberta.

Pople, H., Werner G. 1972, "An Information Processing Approach to Theory Formation in Biomedical Research," *AIFIPS Conference Proceedings,* Vol 40.

Quinlan, J.R. 1979, "Discovering Rules by Induction from Large Collections of Examples," in *Expert Systems in the Micro Electronic Age,* D. Michie (Editor), Edinburgh University Press.

Rychener, M.D. 1976, "Production Systems as a Programming Language for Artificial Intelligence Applications," Ph.d Thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Penn.

Sammut, C. 1981, "Learning Concepts by Performing Experiments," Ph.D. Thesis, Department of Computer Science, University of New South Wales.

Vaidya, P. 1983, "Representaion of Constructive Induction," Report No. x, Department of Computer Science, University of Illinois, Urbana Illinois.

Vere, S.A. 1975, "Induction of Concepts in the Predicate Calculus," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence.*

Waterman, D.H. 1970, "Generalization of Learning Techniques for Automating the Learning of Heurisics," *Artificial Intelligence 1.*

Waterman, D.H. 1974, "Adaptive Production Systems." working paper #385, Department of Psychology, Carnegie Mellon University, Pittsburgh, Penn.

Waterman, D.A. 1975, "Serial Pattern Acquisition: A Production System Approach," working paper #286, Department of Psychology, Carnegie Mellon University, Pittsburgh, Penn.

Winston, D.H. 1970, "Learning Structural Descriptions from Examples," AI-TR-76, MIT Artificial Intelligence Laboratory, Cambridge Mass.

Appendix 1 - Detailed description and user's guide to INDUCE 3

Table of Contents

Appendix 1 - Users Guide

1. Introduction

This appendix provides

further details of the implementation of the program INDUCE 3. This program accepts an environment description, a set of VL decision rules, and a set of parameters. The program produces a set of generalizations of the input decision rules. The basic algorithms and input syntax are given in section 4 of this paper so will not be repeated in full here. In the following pages, the actual commands necessary to use the program are given. Section 2 of this appendix contains a description of the data structures used in the program. The reader is referred to the program listing for more detailed structure. In section 3 of this appendix, the various I/O files are described. Section 4 of this appendix gives a brief outline of the purpose of each procedure and its relation to other procedures in the program.

1.1. High level commands

The following single letter comands can be entered into the program to perform various functions:

H (get help) - Enter this command to obtain a brief explanation of the high level commands and a detailed explanation of one such command by entering 'H X' where x is one of the letters corresponding to a high level command.

M (modify rule base) - This command is used to enter rules into the program or delete rules from memory. Following the M command, the user may enter (A) to add a new rule, (D) to delete an existing rule, or anything else to return to the main level without doing anything. After an A is entered, the system expects a VL_2 rule in correct syntax terminated with a period (.). Since there is no online error correction, this is usually done by placing all rules in a local file (CFILE) with the commands (M and A) interspersed. After the rule has been entered, the program returns to the high level command mode. If a (D) is entered, the program proceeds through the list of all rules asking at each stage whether to delete the rule. The user may enter Y, N, or Q to delete the rule and move to the next rule, to keep the rule and move to the next, or return to the command level. Example:

```
M
A
[SHAPE(X1)=1][P(X1,X2)=2] => [D=2].
```

L (enter logically derived descriptor) - Enter L (carriage return) followed by a logically derived descriptor (L-rule). This rule is put into a set of other L-rules which are related to it (i.e., the applicability of this rule may affect the applicability of the other rules). When the covering procedure is peformed, some of these sets of L-rules are applied to the data rules entered by the M command above. A L-rule is applied by adding the consequence of the L-rule to the data rule, if it is not already there. The left hand side of the L-rule must form a connected graph structure. Each connected subgraph of the right hand side must be connected to the left hand side. As with all rules, the L-rule must end with a period. Example

```
L
[ONTOP(P1,P2)][ONTOP(P2,P3)] => [ONTOP(P1,P3)].
```

E (enter domain generalization structures) - Enter tree structure for such domains. These must be entered in order from lowest level generalization to highest level generalization. For

VL_1 applications, this should be done after a V command has been entered since the V command initializes the symbol table for the special VL_1 mode. Domain generalization structures must be entered before any other rules are added to the rule base via L or M commands. Example:

```
[SHAPE=2 v 4] => [SHAPE=10].
[SHAPE=0 v 1 v 3 v 5] => [SHAPE=11].
[SHAPE=6 v 7 v 8 v 9] => [SHAPE=12].
[SHAPE=10 v 11] => [SHAPE=13].
```

A (Enter an arithmetic derived descriptor) - Enter the derivation rule for an arithmetic derived descriptor. Example:

```
A
GIRTH(X1)=LENGTH(X1)+ WIDTH(X1).
```

Restrictions: the dummy variables of the function on the left hand side must appear on the right hand side of the equation. The arithmetic expression is written in standard algebraic form. The operators which may be used are: + (addition), - (subtraction), - (unary minus), * (multiplication), / (integer division--remainder discarded), and % (integer modulus). *, /, and % are evaluated before - and +. Integer constants may also appear in the expression. The right hand side must contain at least one function or predicate. All functions and predicates are assumed to have interval domains. If more than one value appears in the reference of a function when the expression is to be evaluated, the smallest value is used. The right hand side must form a connected graph structure as well. A connecting predicate can be multiplied to the original expression to accomplish this since predicates have values of 1 when true. Bug: This command will not work correctly if there are two or more occurrences of exactly the same function (with the same dummy variables) on the right hand side.

N (Add arithmetic derived descriptors to the rule base) - The separation of the A and N commands is included to permit users to enter the rules and the arithmetic descriptors in any order and then to apply the arithmetic descriptors when they are desired (after all of the rules have been read in). The N command causes all previously entered arithmetic derived descriptors (since the most recent N command) to be processed and added to all rules in the rule base where they are appropriate.

X (Enter a logical derived descriptor and substitute it into the rule base) - Logical derived descriptors are handled by two separate commands. The L command permits the user to enter a logical derived descriptor which is to be added to each rule for which the premise is true. The X command permits the user to enter a logical derived descriptor which is to be substituted (exchanged) for its premise in each rule in which the premise is true. (The premise is the left hand side of the rule.) Example:

```
X
[BIG(PART1)][BOX(PART1)] => [BIGBOX(PART1)].
```

This example command will substitute [BIGBOX(PARTN)] for every conjunction of BIG(PARTN) and BOX(PARTN) where PARTN is any given PART dummy variable. Each dummy on the right hand side must appear on the left hand side. The right hand side must be a single selector. The left hand side must form a connected graph structure.

C (Cover a set of formulas) - Enter the number of the associated decision after the C command. Be sure to set any trace information using the appropriate parameters before entering the C command.

V (VL_1 mode) - This mode bypasses the VL_2 type structure creation and accepts VL_1 events from the file VL1EVE. After entering V, the program asks for the number of variables which are to be used. Enter this number (it should be 1 less than the number of entries in each line of the VL1EVE file because of the class number in the file). Then, the user is asked to enter another command (E, C, Q, or P). Enter E and then a domain generalization structure for that type of domain, P to change parameters (AQMAXSTAR, LQST, AQCRIT, AQTOLERANCE, or enter VCOST or VTYPE, the latter may be necessary for interval type variables), C to cover a set of events, or Q to return to the high level commands. All of the E and P parameters may be included in CFILE. When C is entered, the program requests the number of the class of events to be covered and then the number(s) of the class(es) against which the cover should be made. To cover against all other classes, enter -1 instead of a list of all other classes. (This is useful for intersecting type covers.) All specifications may be placed in CFILE.

P (Parameters) - This places the user in a parameter examination and modification mode. To get an explanation of each parameter on-line, enter

HELP <parameter name> or HELP

HELP entered alone produces the list of parameters. See the EXPLAIN file for a list of all the parameters and explanations. Specifying a parameter value outside its permitted range causes an error message to be printed and the value is ignored. A missing value is interpreted as the value 0. Most parameters require the parameter name followed by the value. Parameters which may be true or false are set to true by entering the parameter name (e.g., LQST) and are set to false by entering the parameter followed by F (e.g., LQST F). Trace and stop parameters are turned on one at a time by entering TRACE or STP and then the associated number. They are turned off by entering the negative of the number (e.g., TRACE 3 turns on trace 3, STP -6 turns off the program stop at trace level 6).

Functions such as VCOST and VTYPE must have the associated descriptor name in parentheses following the parameter name (e.g., VTYPE(SHAPE)=2 sets the domain of SHAPE to type interval.) All VL_1 type variables have descriptor names X1, X2, ... Xn (so VCOST(X1)=-2 sets the cost of the variable X1 to -2). After all parameters have been set, entering QUIT returns to the previous command. In order to examine the parameters, enter PARA. Besides displaying all control parameter settings, PARA will give the type and cost of all functions for which the two characteristics VTYPE and VCOST are not the default values (type nominal and cost of 1). Enter PRINT D to examine the domains of all functions in the symbol table.

Q (Quit) - This terminates program execution.

D (Dump) - This command, used during debugging, dumps a single rule or the entire the rule base graph structure and the symbol table.

Z (Debug Arith Rules) - This command dumps internal descriptions of arithmetic rules.

In addition, the program ignores all input following a percent character ("%") until another percent character is read. This is useful for including comments in the rule and command input.

1.2. Parameters

This section describes the parameters which can be modified after entering the command P above and the commands required to inspect the parameters in the running version of the program. In general, to set or change a parameter, enter the name of the parameter (only the enlarged left-most characters are necessary) followed by a space or equal sign and the new value of the parameter. The parameters and their meaning are as follows, default values are in parentheses:

TRACE - This parameter may have a set of values in the interval [1..10]. Each value relates to a trace feature of the program. The values currently meaningful are:

1 - Print all of the c-formulas in each untrimmed and each trimmed partial star to examine the process of consistent formula generation and trimming.

2 - Print all the consistent formulas both before the AQ generalization and after this generalization.

3 - Print the best MQ formula; i.e., select the best formula from the output of trace 2.

4 - Print the input events to the AQ procedure and the variable association between the VL_2 c-structure and the VL_1 variables.

5 - Print the output from the VL_1 AQ procedure.

6 - Print the selected meta functions in a table.

7 - Print the LQST2 process during characteristic generalization.

8 - Not used.

9 - Print all generalizations of an event (i.e., the complete set of alternative generalizations which the program has calculated for one event from trace 10). This is the same as the list which comes from trace 2 without the input formulas to AQ.

10- Print the event (c-formula) which is to be covered from F1.

To turn on (off) any trace feature, enter

TRACE i (or TRACE -i)

where i is the number of the trace feature to be turned on (off).

STP - This parameter may also have a set of values in the range [1..10]. Each value corresponds to one trace feature defined above. If STP contains a value of a trace feature and the particular trace feature is set, then the program pauses at the point where the trace information is printed and will provide an explanation of the situation or allow the user to modify parameters. STP may be turned on and off in the same way as TRACE, i.e.,

STP i (or STP -i)

AQCUTF1 (20) - This is a limit on the number of c-formulas examined using the AQ cost function 3.

AQMAXSTAR (2) - This is the AQ maxstar parameter (the number of complexes retained in a partial star in the AQ procedure).

AQCRIT (-1,2) - The criteria list of cost functions to be applied in the AQ procedure. There are

six cost functions available:

1 - Measure the number of events covered by a complex which are not covered by any previously generated L_q complex.

2 - Measure the number of selectors whose reference is not equal to *.

3 - Measure the number of c-formulas which are actually covered by a complex. This is more time consuming than 1 but may give better results.

4 - Sum the costs of all variables in a complex in selectors whose reference is not equal to *.

5 - Measure the number of events in the set F1 which are covered by the complex.

6 - Find the number of events in the set 2 (F0).

To specify a cost criterion, enter AQCRIT(I)=J where j is the number of the criterion (if negative, then the cost is computed as the negative of the value determined by the criterion), and i is the order of application of the criterion.

AQTOLERANCE (0) - This is the tolerance associated with each criterion specified in AQCRIT above. AQTOLERANCE(I) is the tolerance associated with criterion AQCRIT(I). The tolerance can be an absolute tolerance (if it is greater than 1) or a relative tolerance (if it is less than 1). The tolerance is always specified in hundreths, e.g.,

AQTOLERANCE(2)=200

results in a an absolute tolerance of 2 for the criterion applied second.

AQNF (2) - The number of criteria which are to be applied to the complexes.

LQST (false) - If LQST is set, then the resulting complexes from the AQ procedure are stripped to only the necessary values in the reference.

VLMAXSTAR (2) - The maximum number of formulas retained in a partial star.

VLCRIT (3,-1,2) - The criteria list which is to be used for trimming VL_2 formulas. There are five criteria available:

1 - Count the number of c-formulas of the set F1 which are covered by this formula.

2 - Count the number of selectors in the formula.

3 - Count the number formulas of the set F0 which intersect with this formula.

4 - Sum the total cost of all references in all selectors of the formula with reference not equal to *.

5 - Sum the cost of all dummy variables used in the function and predicate selectors of the formulas. This uses the cost of a specific dummy variable (e.g., X1) as originally entered (not as dynamically reassigned by the program). It uses the DPNO field.

This parameter is specified in the same way as AQCRIT above.

VLTOLERANCE (.3,0,0) - The tolerance associated with each VLCRIT specified above. See

AQTOLERANCE above for details about how to enter values for this parameter.

VLNF (3) - The number of VL_2 criteria to apply when trimming a list of formulas.

MAXL (5) - The maximum number of sets of L-rules that will be applied to the input events.

LRCRIT (-1,2,-3,4) - The criteria list which is to be used for trimming the number of L-rule sets down to MAXL. There are 4 criteria available:

1 - The number of c-formulas of F1 which are covered by some L-rule in this set.

2 - The number of c-formulas of F0 which are covered by some L-rule in this set.

3 - The number of selectors in the condition parts of all the L-rules in this set.

4 - The number of selectors in the consequence parts of all the L-rules in this set.

This parameter is specified in the same way as AQCRIT above.

LRTOLERANCE (0,0,0,0) - The tolerance associated with each LCRIT specified above. See AQTOLERANCE above for details about how to enter values for this parameter.

LRNF (4) - The number of L-rule criteria to apply when trimming the number of sets of L-rules.

NCONSIST (4) - The number of consistent alternative generalizations which the program is to produce.

ALTER (2) - The number of alternative new formulas which are produced from one formula when creating a new partial star from an old one.

REGENSTAR (1) - This parameter controls the star regeneration process. If REGENSTAR is less than 2, no regeneration takes place. With values of 2 or more, up to REGENSTAR-1 regenerations may be performed. A regeneration is done if a consistent rule passing through the AQ procedure becomes inconsistent if selectors with a reference of * are removed. To regenerate the rule, the *-reference selectors are removed and the rule (which is not consistent) is returned to the list of rules forming the partial star. Procedure NEWGP is then reapplied to add selectors to the rules in the partial star until they become consistent. If REGENSTAR is set to 1 (or all REGENSTAR-1 regenerations have been consumed) any *-referenced selector that cannot be removed without leaving an inconsistent rule has its reference replaced by the reference value prior to the AQ procedure (i.e., a non-* value). If REGENSTAR is set to 0 then no special processing is performed so the * value remains in the selector reference.

VCOST (1) - The cost of each function in the system. All VL_1 variables when running in VL_1 mode are labelled X1,X2,....,XN. To enter a cost, type VCOST(<fn-name>)=i where <fn-name> is the name of a function which has been in a decision rule which is currently in the program, and i is the cost of the function. Some examples:

VCOST(SHAPE) = 2 or VCOST(X4)= 1

VTYPE (NOM) - This is the structure of each domain:

NOM - nominal
LIN - linear (interval)
STR - tree structured.

The type STR is set automatically when the E command is entered. To make a function domain into an linera type, enter e.g., VTYPE(SHAPE) = LIN.

METATRIM (4) - This specifies the number of different meta functions which are to be selected by the program to be used in descriptions. This value should be less than GSIZE. If it is 0, then no meta-functions are generated.

EXTMTY (false) - This parameter controls the generation of EXTMTY type descriptors (i.e., mst- and lst- types). When first enabled, such descriptors are added to all rules in the rule base. As additional rules are added (via the M command), EXTMTY type descriptors are automatically added to them. When disabled (via EXTMTY=F) all rules in the rule base are edited to remove the EXTMTY type descriptors, and none are added to subsequently added new rules.

EQUIV (false) - This parameter controls the generation of EQUIV type descriptors (i.e., same<attribute> predicates). When first enabled, EQUIV descriptors are added to all rules in the rule base. As additional rules are added, EQUIV descriptors are automatically generated for them. When disabled (via EQUIV=F) all rules in the rule base are edited to remove the EQUIV type descriptors, and none are added to subsequently added new rules.

DESCTYPE (DISCRIMINANT) - This specifies the type of description which the program is to generate. DESCTYPE DISCRIMINANT causes the program to generate the most general description which discriminates events of set F1 from events of set F0. DESCTYPE CHARACTERISTIC causes the program to generate the most specific description which is shared by all events in set F1. F0 must be empty for this to work properly. Thus, only one set of events should be supplied to the program for a characteristic description. For characteristic descriptions, the parameter MINCOVER must be set.

MINCOVER (100) - This specifies the minimum percentage of rules in F1 that a description must cover in order to be considered as a characteristic description. During the rule growing process, each rule is grown (by adding additional selectors) until it fails to cover MINCOVER% of the rules in F1. At that time, it is placed on the MQ star. NCONSIST such MQ rules must be found before the growing algorithm terminates. Thus, if MINCOVER=100, several, fairly trivial rules will be found. If MINCOVER=50, some interesting rules will be found (but this will use more cpu time) but these rules may not cover all of F1.

MAXBACK (0) - This parameter may be used to speed up the generation of characteristic descriptions. During the generation of such descriptions, MAXBACK is used to limit the backtracking in the graph matching algorithm. After backtracking MAXBACK number of times, the match is declared false and the backtracking ends. When MAXBACK is set to 0, there is no backtrack limit. There is never a backtracking limit during the production of discriminant descriptions.

PRULE (true) - This parameters determines whether the program prints the full text of the rule or just rule statistics.

PRINT X - This allows the user to examine certain tables in the program. X may be one of F, R, D, M, L and the system will respond by listing:

F - The set of input decision rules
R - The set of input restrictions
D - The domain table
M - The currently selected meta-functions.

L - All L-rules currently known by the program.

PARAMETERS - This lists the current parameter values.

NOTRACE - This turns off all trace values.

BRIEF - This sets the trace options 3,9,10 and stop option 10.

DETAIL - This sets all traces.

EXPLAIN - This sets all traces and all stop options.

HELP - This allows the user to obtain an explanation on-line of the function of any of the parameters and a list of all parameters accepted under the P high level command.

DEBUG X - This command is used to request debug tracing. X is the name of a procedure or function in the INDUCE 3 program. The form DEBUG -X is used to turn off debug output for procedure/function X. The output produced is intended only for a programmer (i.e., not easily interpreted by the ordinary user).

QUIT - This returns the user to the main command level.

Appendix 1 - Users Guide

2. Data Structures

2.1. Constants

Some constants in the program control the sizes of many structures which may be sensitive to the current problem characteristics. These constants may be increased (to allow larger data structures) or decreased (to permit more copies of a data structure in memory at one time). The constants and their use appear below (suggested values are in parentheses).

SYMSZE(100) - is the size of the symbol table. It can be estimated by finding the sum of the number of functions, predicates, and distinct variables plus the number of groups of variables plus 2 (for meta functions #PT and FORALL) plus 2 times the number of binary predicates (for MST-, LST- type predicates). In VL_1 mode, SYMSZE is the number of VL_1 variables plus 1.

NDES(15) - is the size of the DSTRUC table. One row is required in this table for each internal node in each generalization structure (i.e. one row for each rule which is input with the E command.)

GSIZE(99) - specifies the size of all graph structures in the program and the number of VL_1 type variables which are allowed in the program. This number being too small is probably the cause of an 'array index out of bounds' message and may be remedied by increasing the parameter. Its value can be estimated by finding the sum of the number of selectors in the longest rule which must be stored plus the number of variables in the rule plus 1 (not including meta selectors). An estimate which is too large will use up memory very quickly and cause a message 'runtime stack overflow' therefore, the parameter should be approximated rather closely.

MNVAL(30) - is the maximum value in a set of values. A set of values (VALTP) is used in several places (GRAPH, CPX, DSTRUC) in the program. Each set is allowed to contain values from 0 to MNVAL. The maximum value of this parameter may be determined by the architecture of the machine (CDC is 58, DEC is about 30, but the VAX 11/780 is unlimited).

MLNK(20) - is the number of links to any node of a graph structure. This may be estimated by finding the maximum number of times that a particular variable occurs in a rule and using either this figure or the larger number of arguments of any one function, which ever is largest. MLNK must be one larger than either of these numbers since links are stored as an array of numbers which terminates with a 0 value.

MRULE(25) - is the maximum number of rules in either F1 or F0.

ASSZE(20) - is the maximum number of entries in an arithmetic expression stack. There is one entry on the stack for each function and value in the expression and one entry on the stack for each operator. There is no compiler or system limit to this parameter.

BSIZE(110) - is the maximum length of any input line (from the CFILE or the user at a terminal).

2.2. Parse table (PT)

The parse table consists of a data structure which represents the productions in the VL_2 grammar (RHS and CONT) along with information about which semantic routines are invoked with the recognition

of one non-terminal in the grammar (SRULE). The array RHS contains a row for each alternative in each production where each element in a row is a positive or negative integer or zero. If the number is positive, it represents a token in the input (it is either the machine representation of a character or 1 - a function symbol, 2 - a variable, or 3 - a number). If the entry of RHS is negative, it represents a non-terminal whose definition is found beginning in the row corresponding to the absolute value of the entry (e.g., -3 represents the non-terminal beginning in row 3 of the table). A zero value signifies the end of the alternative. The boolean array CONT indicates whether a row of RHS is a continuation of a previous row in a production (value true) or the first alternative of a production (value false). Finally, the array SRULE contains a number indicating the semantic rule (element in a case statement in the procedure PROCESS) which is to be applied if the production in the corresponding row of the table is matched. Example: (see file TABLES for the complete input grammar)

```
<VLRULE>      ::=  <NUMBER> <RULE>
                |  <RULE>
<RULE>        ::=  <CONDITION> ⇒ <SELECTOR>
<CONDITION>   ::=  <SELECTOR> <CONDITION>
                |  <SELECTOR>
<SELECTOR>    ::=  [ <VARIABLE> = <REF> ]
                |  [ <FN-SYM> ( <ALIST> )= <REF> ]
```

Parse Table in the program: (The actual table in the program contains numbers instead of characters)

ROW	SRULE	CONT	RHS	NAME
1	1	F	3 -3	A VL2 DECISION RULE
2	2	T	-3 0	A VL2 DECISION RULE
3	3	F	-4 ⇒ -6 0	A VL2 DECISION RULE
4	4	F	-6 -4 0	A CONJUNCTION OF SELECTORS
5	4	T	-6 0	A CONJUNCTION OF SELECTORS
6	14	F	[-19 = -10] 0	A VARIABLE SELECTOR
7	7	T	[-21 (-14) = -10] 0	A FUNCTION SELECTOR

2.3. Symbol Table (SYMTAB)

The symbol table is a table with an entry for each function, variable, and symbolic value in the VL_2 decision rules. One entry (NELT) specifies the number of rows which are actually used. The first two rows always contain the information for the meta functions #PT and FORALL. The columns contain:

NAME - the character string representing the name of the entry

PNO - the function number associated with the entry (normally this just points to the row which contains the entry).

NARG - the number of arguments of a function.

VTYPE - type of symbol (NOM-nominal domain variable, LIN-linear domain variable, STR-tree structured domain variable, VAL-value of a function, VBL-existentially quantified variable).

VCOST - variable cost used in cost functions 4 and 5 and selection of alternative selectors (ALTER parameter) in the procedure NEWGP.

NVAL - maximum value in the domain (not including generalized values of tree structured domains).

Appendix 1 - Users Guide

MVAL - minimum value in the domain.

Example: NELT=7

NAME	PNO	NARG	VTYPE	VCOST	NVAL	MVAL
FORALL	1	0	NOM	1	1	1
#PT	2	0	LIN	1	6	0
SHAPE	3	1	STR	1	6	1
X	4	0	VBL	1	-	-
X1	5	0	VBL	1	-	-
X2	6	0	VBL	1	-	-
P	7	2	NOM	1	1	1

2.4. Domain Structures (DSTRUC)

The generalization structures of each tree structured domain are stored in this record. NELE specifies the number of rows in the table which are used. PREM is a set of all descendents of the node in CONS for the domain of the function which is defined in the row PNO of the symbol table. Example:

[SHAPE=1 v 2 v 3] => [SHAPE=7].
[SHAPE=0 v 5 v 6] => [SHAPE=8].

PREM	CONS	PNO
1,2,3	7	3
0,5,6	8	3

2.5. Meta selector Table (MSTR)

This table records the meaning of meta selectors which are used in the formulas. The values of the selector themselves are stored in a structure referenced by MSEL in the GRAPH record. The table contains two integers (METATRIM and NMST) the latter indicates the number of current entries in the table. Elements of the table are accessed indirectly through the array PTR to facilitate sorting of the array with a minimum amount of effort. (e.g., the third element logically in the array PNO is the element PNO[PTR[3]]). Elements are sorted in descending order using PTR as an index according to the values of F1COV (primary field) and -F0COV (the secondary field). The columns are interpreted:

SYMPTR - is the index in the symbol table of the name of the meta function (e.g., a pointer to either FORALL or #PT).

VARPTR - is the index into the symbol table of the dummy variable associated with the unary function from which the metaselector is derived (e.g. for [shape(X1)=...] VARPTR points to X).

PNO - is the index in the symbol table of the referee associated with the particular meta function (e.g., a pointer to SHAPE in the symbol table for a function which counts the number of occurrences of a selector of the form [shape(x1)= ...]).

VAL - is the set containing the reference of the function associated with PNO (e.g., the reference in a selector [SHAPE(X1)=2,3]).

PTR - is the location in PNO, SYMPTR etc. of the information for each selected meta selector in

the order of preference (e.g., information for MS2 would be found in PNO[PTR[2]], SYMPTR[PTR[2]] etc).

F1COV - the maximum number of formulas in F1 covered by one value of this meta function.

F0COV - is the number of formulas of F0 covered by the meta function with the value found in F1COV.

Example: (NMST=3)

PNO	VAL	SYMPTR	VARPTR	PTR	F1COV	F0COV
3	1	1	4	2	3	0
3	0	2	4	1	4	0
3	1	2	4	3	3	2

with the three meta functions:
MS1 = [#XS(SHAPE=0)=?]
MS2 = [FORALL-X(SHAPE=1)]
MS3 = [#XS(SHAPE=1)=?]

2.6. Formula for Graph Structure (GRAPH)

This is the structure used to store each formula. It is composed of 4 parts, the single parameters (COEF, RIGHTS, LEFTS, RNO, COST, ESET, NXTN), a pointer to a set of meta selectors (MSEL), and information about each node and the links between nodes. Each node has a number (the subscript value of each array below) which is used in the LNK array to refer to any node in the graph so that for example, VAL[3] is the value set associated with the node number 3.

COEF - not used

LEFTS - the size of the left-hand side of the graph in nodes. Left-hand nodes are given the lowest node indices from 1 up to LEFTS.

RIGHTS - the node number of the first right-hand side node. Right-hand nodes are given the highest possible node indices from RIGHTS up to GSIZE.

RNO - the unique rule number associated with the graph.

FP - a flag which is used in absorption and the COVER routine.

COST - the cost of the formula (COST[I] is the value associated with cost criterion number I).

ESET - the decision value associated with this rule

NXTN - the pointer to the next graph structure in a list or set of such structures.

MSEL - a pointer to the meta selectors associated with the graph. The metaselectors are stored in an AQ complex corresponding to the MST.

FLAGS (array) - various flags to mark the node. The flags include VBL-node represents an existentially quantified variable; ORDIRR-edge order is irrelevant (i.e., unlabelled edges from this node); P1,P2,P3-three different "node has been processed" indicators used by various routines as temporary node markers; LRL-this node was added by application of l-rules; EQIV-this node contains an EQUIV type descriptor; MSTLST-this node contains an EXTMTY type descriptor.

Appendix 1 - Users Guide

VAL (array) - the set of values associated with the node (this may be a subrange corresponding to [X1=3..6] for example).

PNO (array) - a pointer to the domain definition for the function in the symbol table. Points to the dummy variable family name (e.g., PART instead of PART1).

DPNO (array) - A pointer to the domain definition of the dummy variable itself. It points to, e.g., PART1 rather than PART (unlike PNO). It is used by VCOST function 5 to derive the correct cost.

LNK (array) - links between nodes. Edges are not given an explicit direction, instead, certain routines infer the direction of an edge by the types of node at each end of the edge. All nodes which are connected are doubly linked; if incomming edges are labeled, these labels are indicated by the location in the link array (LNK) for the node.

Example

For the expression: [P(X1,X2)][SHAPE(X1)=2], the link structure is

ROW	FUNCTION	LINKS
1	X2	3
2	X1	3 4
3	P	2 1
4	SHAPE	2

A partial example using the symbol table above is:

[SHAPE(X1)=1][P(X1,X2)][MS2=2]

NODE	PNO	VAL	FLAG	LNK
1	4	*	VBL,ORDIRR	2 3
2	3	1		1
3	7	1		1 4
4	4	*	VBL,ORDIRR	3

2.7. VL_1 Complex Storage (CPX)

This structure is a simple list of references (CVAL) in bit positional notation along with certain flags (FP and FQ), a link to the next such structure in a set (NXTC) and the cost of the complex (COST). The interpretation of each variable is found in the symbol table through the index SLOC in AQPARM (e.g., the set contained in CVAL[3] is the reference of the variable in row SLOC[3] of the symbol table).

2.8. AQ Parameters (AQPARM)

The structure contains several parameters relevant to the AQ procedure.

NVAR - the number of variables for the run.

CSTF - the list of cost functions in the order of application.

TOLER - the tolerance asociated with each cost function (TOLER[3] is the tolerance of the cost

Appendix 1 - Users Guide

function which is applied third — i.e., CSTF[3]).

NF - the number of cost functions to apply

FREEC - a pointer to a list of free complex storage structures (CPX's)

SLOC - the location in the symbol table of the domain definition for each VL_1 type selector in CVAL.

CUTF1 - a parameter which limits the number of formulas examined with AQCRIT of 3.

LQST - if true, then VL_1 complexes are stripped.

MAXSTARAQ - the maximum size of a partial star in AQ

2.9. VL Parameters (PARM)

This structure contains parameters relevant to the VL_2 portions of the program.

CSTF - the cost function indices in order of application

TOLER - the tolerance associated with each cost function

NF - the number of cost functions used

MAXSTAR - the maximum number of elements in a partial star.

ALTER - the number of new elements which are generated from one formula in a partial star P_i when forming a new partial star P_{i+1}.

EXTMTY - a flag indicating whether EXTMTY type predicates have been added.

EQUIV - a flag indicating whether EQUIV type predicates have been added.

NCONSIST - the minimum number of consistent generalizations produced.

DESCTYPE - the type of generalization, either discriminant or characteristic.

MINCOVER - for characteristic generalization, the minimum percentabe of the rules that must be covered.

MAXBACK - for characteristic generalization, the maximum number of backtrack steps allowed in the graph matching algorithm before declaring a mismatch. Zero denotes unlimited backtracking.

SUBGLIMIT - the limit on graph matching backtracking, normally 0 (no limit). During characteristic generalization SUBGLIMIT is set to MAXBACK.

REGENSTAR - the maximum number of regeneration cycles possible to remedy rules that would become inconsistent if *-referenced selectors were removed.

Appendix 1 - Users Gulde

2.10. L-rule parameters (LPARM)

This structure contains parameters relevant to the use of L-rules.

CSTF - the cost function indices in order of application

TOLER - the tolerance associated with each cost function

NF - the number of cost functions used

MAXL - the maximum number of sets of L-rules

2.11. Arithmetic Expression Variables

Arithmetic expressions are parsed by VLINT using the second half of the parse table. VLINT is passed the starting row in the parse table where it is to start parsing. For arithmetic expressions, this row is a constant defined as ARITHDD. Arithmetic expressions are parsed onto an ARITHSTACK in reverse polish notation. The program uses a grammar which actually causes the order of execution to be from right to left. The ARITHSTACK entry contains the following fields:

ACTION - is a code telling what to do with this entry. It takes on the values ADD (perform addition), SUBTRACT (perform subtraction), MULTIPLY (perform multiplication), DIVIDE (perform division), MODULO (perform modular division), MINUS (perform a unary minus), FUNCT (this entry is a function to look up the value of), and NUMBER (this entry is an integer). If ACTION is an operator, then the other fields of the record are meaningless.

ARGUMENT - if ACTION is NUMBER then this field contains the integer value of the integer. If ACTION is FUNCT then this field contains the PNO (index into symbol table) of the corresponding function or predicate. During the computation process in CALCARITH, the ARGUMENT fields are updated to point to the graph index of the corresponding function or predicate in TOFIND.

DUMMY - is an array of pointers to the symbol table for each dummy variable of the function or predicate in ARGUMENT. It is only meaningful if ACTION is FUNCT. The function is assumed to have ordered dummy variables (ORDIRR is FALSE). The list is terminated by a zero index.

2.12. Additional Variables

INFILE - an integer specifying whether input is from the terminal or from CFILE.

NMQ - the number of elements in MQ

FREEG - pointer to the list of available graph structures

LSLIST - pointer to the list of sets of L-rules

STAR - pointer to the list of formulas in a star

MQ - pointer to the list of consistent formulas

Appendix 1 - Users Guide

GSET - pointer to the list of input formulas

COVSET - pointer to the list of output formulas

STP,TRACE - sets of values for trace features

DBUG - set of procedure/function debug output indicators.

Appendix 1 - Users Guide

3. I/O Files

3.1. TABLES

This file contains the parse table information. Terminals in the grammar which are characters immediately follow any number (i.e. non-terminal). The end of each row of the parse table has a 0 followed by a (up to) 60 character name which describes this production (used in printing error messages). The column CONT has the value 1 if the line is an alternative RHS for the preceeding rule, 0 otherwise. The component parts of RHS denote either a terminal symbol (actual punctuation or the numbers 1-function symbol, 2-variable symbol, or 3-number) or the negative of the non-terminal number. Below is the parse table as it currently stands:

Non-terminal Number	Cont Flag	Action Code	RHS	DESCRIPTION
1	0	1	3 -3 0	a vl2 decision rule
2	1	2	-3 0	a vl2 decision rule
3	0	3	-4⇒ -6 0	a vl2 decision rule
4	0	4	-6 -4 0	a conjunction of selectors
5	1	4	-6 0	a conjunction of selectors
6	0	14	[-19= -10] 0	a variable selector
7	1	7	[-21(-14)= -10] 0	a function selector
8	1	18	[-21(-14)] 0	a predicate selector
9	1	7	[-21= -10] 0	a niladic function selector
10	0	8	-43 v -10 0	a list of numbers
11	1	9	-43.. -43 0	an interval of numbers
12	1	19	* 0	an asterisk (symbolizing the entire domain)
13	1	10	-43 0	a single number
14	0	11	-19, -14 0	a dependent variable list
15	1	20	-19. -14 0	a dependent variable list (order irr)
16	1	12	-19 0	a subscripted variable
17*	0	13	-19* -10; -17 0	a list
18*	1	14	-19= -10 0	a list
19	0	15	2 0	a subscripted variable
20	0	16	3 0	a number
21	0	17	1 0	a function symbol
22	0	32	-32= -23 0	an arithmetic derived descriptor
23	0	25	-25 -37 -23 0	an arithmetic expression
24	1	31	-25 0	an arithmetic expression
25	0	25	-27 -39 -25 0	a term
26	1	31	-27 0	a term
27	0	31	(-23) 0	a factor
28	1	31	-33 0	a factor
29	1	28	- -33 0	a factor
30	1	31	-32 0	a factor
31	1	28	- -32 0	a factor
32	0	31	-34(-35) 0	a function call
33	0	23	3 0	a number
34	0	21	1 0	a function symbol
35	0	31	-42, -35 0	a list of dummy variables
36	1	31	-42 0	a list of dummy variables
37	0	27	+ 0	an addition operator
38	1	29	- 0	an addition operator
39	0	24	* 0	a multiplication operator

40	1	26	/ 0	a multiplication operator
41	1	30	% 0	a multiplication operator
42	0	22	2 0	a dummy variable
43	0	5	-20 0	a number
44	1	33	1 0	a symbolic value
45	0	3	-46=> -48 0	a l-rule (added by wah 10/82)
46	0	4	-6 -4 0	a conjunction of selectors
47	1	4	-6 0	a conjunction of selectors
48	0	5	-50 -48 0	the consequence of the l-rule
49	1	5	-50 0	the consequence of the l-rule
50	0	14	| -19= -10| 0	a variable selector
51	1	7	| -21(-54)= -10|	a function selector
52	1	18	| -21(-54)| 0	a predicate selector
53	1	7	| -21= -10| 0	a niladic function selector
54	0	35	-19, -54 0	a dependent variable list
55	1	36	-19. -54 0	a dependent variable list (order irr)
56	1	37	-19 0	a subscripted variable

* denotes unused entries

3.2. EXPLAIN

This file contains text for explanation. Each explanation has a number and is delimited by a ! in column 1 followed by the number of the explanation preceeding the text and a ! in column 2 - 80 following the text. If a line ends with *, the program stops printing to allow the user to read the material.

3.3. CFILE

This file contains a set of input commands and data which is to be executed before the system asks for user input. Normally, input rules and certain parameters are included in this file. Comments can be placed in the file by enclosing them in percent signs ("%").

3.4. VL1EVE

This file contains a list of VL_1 type events. The file is in the format for AQ7 except that each event specification is preceeded with the class number of the associated decision. A -1 indicates a value which is irrelevant.

Appendix 1 - Users Guide

4. Program Structure

The program INDUCE 3 (Appendix 2) contains about 8000 Pascal statements and 80 procedures. These procedures may be grouped into these classes: 1) control and user interface, 2) VL to internal formula representation, 3) graph manipulation, 4) add new functions, 5) AQ complex manipulation, 6) L-rule procedures and 7) supporting procedures. Each group of procedures operates nearly independently of the others thus giving the possibility of implementation on a smaller machine.

The main program accepts high level commands and calls the appropriate procedures to perform the requested action. Any input in the form of a decision rule passes through the VLINT procedure for translation to internal format. On some occasions, information is then copied from one internal form to another (E command) but most of the work is done in VLINT. All other user interaction takes place in ENTERP (enter parameters). The VL_1 mode uses the VL_1 procedure and AQ, bypassing all procedures dealing with graph manipulation. To cover a set of formulas, the COVER procedure is called which in turn, calls NEWGP to grow generalizations and AQSET to apply AQ to the consistent generalizations in MQ.

4.1. Control and User Interface

MANE - process high level commands

ENTERP - Decode commands at the parameters command level.

ENTERD - Read domain structure definitions.

PGRAPH - Print the graph structure as VL_2 formula. Assign indices to all variables. Write out function and arguments if any. Then, write out reference. If tree structured domain and the value is an internal node, then only print out the internal node.

EXPLN - Find requested text from the file EXPLAIN and print it stopping at (*) for carriage return from user.

4.2. VL Translation to Internal

TOKEN - Read an input line and add the terminator (?). Scan over the letters and digits and set CTYPE (0-delimiter, 1-function symbol, 2-variable, 3-number). If CTYPE was 0 then determine internal representation of the delimiter. If CTYPE is 1 or 2, then find the row in the symbol table (FINDROW). If it is not there, then add a new row to the symbol table FIXSYM (the name of the symbol is located between FCURS and LCURS in BUF). In the case of a variable, add an extra row for the domain of the variable in addition to a new row for the variable itself (i.e., a row for X in addition to a new row for X1). If CTYPE is 3, then compute the value of the number. Return the location in the symbol table or the computed number in the parameter SROW and delimiter type in CTYPE.

VLINT - Translate VL_2 formula into graph structure. Maintain a value stack (VSTK), a function stack (FSTK), semantic stack (SSTK) and a parse stack (PSTK).

PSTK - Contains a stack of all non terminals not yet completed.

SSTK - Contains the tokens from the input buffer which have not been matched with an element of a completed production.

Appendix 1 - Users Guide

VSTK - the stack of numbers not already placed into the graph.

FSTK - the stack of arguments of a function (FSTK[1] is always the function symbol of the selector being parsed).

As tokens are accepted from the input buffer, they are matched with productions in PT. If a token does not match an element of a production which is a non terminal, the location of the non terminal is placed on PSTK and the production defining the nonterminal is tried (PROD and LOC determine the current element in PT under consideration). If there is no match, then try an alternative definition of the non terminal. If there is no alternative, back down PSTK and try another alternative of this non-terminal.

If a token matches the element of PT under consideration, put this token on SSTK and try the next element in the production. If the complete production is matched, replace the matching tokens on SSTK with the appropriate nonterminal, back down PSTK to the previous location, process the indicated semantic rule (PROCESS) and proceed. Once the productions in row 1 of PT are completed, the expression is said to be syntactically correct.

PROCESS - Execute the semantic rule for the production (-PROD). Briefly, node assignments are made using the elements in FSTK, values in the reference are assigned from elements in VSTK. The MNVAL and EVAL fields of the symbol table are updated and the type of a node is determined. Links between variables and functions are assigned recalling that FSTK[1] contains the location of the function.

PARSEARITH - Execute semantic rules for arithmetic derived descriptors. A data structure called an ARITHSTACK is built which contains the arithmetic expression in reverse polish notation. The first element on the stack is the new variable to which the expression value should be assigned.

4.3. VL_2 Formula Manipulation

SUBG1 - Determine if the graph in G1 is a subgraph of the graph in G2. The procedure SUBG1 selects a starting node of G1 and a matching node of G2. SUBG ties to find a node in G2 to match each node in G1. If such a matching is found, it is reflected in ASSIGN records that record corresponding nodes in the two graphs. If INSD is true, two nodes (selectors or variables) are matched only if the values of the first cover the values of the second. If INSD is false, the values of the two nodes need only intersect.

ADDCONS - Add consequence of restriction to graphs.

TRIMG - Trim a list of formulas to MAXS elements, return other formulas to FREEG. Place formulas with COST[3] into MQ (consistent formulas). Instead of sorting a linked list, the array CA is sorted. Costs are assumed to be stored with each formula (calculated in COVER).

COSTG - Determine the cost function CT specified for the formula P.

COVER - Cover the set of formulas ES. First, select an element of F1 to cover (G) and compute the initial partial star. For all nodes in a graph, the flag COUNT is set to 1. Trim the partial star and apply absorption. Form a new partial star by calling NEWGP for each remaining element of the trimmed partial star. Once NCONSIST elements are in MQ, apply AQ (via AQSET) to each consistent formula. Trim the list to one best element and remove elements of F1 covered by this formula (set FP to false). Select a new element of

F1 and repeat until F1 is exhausted.

NEWGP - Add new selectors to the input graph to form a list of ALTER or less new formulas. G0 is the old generalization of G1; direct association exists betwen nodes of G0 and nodes of G1 (i.e., correspondence is 1-1 by row, not through ASSGN as with other correspondences). The procedure tries to form new connected graphs but if that fails a disconnected graph may result. A list of selectors which may be connected to the current graph is created in CANDID and sorted with respect to VCOST and NARG. All variables connected to existing nodes are flagged (P1) and then all function nodes connected to flagged variables are marked (P3). All "P3" selectors are placed in CANDID. Then, a new graph (in SLST) is formed from the old one with a new selector and any relevant variables. EQUIV type functions are discarded if they have no more than 1 argument. The list SLST is returned to the calling procedure (COVER).

4.4. AQ Complex Manipulation

AQ - perform the AQ algorithm on the sets F1 and F2 of complexes obtained from the sets F1 and F0 of rules. This routine is much like AQ and is not further explained here.

LQST2 - perform the LQST function during characteristic generalization. During characteristic generalization, it becomes necessary to have a minimum sized cover which covers all rules (not complexes) in F1. Since there is often a many-to-one relationship between complexes in F1 and rules in F1, this is a non-trivial task and LQST2 performs this task. During the ALLC procedure, a CPXTABLE is attached to each complex which lists the rule numbers of the original rules in F1 which the complex covers. LQST2 loops finding the complex which covers the most rules, combining its reference values with the complex currently being derived and eliminating all complexes which cover the rules it covers from further consideration. When the set of complexes is exhausted, a quasi-minimal cover has been found. Trace 7 causes various information to be printed out during this covering process.

AQSET - Translate from VL_2 representation (graph structure) to VL_1 representation (sequence of sets of values). Create two sets of complexes, F1 containing subgraphs of graphs with VL_2 set F1, and F2, the set of complexes associated with c-structures (GSUB) isomorphisms with elements of the VL_2 set F0. The first element of F1 corresponds to the part of the graph GSUB which was consistent. The two sets of events are passed to the AQ procedure which returns a complex covering the first element of F1 but no element of F2. This is copied back into GSUB to form the extended reference generalization.

ALLC - Translate from graph to complex and add to the list of complexes if not already there. Also, set up SLOC to relate VL_1 variables to symbols and find NVAR (number of variables). Use assignments from the c-structure GSUB and the graph G1 for nodes with flag P1 in GSUB. All meta-selectors are loaded in the first METATRIM VL_1 variables, the remainder are nodes with flag P1 in GSUB. A CPXTABLE is maintained for each complex which contains a list of the rule numbers of the rules which that complex covers. This is used by the LQST2 routine.

VL1 - Input VL_1 events from the file VL1EVE and translate to complex storage. Call AQ to find generalization and then print result.

TRIMF - Trim a list of complexes with respect to AQCSTF etc. This is nearly the same as

TRIMG but uses CPX structures.

COSTF - compute the cost of a complex.

PCPX - Print in VL_1type format indexing into SYMTAB using AQ^.SLOC array to find the maximum and minimum values.

4.5. Add New Functions

ADDSEL - find sets of unary function nodes which have the same value in the graph. If several nodes have the same value, then a new symbol is added to SYMTAB and a new node is added for the predicate SAME<sym>(p_1.p_2...) where p_1, p_2, etc. are dummy variables that have links to the predicates having the same values. <sym> is the name of the function, e.g., several COLOR function nodes may give rise to a SAMECOLOR predicate.

ADDML - Add MST, LST, EXTMTY type predicates. For each binary predicate whose arguments assume values from the same domain, add extremity predicates.

ADDMETA - add meta-selectors to each formula in F1 and F0. For each unary function and function value, count the number of occurrences of this pair in a formula and add a selector of that type to the formula (COMPMS). Calculate F1COV and F0COV and sort the list of meta selectors (TRIMM).

PROCARITH - loop thru F1 and F0 adding an arithmetic derived descriptor to each graph in turn. This is accomplished by first creating a temporary graph (TOFIND) which contains the necessary functions and dummies from the right hand side of the arithmetic derived descriptor rule. This forms a connected graph structure. Then we call CALCARITH to find all isomorphisms between TOFIND and the rules in the rule base. PROCARITH contains the internal procedure BUILDG which builds a graph corresponding to the arithmetic expression.

4.6. L-rule Procedures

LSINSERT - inserts a new L-rule into an already existing set of L-rules. It is inserted into the first set of L-rules to which it is related to some L-rule in that set. Two L-rules L1 and L2 are related if the consequence of L1 shares some selector (with the same function symbol and intersecting reference) with the condition of L2, or vice-versa.

LSCHOOSE - chooses MAXL or fewer sets of L-rules. The L-rules are chosen according to the cost criteria for L-rules. This procedure is very similar to TRIMG, the analagous procedure for VL2 formulas.

ADDLRULES - adds the consequence of a L-rule to an event.

4.7. Supporting Routines

ILINE - input a new line from CFILE or the terminal placing it into the main input buffer MAINBUF.

Appendix 1 - Users Guide

GETCHRR - get the next character from MAINBUF.

GETINT - get an integer from the next characters in MAINBUF. If the next character is not a digit or minus sign, assume 0 was found. The buffer pointer is left pointing to the non-digit following the integer.

INIT - All parameter initialization is done here.

Appendix 1 - Users Guide