FINAL REPORT

RESEARCH ON SECURE SYSTEMS AND AUTOMATIC PROGRAMMING

Period: March 1, 1973 to August 31, 1977

Co-Principal Investigators:
Saul Amarel
(201) 932-3546/2001

C. V. Srinivasan
(201) 932-2019/2001

ARPA Order Number 2406
Program Code Number 3030
Grant Number DAHC15-73-G-6

Department of Computer Science
Rutgers, The State University
New Brunswick
New Jersey 08903

October 14, 1977
RELATIONS BETWEEN RECURSIVE DEFINITIONS AND THEIR EFFICIENT ALGORITHMS

M. C. Paull

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
1. INTRODUCTION

Typically there are significant differences between the formulation of an algorithm and its ultimate implementation. For example, the minimum path between two nodes in a weighted di-graph can be found by enumerating all paths between the two nodes and choosing the smallest. This approach can easily be formulated as a recursively defined function, which may in turn be implemented in a standard way. This is significantly different than Dykstra's algorithm, the favored shortest path implementation. On the one side, close to the problem statement, then there is an initial, simply formulated, but often inefficient algorithm. On the other side, nearer to the final implementation, is an efficient algorithm. The study of the connection between these two is the subject of this paper.

It will be assumed that the initial formulation of an algorithm is as a recursive definition and that this definition is in a standard form (to be given). The standard form was chosen because, firstly, it is one which, in our experience, has frequently arisen naturally as an initial algorithm formulation. Secondly the chosen form lends itself nicely to an overview of a variety of possible implementations of the algorithm thus formulated. The recursive definition though sufficient to provide the value of the function anywhere in its domain is non-deterministic as to which of a variety of sequential implementations are to be used to determine that value. The variety of implementations correspond to the various orders of substitution which are equally valid in evaluating such a definition. Some orders of evaluation
-2-

become possible only if the primitive functions which enter into the recursive definition have appropriate properties. Different orders of evaluation will result in different memory requirements, but will not cause significant time differences in the resultant implementations. This dependence of memory requirements on the order of evaluation is the main subject of section 2 of this paper.

Implementation of the recursive definition generally requires the repetitive execution of similar operations. If it can be shown that some pairs of these operations will yield the same or similar intermediate results at different points in the computation—then only one such intermediate result need be computed and remembered. It may then be accessed from memory when needed again instead of being recomputed. This can happen many times in a sufficiently systematic way so that a significant time saving can be realized. The existence of this situation depends on properties of the primitive functions which compose the recursive definition. In section 3 a significant class of problems for which time efficient implementations are available is considered.

Related Work

The work reported here is in an area of study in which there have been a number of significant publications. Strong has identified a class of recursive definitions for which memory efficient implementations [5,6] (called 'flowcharts') are available. This class is defined in terms of a recursive scheme whose constituent primitive functions are virtually unrestricted. If the properties of these primitive functions are restricted somewhat, a wider class of recursive definition forms will yield similar memory
efficient implementations. Such restrictions are considered here because they arise naturally in practice. So this aspect of the work can be considered an extension of Strong’s results.

Burstall and Darlington studied properties of recursive definitions whose existence allows efficient implementation, with one objective being the incorporation of a search for such properties in an optimizing compiler. Later Burstall and Darlington extended this study to consideration of transformations of recursive definitions which are likely to produce better implementations. The spirit of our work here is largely in tune with that of these investigators with some significant differences in emphasis and in the particular properties studied. Our emphasis has been mainly on understanding the complete set of properties which allow the transformation from an initial recursive definition to the best algorithms actually known and to the proof of this connection. Thus we tend to consider relatively complex sets of properties and transformations as opposed to many simple properties and transformations. We also study mainly one form of first order recursive definitions, rather than the many forms they consider.

In a more general way this work is also related to work in AI and the design of algorithms, to which specific reference will be given at the appropriate point in the paper.

The remainder of this introduction is devoted to a sketch of the definitions and results to be detailed in the subsequent sections of the paper.

First-order means a definition in which the defined function symbol never appears nested on the right.
Appendix I contains a summary of most of the notation used in the paper. (This notation is also defined on first use in the paper.)

The Standard Form

This paper concerns the implementation of recursive definitions of a function \( f(X) \) in a class \( F \) in which every definition has the following form:

\[
\begin{align*}
  f(X) &= q(X) & \text{if } T(X) \text{ (terminal condition and values)} \\
  f(X) &= w(f(o_1(X)), \ldots, f(o_m(X))(X))) & \text{if } \bar{T}(X) \text{ (body)} \\
  \text{initially } X \in D_f
\end{align*}
\]

where the data structure \( X \in D_f \), primitive functions \( w, q, o_1, o_m \), and predicates \( T \) in the definition collectively designated by the tuple \( <D, w, q, o, m, T> \) must be constrained so as to make \( I \) a terminating definition.

A definition is terminating if for each \( d \in D_f \) the sequence of expressions resulting from substitution for forms \( f(a) \) (where \( a \) is any expression using \( I \) which starts with \( f(d) \), \( d \in D_f \), and next produces \( w(f(o_1(d)), \ldots, f(o_m(d)(d))) \), etc. has the properties:

1. It is always possible to evaluate \( T(a) \) and if \( T(a) \) is false it is always possible to evaluate \( m(a) \), and \( o_i(a) \) for \( 1 \leq i \leq m(a) \).
(2) Independent of the order of substitution for the different appearances of the form \( f(a) \) after the same finite number of such substitutions a 'terminal' expression will be obtained in which, for every appearance \( f(a) \), \( a \) is terminal (i.e. \( T(a) \) is true) and \( q(a) \) can be evaluated.

(3) The function \( w \) is defined so as to make it possible to evaluate the terminal expression in any order consistent with its parentheses structure.

The tuples \( <D,w,q,O,m,I> \) which satisfy the above constraint are members of the set \( V \). The set of definitions of form \( I \) which satisfy these constraints constitute the recursive scheme \( F(V) \).

This form of definition often arises in practice as an initial solution to an algorithm design problem, particularly when the problem can be viewed as requiring an enumeration or an enumeration followed by a selection (search). The examples of recursive definitions in \( F(V) \) given below arose from adopting such a point of view. Their structure can be easily seen by evaluating them for some small initial values of their arguments.

Examples:

Ex. 1.1 If \( f(X) \) is to be the set of all \( n \) bit binary numbers (let \( N \) be the set of positive integers), then:

\[
X = \{<a,n> | a \text{ a string of 0's and 1's, } n \in N \}
\]

\[
f(X) = f(a,n) = \{a\} \quad \text{if } n = 0
\]

\[
f(X) = f(a,n) = f(a/<0>,n-1) \cup f(a/<1>,n-1) \quad \text{if } n > 0
\]

where \( / \) is string catenation, and \( \cup \) set union.

\( X \) initially \( \in \{<\lambda,n> | n \in N \} \)

Then ex. \( f(\lambda,2) = f(<0>,1) \cup f(<1>,1) = (f(<0,0>,0) \cup f(<0,1>,0)) \cup f(<1>,1) \)

\( = \{<00>\} \cup f(<0,1>,0) \cup f(<1>,1) = \text{ etc.} \)
Ex. 1.2 If \( f(X) \) is the set of all permutations of the first \( n \) integers,

\[
X = \{a, \alpha\mid \alpha \text{ is a string of positive integers}\}
\]

\[
\begin{align*}
f(X) &= f(n, \alpha) = \{a\} & \text{if } n = 0 \\
\quad \text{and if } p = |\alpha| = \text{the length of } \alpha \text{ then} \\
f(X) &= f(n, \alpha) = f(n-1, a[a_p+n]) \cup \ldots \cup f(n-1, a[a_p+n]) & \text{if } n > 0 \\
\end{align*}
\]

where \( a[a_i+n] \) is an inserting function; i.e.

- if \( \alpha = \langle a_1, \ldots, a_p \rangle \) then \( a[a_i+n] \) is the result of inserting the integer \( n \) after component \( a_i \) in \( \alpha \) or is \( \langle a_1, \ldots, a_{i-1}, n, a_i+1, \ldots, a_p \rangle \).

\( X \) is initially \( \epsilon \{\langle n, \lambda\mid n \in N\} \)

Then ex. \( f(2, \lambda) = f(1, \langle 2 \rangle) = f(0, \langle 12 \rangle) \cup f(0, \langle 21 \rangle) = \{\langle 12 \rangle\} \cup f(0, 21) \)

\( X \) is initially \( \epsilon \{\langle 12 \rangle\} \cup \{\langle 21 \rangle\} \)

Ex. 1.3 \( f(X) \) is the string of moves (each a pair of numbers \( <a, b> \) meaning move a disc from pin \( a \) to pin \( b \)) necessary to optimally solve the, now classical, Tower of Hanoi puzzle. To move \( n \) discs initially on pin 1 to pin 2:

\[
X = \{\langle x, y, z\rangle, n\mid<x, y, z> \text{ is a permutation of } \langle 1, 2, 3\rangle, n \in N\}
\]

\[
\begin{align*}
f(\langle x, y, z\rangle, n) &= \langle x, y \rangle & \text{if } n = 1 \\
f(\langle x, y, z\rangle, n) &= f(\langle x, z, y\rangle, n-1) \cup f(\langle x, y, z\rangle, 1) \cup f(\langle z, y, x\rangle, n-1) & \text{if } n > 1 \\
\end{align*}
\]

\( X \) is initially \( \epsilon \{\langle 1, 2, 3\rangle, n\mid n \in N\} \)

Algorithms to Implement Definitions in \( F(V) \) which are Efficient in Use of Memory

An 'algorithm scheme' defining a set of algorithms is defined in a manner analogous to that used in defining a recursive scheme like \( F(V) \). In this paper algorithm schemes generally will involve standard assignment and conditional statements using the same unspecified set of data-structures \( D \), primitive functions \( w, q, O, m \) and predicate \( T \) (designated by the tuple \( <D, w, q, O, m, T> \)
used in defining the recursive scheme $F(V)$. If we constrain the selection of tuples to be a member of a set $V$, the set of algorithms thus defined is designated $S(V)$ and a particular algorithm $c S(V)$, corresponding to a tuple $v \in V$ is designated $S(v)$. The recursive and algorithm scheme $F(V)$ and $S(V)$ are equivalent iff for each $v \in V$, $F(v)$ is equivalent to $S(v)$. A recursive function definition $F(v)$ and an algorithm $S(v)$ are equivalent if with domain $D$ in $v$, for every $d \in D$, the value of $f(d)$ as computed with recursive definition $F(v) = \text{value of the result of running the algorithm } S(v) \text{ with } d \in D \text{ as its initial value}$. It is easy to find a number of algorithm schemes equivalent to $F(V)$.

A main purpose of this paper is to show that for a set $V'$, built from $V$ by constraining the function $w$ to be 'associative' and the set of functions $O$ to have an 'inverse', there is an algorithm scheme $S(V')$ equivalent to $F(V')$ which is particularly efficient in its use of memory. The algorithm scheme available when these conditions are satisfied is given in figure 2.2. The algorithm scheme $S(V')$ is given in terms of the data-structures, primitive functions and transformations of these primitive functions (inverse of $O$ for example) which are immediately available under the assumption of the existence of an 'inverse', that appear in the equivalent recursive scheme $F(V')$.

For many of the recursive definitions in the class $F(V')$, the equivalent member of the class $S(V')$ - which can be obtained mechanically from the recursive definition is the 'good' algorithm usually used to realize that definition. Thus corresponding to example 1.1, the algorithm obtained by instantiation of that particular $D, w, q, O, m$ and $T$ in $S(V')$ is one in which:

---

1 Theorems 2.1 and 2.2

2 These terms are defined in section 2. An inverse operation plays a similar role in [6]. Our 'inverse', however, is different, having been independently developed [7,8] in combination with associativity to delineate another class of definition with efficient implementations.
First a string of $n$ 0's is formed and outputted - being the first binary number produced, then, because the rightmost symbol in the string is a 0 it is changed to a 1 and the result outputted. In general, the algorithm remembers the last binary number formed and outputted, say $X$. The next binary number is formed by a scan of the bits of $X$ starting with the rightmost bit, and changing them by the following scheme. Let $b_1$ the bit under scrutiny - if $b$ is a 0 it is changed to a 1 and the result is the next binary number to be outputted - if it is a 1 it is changed to a 0, $b$ becomes the bit in $X$ one position to the right of the current $b$ and the scrutiny is repeated. When the leftmost bit of a number $X$ becomes $b$ and that bit $= 1$ then the process terminated. In summary this algorithm for producing all $n$-bit binary numbers, consists simply in 'adding 1' to produce successive members of the set. It is the 'good' algorithm for producing the set. It keeps in memory only the last number produced thus using an amount of storage roughly equal to that required to hold the argument of $f$ in its recursive definition. This is characteristic of all the algorithms in $S(V')$ in relation to the equivalent member of $F(V')$ and is the 'memory efficiency' mentioned.

In a similar way, the algorithm for example 1.2 obtained by instantiation of the primitives that appear in the recursive definition in example 2.2 produces one permutation at a time. A permutation is produced from the previous permutation by interchange of adjacent terms. This again is the 'good' algorithm for generating permutations.

Creating an Inverse

In examples 1.1 and 1.2, the given 0-functions had an inverse - in example 1.3 the 0-function as given does not have an inverse and thus the
algorithm scheme $S(V')$ is not available. However, as will be shown—when in a recursive definition, $f \in F(V)$, the $O$-function does not have an inverse—a simple transformation of $f$ to an equivalent definition, say $f'$, involving an O-function having an inverse can always be found in $F(V')$. Thus $f'$ will have an equivalent in $S(V')$. This new definition $f'$ is equivalent to $f$ in the sense that to each argument $d$ of $f$ there is a 'simply' computed argument $d'$ of $f'$ such that $f'(d') = f(d)$. Using this transformation, an equivalent definition to that of example 1.3 will be given subsequently, whose equivalent algorithm in $S(V')$ will produce the moves necessary to solve the Tower of Hanoi problem—one at a time, the only temporary memory necessary being that for a record of the previous move and its number.

Memory Efficiency

In the standard compiler implementation of a recursive definition of the form of I, that definition is taken to describe a procedure which calls itself. The procedure uses a stack to temporarily remember, amongst other things the set of arguments—the data structure—associated with the call. The size to which the stack grows varies and depends on the depth of the calls. In general, if the definition is non-linear, i.e. has more than 1 call of the defined function on the right, then the arguments of the $w$ functions will have to be stacked also. When the memory efficient algorithm to be described here is applicable then both of these stacks can be eliminated. Instead only 1 copy of the argument of the calling function will be saved. All other temporary memory uses in the algorithm are comparable to those in the standard implementation. It will be possible to eliminate the need for these stacks for any definition of form I, provided, only, as we have said that the $w$ function is associative and the O functions have a uniform inverse.

Although the 'memory efficient' algorithms of $S(V')$ are honestly so for the most part, the nature of the memory efficiency can be misleading. The implementing algorithm available when $w$ is 'associative' and the O-function has an 'inverse' is efficient in the sense that the memory required is usually of the order of the largest storage required for the argument (also called a data structure) of $f$ which arises if $f$ is evaluated by successive substitutions.

Usually this largest data-structure for which memory need be provided is Theorem 2.3.
requires a small amount of memory relative to the total of all data-structures produced during the implementation of the definition for a given initial data-structure - ex. of the order of a single member of a set when a set is being enumerated. Even when the 'inverse' does not exist it can be incorporated as previously noted, leaving the 'memory efficiency' notion still viable. However there is another way of obtaining a 'memory efficient' equivalent algorithm which is deceiving.

This technique involves obtaining a technically correct equivalent recursive definition of \( f \), say \( f' \) having only one occurrence of \( f' \) on the right, but in compensation involving much larger data structures \( X' \) and complex function \( o' \) than the corresponding \( X \) and \( o \) of \( f \). That is, for each definition of form I there is an equivalent definition of the form:

\[
\begin{align*}
\text{if } T(X') & \quad \text{then } f'(X') = q'(X') \\
\text{if } \overline{T}(X') & \quad \text{then } f'(X') = w(f'(o'(X')))
\end{align*}
\]

Initially, \( X' \in D_f \).

By equivalent, we mean that there is a 1-1 correspondence \( g \) between \( D_f \) and \( D_f' \), so that for each \( d \in D \):

\[ f(d) = f'(g(d)) \]

If \( f' \) has an inverse then it can be realized in the same memory efficient manner as other definitions in \( F(V') \) and if not it can easily be modified so as to have one while still keeping the result in the form of II. Memory efficiency, however, means that the memory requirement will not exceed the size of the largest data-structure which arises as an argument of \( f \) during evaluation of \( f' \). But in this equivalent definition that

* Theorem 2.1 and theorem 2.2 give the two classical ways this is done, called breadth-first and depth-first respectively.
Data-structure is typically much larger (often exponentially) than that which could arise in the original definition.

The term 'memory efficiency' as used here then requires caution in its application.

**Time Efficient Implementations**

As noted earlier the opportunity for time efficient implementations of recursive definitions in $F(V)(= F$ from here on) arises when repetitive use of the same operations are necessary in the evaluation of the function. In the time efficient implementations the originally repeated operations are done once - the result being remembered for later use. This is classically called 'pruning' in the Artificial Intelligence literature. For some subclasses of $F$ the nature of the repeated operations are sufficiently independent of the particular initial data-structure so that one can design a class of implementations guaranteed to be more efficient in all cases than the standard implementation. An example of such a subclass is all functions which are substitutionally solvable and are of the form given below.

Let $\hat{V} = \langle x_1, x_2, \ldots, x_n \rangle$ = a vector whose values are integers,

$\hat{c}_i = \langle c_{i1}, \ldots, c_{in} \rangle$ = a constant vector of integers - is vector subtraction.

$$\begin{align*}
  f(\hat{V}) = q(\hat{V}) & \quad \text{if } \hat{V} \text{ is any of a finite set of integer vectors, say } K \\
  f(\hat{V}) = w(f(\hat{V} - \hat{c}_1), \ldots, f(\hat{V} - \hat{c}_n)) & \quad \text{if } \hat{V} \notin K \\
  \text{Initially } \hat{V} \in \text{ a defined set of integer vectors, say } L
\end{align*}$$

A specific simple member of this subclass is the definition for the
Fibonacci series \((\vec{V} \text{ is a one dimensional vector})\)

\[
\begin{cases}
  f(n) = 1 & \text{if } n = 1, \text{ or } 0 \\
  f(n) = f(n-1) + f(n-2) & \text{if } n > 1 \\
  \text{Initially } n \in N
\end{cases}
\]

For any definition in this subclass in which \(\vec{V}\) has \(m\) components it is easy to see that the function can be implemented by \(m\) nested DO-loops. This implementation requires much less than the exponential time involved in a straightforward implementation of the recursive definition - which does not take advantage of repeated suboperations. Although detection of a member of this subclass is relatively easy, the detailed dependence of the parameters of the nested DO implementation on the properties of \(w\) and the and the sets \(K\) and \(L\) in this subclass of definition is an interesting study. This study however is not carried out here.

Another such subclass called the 'explicit history' class will be considered here. The evaluation of members of this class will be shown to be equivalent to solving a set of equations* analogous to sets of linear equations. Consequently an algorithm analogous to Gaussian elimination will be shown to be an available implementation of explicit history definitions. This algorithm has polynomial complexity as opposed to the exponential time required in a straightforward implementation of the recursive definitions in this class.

* Theorem 3.1.
Equivalent Formulations of Recursive Definitions

To be in the class F(V) recursive definitions must be of form I. Despite this constraint a function may have a number of different definitions all in F(V). Some of these may have desirable properties absent in others. Transformations which can be used to obtain equivalent definitions in F(V) with desirable properties are developed in section 3 of this paper. Actually, example 1.1 gives a definition of the set of all n-bit binary numbers which may not be entirely natural. One which may be considered is based on the fact that:

The set of all n-bit binary numbers = the set of all (n-1)-bit binary numbers each with a 0 appended together (unioned) with the set of all (n-1)-bit binary numbers each with a 1 appended.

A formal statement of this definition is:

Ex. 1.5  \[ f(n) = \begin{cases} \lambda & \text{if } n = 0 \\ (\langle 0 \rangle \biguplus f(n-1)) \cup (\langle 1 \rangle \biguplus f(n-1)) & \text{if } n > 0 \end{cases} \]

where if B is a set of strings and a a string a \( \biguplus B = \{ a \biguplus b \mid b \in B \} \)

Initially n \( \in N \)

Then ex. \( f(2) = \langle 0 \rangle \biguplus f(1) \cup \langle 1 \rangle \biguplus f(1) = \langle 0 \rangle \biguplus (\langle 0 \rangle \biguplus f(0)) \cup \langle 1 \rangle \biguplus f(1) \)

\( = \langle 0 \rangle \biguplus (\langle 0 \rangle, \langle 1 \rangle) \cup \langle 1 \rangle \biguplus f(1) = \{\langle 00 \rangle, \langle 01 \rangle\} \cup \langle 1 \rangle \biguplus f(1) = \text{etc.} \)

This definition, though still strictly in form I is not in F(V') because w is not associative. Therefore the scheme S(V') is not directly available for its realization. However, there are theorems* which will

* Theorem 3.2
In this case give an equivalent definition in \( F(V') \) thus once again making the realization of \( S(V') \) available.

In fact, the definition of example 1.1 can be obtained by the application of such theorems to example 1.5. It is interesting to compare the above interpretation of the definition in example 1.5 with one for the equivalent definition in example 1.1 which was originally claimed to be natural.

Interpretation of Definition 1.1:

The set of all \( n+|a| \)-bit binary numbers which have a prefix \( a = \)

The set of all \( n+|a| \)-bit binary numbers having a prefix \( a \) followed by 0 together (unioned) with the set of all \( n+|a| \)-bit binary numbers having a prefix \( a \) followed by 1.

2. MEMORY EFFICIENT IMPLEMENTATIONS

The first part of this section, thru page 29, is largely devoted to material which is probably familiar. This is done inorder to develop the definitions of a number of terms which are used later in this section. Altho the concepts are familiar the terms we use may not always be so. Two well known, 'classical' implementations of recursive definitions, both of which use stacks are shown to valid in these preliminary pages. This is done for comparison with the 'inverse' implementation, which uses no stacks, and whose description and justification is the main objective of this section. The 'classical implementations are described in a somewhat unusual way, different than the flowchart form used for the 'inverse' implementation. Their validity is established in this form which was thought to be an excercise of sufficient interest to justify inclusion here since the form in which these implementations are given is generally applicable. The inverse implementation, for example, could be given in
this form. In any case, the material in the preliminary pages can easily
be skipped and only referred to to pick up definitions of terms used later,
without losing the main point of the paper.

**Definition of Standard Recursive Scheme** $F$:

Consider the set $F'$ of all functions $f$ that can be defined as
follows:

**Def. 2.1**

$$
\begin{cases}
  f(X) = q(X) & \text{if } T(X) \\
  f(X) = w(f(o_1(X)), \ldots, f(o_m(X))) & \text{if } \overline{T}(X) \\
  \text{initially } X \in D_f
\end{cases}
$$

where the primitive functions and predicates which are used in the de-
finition are weakly constrained as to the nature and extent of their
domains and ranges. $D_f$ is the set of initial data-structures and may be
any set. Other sets must be included in some of the domains of some of
the primitive functions. These other sets are defined recursively, using
the primitive functions. First these sets are named and their relation to
the primitive functions given, then they are defined.

- $m$ is a function whose domain must include the set $\Delta_f$ and whose range is the
  positive integers $\geq 1$. $m(X) \geq 1$ for all $X \in \Delta_f$.

- $O$ is a set of functions: $\{o_1, o_2, \ldots\}$.

  The domain of $O$ must include the set $\Delta^1_f$. $\Delta_f$ is the union of all the domains
  of the functions in $o_1$ and is called the domain of the $O$-function.

  The range of $o_1$ must include the set $p^1_f$.

  The union of the sets $p^i_f$ of all the functions in $O$ is the range
  of the $O$-function and is called $P_f$.

- $T$ is a predicate whose domain includes $D_f \cup P_f$. Its range is
  \{true, false\}.

- $q$ is a function whose domain must include $Q_f$. Its range may be
  any set, say $W_f$. 
w is a function whose range is called $W_f$ and whose domain must include $V_g w$. The sets named above are defined as follows (the subscript $f$ is dropped where it is not essential):

$$\Delta^1 = \{d|d \in D \text{ and } T(d)\}; \text{ and for } j > 1$$

$$\Delta^j = \{o_1(x)|x \in \Delta^{j-1} \text{ and } i \leq m(x) \text{ and } T(o_1(x))\}$$

$$\Delta = \bigcup_{i=1}^{+\infty} \Delta^i$$

The set $\delta^i$ of $o_1 \in 0$ is:

$$\delta^i = \{x|x \in \Delta \text{ and } i \leq m(x)\}$$

The range of 0 is:

$$P = \{o_1(x)|x \in \Delta \text{ and } i \leq m(x)\}$$

The range $p^i$ of $o_1 \in 0$ is:

$$p^i = \{o_1(x)|x \in \delta^i\}$$

The set of terminal data-structures $Q$ is:

$$Q = P - \Delta$$

The set $W$ is defined as follows:

$$W^1 = \{w(x_1,\ldots,x_n)|x_1 \in W_f, n = \text{ a positive integer } N\}; \text{ for } j > 1,$$

$$W^j = \{w(x_1,\ldots,x_n)|x_1 \in W^{k-1}, k(j); n \in N\}$$

$$W = \bigcup_{i=1}^{+\infty} W^i$$

If in addition to being a member of the set $F'$, a recursive definition is terminating; as defined below it is a member of the set $F$. We need some preliminary definitions.

If $i_1, \ldots, i_n = 1$ is a sequence of integers then $o_{i_1,\ldots,i_n}(x) = o_1(x)$ is an abbreviation for $o_{i_1}(\ldots o_{i_2}(o_{i_1}(x)) \ldots); o_\lambda(x) = x.$
A length 1 sequence of integers \(i_1\) is applicable to a data-structure \(X \in T_f\) if \(i_1 \leq m(X)\). A length \(n\) sequence of integers \(i_1, \ldots, i_n\) is applicable to a data-structure \(X\) if \(i_1, \ldots, i_{n-1}\) is applicable to \(X\) and \(i_n\) is applicable to \(o_{<i_1, \ldots, i_{n-1}>}(X)\).

\(I(X)\) is the set of all integer sequences applicable to \(X \in T_f\).

\(F\) is terminating if \(\forall d, d \in D\) implies \(I(d)\) is finite.

Note that if \(I(X)\) is finite it cannot contain an infinite sequence, because it always contains all prefixes of any sequence it contains.

This completes our definition of \(F\). Next we give some simple consequences of the definition which will be used later. First, the substitutionally solvable property that \(d \in D, I(d)\) is finite can be extended to any \(X \in T_f\). This is done in lemmas 2.1 and 2.2.

**Simple Properties of \(f \in F\)**

**Lemma 2.1:** If \(f \in F\) and \(X \in T_f\), then \(\exists\) an integer sequence \(I \in I(d)\) and \(\exists\) a data-structure \(d \in D\) such that \(o_{\downarrow}(d) = X\).

**Proof:** If \(X \in T_f\) then obviously there exists some \(c\) (at least 1) such that \(X \in T_c\). The lemma is proven by induction on the sets \(T_c\). Assuming there is a length \(k-1\) sequence \(I_Y\) for each data-structure \(Y \in T_{k-1}\) and \(d \in D\) such that \(o_{\downarrow}(d) = Y\). Then it follows, by definition of \(T_k\) that if \(X \in T_k\) then \(X = o_{\downarrow}(Y)\) for some \(i \leq m(Y)\), and \(Y \in T_{k-1}\). Thus \(X = o_{\downarrow}(o_{\downarrow}(d)) = o_{<i>}(d)\). Since also \(D = T_1\), and \(o_{\downarrow}(d) = d\) for each \(d \in D\), the proof is complete.

**Lemma 2.2:** If \(f \in F\) and \(X \in T_f\), then \(I(X)\) is finite.

**Proof:** From the previous lemma the data-structure \(X = o_{\downarrow}(d)\) for some \(d \in D\) and integer sequence \(I\). Therefore \(I(d) \geq \text{the}\)

* \(F\) as defined here is the same as \(F(V)\) as defined in the Introduction.
set consisting of $I$ concatenated with each member of $I(X)$. Thus if $I(X)$ is not finite, $I(d)$ cannot be finite but this contradicts the condition that $f \in F$ is substitutionally solvable.

Another consequence of the definition of $F$ is that the data-structures in $\Delta_f$ can be usefully ordered in another, almost reverse, manner than the ordering by membership in the subsets $\Delta^j$. In most of the subsequent inductive proofs, induction will be carried out on this ordering.

Ordering the Data-Structures in $\Delta$ (Remoteness):

For any function $f$ in $F$:

We say a data-structure $X$ in $\Delta_f \cup Q_f$ is of remoteness 0 (or is terminal) if $X \in Q_f$.

We say a data-structure $X$ in $\Delta_f \cup Q_f$ is of remoteness $n$ if:

1. $\exists i : i \leq m(X)$ and $o_i(X)$ is of remoteness $n-1$ and
2. $\forall i : i \leq m(X)$ implies $o_i(X)$ is of remoteness $n-k$ and $k \geq 1$.

Lemma 2.3: If $f \in F$, then there is a function $r$ with domain $\Delta_f \cup Q_f$ such that if $X \in \Delta_f \cup Q_f$ then $r(X) = \text{the remoteness of } X$.

Proof: For each $X \in \Delta_f \cup Q_f$ let $r(X)$ be the maximum of the length of all the sequences in $I(X)$. For each $X \in \Delta_f \cup Q_f$, $X$ is of

* Alternately this can be phrased 'of remoteness < $n$.}
remoteness \( r(X) \). This is shown by induction. If \( T(X) \) then \( I(X) \) is empty and \( r(X) = 0 \). Assume that if \( r(X) < n \), \( X \) is of remoteness \( r(X) \). Let \( r(X) = n \), i.e. there is a longest sequence of length \( n \), say \( I = \langle i_1, \ldots, i_n \rangle \) in \( I(X) \). Let \( o_{i_1}(X) = Y \). Then \( I' = \langle i_2, \ldots, i_n \rangle \) is in \( I(Y) \). Furthermore, no sequence applicable to \( Y \) is longer than \( I' \) because otherwise \( I \) could not have been a longest sequence in \( I(X) \). So \( r(Y) = n-1 \) and \( Y \) is of remoteness \( r(Y) = n-1 \). Therefore, since \( o_{i_1}(X) = Y \) and for all \( j \neq i_1 \), \( j \leq m(X) \), \( r(o_{j}(X)) \leq n-1 \), \( X \) is of remoteness \( r(X) = n \) by definition of remoteness.

**Interpretation of the Recursive definitions in \( F \)**

In the next paragraphs we briefly sketch some important well known facts about the interpretation of a terminating recursive definitions such as \( f \in F \).

A recursive definition \( f \in F \) defines the function \( f \) on the domain \( D_f \) by giving a relation (in terms of the primitive function \( w \)) that \( f(d) \) must satisfy with the same function \( f \) at some different argument value namely with \( f(o_{i_1}(d))'s \) for \( 1 \leq i \leq m(d) \). The same definition is applicable to define \( f(o_{i_1}(d)) \) for each \( 1 \leq i \leq m(d) \), of \( f \) at still other arguments. This process of repeated redefinition of \( f \) with different arguments will eventually end (because \( f \in F \) is substitutionally solvable) with arguments \( X \) which are terminal, at which point the definition of \( f \) gives a definite value = \( q(X) \) to be assigned to \( f(X) \). Thus this process will close, and a definite value will be assigned to \( f(d) \). It can easily be shown that this is a unique value.
This process of re-definition can be formulated as a non-deterministic procedure involving successive substitutions in an expression whose evaluation will give the value of \( f(d) \) for \( d \in D_f \).

Let \( E_i \) be an expression involving a composition of the \( w \), and \( o_i \) functions, \( a \in D, \) and occurrences of \( f(\alpha) \), \( f \) being the symbol for the defined function, and \( \alpha \) its argument. Let \( E_1 = f(d) \) and in general to get \( E_{i+1} \) from \( E_i \) do the following:

Choose any occurrence of \( f(\alpha) \) in \( E_i \). Note that \( \alpha \) itself will never contain any occurrence of \( f \), \( \alpha \) will just involve a composition of \( w \)'s, \( o_i \)’s and \( d \). Evaluate \( \alpha \), this can be done because it only involves given primitive functions and a given data-structure \( d \).

If the evaluated \( \alpha \) is terminal, i.e. \( T(\alpha) \) is true, then \( f(\alpha) \) is replaced by \( q(\alpha) \) to obtain \( E_{i+1} \). Thus the right side of the 1st equation of the definition of \( f \) is substituted for \( f(\alpha) \). If on the other hand \( T(\alpha) \) then the value of \( \alpha \) is substituted for \( X \) on the right side of the second equation in the definition of \( f \), and then this entire resulting right side is substituted for \( f(\alpha) \) in \( E_i \) to produce \( E_{i+1} \).

Substitutions are continued until \( E_i \) contains no occurrences of \( f \). This must eventually occur because \( f \) is substitutionally solvable. At this point in the evaluation \( E_i \) is the value of \( f(d) \).

The result of this non-deterministic procedure starting with \( f(d) \) may be interpreted as the definition of \( f(d) \).

This definition is non-deterministic because any occurrence of \( f(\alpha) \) in \( E_i \) may be legitimately chosen to be substituted for next. No order is prescribed.
Although the meaning of the recursive definition is tied to this non-deterministic unordered procedure, the common connotation of 'recursive implementation' involves a fixed ordering of the substitutions for occurrences of \( f(a) \) in the successive expressions \( E^i \). This order requires substitution always for the leftmost occurrence of \( f(a) \) in \( E^i \). This is the order implemented in virtually all compilers which accept recursive definitions. It is sometimes called depth-first ordering. This ordering amongst others will be investigated here. We call the depth-first ordering the standard implementation—recognizing that strictly there is not a single implementation entitled to be called the recursive implementation.

So given a recursive definition—and the order in which the \( f(a) \) occurrences are to be substituted for—the basis of a deterministic implementation is established. This can be detailed in a flowchart and is one of the ways in which we will specify such an implementation.

There is a subset of recursive definitions of \( f \in F \), however, in which one need not specify the order explicitly. There can be no question of the order of substitution for occurrences of \( f(a) \) when each expression \( E^i \) has only one occurrence of \( f(a) \). This will occur for any definition of \( f \in F \) whose second equation has only one occurrence of \( f(a) \) on the right. Such a unary recursive definition itself then is a second way to specify an implementation.

An implementation which can be specified in one of these forms can also be specified in the other form.

In the subsequent sections both ways of specifying an implementation are used.
In the next section unary definitions are developed which represent the classical depth-first and breadth-first implementations of functions $f \in F$. By showing the equivalence of these unary definitions to $f \in F$ the validity of these implementations is established.

Later another implementation for $f : F$ called an inverse implementation will be described by a flowchart and will be proven to be valid. This inverse implementation, unlike the classical implementations, does not require a stack.

Classical Implementations:

Notation:

We need notation for operations which replace a component of a vector with single or multiple components which are functions of the replaced component.

Let $L$ be a vector (list); $L = \langle l_1, \ldots, l_n \rangle$.

Let $t_i$ denote an individual member or subsequence of $L$ which has some specified properties $P_i$. The notation:

$L[t_1 + x_1; \ldots; t_n + x_n]$ denotes the list obtained by replacing all components of $L$ which have property $P_i$ by $x_i$. $x_i$ may be an individual component or a number of components. For example, if $P_1$ is the property of being an odd indexed component of $L$ and if $n$ (the number of components of $L$) is even then the meaning of $L[t_1 + a]$ is given by:

$L[t_1 + a] = \langle a, l_2, a, l_4, a, \ldots, l_n \rangle$

The next two theorems will show that for each $f \in F$ there are two unary definitions $F_B$ and $F_D$ both also members of $F$ and both equivalent to $f$. Since they are unary the implementation...
for evaluating these definitions is deterministic (see pg 19). The
algorithm \( F_B \) is similar to the classical 'breadth-first' algorithm and
\( F_D \) is similar to the classical 'depth-first' algorithm.

**Breadth-First Implementation**

Let \( f \in F \), thus

\[
\begin{align*}
1) & \quad f(X) = q(X) \quad \text{if } T(X) \\
2) & \quad f(X) = w(f(o_1(X)), \ldots, f(o_m(X)(X))) \quad \text{if } \overline{T}(X) \\
3) & \quad \text{initially } X \in D
\end{align*}
\]

To define the function \( F_B \) which is equivalent to \( f \) we first need to de-
fine a number of new primitive functions and predicates in terms of the
primitive functions \( q, w, q, m \) and predicate \( T \) of \( f \). For this the notation
just introduced is used.

\[
L = \langle l_1, \ldots, l_n \rangle, \quad Z = \langle z_1, \ldots, z_p \rangle
\]
are both vectors. The components
of \( L \) are either brackets in the set
\[
\text{BRACK} = \{', '], '<', '>', '\}
\]
or members of \( \Delta_f \) the domain of \( f \), or of \( Q_f \) the terminal data-structures
of \( f \). (\( \Delta_f \) and \( Q_f \) are assumed not to contain any of the brackets in \( \text{BRACK} \).)

The components of \( Z \) are either members of \( \text{BRACK} \) or of \( w_f = \) the range of \( q \),
or of \( w_f = \) the range of \( w \).

With \( X \in \Delta_f, \ t \in Q_f, \) and \( t_i \in w_f \cup w_f \) let:

\[
\begin{align*}
Q_B(L) & = L[X + '\{', o_1(X), \ldots, o_m(X)(X), ']' ] \\
T_B(L) & = \text{true if every component of } L \text{ is a member of } \text{BRACK} \text{ or of } Q_f \\
Q_B(L) & = L[t + q(t)] \\
W_B(L) & = 1 \\
W_B(Z) & = Z['{', t_1, \ldots, t_n, '}'] + w(t_1, \ldots, t_n)
\end{align*}
\]
Now we can define $F_B$:

1. $F_B(L) = Q_B(L)$ if $T_B(L)$
2. $F_B(L) = W_B(F_B(Q_B(L)))$ if $T_B(L)$
3. Initially $L = <X>$, $X \in D$

Theorem 2.1: For each $f \in F$ (as I above) there is a function $F_B \in F$ (as II above) such that $F_B(<X>) = \langle f(X) \rangle$ for all $X \in D$.

Proof: The proof uses induction on the remoteness of the data-structures $X \in \Delta_f \cup Q_f$ which, along with brackets, constitute the significant components of the vector data-structures $L \in \Delta_{F_B}$. What we will show is first that for any $L$ in $\Delta_{F_B}$ whose components are all members of $Q_f$, designated by $t$, or are in BRACK, that:

$$F_B(L) = L[t + f(t)]$$

This is true since if $T_B(L)$ is true then with $t \in Q_f$

$$F_B(L) = Q_B(L) \quad \text{by II 1}$$

$$= L[t + q(t)] \quad \text{by definition of } Q_B$$

$$= L[t + f(t)] \quad \text{by I 1}$$

Secondly we show inductively that if $L$ contains component of $\Delta_f$ designated $X$ then

$$(H) \quad F_B(L) = L[X + f(X)]$$

Assume that as long as all components of $L$, other than those in BRACK, are of remoteness $< n$ that statement $(H)$ is true. If $T_B(L)$ is not true and all components of $L$ other than those in BRACK are of remoteness $\leq n$, $n > 0$, and at least one such component is of remoteness $n$, then: ($X$ is used to designate a member of $\Delta_f$, $t$ a member of $Q_f$)
\[
F_B(L) = W_B(F_B(O_B(L))) \text{ by II(1)}
\]
\[
F_B(L) = W_B(F_B(L[X + \{'o_1(X), \ldots , o_m(X))(X), \}'])
\text{ by the definition of } O_B
\]
\[
F_B(L) = W_B(F_B(L')) \text{ abbreviating the expression above with } L'
\]

Clearly all components in \( L' \) are of remoteness \(< n \) and at least one has remoteness \( n-1 \).

So the inductive hypotheses may be used for all \( X \in A_x \) in \( L' \).

\[
F_B(L) = W_B(L'[X + f(X)])
\]

But \( L' = L[X + \{'o_1(X), \ldots , o_m(X))(X), \}'] \)

So \( L'[X + f(X)] = L[X + \{'o_1(X), \ldots , f(o_m(X))(X), \}'] \)

So \( W_B(L'[X + f(X)]) = W_B(L[X + w(f(o_1(X)), \ldots , f(o_m(X))(X))]) \)

by the definition of \( W_B \).

Thus:

\[
F_B(L) = L[X + f(X)]
\]

Thus for \( L = <X>, X \in D : \)

\[
F_B(L) = <X>[X + f(X)] = f(X)
\]

Depth-First Implementation

The depth-first function \( F_D \) equivalent to \( f \) in I above is defined as follows:

1) \( F_D(L, k) = \bar{O}_D(L, k) \) if \( \bar{T}_D(L) \)

2) \( F_D(L, k) = F_D(O_D(L, k)) \) if \( \bar{T}_D(L) \)

3) initially \( L = <X>, X \in D, \) \( k = 1 \) (we say then \( <L, k> \in D_D \)

where with \( l_k \) the \( k \)th component of \( L \) the definition of \( O_D, T_D \) and \( Q_D \) are:

1) \( O_D(L, k) = <L[l_k + \{'o_1(l_k), \ldots , o_m(l_k)(l_k)\}'], k> \) if \( l_k \in A_x \)

2) \( = <L[l_k + q(l_k)], k+1> \) if \( l_k \in Q_x \)

3) \( = <L[\{'t_1, \ldots , t_n\}' + w(t_1, \ldots , t_n)], k-n-1> \)

.. if \( l_k = ')' \) and \( t_1, t_2, \ldots , t_n \) are all in \( W_f \cup \bar{W}_f \)

(and \( ')' \), \( t_1, \ldots , t_n \) precedes \( \) in \( L \) as assumed)
Theorem 2.2 If $f$ (of definition I above) $\in F$, then $F_D$ (of definition III above) is also $\in F$ and for $d \in D$, $F_D(<d>,1) = f(d)$.

Proof: Again the proof will be by induction on remoteness.

It will be shown that if $l_k$ is the $k^{th}$ component of $L$ and $l_k \in \Delta_f \cup Q_f$ then:

(H) $F_D(L,k) = F_D(L[l_k + f(l_k)],k)$

This is certainly true if $l_k$ is of remoteness 0, i.e. if $l_k \in Q_f$ because then:

$$F_D(L,k) = F_D(L[l_k + q(l_k)],k+1)$$

by definition of $O_D(2)$ and III 1

$$= F_D(L[l_k + f(l_k)],k+1)$$

by definition of $f$, I 1

$$= F_D(L[l_k + f(l_k)],k)$$

by definition of $O_D(4)$

Assume (H) is true if $l_k$ has remoteness $< n \geq 0$. Then if $l_k$ is of remoteness $n > 0$, and is $\in \Delta_f$, it follows from $O_D(1)$ that

$$F_D(L,k) = F_D(L[l_k + \{'$\cdot$$'$, $o_1(l_k)$, $\ldots$, $o_m(l_k)(l_k), \ldots$'$],k)$$

with $o_i(l_k)$ for $1 \leq i \leq m(l_k)$ each being of remoteness $< n$. 

(4) $O_D(L,k) = <L,k+1>$

if $l_k = \sim \{\}$ or if $l_k \in W_f \cup W_f$

$T_D(L,k) =$ true if $|L| = 1$ and $k = 2$.

$Q_D(L,k) = 1$.
Rewriting the above in expanded notation and indicating $l_k$ (the $k$th component of $L$) by underline:

$$F_D(L, k) = F_D(1_1, \ldots, 1_{k-1}, ''t', o_1(1_k), \ldots, o_m(1_k) (1_k), 't'' \rangle / L', k)$$

Then by definition $O_D(4)$:

$$F_D(L, k) = F_D(<1_1, \ldots, 1_{k-1}, ''t', o_1(1_k), \ldots, o_m(1_k) (1_k), 't'' \rangle / L', k+1)$$

Since $o_1(1_k) \in \Delta^c \cup Q_f$ and is of remoteness $< n$ we have by (H), and then by an application of $O_D(4)$ again:

$$F_D(L, k) = F_D(1_1, \ldots, 1_{k-1}, ''t', f(o_1(1_k)), f(o_2(1_k)), \ldots, o_m(1_k) (1_k), 't'' \rangle / L', k+2)$$

And since in fact for $i = 1$ to $m(1_k)$, $o_i(1_k) \in \Delta^c \cup Q_f$, and is of remoteness $\leq n$ by repeated application of (H) and $O_D(4)$:

$$F_D(L, k) = F_D(1_1, \ldots, 1_{k-1}, ''t', f(o_1(1_k)), \ldots, f(o_m(1_k)), 't'' \rangle / L', k+m(1_k)+1)$$

And finally by $O_D(3)$:

$$F_D(L, k) = F_D(1_1, \ldots, 1_{k-1}, w(f(o_1(1_k)), \ldots, f(o_m(1_k) (1_k))) / L', k)$$

Since $f(1_k) = w(f(o_1(1_k)), \ldots, f(o_m(1_k) (1_k)))$

$$F_D(L, k) = F_D(1_1, \ldots, 1_{k-1}, f(1_k) / L', k)$$

And compacting the notation:

$$F_D(L, k) = F_D(L' [1_k + f(1_k)], k)$$
For \( L = \langle d \rangle \), \( k = 1 \):

\[
F_D(\langle d \rangle, 1) = F_D(\langle d \rangle[d + f(d)], 1)
\]

\[
= F_D(\langle f(d) \rangle, 2) \quad \text{since } f(d) \text{ is in the domain of } w, \quad O_D(4)
\]

is applicable

\[
= F_D(\langle f(d) \rangle, 2) \quad \langle \langle f(d) \rangle, 2 \rangle \text{ is terminal so}
\]

\[
= f(d')
\]

Properties of \( f \in F \). Sufficient for Memory Efficient Implementations

Another implementation more efficient that the two classical ones is available when the recursive definition \( f \in F \) has some special properties. These properties are now defined.

**Associativity:** Associativity has the usual meaning here. The function \( w \) is associative if:

\[
w(a_1, a_2, \ldots, a_m) = w(w(a_1, a_2), a_3, \ldots, a_m) \quad \text{for } m \geq 3
\]

\( w = \) minimum, sum, catenation and union provide examples of \( w \)-functions with this property. In each case one can compute \( w(a_1, \ldots, a_m) \) as follows:

\[
\begin{align*}
X & \leftarrow X \\
\text{For } i = 1 \text{ to } m \\
Y & \leftarrow w(X, a_i) \\
X & \leftarrow Y
\end{align*}
\]

thus requiring at any one time memory for at most 2 copies of the result of \( w(a_1, \ldots, a_j), j \leq m \). If \( w \) is the function minimum, this memory does not increase on the number, but only on the value of its arguments, \( a_i \). If \( w \) is catenation, sum, or union the memory required will increase, albeit at different rates, with the number of arguments. There is, however, a significant
difference in use of the memory, between a computation of catenation and of union. To obtain catenate \((a, b)\), \(b\) needs only be attached at the end of \(a\). To obtain the union \((a, b)\), \(a\) must be searched for an occurrence of a member of \(b\). If \(a\) represents the result of a previous computation then in the union case it is necessary to re-access this memory whereas this is not necessary in the catenation case. This is an important consideration because memory that is not re-accessed can be located in areas of memory (disc) which need not be easy to access (as is core). The temporary memory requirements for the implementation of a function then do not depend on the usual mathematical properties of that function only, but also depend on the means available for accessing the memory. Nevertheless, for compactness our results are given in terms of the usual mathematical properties--so caution is needed in their interpretation.

Uniform Inverse:

Consider a set of functions \(H = \{h_1, \ldots, h_M\}\). Let \(D_i\) be the domain over which \(h_i\) is defined and let \(R_i\) be the corresponding range of \(h_i\). Then we will say \(D = \bigcup_i D_i\) is the domain of \(H\) and \(R = \bigcup_i R_i\) is its range.

The set of functions \(H\) is said to have a uniform inverse on the domain \(D\) if:

1. Every \(h \in H\) has an inverse and
2. \(R_i R_j = \emptyset\) for every \(R_i \neq R_j\) in \(R\).

If \(H\) has an uniform inverse then it is easy to see that the following two 'uniform inverse' functions on \(R\) exist for \(r \in R_i \subset R\):

---

1. It is also true that there may be some advantage in time efficiency in one grouping of the arguments of \(w\) over another though both give the same result when \(w\) is associative. An example of such a function is merge, i.e. \(\text{merge}(a_1, \ldots, a_M)\) in which \(a_i\) are each finite sorted sets of numbers.
(1) \( H^{-1}(r) = d \in D \) such that \( H_i(d) = r \)

(2) \( i_H(r) = i \) the index of the range \( R_i \) of which \( r \) is a member.

A recursive definition, \( f \in F \), has a uniform inverse if the set of functions \( O_i \in O \) in \( f \) has a uniform inverse.

For a given function set \( O \) it is possible that none, one, or two of the pair \( H^{-1}, i_H \) exist. Despite the fact that the uniform inverse is a strong condition it does often occur. Furthermore when it doesn't, there is always a strongly equivalent definition which does have a uniform inverse. This is shown after a short degression required to define strong equivalence.

**Equivalence of Recursive Definitions:**

Consider two definitions in \( F \):

1. \( f \) on domain \( D \)

\[
\begin{cases}
  f(X) = q(X) & \text{if } T(X) \\
  f(X) = w(f(o_1(X)), ..., f(o_m(X))) & \text{if } \overline{T}(X) \\
  \text{initially } X = d \in D
\end{cases}
\]

2. \( g \) on domain \( D' \)

\[
\begin{cases}
  g(X') = q'(X') & \text{if } T'(X') \\
  g(X') = w'(g(o_1'(X')), ..., g(o_m'(X')))) & \text{if } \overline{T}'(X') \\
  \text{initially } X' = d' \in D'
\end{cases}
\]

If there is a 1-1 correspondence between \( D \) and \( D' \) such that whenever
d \in D \text{ and } d' \in D \text{ are two corresponding data-structures } f(d) = g(d')
then the two definitions are equivalent. The above correspondence may be
extended to one between \( \Delta_f \) and \( \Delta_g \) with \( \delta \in \Delta_f \) corresponding to \( \delta' \in \Delta_g \)
by having \( o_{i}(\delta) \) correspond to \( o'_{i}(\delta') \) whenever \( \delta \) corresponds to \( \delta' \) and
\( o_{i}(\delta) \) \text{ and } \( o'_{i}(\delta') \) \text{ are both defined. This is called a structural correspondence. If in addition to such a structural correspondence of } \Delta_f \text{ to } \Delta_g \text{ the following conditions hold}

(1) \( T(\delta) = T'(\delta') \)
(2) \( q(\delta) = q'(\delta') \text{ if } T(\delta) \text{ (and } T'(\delta') \text{)} \)
(3) \( m(\delta) = m'(\delta') \text{ if } \overline{T}(\delta) \text{ (and } \overline{T}'(\delta') \text{)} \)
(4) \( w = w' \)

then \( f \) \text{ and } \( g \) are strongly equivalent. (Note that the structural correspondence of } \Delta_f \text{ to } \Delta_g \text{ need not be 1-1. It will not even necessarily be defined on all members of } \Delta_f \text{ and } \Delta_g \text{ unless the conditions (1) through (4) are satisfied.}

Strong equivalence of two definitions implies that they not only give the same results but also require the same number of substitutions in their evaluation for corresponding initial arguments.

As an example of a strong equivalence, consider the two functions \( f \) and \( g \) each in \( F \):

(1) \[
\begin{cases}
    f(X) = q(X) & \text{if } T(X) \\
    f(X) = w(f(o_1(X)), \ldots, f(o_m(X))(X))) & \text{if } \overline{T}(X) \\
\end{cases}
\text{ initially } X = d \in D
\]

(2) \[
\begin{cases}
(a) \quad g(X, Y) = q(X) & \text{if } T(X) \\
(b) \quad g(X, Y) = w(g(o_1(X), h_1(Y)), \ldots, g(o_m(X), h_m(X)(Y))) & \text{if } \overline{T}(X) \\
\end{cases}
\text{ initially } <X, Y> = <d, y_d> \in D' \text{ with } d \in D \text{ and } y_d \in \text{ a set } \mathcal{Y}_d \\
(H = \{h_1, \ldots, h_N\} \text{ is a set of primitive functions})
Let data-structures \( d \in D \) correspond to \( <d, y_d> \in D' \). Extend this correspondence to one between \( \Delta_f \) and \( \Delta_g \) by letting \( o_i(X) \in \Delta_f \) correspond to \( <o_i(X), h_i(Y)> \in \Delta_g \) whenever \( X \in \Delta_f \) corresponds to \( <X, Y> \in \Delta_g \) and \( i \leq m(X) \). For example if \( d \in D \) and \( o_i(d) \) is defined then it corresponds to \( <o_i(d), h_i(y_d)> \in \Delta_g \).

For each member of \( \Delta_f \) this correspondence defines a corresponding member of \( \Delta_g \). This follows because every member in \( \Delta_f \) is either in \( D \), for which the correspondence is given explicitly, or it is \( o_i(X) \) for \( X \in \Delta_f \) and \( o_i \) is defined and \( T(X) \), in which case the correspondence to a member of \( \Delta_g \) is given since \( o'_i(X,Y) \)'s existence just depends on \( X \), because \( m'(X,Y) = m(X) \), \( T'(X) = T(X) \).

Conditions (1) through (4) are obviously satisfied for this correspondence in the above definitions. Furthermore, the function \( g(X,Y) \) is independent of \( Y \), its second argument. This is shown inductively as follows. Directly from the definition (2a) we see that \( g(X,Y) \) is independent of \( Y \) when \( (X,Y) \) is of remoteness \( 0 \). Its being of remoteness \( 0 \) is also independent of \( Y \). Referring to (2b), if it is assumed that each term \( g(o_i(X), h_i(Y)) \) appearing on the right is independent of its second argument then it follows certainly that \( g(X,Y) \) on the left of (2b) is independent of \( Y \). If the argument on the left side of (2b) is of remoteness \( n \) from terminal then all the arguments of terms on the right are of remoteness \( < n \) from terminal. Thus the inductive argument is completed concluding that \( g(X,Y) \) is independent of \( Y \) if \( X \) and thus if \( (X,Y) \) is of remoteness \( 0,1,2, \ldots, n \).

Thus definition (2) can be rewritten removing \( Y \) which with \( f \) replacing \( g \) is the same as (1). Therefore
Lemma 2.5  \( g \) and \( f \) above are strongly equivalent

Since the value of \( g(X,Y) \) is independent of \( Y \) it may seem silly to ever construct such a definition, with a 'redundant' \( Y \), to replace \( f \), or alternatively that such a redundant \( Y \) would arise inadvertently in \( g \) to be removed by replacement with the equivalent \( f \). The following theorem, however, demonstrates that such 'redundant' additions can be of considerable use.

Theorem 2.3: For any recursive definitions \( f \) in \( F \) there is a strongly equivalent definition in \( F \) which has a uniform inverse.

Proof: If \( f \) already has a uniform inverse it serves as its own strongly equivalent definition. If not the following definition serves that purpose. Referring to Def. 2.1 for the definition of \( f \), the following function \( g \) defined in terms of the same sets, primitives and predicates is strongly equivalent to \( f \). (\( \rho = <p_1, \ldots, p_n> \) is a vector which records indices, and \( d \) is the initial data structure.)

\[
\begin{align*}
g(X, \rho, d) &= q(X) & \text{if } T(X) \\
g(X, \rho, d) &= w(o_1(X), <1> \parallel \rho, d), \ldots, g(o_m(X), <m(X)> \parallel b, d) & \text{if } \overline{f}(X) \\
\text{initially } <X, \rho, d> &= <d, \lambda, \lambda, d> \text{ with } d \in D.
\end{align*}
\]

\( g \) is strongly equivalent to \( f \) by application of lemma 2.5, with \( Y \) of that lemma corresponding to \( \{<p,d> | p \text{ a sequence of integers, } d \in D\} \), and \( y_d \) corresponding to \( <n,d> \) with \( d \in D \). Furthermore \( g \) has a uniform inverse which is given by the following:
The $0^{-1}$ function is quite complex, requiring recreating a sequence of data-structures starting with the initial data structure. In practice one wants to construct a strongly equivalent definition which gives an inverse but entails the creation of an $0^{-1}$ function which is simple. Simpler, hopefully, than that given in the above theorem. This can often be done. If, for example,

Corollary 2.1: For a given recursive definition $f \in F$ there is no uniform inverse, but each function $o_i \in \varnothing$ has an inverse $= o_i^{-1}$, then the definition for $g$ given above with the third component $d$ deleted from its arguments will serve with the additional benefit that an alternative simpler definition of $0^{-1}(X,o) = <o_i^{-1}(X), o[o_1+n]>$ can be used.

This corollary can be applied to the 'Tower of Hanoi' definition ex.1.3. In that example, $o_i \in \varnothing$ has an inverse for $i = 1$ and 3 but does not quite have an inverse when $i = 2$: 

$$o_2^{-1}(<x,y,z>,n) = <x,y,z>,n+1>$$

$$o_3^{-1}(<x,y,z>,n) = <z,y,z>,n+1>$$

So first we slightly modify the definition of $f$ so there will be an inverse for $o_2$, Lemma 2.5 justifies this simple modification in which a component $s$ is added to store the quantity $A$ above when $i = 2$, and otherwise to remain equal to 0.
\[
\begin{cases}
    f'(<<x, y, z>, n>, s) = <x, y> & \text{if } n = 1 \\
    f'(<<x, y, z>, n>, s) = f'(<<x, z, y>, n-1>, s) \parallel f'(<<x, y, z>, 1>, n) \parallel f'(<<z, y, x>, n-1>, s) & \text{if } n > 1 \\
\end{cases}
\]

initially \(<<x, y, z>, n>, s> = <<1, 2, 3>, n>, 0>, n \in N\)

Now \(f\) is equivalent to \(f\) in 1.3 and \(o_i\) has an inverse for \(i = 1, 2,\) or 3.

These inverses are:

\[
\begin{align*}
    o_1^{-1}(<x, y, z>, n>, s) &= <<x, z, y>, n+1>, 0> \\
    o_2^{-1}(<x, y, z>, n>, s) &= <<x, y, z>, s>, 0> \\
    o_3^{-1}(<x, y, z>, n>, s) &= <<z, y, x>, n+1>, 0>
\end{align*}
\]

Corollary 2 now applies to \(f'\). Its application yields \(g\) below.

(Some unnecessary >'s and <'s have been dropped.)

\[
\begin{cases}
    g(<x, y, z>, n, s, \rho) = <x, y> & \text{if } n = 1 \\
    g(<x, y, z>, n, s, \rho) = g(<z, y, x>, n-1, s, \rho) \parallel g(<x, y, z>, 1, n, s\rho) \parallel g(<x, y, z>, n, s\rho) \parallel g(<x, y, z>, n, s\rho) & \text{if } n > 1 \\
\end{cases}
\]

initially \(<x, y, z>, n, s, \rho> = <<123>, n, 0, \lambda>\)

and the uniform inverse is given by

\[
\begin{align*}
    i_0(<x, y, z>, n, s, \rho) &= \rho_1 \\
    o_1^{-1}(<x, y, z>, n, s, \rho) &= o_1^{-1}(<x, y, z>, n, s), \rho[\rho_1^{-1} \lambda> \]
\]
Implementation of $f \in F$ with Associativity and Uniform Inverse

We will first give a way of implementing any definition $f$ in $F$ which has a uniform inverse and is otherwise unrestricted. Then we will give a way of implementing any $f$ in $F$ which has a uniform inverse and in which $w$ is associative. This is done to contrast the means necessary for implementation in these two cases. In both cases the implementation is described by a flowchart containing, as usual, interconnected assignments and decision statements. In both cases the expressions in the assignment statements and decisions are compositions involving the primitive functions and predicates $w$, $o$, $m$, $q$ and $T$ and the inverses $0^{-1}$, $i_0$ which enter the definition of $f \in F$. In both cases, in addition to the above functions from the definition of $f$, the repertoire of flow chart expression is completed by an add 1 function, a push and pop and an $=$ predicate.

In both cases there is a storage cell $X$ which is assumed adequate to hold any member in $A_f \cup Q_f \cup D_f$.

In the case that $f$ has both a uniform inverse and an associative $w$ there is also a storage list $V$ which can hold at most any two members in $W_f \cup W_f$. In the case that $f$ only has a uniform inverse the list $V$ is still necessary but it cannot be bounded in size. It may be required to hold any number of members in $W_f \cup W_f$. The size actually used will be dependent on the specific function $f \in F$ realized as well as the initial data-structure. In this case an auxiliary storage ARG is also used. It holds at most a number of members in $W_f \cup W_f$ to the largest value in the range of $m$.

1. See fig. 2.1
2. See fig. 2.2
Flowcharts 1 and 2 which follow describe a computation for each \( d \in D \).

It is necessary to give a concrete interpretation of the sense in which a flowchart describes a computation. We imagine a traveler who starts by entering block (0) of the flowchart. The traveler carries out the computation described in that block then, depending on the nature of the block, proceeds to the appropriate next block. The traveler continues following the block instructions and proceeding through the flowchart until FINI is reached completing the voyage. The value found in \( V \) when the traveler has completed the voyage is the value computed by the flowchart.

Flowcharts: notation and assumptions

In these flowcharts we will use the following notation. General:

\((e \text{ is an expression})\)

\[
\begin{align*}
X \leftarrow e & \quad \text{the value of } e \text{ is assigned to } X \\
V \leftarrow \text{PUSH } e & \quad \text{the value of } e \text{ is pushed into list } V \\
X \leftarrow \text{POP } V & \quad \text{the top member of } V \text{ is popped and assigned to } X \\
X \leftarrow \text{POP } V[n] & \quad \text{the top } n \text{ members of } V \text{ are popped and assigned to } X \\
\end{align*}
\]

If \( V \) is a list \( = <v_1,v_2, \ldots, v_n> \) then \( w(V) \) stands for the expression \( w(v_1,v_2, \ldots, v_n) \).
Primitives and their Compositions: (Some of the definitions are extended to \( Q_f \) to make the flowcharts work if the initial data-structure is terminal.)

<table>
<thead>
<tr>
<th>Flow chart Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{FIRST.KID}(X) )</td>
<td>( o_1(X) ) if ( X \in \Delta_f )</td>
</tr>
<tr>
<td>( #\text{KIDS}(X) )</td>
<td>( m(X) ) if ( X \in \Delta_f ); ( \equiv 1 ) if ( X \in Q_f )</td>
</tr>
<tr>
<td>( X = \text{TERMINAL?} )</td>
<td>( T(X) ) if ( X \in \Delta_f \cup Q_f )</td>
</tr>
<tr>
<td>( \text{PARENT}(X) )</td>
<td>( o^{-1}(X) ) if ( X \in \Delta_f ); ( \equiv X ) if ( X \in Q_f )</td>
</tr>
<tr>
<td>( #\text{SIB}(X) )</td>
<td>( i_0(X) ) if ( X \in \Delta_f ); ( \equiv 1 ) if ( X \in Q_f )</td>
</tr>
<tr>
<td>( \text{NEXT.SIB}(X) )</td>
<td>( o_{#\text{SIB}(X)}(\text{PARENT}(X)) ) if ( X \in \Delta_f )</td>
</tr>
<tr>
<td>( #\text{SIBS}(X) )</td>
<td>( #\text{KIDS}(\text{PARENT}(X)) ) if ( X \in \Delta_f ); ( \equiv 1 ) if ( X \in Q_f )</td>
</tr>
</tbody>
</table>

If \( w \) is associative we assume that there is a member \( q_w \) in the range of \( w \) such that \( w(X, q_w) = X \) for all \( X \) in the range of \( w \). This definition is used in the second flowchart following.
Flowchart 1:
For \( f \in P \) and \( f \) has a \underline{uniform} inverse.

1. \( X \leftarrow \text{TERMINAL?} \)
   
2. \( V \leftarrow \text{PUSH} \, q(X) \)
   
3. \( X \leftarrow \text{FIRST.KID}(X) \)
   
4. \( \text{SIB}(X) = \text{#SIBS}(X) ? \)
   
5. \( X \leftarrow \text{NEXT.SIB}(X) \)
   
6. \( \text{ARG} \leftarrow \text{POP} \, V[\text{#SIBS}(X)] \)
   
7. \( V \leftarrow \text{PUSH} \, \omega(\text{ARG}) \)
   
8. \( X \leftarrow \text{PARENT}(X) \)
   
9. \( X = \text{d?} \)
Flowchart 2:

For $f \in F$ and $f$ has a uniform inverse and $w$ is associative.
When we say a flowchart implements or realizes an \( f \in F \) we mean that for each \( d \in D \) the evaluation of the function \( f(d) \) is = to the value computed by the flowchart with traveler starting at block \( 0 \) and \( d \) in the flowchart = to \( d \) in \( f(d) \).

We now present proofs that the given flowcharts, figures 2.1 and 2.2, do in fact implement \( f \in F \) under the appropriate constraints. The proofs are very similar, both using induction on the remoteness of the data-structures in \( A_f \).

**Theorem 2.4:** If \( f \in F \) and \( f \) has an uniform inverse then it is implemented by flowchart 1. (figure 2.1)

**Proof:** First we wish to show that if block \( 1 \) of flowchart (1) is entered by the traveler, with storage cell \( X \) containing data structure \( A \), and the list storage facility \( V \) containing the sequence of elements \( a \), then the traveler will eventually arrive at block \( 9 \) with the value of \( f(A) \) in \( X \), and with \( a \neq f(A) \) in \( V \).

This is done by induction on the remoteness of \( A \). It is obvious that if the remoteness of \( A \) is 1 then with traveler starting by entering block \( 1 \) of the flowchart with \( A \) in \( X \) and \( a \) in \( V \), the traveler will execute blocks \( 1, 3, 1 \) then \( 2 \) because \( o_1(A) \) must be of remoteness 0, then \( 4 \), and if \( 1 = \text{SIB}(A) = \#\text{SIBS}(A) \) then \( 5 \) will be next. After \( 5 \) the cycle \( 1 \ 3 \ 1 \ 2 \ 4 \ 5 \) will be repeated. Altogether, the cycle repeats \( \#\text{SIBS}(A) = p \) times. Then the traveler will proceed through \( 1 \ 3 \ 1 \ 2 \ 4 \) and this time
continue with ⑥⑦⑧ to ⑨. Tracing the change in the content of X and V during this journey; X contains $o_{i+1}(A)$ after the $i^{th}$ cycle of ①③①⑦①⑤; when the traveler reaches ⑥ X contains $o_p(A)$, but after ⑧ it contains A on arrival at ⑨. Simply tracing the blocks on travelers path shows that V will contain α//<f(A)> when traveler enters ⑨. Assume that if A is of remoteness < n, A is in X, α in V, and the traveler starts by entering ①, the traveler will eventually arrive at block ⑨ with $f(A)$ in X and α//<f(A)> in V. Consider then that A is of remoteness n, A is in X, α in V, and the traveler enters ①. Since X is not TERMINAL, the traveler goes to ③. As a result, the traveler enters ① again with $o_1(A)$ in X, and α still in V. Thus by the inductive hypothesis, the traveler will eventually enter ⑨ with $o_1(A)$ in X and α//<f(A)> in V. Next since $o_1(A)$ cannot = d or the uniform inverse would not exist, the traveler will pass through ⑨ back to ④. Assuming that 1 = SIB#(X) = #SIBS(X) where X = $o_1(A)$, then the traveler passes through ④ to ⑤ where X is made to the NEXT.SIB($o_1(A)$) or $o_2(A)$. When now the traveler re-enters ① then X is $o_2(A)$ and V is α//<f($o_1(A)$)> . Again the traveler passes through ① and by the inductive hypotheses eventually to ⑨ with X containing $o_2(A)$ and V now containing α//<f($o_1(A)$), f($o_2(A)$)>. This process continues until if p=#SIBS(A) traveler arrives at ① with X containing $o_p(A)$ and V containing:
Then by the inductive hypothesis the traveler eventually arrives at 9 with X containing \( o_p(A) \) and V containing:

\[
\langle f(o_1(A)), f(o_2(A)), \ldots, f(o_{p-1}(A)) \rangle
\]

The traveler then passes through 9 entering 4, and because \( \text{SIB}(o_p(A)) = p = \#\text{SIBS}(o_p(A)) \) the traveler will then enter 6, then 7 as a result of which V now contains:

\[
\langle f(o_1(A)), \ldots, f(o_p(A)) \rangle = \langle f(A) \rangle
\]

Then the traveler goes through 8 finally arriving at 9 with V still containing \( \langle f(A) \rangle \)

\[
\text{PARENT}(o_p(A)) = A.
\]

Thus the first point to be made is proven.

Now let the traveler start by entering block 0, thus setting V to \( \lambda \) and X to \( d \in D \). The traveler then enters 1, and by the first point made above the traveler will eventually arrive at 9 with \( V = \lambda/\langle f(d) \rangle = f(d) \) and \( X = d \). Then at 9 the test will succeed leaving the traveler at FINI with \( V = f(d) \).

The proof covers the case that \( d \) is of remoteness \( > 1 \).

For remoteness of \( d = 0 \), a direct trace of the flowchart will verify its adequacy.

**Theorem 2.5:** If \( f \in F \) and \( f \) has a uniform inverse and \( w \) in the definition of \( f \) is associative then \( f \) is implemented by flowchart 2.

(figure 2.2)
Proof: The proof is very similar to that of Theorem 2.2. The difference is in the value that will be in \( V \) when the traveler reaches \( 9 \).

First we need to show that if \( A \) of remoteness \( n \), block 1 of flowchart 2 is entered with \( A \) in \( X \) and \( B \) in \( V \) then eventually the traveler arrives at \( 9 \) with \( A \) still in \( X \) and \( V \) containing

\[
[w \ldots w[w[B, f(o_1(A))], f(o_2(A))], \ldots, f(o_m(A))]) =
\]

by associativity

\[
w[B, w[f(o_1(A)), \ldots, f(o_m(A))]] = w[B, f(A)]
\]

Again we use induction. The case when \( A \) is of remoteness 1 is easily verifies by tracing the flowchart through the sequence of blocks \(<1 3 1 2 6 4 5> m(A)-1 \) times and then through \( 1 3 1 2 6 4 7 9 \).

Assume First is correct if the remoteness of \( A \) is \(< n \). Now let \( A \) be of remoteness \( n \); \( X \) is \( A, V \) is \( B \) and the traveler is at \( 1 \). The traveler goes to \( 5 \) where \( X \) becomes FIRST.KID(X) = \( o_1(A) \) and the traveler returns to \( 1 \). Since \( o_1(A) \) is of remoteness \(< n \), the inductive hypothesis applies. Thus the traveler arrives at \( 9 \) with \( X \) being \( o_1(A) \) and \( V = w[B, f(o_1(X))] \).

\( o_1(A) \) cannot be initial because of the inverse so the traveler goes next to \( 4 \). If we assume now that

\( l = SIB#(X) \neq \#SIBS(X) \) where \( X = o_1(A) \), the traveler will pass through \( 4 \) and \( 5 \) updating \( X \) to contain \( o_2(A) \) and then enter \( 1 \). By inductive hypothesis again the traveler will eventually arrive at \( 9 \) with \( V \) containing:

\[
w[w[B, f(o_1(A))], f(o_2(A))]
\]

and \( X \) containing \( o_2(A) \). Assuming without loss in generality that \( p = m(A) \), the traveler will eventually arrive at \( 9 \).
after p repeats of the journey from 1 to 9 with V containing:

\[ w[... w[w[B,f(o_1(A))],f(o_2(A))], ..., f(o_p(A))] = w[w[B,f(o_1(A))], ..., f(o_p(A))] \]

by associativity and \( = w[B,f(A)] \) by definition of \( f(A) \).

\( X \) contains \( o_p(A) \) at this time. So the traveler goes to 4 where the decision is yes; 7 is next with \( X \) becoming its PARENT, i.e. PARENT(\( o_p(A) \)) = A. Thus the traveler arrives at 9 again with \( X \) containing A, V still containing \( w[B,f(A)] \). Thus the First result is proven. Now let the traveler start by entering 9 thus setting \( X \) to d and V to \( B = O_w \). Next the traveler enters 1 with these values in \( X \) and V and so by the First result the traveler will eventually arrive at 9 with \( X \) containing d and V containing \( w[O_w,f(d)] = f(d) \) by definition of \( O_w \).

As before the proof is for \( d \in D \) having remoteness \( \geq 1 \) and is verified to include remoteness 0 by tracing flowchart 2 explicitly for this case.

The necessity for a 'uniform inverse' as opposed to a simple inverse in developing these theorems results from the fact that in the standard form of recursive definition considered here the number of appearances of the defined function symbol \( f \) is determined (=\( m(X) \)) by \( X \) the argument of \( f \).

This dependence was incorporated so that many common problems could be naturally expressed in that form.

We have not discussed the higher order recursive definitions having nesting on the right - largely because in our experience such definitions rarely occurred in practice. Such definitions are considered in [6]. The techniques given in [6] in combination with those here can be used to extend the above results to higher order recursive definitions not covered in [6].
3. A CLASS OF FUNCTIONS IN F WITH TIME EFFICIENT IMPLEMENTATIONS

Types of Time Efficient Implementations--Patterned Comparability

In the Introduction the existence of a time-efficient implementation of a function $f \in F$ was traced to the fact that in the standard evaluation of $f$ amongst the many sub-computations necessary, there are pairs which are virtually identical. In the implementation then it becomes possible to use the remembered result of computing one member of such a pair in computing the second member. Thus the time cost of recomputation is minimized. Examples illustrating this general assertion are given below.

Consider a function $f \in F$ for which the following properties hold.

(1) There is a relation called dominance between some pairs of members of $W_f$ (the range of $w$ in the definition of $f$) such that:

(2) Whenever two members of $W_f$ with one dominating the other both appear as arguments of a $w$ function in a given order(s), then the dominated argument may be removed--(the non-dominated one possibly requiring concurrent simple alteration) and the $w$ function will still give the same result.

Properties (1) and (2) alone are sometimes sufficient to allow significant time saving as when $w$ is a logical 'and' function with the arguments 0 or 1. If any argument of $w$ is known to be 0 then the other arguments need not be computed. However, the existence of (1) and (2) is not always sufficient to guarantee a time-saving. But if the following property also holds, time saving can be guaranteed:

(3) For a substantial set of pairs $x$ and $y$ such that $x = o_1(d)$, and $y = o_J(d)$ where $I$ and $J$ are sequences applicable to the same initial data structure $d$, it is simple to determine whether $f(x)$ dominates $f(y)$. (A pair of data structures $x$ and $y \in \Delta_f$ for which such a determination is possible are called comparable.)
Consider the following example of the existence of all three of these above properties. If \( w \) is a minimum, and \( W_f \) is the set of positive integers then we can define \( x \) dominates \( y \) to mean that \( x \) is less than \( y \) and (1) and (2) will be satisfied. This in itself is not enough to guarantee any time saving.

Let \( \alpha \) and \( \beta \), be partial paths in an integer weighted digraph \( G \) and let 
\[
\frac{f(\alpha, c(\alpha))}{f(\beta, d(\beta))}
\]
be the cost of a path from node \( A \) to node \( B \) in starting with the partial path \( \alpha/\beta \) whose cost is itself \( c(\alpha)/c(\beta) \). Then with the current definition of dominance one can determine whether \( f(\alpha, c(\beta)) \) dominates \( f(\beta, c(\beta)) \) whenever \( \alpha \) and \( \beta \) end on the same node. If they do, \( f(\alpha, c(\alpha)) \) dominates \( f(\beta, c(\beta)) \) if \( c(\alpha) \geq c(\beta) \) otherwise \( f(\beta, c(\beta)) \) dominates \( f(\alpha, c(\alpha)) \). Thus (3) is satisfied and it is easy to see that dominated functions need not be computed.

In fact whenever the three properties above hold, time-savings are possible. To see this requires a brief review of implementation techniques for \( f \in F \). The standard depth first or breadth first implementations of a recursive definition \( f \in F \) is a simulation of the substitution process described in Section 2. Initially the substitution starts with the 'evaluation form' \( f(d), d \in D_f \). If \( T(d) \) is not true this is replaced by the 'evaluation form' \( w(f(o_1(d), \ldots, f(o_m(d)(d))) \). Then substitution is made for \( f(o_j(d)) \) for some \( 1 \leq j \leq m(d) \) is made to get a next 'evaluation form'.

The process continues with subsequent substitutions for occurrences of the form \( f(\alpha) \). Now if the above three conditions hold one can include in the evaluation an examination to determine whether two appearances of \( f \), say \( f(\alpha) \) and \( f(\beta) \) appearing in a subexpression, like \( w(f(\alpha), \ldots, f(\beta), \ldots) \),
of an evaluation form are comparable. If they are and if \( f(\alpha) \) dominates \( f(\beta) \), then \( f(\beta) \) can be eliminated from the subexpression. An entire course of substitutions is thus eliminated.

When the three conditions above hold, it is of advantage then to incorporate in the implementation a means for comparing pairs of data-structures \( \alpha \) and \( \beta \) in \( \Delta_f \). The details on how this is done will depend on the details of the definition but two broad classes can be distinguished. For a definition \( f \in F \) in which comparable data-structures arise, the pattern and frequency of their occurrences may be highly dependent on the initial data-structure, or alternately their occurrences may follow a fixed, predictable pattern largely independent of the input. In the input dependent case a facility for testing for comparability can be incorporated. It must be able to handle comparisons in a general way. Many partial results will have to be saved for comparison, even though the benefit derived from the comparison may be small. In the patterned or systematic case the implementing algorithm can often be tailored to take advantage of this fixed pattern—avoiding the need for a general comparison facility.

In this section, a significant subclass of functions in \( F \) which have a patterned structure of comparable data-structures will be studied. This subclass is called the 'explicit history' class. Corresponding to each member of this class is a set of equations whose solution is equivalent to the evaluation of the corresponding function. If this set of equations has the property of 'open-loop consistency' it can be solved by a process similar to Gaussian Elimination. This in turn will immediately provide a

* Problems whose solution can be obtained by effectively solving a set of equations with 'linear' like properties form a significant class. Such a class is carefully considered in [1]. Here we are interested in how recursive definition formulations to these and even some 'non-linear' problems are related to their set of equations formulation.
relatively efficient algorithm for implementation of such functions in F.

The definition of 'open-loop consistency', a property of equation sets, will be developed first then that of the explicit history function, and then in Theorem 3.1 their relation will be established.

Throughout the subsequent development we will make extensive use of notation similar to that introduced in section 2 for indicating replacements of components of a vector. Here we will extend that notation to describe replacement of a component of any expression. So if e is an expression, \(X_2\) a variable which may appear in e and \(e_1\), another expression then \(e[X_2 + e_1]\) means the result of replacing each occurrence of \(X_2\) in the expression e by the expression \(e_1\). The notation can be further extended to cover sets of such substitutions; for example, \(e[X_k + e_k; k \in N]\) is the result of substituting the expression \(e_1\) for all occurrences of \(X_1\), \(e_2\) for all occurrences of \(X_2\), \ldots, and \(e_n\) for all occurrences of \(X_n\) in e. Also the notation allows composition; so \(e[X_2 + e_1][X_1 = 0]\) is the result of first replacing each \(X_2\) in e by \(e_1\) and then every \(X_1\) in the result by 0.

Note that the expression \(e[X_2 + e_1[X_1 + 0]]\) gives the same result as that of the notation in the previous sentence. Such reorderings yielding the same result will be used in the proof of Theorem 3.1.

**Open-Loop Consistency and Explicit History Definitions**

**Equation Sets**

\(W_n\) is a set of functions. If \(w \in W_n\) then \(w\) is defined on 0 arguments, 1 argument, \ldots, \(n\) arguments. Each argument is drawn from a set \(S\).
The range of each \( w \) is also \( S \). \( S \) contains a 0 element with the property that \( w(X_1, \ldots, X_{c-1}, X_c, X_{c+1}, \ldots, X_n) = w(X_1, \ldots, X_{c-1}, X_{c+1}, \ldots, X_n) \) if \( X_c = 0 \).

\( E_n(W_n) \) is a set of equations involving \( n \) functions from \( W_n = \{w_1, \ldots, w_n\} \), and the variables \( X_1, \ldots, X_n \).

\( E_n(W_n) \) contains a subset called the **basis** subset.

The basis of \( E_n(W_n) = \{X_j = w_j(X_j, \ldots, X_{m_j}) \mid j \in N, j_1 \in N, j_k > j_1 \text{ if } i > k\} \)

In addition, \( E_n(W_n) \) satisfies the closure conditions that:

**C1:** If \( X_j = e \) is in \( E_n(W_n) \) then so is \( X_j = e[X_k + 0] \) for any \( X_k \) in \( e \) and:

**C2:** If for any given \( j \) and \( k \in N, X_j = e_1 \) and \( X_k = e_2 \) are each members of \( E_n \), then so is \( X_j = e_1[X_k + e_2] \).

\( E_n(W_n) \) consists only of those equations in the basis together with those constructable from **C1** and **C2**.

From here on we use \( E_n \) to stand for \( E_n(W_n) \).

**Open-Loop Consistent Equation Sets**

Let \( Q = \{X_j = e_j \mid j \in N\} \) be a set of equations in \( E_n \).

Let \( Q_c = \{X_j = e_j \mid 1 \leq j \leq c-1\} \cup \{X_c = e_c[X_c + 0]\} \cup \{X_j = e_j \mid c+1 \leq j \leq n\} \)

If every solution to \( Q \) is also a solution to \( Q_c \), then \( Q \) in \( E_n \) is open-loop consistent in \( X_c \). If \( Q \) of \( E_n \) is open-loop consistent in all variables \( X_c, c \in N \), then \( Q \) is open-loop consistent in \( E_n \). If every subset in \( E_n \) of the form of \( Q \), is open-loop consistent, then \( E_n \) is open-loop consistent.
The significance of open-loop consistency follows from the following. On the one hand, if the basis set of $E_n$ is:

$$\{x_i = e_i | i \in N\}$$

and $E_n$ is open-loop consistent then a process analogous to Gaussian Elimination will be adequate to solve the basis equations for $\{x_i | i \in N\}$. This process is based on the alternate application of two operations applied to a set of equations which is initially the basis of $E_n$. The first operation is used to remove the recursive appearance in an equation of the form $x_j = e_j$ of any occurrence of $x_j$ in $e_j$. This is made possible if the set of equations is Open-Loop Consistent, by setting all occurrences of $x_j$ in $e_j$ to 0. (Open-loop consistency is a specialization of a considerably more general property which would allow removal of such recursive appearances, which we hope to develop at a future time.) This first operation is only necessary if there is such a recursive appearance to be removed in one or more equations of the set. The second operation uses an equation of the form $x_j = e_j$ with $x_j$ appearances removed from $e_j$ to substitute for all appearances of $x_j$ on the right of other equations in the set, thus eliminating all occurrences of $x_j$ on the right of all equations in the set. The two operations can be repeated $n$ times as needed to produce a solution to the basis set from $E_n$. Note that every equation in the sets that result from different steps in this process is in $E_n$ so that the open-loop consistency condition is always applicable. In any case this process provides a relatively efficient means of solving such a set of equations.

On the other hand the evaluation of an 'explicit history' recursive definition will be shown to be equivalent to the solution of a corresponding
set of equations forming the basis of a set $E_n$ if $E_n$ is open-loop consistent. Thus the relatively efficient solution will be applicable to 'explicit history' definitions having this property.

Explicit History Recursive Definitions

The 'explicit history' recursive definition will initially be given in a form which allows the simple description of the corresponding set of equations. Later we will show other equivalent forms for such definitions.

Recalling that $N = \{1, 2, \ldots, n\}$, let $H$ be a vector all of whose components are in $N$, and no two of which are the same. $H$ is the history vector. $H/c$ will be used as a shorthand for $H/(c> = \text{the concatenation of } c\text{ with vector } H$. If $V$ is a vector, $\{V\}$ is the set of components of $V$. $w_i \in W_n$, a set of functions defined earlier. $c_i \in N$, $c_i > c_j$ if $i > j$.

An 'explicit history' recursive definition has the form:

$$
def 3.1 \begin{cases} 
X^H = 0 & \text{if } c \in \{H\} \\
X_c^H = w_c(X^c_{c_1}^H, \ldots, X^c_{c_m}^H) & \text{if } c \in N - \{H\} \\
\text{initially } H = \lambda, c \in N 
\end{cases}$$

If $f$ is an explicit history recursive definition then the corresponding set of equations designated $E_n(f)$ is built on the following basis:

$$
def 3.1.1 \left\{ X_c = w_c(X_{c_1}^c, \ldots, X_{c_m}^c) = e_c | c \in N \right\}$$
Theorem 3.1: If \( f \) is an explicit history recursive definition and \( E_n(f) \) its corresponding equation set is open-loop consistent then the set of values \( \{ x^\lambda_c | c \in N \} \) as determined by evaluating \( f \), will satisfy the basis of \( E_n(f) \) with \( x^\lambda_c = x^\lambda_c \) for all \( c \in N \).

Proof: A partly inductive argument will be used.

First we define two kinds of expressions \( Z_j^\alpha \) and \( Y_j^\alpha \):

If \( N-\{a\} = \{j\} \):

1. \( Z_j^\alpha = e_j = \) the right side of the equation \( X_j = e_j \) in \( E_n \)
2. \( Y_j^\alpha = 0 \)

If \( j \in a \):

1. \( Z_j^\alpha = \) an expression giving \( X_j \) as a composition of functions in \( W_n \) in which only the variable \( X_k \) with \( k \in \{a\} \) appear
2. \( Y_j^\alpha = Z_j^\alpha(x_k = 0 : k \in \{a\}) \)

\( Z_j^\alpha \) then gives \( X_j \) as an expression involving \( n \) variables and thus as a composition of functions from \( W_n \) with no arguments. \( Y_j^\alpha \) must be defined as a constant and so the expression \( Z_j^\alpha \) must be evaluated as a constant. By definition of \( Y_j^\alpha \):

\[ Y_j^\alpha = Z_j^\alpha \]

Note that by definition when \( N-\{a\} = \{j\} \) the equation \( X_j = Z_j^\alpha \) is in \( E_n \), thus since \( E_n \) is open-loop consistent so is \( X_j = Y_j^\alpha \) (by \( d_2 \) above).

(H) Assume that for \( a = \) any sequence of \( k < n \) integers taken from \( N \) with no two equal and \( c \in N-\{a\} \) that:

\[ X_j = Z_j^{\alpha/c/} \quad \forall j \in N-\{a//c\} \]

are each true equations in \( E_n \).
Then for \( c \in N \):

\[
X_c = e_c.
\]

is in \( E_n \) by def 3.1.1

\[
X_c = e_c[X_j + z^a_j/\langle c \rangle] : j \in N - \{a/\langle c \rangle\}
\]

is in \( E_n \) by definition of \( E_n \) and (H)

\[
X_c = e_c[X_j + z^a_j/\langle c \rangle] [X_c + 0] : j \in N - \{a/\langle c \rangle\}
\]

is in \( E_n \) because \( E_n \) is open-loop consistent and \( X_j = z^a_j/\langle c \rangle \) is in \( E_n \) by (H).

\[
X_c = e_c[X_j + z^a_j/\langle c \rangle] [X_c + 0] : j \in N - \{a/\langle c \rangle\}
\]

is in \( E_n \) by reordering and substitution.

\[
X_c = e_c[X_j + z^a_j/\langle c \rangle] [X_c + 0] : j \in N - \{a\}
\]

by which \( z^a_j/\langle c \rangle = 0 \) when \( j \in \{a/\langle c \rangle\} \).

So if \( z^a_j, j \in N - \{a\} \) is defined as:

\[
z^a_c = e_c[X_j + z^a_j/\langle c \rangle] [X_c + 0] : j \in N - \{a\}
\]

then \( X_c = z^a_c \) is true and is in \( E_n \) for \( c \in N - \{a\} \) and by definition of \( Y^a_c \) and \( X^a_c = Y^a_c \) for \( c \in N - \{a\} \). In fact we have that:

\[
Y^a_c = e_c[X_j + Y^a_j/\langle c \rangle] : j \in N - \{a\}
\]

by \( d_2 \)

\[
Y^a_c = e_c[X_j + z^a_j/\langle c \rangle] [X_c + 0 : k \in \{a\}] : j \in N - \{a\}
\]

by reordering

\[
Y^a_c = e_c[X_j + z^a_j/\langle c \rangle] [X_c + 0 : k \in \{a\}] : j \in N - \{a\}
\]

by reordering

(1) \[
Y^a_c = e_c[X_j + Y^a_j/\langle c \rangle] : j \in N - \{a\}
\]

or since by \( d_1 \):

\[
\begin{align*}
Y^a_c &= 0 & \text{if } c \in a \\
Y^a_c &= e_c[X_j + Y^a_j/\langle c \rangle] & \text{if } c \in N - a
\end{align*}
\]

These two relations give the recursive definition we seek.

The recursive definition, def 3.1, is not, as written, in the form required for a member of \( F \). However, by simple transformations a definition
which is a member of $F$ can be generated.

Let $f(H,c) = x^H_c$ and make the subscript of $w_c$ an argument of a modified $w$ function.

\[
\begin{align*}
    f(H,c) &= 0 & \text{if } c \in \{H\} \\
    f(H,c) &= w(c,f(H/c,c_1), \ldots ,f(H/c,c_n)) & \text{if } c \in N-\{H\} \\
    \text{initially } H = \lambda, c \in N
\end{align*}
\]

Still one more small change is needed to create a definition in $F$. The first argument $c$ of $w$ is not in the required form. It is put in the correct form by adding another argument, $s$, which is either $0$ or $1$.

\[
\begin{align*}
    f(H,c,0) &= 0 & \text{if } c \in \lambda \\
    f(H,c,s) &= c & \text{if } s = 1 \\
    \text{def 3.1b} f(H,c,s) &= w(f(H,c,1),f(H/c,c_1,0), \ldots ,f(H/c,c_n,0)) & \text{if } c \in N-\{H\} \\
    \text{initially } H = \lambda, c \in N, s = 0
\end{align*}
\]

Theorem 3.1 is the Main Result of this section. Its application will be illustrated by an example. In order, however, to make this result generally applicable, it would be necessary to further elucidate two questions concerning the set of equations which correspond to a given 'explicit history' definition.
The first of these is: do sets of equations which are open-loop consistent arise with significant frequency in practice. The second question arises after one has an open-loop consistent set at which point one would like to know if the solution is unique. This would guarantee that evaluation of the recursive definition would give the same result as would any procedure for solving the corresponding equation set. If the solution is not unique it would be well to know to which solutions of the equation set, obtained by what procedure, the evaluation of the corresponding recursive definition (which is always unique) is equivalent. These two questions are not considered further here. Now, however, we note some very special cases in which the answers to these questions are evident.

We say $E_n$ has no loop if the equations in the basis of $E_n$:

$$X_1 = \{e_i | i \in N\}$$

have the property that $e_i$ contains no occurrence of any variable $X_{i-k}$, $k \geq 0$. If this is the case then it is easy to see that no equation in $E_n$ can be recursive - i.e. have the variable appearing on the left of the equation also occur on the right. From this it follows directly that $E_n$ is open-loop consistent. Furthermore, it is easy to show that in this case the solutions are unique.

Lemma 3.1: If the basis of $E_n$ has no loops then $E_n$ is open-loop consistent and its solution is unique.

Examples of Applications of Theorem 3.1

The maximum and minimum path problems, each defined on a directed graph will be used to illustrate the application of Theorem 3.1.
The following graph definition will be used:

\( G \) is a directed graph with a set of nodes \( N = \{1, 2, \ldots, n\} \)

If \( c \) is a node in \( G \) then:
- \( c_i \) is the \( i \)th neighbor of \( c \) (the \( i \)th node reachable from \( c \) by traversing a single directed branch)
- \( c_m \) is the last neighbor of \( c \)

\( W(c_i) \) is the weight of the branch from \( c \) to \( c_i \). It is always > 0.

If \( X_i \) is a number then:
- \( \max(X_1, \ldots, X_n) \) is the maximum of \( X_i, i \in N \)
- \( \min(X_1, \ldots, X_n) \) is the minimum of \( X_i, i \in N \)

A path \( h = <h_1, \ldots, h_p> \) in \( G \) is a sequence of nodes in \( G \) such that there is a branch \( <h_i, h_{i+1}> \) in \( G \) for each \( p-1 \leq i \geq 1 \). The weight of path \( i \) is the sum of the weights of the branches \(<h_i, h_{i+1}>, \ldots \geq i \geq 1\).

Interpret \( \text{MAW}(h,c)* \) to be the maximum weight loopless path starting with the path \( h/<c> \) and ending on node \( t \). Since continuation of such a path must be by passage to some neighbor, \( c_* \), of \( c \) which cannot be in \( h/<c> \) if there are to be no loops we have:

\[
\text{Ex. 3.1} \quad \begin{cases} 
\text{MAW}(h,c) = 0 & \text{if } c \in \{h\}, \text{ or } c=t \\
\text{MAW}(h,c) = \max(W(c_1)+\text{MAW}(h//c,c_1), \ldots, W(c_m)+\text{MAW}(h//c,c_m)) & \text{if } c \in N-\{h\} 
\end{cases}
\]

and analogously for the minimum path we have if \( \text{MIN}(h,c) \) is the minimum weight of all paths starting with path \( k //<c> \) and ending on node \( t \) then its recursive definition can be given by the equations:

*Strictly this is \( \text{MAW}_c(h,c) \) since the path's end on \( t \).
Ex 3.2
\[
\begin{align*}
\text{MIW}(h,c) &= 0 & \text{if } c \in \{h\} \text{ or } c = t \\
\text{MIW}(h,c) &= \min(W(c_1) + \text{MIW}(h/c, c_1), \ldots, W(c_m) + \text{MIW}(h/c, c_m)) & \text{if } c \in N - \{h\}
\end{align*}
\]
initially \( h = \lambda, c \in N \)

With \( w(c, X_1, \ldots, X_m) = \max/min(W(c_1) + X_1, \ldots, W(c_m) + X_m) \), both examples are in the form of definition 3.1a and thus are equivalent to the recursive definition in theorem 3.1. Therefore, in both cases we may speak of the corresponding set of equations def 3.1.1. Consider the equations in the basis of \( E_n(\text{MAW}) \) corresponding to ex 3.1 first:

Ex 3.1.1 \( \{ X_c = \max(W(c_1) + X_{c_1}, W(c_2) + X_{c_2}, \ldots, W(c_m) + X_{c_m}) \mid c \in N \} \)

where as noted \( c_i \) is the \( i \)th neighbor of node \( c \) in \( G \). Now in general because of the \( \max \) function, \( E_n(\text{MAW}) \) with the above basis will not be open-loop consistent. In fact if the basis of \( E_n(\text{MAW}) \) contains an equation in which the same variable appears on the right then \( E_n(\text{MAW}) \) is not open-loop consistent because suppose \( c_j = c, a_{c_i} > 0 \) and:

(a) \( X_c = \max(a_{c_1} + X_{c_1}, \ldots, a_{c_j} + X_{c_j}, \ldots, a_{c_m} + X_{c_m}) \)

then if \( E_n(\text{MAW}) \) was open-loop consistent \( X_c \) would also satisfy:

(b) \( X_c = \max(a_{c_1} + X_{c_1}, \ldots, a_{c_j-1} + X_{c_j-1}, a_{c_j} + X_{c_j+1}, \ldots, a_{c_m} + X_{c_m}) \)

but it is easy to see by substitution for \( X_{c_j} = X_c \) using (b) in (a) that this cannot be.

However, if the graph \( G \) does not have any loops, then \( E_n(\text{MAW}) \) is open-loop consistent, since then the conditions of lemma 3.1 are met. So if \( G \) is loopless, theorem 3.1 can be applied and solutions to the equations of ex 3.1.1 will be equivalent to an evaluation of the recursive definition in ex 3.1. The number of equations in ex 3.1.1 is equal to the number of nodes in the
graph $G$. No variable appears on both the left and right sides of an equation and cannot do so as the result of any substitutions because $G$ has no loops. One of the equations in the set, say $X_n = e_n$, will have only a constant on its right side, i.e. $e_n = a_n$. This constant may be substituted for all occurrences of $X_n$ on the right of the other equations. As a result, a second equation, say $X_{n-1} = e_{n-1}[X_n + a_n] = a_{n-1}$ will have only a constant on its right. This process continues until all resultant equations up to the one with the variable whose value is sought have only a constant on their right. The $i^{th}$ substitution involves substituting for at most $i$ variables, $i$ additions and taking the maximum over $i$ numbers, or the order of $i$ steps, $i$ ranges from 1 to the number of equations at most. So after the equations are set up, the solution algorithm has a complexity of the order of $n^2$.

In ex 3.2, the set of equations $E_n(MIW)$ is open-loop consistent. This is easily seen for if $c_j = c$, $a_{c_j} > 0$ and:

(a) $X_c = \min(a_{c_1} + X_{c_1}, \ldots, a_{c_j} + X_{c_j}, \ldots, a_{c_m} + X_{c_m})$

and consider:

(b) $X_c = \min(a_{c_1} + X_{c_1}, \ldots, a_{c_j} + X_{c_j}, a_{c_j + X_{c_j + 1}}, \ldots, a_{c_m} + X_{c_m})$

substituting with (b) in the right of (a) we get:

$X_c = \min(a_{c_1} + X_{c_1}, \ldots, \min(a_{c_1} + X_{c_1}, \ldots, a_{c_j} + X_{c_j}, a_{c_j + X_{c_j + 1}}, \ldots, a_{c_m} + X_{c_m}))$

which because of the properties of $\min$:

$X_c = \min(a_{c_1} + X_{c_1}, \ldots, a_{c_j} + X_{c_j}, a_{c_j} + X_{c_j + 1}, \ldots, a_{c_m} + X_{c_m})$

The same equation would result from substituting for $X_c$ on the left of (a) with the right of (b). Thus (b) satisfies (a) verifying the claimed
open-loop consistency. Not only is $E_n^{(MIW)}$ open-loop consistent, but the basis of $E_n^{(MIW)}$ can be shown to have a unique solution.

Thus a substituional solution to this set of equations will give a solution to the corresponding recursive definition. This solution has a time complexity $\leq n^3$, in part because after each substitution step the terms can be gathered in this case so they will again be a simple-min of constants + 'ed with variables, in the same form as the initial basis equations. Furthermore in this case, again - primarily because of the properties of the min function, the complexity of the gaussian elimination solution can be reduced to order $n^2$ by adopting the appropriate order of substitution. The appropriate order is given by the following considerations.

First the $X_t$ equation will be:

$X_t = 0$, because $t$ is by definition the last node on a path

The right side is a constant. So substitute throughout the other equations right sides this constant value for $X_t$. In these other $n-1$ equations carry out the + and min operations with the substituted value. That will leave the right side of these equations again in the form of a min of terms. All except one of these terms being a constant + 'ed with a variable. The exceptional term being simply a constant. The variable $X_t$ will not appear on the right side of any equation. The key step now is to examine the $n-1$ equations for the one whose constant term is the smallest.

Say this is:

$X_j = \min(c_{j_1} + X_{j_1}, ..., c_{j_n} + X_{j_n}, d_j)$

where $c_{j_1}$ and $d_j$ are constants. Then that equation can be simply replaced by:

$X_j = d_j$
That is, all the variables can be removed from the right side of this equation. Roughly this is justified because, since $d_j$ is the smallest constant in the set of equations and consideration of the substitutional method of solution indicates that no variable can ultimately be assigned a lower value. Next in the n-2 equations (excepting the $X_i$ and $X_j$ equations which are solved) $d_j$ is substituted for all appearances of $X_j$. Terms are then gathered in the remaining n-2 equations. Again the one of those having the smallest constant is chosen and all its variable terms removed, and so forth thus providing the solution for another variable. The process is repeated until the variable whose value is sought becomes the one having the smallest constants on the right, of all remaining unsolved equations. That is the value of the variable.

In this method of solution one is always substituting a constant thus greatly reducing the effort necessary to simplify the right sides of equations. It is essentially Dykstra's algorithm and its complexity of order $n^2$.

Some Simple Equivalences

Throughout this paper we have presented recursive definitions purported to be 'natural'. In fact, however, for any given algorithm design problem one may develop a number of recursive definitions each being equally entitled to being called natural. The differences in these equally natural descriptions may range from very small details to deeper differences representing radically different points of view. We have seen some small adjustments of detail in getting equivalent formulations of the recursive definition of theorem 3.1. Deeper differences are represented by the
alternate formulations for the recursive definition of the set of binary numbers given in the introduction. The path examples of the previous pages were designed to look natural - but also to fit the theorem formulation - many other formulations - which did not quite fit - the theorem formulation could have been represented as natural. There is no deception here; they are all natural. There are many equivalent ways of defining the same function. A number of simple equivalences will now be developed which account for some common alternatives. Possession of such equivalences allow one to move easily from definition to definition. As we have seen, some definitions are closer to good algorithms than others. Some definitions, unary ones, can even be considered as specifying algorithms. In fact, all the theorems given thus far can be viewed as equivalences which allow one to move from an initial definition to a better one. The equivalences given now differ only in being somewhat simpler than those developed previously, and seeming to have more to do with initial formulation.

The equivalences to be given all involve trade-offs between the complexity of the \( w \) and \( o \)-functions, or between the complexity within the argument of the function being defined (say \( f \)) and the complexity outside that argument. If the structure of the argument of \( f \) is considered the data-structure, and the way in which \( f \) enters as an argument (of \( w \)) in the recursive definition is called the control structure, then these equivalences give trade-offs between data and control structures.

The significance of the following two theorems can probably be better appreciated if the example following the theorems are scanned before the theorems and proofs are read.
Theorem 3.2: If operations \( \cdot \) and \( + \) have the properties:

\[
P_1 \quad X_1 \cdot (X_2 \cdot X_3) = (X_1 \cdot X_2) \cdot X_3
\]

\[
P_2 \quad X \cdot (\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} (X \cdot X_i)
\]

and if:

\[
d_1 \quad \begin{cases}
g(X) = q(X) & \text{if } T(X) \\
g(X) = \sum_{i=1}^{m} (t_i(X) \cdot g(o_i(X))) & \text{if } T(X)
\end{cases}
\]

initially \( X \in D \)

and:

\[
d_2 \quad \begin{cases}
f(y, X) = y \cdot q(X) & \text{if } T(X) \\
f(y, X) = \sum_{i=1}^{m} f(y, t_i(X), o_i(X)) & \text{if } T(X)
\end{cases}
\]

initially \( X \in D \).

then for each \( X \in \Delta_g \):

\[
f(y, X) = y \cdot g(X)
\]

Proof: We simply show that \( y \preceq g(X) \) with \( g(X) \) as defined in \( d_1 \) satisfies or makes true the relations of \( d_2 \) when appropriately substituted for \( f(y, X) \) in \( d_2 \).

Thus if \( X \) is of remoteness 0, \( d_2 \) defines \( f(y, X) \) so that substituting:

\[
y \cdot g(X) = y \cdot q(X)
\]

\[
y \cdot q(X) = y \cdot q(X) \quad \text{since } g(X) = q(X) \text{ when } X \text{ is of remoteness 0 by } d_1
\]

For \( X \) having remoteness > 0, \( d_2 \) defines \( f(y \cdot X) \). Substitute \( y \cdot g(X) \) for \( f(y \cdot X) \) throughout that equation to determine if it is thus satisfied. Note that if \( X \in \Delta_g \) then certainly \( o_1(X) \) is also a member of \( \Delta_g \).
\[ y \cdot g(X) = \sum_{i=1}^{m(X)} [(y \cdot t_i(X)) \cdot g(o_i(X))] \]

\[ = \sum_{i=1}^{m(X)} [y \cdot (t_i(X) \cdot g(o_i(X)))] \quad \text{by p1} \]

\[ = y \cdot (\sum_{i=1}^{m(X)} [t_i(X) \cdot g(o_i(X))]) \quad \text{by p2} \]

\[ y \cdot g(X) = y \cdot g(X) \quad \text{by d} \]

Our first example of the application of theorem 3.2 requires some definitions. Let \( B = \{b_1, ..., b_m\} \) and \( A = \{a_1, ..., a_n\} \), both be sets whose components are vectors:

\[ B \otimes A = \{b_1/a_1, ..., b_m/a_n\} \cup \{b_2, ..., b_m\} \otimes A \]

With this definition of \( \otimes \) the following property clearly holds for \( A, B \) and \( C \) each sets of vectors:

\[ C \otimes (B \otimes A) = (C \otimes B) \otimes A \quad \text{by p1} \]

Also if \( \cup \) is the usual union operation then if each \( X_i, i \in N \), as well as \( A \) is a set of vectors:

\[ X \otimes (X_1 \cup X_2 \cup ... \cup X_n) = (X \otimes X_1) \cup (X \otimes X_2) \cup ... \cup (X \otimes X_n) \quad \text{by p2} \]

Now the set of all \( n \) bit binary numbers (vectors) = \( B(n) \) is the set of all \( n-1 \) bit binary numbers, \( B(n-1) \), with a 0 attached in front of each member of the set, \( \langle 0 \rangle \otimes B(n-1) \), together with \( \langle 0 \rangle \) \( B(n-1) \) with a 1 attached in front of each member of the set, \( \langle 1 \rangle \otimes B(n-1) \).

More formally the definition is:

\[
B(n) = \begin{cases} 
\{\lambda\} & \text{if } n = 0 \\
\{\langle 0 \rangle\otimes B(n-1)\} \cup \{\langle 1 \rangle\otimes B(n-1)\} & \text{if } n > 0
\end{cases}
\]

(1)

\[ \text{initially } n = \text{the number of bits of the binary number} \]

Since \( \otimes \) and \( \cup \) satisfy p1 and p2, it follows by theorem 3.2 that
A \( y, n \) given in the following definition is a member of \( F \) and is related to the definition above in that \( B'(y, n) = y \cup B(n) \).

\[
\begin{align*}
B'(y, n) &= y \quad \{ \lambda \} = y & \text{if } n = 0 \\
B'(y, n) &= B'(y \cup \{0\}, n-1) \cup B'(y \cup \{1\}, n-1) & \text{if } n > 0 \\
\end{align*}
\] (2)

initially \( n \) = the number of bits of the binary number, \( y = \{ \lambda \} \)

Note that (2) is essentially the definition given in the Introduction for binary numbers (ex 1.1).

As a second example consider the application of this theorem to the previous path examples. Thus equivalent to the definition of ex 3.1 we have by theorem 3.2 since \( \max \) (corresponding to +) and \( + \) (corresponding to .) have the appropriate properties the following function MAN (the pair \(<h, c>\) corresponds to \( X, W(c_1)\) corresponds to \( t_i(X), <h/c, c_1>\) corresponds to \( \sigma_i(X).\)):

\[
\begin{align*}
\text{MAN}'(y, h, c) &= y + 0 & \text{if } c \in \{h\} \text{ or } c = t \\
\text{MAN}'(y, h, c) &= \max\left(f(y + W(c_1), h/c, c_1), \ldots, f(y + W(c_m), h/c, c_m)\right) & \text{if } c \in N \setminus \{h\} \\
\end{align*}
\]

initially \( h = \lambda, c \in N, y = 0 \)

The next equivalence allows the replacement of a vector in the data-structure by an individual component at the expense of a more complex control structure.

Definitions:

\( \alpha \) is a sequence of symbols in the alphabet \( A \).

If \( a \in A, R_i(a)\) for \( i \leq m(a) \) is a sequence of symbols in \( \alpha \). There is a subset of \( A \), say \( A' \), such that for each \( a' \in A' \), \( m(a') = 1 \), and \( R_1(a') = \lambda \). This is necessary if the definition of \( f \) in the following theorem is to be legitimate.
Theorem 3.3: Given the binary operations \( \cdot \) and \(+\) with the following properties:

\[
\begin{align*}
p1: \quad (X_1 \cdot X_2) \cdot X_3 &= X_1 \cdot (X_2 \cdot X_3) \\
p2: \quad (\Sigma_i X_i) \cdot Z &= \Sigma_i (X_i \cdot Z) \\
p3: \quad 0 \cdot X &= X \cdot 0 = X
\end{align*}
\]

and given the definition:

\[
d_1 \begin{cases} 
  f(a) = 0 & \text{if } a = \lambda \\
  f(a) = \Sigma_{i=1}^{m(a)} r_i(a) \cdot f(a[ a_i + R_i(a_i) ]) & \text{if } a \neq \lambda \\
\end{cases}
\]

initially \( a = \langle a \rangle \) where \( a \in A \)

then:

\textbf{fact1:} \quad f(a) = f(a_{1:j}) \cdot f(a_{j+1:n}) \quad \text{if } j \in \mathbb{N}

and further given the definition:

\[
d_2 \begin{cases} 
  g(a) = 0 & \text{if } a \in A' \\
  g(a) = \Sigma_i^{m(a)} r_i(a) \cdot g([R_i(a)]) \cdot g([R_i(a)]_2) \cdots g([R_i(a)]_n) & \text{if } a \notin A' \\
\end{cases}
\]

initially \( a \in A \)

then:

\textbf{fact2:} \quad \text{for each } a \in A', \quad f(\langle a \rangle) = g(a)

Proof: First we show that:

\[
f(a) = f(a_{1:j}) \cdot f(a_{j+1:n})
\]

(H) Assume since \( a[ a_i + R_i(a_i) ] \) is of lesser remoteness than \( a \) that \textbf{fact1} is true, i.e.:

\[
f(a[ a_i + R_i(a_i) ]) = f(R_i(a)) \cdot f(a_{2:n})
\]

Thus:

\[
f(a) = \Sigma_{i=1}^{m(a)} r_i(a) \cdot f(R_i(a)) \cdot f(a_{2:n})
\]

by \( p1 \)

\[
= \Sigma_{i=1}^{m(a)} ((r_i(a) \cdot f(R_i(a_i)) \cdot f(a_{2:n}))
\]

by \( p2 \)

\[
= f(a_{1:j}) \cdot f(a_{j+1:n})
\]

* \( a_{1:j} = \langle a_1, a_{1+1}, \ldots, a_j \rangle \) if \( a = \langle a_1, \ldots, a_n \rangle \); \( a_{1:j} = \langle a_1 \rangle \)
The basis for this induction is given next.
If \( \alpha \) is of remoteness 1 then: \( \alpha[a_1 + R_i(a)] \) must be of remoteness 0, must = so:

\[
\begin{align*}
f(\alpha) &= \sum_{i=1}^{m(a_1)} r_i(a_1) \cdot f(\alpha[a_1 + R_i(a_1)]) \\
&= \sum_{i=1}^{m(a_1)} r_i(a_1) \cdot f(\lambda) \\
&= \sum_{i=1}^{m(a_1)} r_i(a_1) \quad \text{since } f(\lambda) = 0 \text{ by } d_1 \text{ and } p3
\end{align*}
\]

Since \( \alpha[a_1 + R_i(a_1)] = \lambda \), \( \alpha \) must = \( \langle a' \rangle \) with \( a' \in A' \).
Therefore if \( \alpha \) is of remoteness 1 then \( \alpha = \langle a' \rangle \), \( a_1 = a' \).

\( a_{2;n} = \lambda \), and:

\[
\begin{align*}
f(\alpha) &= f(\langle a' \rangle) = \sum_{i=1}^{m(a')} r_i(a') \\
&= \sum_{i=1}^{m(a')} r_i(a') \cdot f(\lambda) \quad \text{by } p1 \\
&= f(\langle a' \rangle) \cdot f(\alpha_{2;n})
\end{align*}
\]

(H1) Now assume that for \( 1 \leq j \leq m \) that:

\[
\begin{align*}
f(\alpha) &= f(\alpha_{1;j}) \cdot f(\alpha_{j+1;n}) \quad \text{this has already been shown} \\
&\quad \text{true when } j = 1
\end{align*}
\]

then:

\[
f(\alpha_{j+1;n}) = f(\alpha_{j+1}) \cdot f(\alpha_{j+2;n})
\]

so:

\[
\begin{align*}
f(\alpha) &= f(\alpha_{1;j}) \cdot (f(\alpha_{j+1}) \cdot f(\alpha_{j+2;n})) \quad \text{by assumption } (H1) \\
&= (f(\alpha_{1;j}) \cdot f(\alpha_{j+1})) \cdot f(\alpha_{j+2;n}) \quad \text{by } p2 \\
&= f(\alpha_{1;j+1}) \cdot f(\alpha_{j+2;n}) \quad \text{by } (H1)
\end{align*}
\]

This completes the proof of fact1, fact1 may be used to establish fact2. By \( d_1 \) if \( \alpha \not\in A' \):

\[
\begin{align*}
f(\alpha) &= \sum_{i=1}^{m(a)} r_i(a) \cdot f(R_i(a)) \\
&= \sum_{i=1}^{m(a)} r_i(a) \cdot (f([R_i(a)]_1) \cdot \ldots \cdot f([R_i(a)]_n))
\end{align*}
\]
and if $a \in A'$:

$$f(<a>) = 0$$

Note that these last two equations for $f(a)$ are almost identical to the equations for $g(a)$ in $d$. A simple inductive argument based on this near identity shows that:

$$f(<a>) = g(a)$$

Example of Application of Theorem 3.2

This example involves a problem which together with its 'good' solution is discussed in [3]. We will see how one could get from an initial formulation of a recursive solution to this problem to the 'good' algorithm of [3] via theorem 3.2. The problem involves the multiplication of a set of matrices; $M_1 \times M_2 \times \ldots \times M_n$. The dimensions of each matrix is given. The dimensions of $M_i$ are $(r_i, c_i)$. The number of multiplications necessary to multiply $M_i$ by $M_j$ is then $r_i \times (c_i = r_j) \times c_j$. The multiplication $M_1 \times M_2 \times \ldots \times M_n$ may be associated in any way to get the same answer; so $(M_1 \times M_2) \times M_3 = M_1 \times (M_2 \times M_3)$ for example; but with differing numbers of multiplications required. The problem is to design an algorithm to find an association which will give the minimum number of multiplications. In fact, we will design an algorithm to find that minimum number of multiplications - which is the heart of the matter. The initial approach is to somehow enumerate all ways of associating $M_1 \times \ldots \times M_n$ and the corresponding costs in number of multiplications and then to choose the association giving the minimum of these. Different associations are equivalent to different ways in which the expression $M_1 \times \ldots \times M_n$ can be parenthesized. The last matrix multiplication in any such association must involve one of the following alternatives:
Let $M_1$ with the result of $(M_2 \times \ldots \times M_n)$ with a cost of
\[ x_1 \times (c_1 = r_2) \times c_n + \text{the cost} \left(=0\right) \text{of doing } (M_1, M_1) + \text{the cost of doing } (M_2 \times \ldots \times M_n) \text{ or}
\]
The result of $(M_1 \times M_2)$ with the result of $(M_3 \times \ldots \times M_n)$
with a cost of $x_1 \times (c_2 = r_3) \times c_n + \text{the cost of doing } (M_1 \times M_2) + \text{the cost of doing } (M_3 \times \ldots \times M_n) \text{ or } 
$
The result of $(M_1 \times \ldots \times M_{n-1})$ with $M_n$ with a cost of $x_1 \times (c_{n-1} = r_n) \times c_n + \text{the cost of doing } (M_1 \times \ldots \times M_{n-1}) + \text{the cost} \left(=0\right) \text{of doing } (M_n, M_n)$.

If $a = <i_1, j_1>, <i_2, j_2>, \ldots, <i_n, j_n>$ and $f(a)$ = the cost of doing $(M_{i_1} \times \ldots \times M_{j_1}) + \text{the cost of doing } (M_{i_2} \times \ldots \times M_{j_2}) + \ldots + \text{the cost of doing } (M_{i_n} \times \ldots \times M_{j_n})$, then:

\[
\begin{cases}
  f(a) = 0 & \text{if } a = \lambda \\
  f(a) = f(a[a_1 + \lambda]) & \text{if } a \neq \lambda, i_1 = j_1 \\
  f(a) = \text{Min}((x_{i_1} \times c_{i_1} \times c_n) + f(a[a_1 + <i_1, i_1>, <i_1+1, j_1>], \ldots \\
  \ldots, (x_{i_1} \times c_{j_1-1} \times c_{j_1}) + f(a[a_1 + <i_1, j_1-1>, <j_1, j_1>])) \\
  \text{initially } a = (<1, n>)
\end{cases}
\]

Let $a = <i, j>$ in which $i, j \in N$ and $j \geq i$. If $m(a) = 1$ and $R_k(a) = \lambda$
when $i = j$, and $m(a) = j - i$ and $R_k(a) = <i, i+k-1>, <i+k, j>$ when $j > i$
then the above recursive definition can be rewritten:

\[
\begin{cases}
  f(a) = 0 & \text{if } a = \lambda \\
  f(a) = \bigcup_{k=1}^{m(a)} ((x_{i_1} \times c_{i_1+k-1} \times c_{j_1}) + f(a[a_1 + R_k(a_1)])) & \text{if } a \neq \lambda \\
  \text{initially } a = <<1, n>>
\end{cases}
\]

* Means remove first component of $a$
Now this definition fits $d_1$ of theorem 3.3 with $\cdot$ being addition and $\Sigma$ being Min. These operations have the appropriate properties so applying the theorem we get the following definition in which we have resubstituted for $R_j(a_i)$ being equivalent to the previous definition, with $a = <i,j>$, $i,j \in N$ and $j > i$.

$$
\begin{align*}
g(a) &= 0 \quad \text{if } i = j \\
g(a) &= \min_{k=1}^{n}(r_i \times c_{i+k-1} \times c_j) + g(<i,i+k-1>) + g(<i+k,j>) \quad \text{if } j > i \\
\text{initially } a &= <1,n>
\end{align*}
$$

If we replace $g(<i,j>)$ by $X_{ij}$ and $r_i \times c_{i+k-1} \times c_j$ by $a_{ijk}$ in the above definition then that definition amounts to a set of $n^2$ equations, each equation being of the form:

$$
X_{ij} = \min(a_{ij1} + X_{ii} + X_{i+1,j} a_{ij2} + X_{i+1,j}, \ldots)
$$

This set of equations, like that for maximum path in a graph with no loops, forms the basis of a set $E_n$ that is open-loop consistent. The reason is that no equation in the equation set $E_n$ will have the same variable on both sides of the equation. This is because in the basis equations when $X_{ij}$ appears on the left and $X_{kp}$ appears on the right of an equation $j-i > k-p$, and so the same variable cannot appear on the right as on the left of any equation in $E_n$ since this property will be preserved even after substitution.

A process analogous to that described for maximum path can be used. The values of $X_{ij}$ with $i = j$ is known. By substituting these values, the solution of all $X_{ij}$ with $j-i = 1$ can be found. In general $X_{ij}$ with $j-i = k$ can be determined from the solutions for $X_{ij}$, $j-i = k+1$. 

II

until $x_{ln}$ is found. This process has a complexity of $n^5$.

Conclusions:

The study of the relation of recursive definitions to their 'good' implementation is a place in which many important concepts and results, developed in diverse regions of computer science, seem to come together. One has here a natural way of classifying algorithms according to properties of their recursive definitions which cuts across relatively superficial classifications according to applications areas. The strong analogy between recursive definitions and differential equations on the one hand and 'good algorithms' and closed (or otherwise good) solutions on the other hand, supports the expectations that this study is a place to bring it all together.
References:

1. Aho, Hopcroft, Ullman; The Design and Analysis of Computer Algorithms; Addison-Wesley, 1975; pp 195,222

2. Darlington, J. and Burstall, R.M.; A System Which Automatically Improves Programs; Proceedings Third International Joint Conference on Artificial Intelligence; Stanford, California, 1973; pp 479-485


7. Paull, M.C.; Formulation and Manipulation of Enumeration Based Algorithms; Research Report SOSAP-TR-4; December, 1973

Appendix I

Summary of Frequently used Notation

If P is a predicate the $\neg P$ means not P.

$\mathbb{N}$ is the set of all positive integers = \{1, 2, ..., \}

$\mathbb{N}$ is the finite set of integers from 1 to n = \{1, ..., n\}

If A and B are sets

$A \cup B$ is set union

$A \cap B$ is set intersection

$\overline{A}$ is the complement of A

$A - B = A \cap \overline{B}$

$|A|$ is the number of elements in A

$<a_1, ..., a_n>$ is an ordered set or vector with components

$a_i : i \in \mathbb{N}$ and $a_{i:j}$ represents the subvector $<a_i, a_{i+1}, ..., a_j>$; $a_{i:i} = <a_i>$

If A and B are ordered sets $=<a_1, ..., a_n>$ and $=b_1, ..., b_n>$ respectively

$A//B = <a_1, ..., a_n, b_1, ..., b_n>$

$\{A\}$ is the set of all components in A

If E, x and y are each an expression, i.e. a string or ordered set of symbols from a given alphabet, usually satisfying some constraints as to form, then

$E[x \to y]$ is the expression that results when each occurrence of x is replaced by y in E.

The notation is extended to allow the specification of a number of replacements $E[x \rightarrow y, z \rightarrow w]$ is the expression which results when each x is replaced by y and each z by w in E.
An entire set of replacements can also be specified, i.e. if $E$, $x_i$ and $y_i$ for all $i \in N$ are expressions

$E'[x_i + y_i | i \in N]$ is the expression that results when each occurrence of $x_i$ is replaced by $y_i$ in $E$ for all $i \in N$

The notation also composes to allow specification of multiple replacements, i.e.

$$(E[x + y]) [x_i + y_i | i \in N] = E'[x_i + y_i | i \in N]$$

where $E' = E[x + y]$.

The notation is also extended to specify replacement of 1 component of $E$.

Thus if $E$ and $x$ are expressions and $i$ an integer

$E[E_i + x]$ is the result of replacing the $i^{th}$ symbol in $E$ by the expression $x$

$E[E_i:i+k + x]$ is the result of replacing the sub expression of the $i^{th}$ thru $i+k^{th}$ symbols in $E$ with $x$.

A further extension allows specifying the insertion of a string between symbols

$E[E_{i+} + x] = E[E_{(i+1)-} + x]$ is the result of inserting $x$ between the $i^{th}$ and $(i+1)^{st}$ symbol of $E$. 
MEMORY EFFICIENT IMPLEMENTATIONS OF RECURSIVE DEFINITIONS

M. C. Paul

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
1. INTRODUCTION

Typically there are significant differences between the initial formulation of an algorithm and its ultimate implementation. For example the minimum path between two nodes in a weighted di-graph can be found by enumerating all paths between the two nodes and choosing the smallest. This approach can easily be formulated as a recursively defined function, which may in turn be implemented in a standard way. This is significantly different than Dykstra's algorithm, the favored shortest path implementation. On the one side, close to the problem statement, then there is an initial, simply formulated, but often inefficient algorithm. On the other side, nearer to the final implementation, is an efficient algorithm. The study of the connection between these two is the subject of this paper.

It will be assumed that the initial formulation of an algorithm is as a recursive definition and that this definition is in a standard form (to be given). The standard form was chosen because, firstly, it is one which, in our experience, has frequently arisen naturally as an initial algorithm formulation. Secondly the chosen form lends itself nicely to an overview of a variety of possible implementations of the algorithm thus formulated. The recursive definition though sufficient to provide the value of the function anywhere in its domain is non-deterministic as to which of a variety of sequential implementations are to be used to determine that value. The variety of implementations correspond to the various orders of substitution which are equally valid in evaluating such a definition.

Some orders of evaluation
become possible only if the primitive functions which enter into the recursive definition have appropriate properties. Different orders of evaluation will result in different memory requirements, but will not cause significant time differences in the resultant implementations. This dependence of memory requirements on the order of evaluation is the main subject of this paper.

Related Work

The work reported here is in an area of study in which there have been a number of significant publications. Strong has identified a class of recursive definitions for which memory efficient implementations \([5, 6]\) (called 'flowcharts') are available. This class is defined in terms of a recursive scheme whose constituent primitive functions are virtually unrestricted. If the properties of these primitive functions are restricted somewhat, a wider class of recursive definition forms will yield similar memory
efficient implementations. Such restrictions are considered here because they arise naturally in practice. So this aspect of the work can be considered an extension of Strong's results.

Burstall and Darlington studied properties of recursive definitions whose existence allows efficient implementation, with one objective being the incorporation of a search for such properties in an optimizing compiler. Later Burstall and Darlington extended this study to consideration of transformations of recursive definitions which are likely to produce better implementations. The spirit of our work here is largely in tune with that of these investigators with some significant differences in emphasis and in the particular properties studied. Our emphasis has been mainly on understanding the complete set of properties which allow the transformation from an initial recursive definition to the best algorithms actually known and to the proof of this connection. Thus we tend to consider a few relatively complex sets of properties and transformations as opposed to many simple ones. We also study mainly one form of first order recursive definitions, rather than the many forms they consider.

The remainder of this introduction is devoted to a sketch of the definitions and results to be detailed in the body of the paper.

---

1 First-order means a definition in which the defined function symbol never appears nested on the right.
Appendix I contains a summary of most of the notation used in the paper. (This notation is also defined on first use in the paper.)

The Standard Form

This paper concerns the implementation of recursive definitions of a function $f(X)$ in a class $F$ in which every definition has the following form:

\[
\begin{align*}
\text{if } T(X) & \quad \text{(terminal condition and values)} \\
f(X) &= q(X) & \text{(initially } X \in D_f) \\
\{ & \quad \text{if } T(X) \quad \text{(body)} \\
& \quad f(X) = w(f(o_1(X)), \ldots, f(o_m(X)(X))) \\
\} \\
\end{align*}
\]

where the data structure $X \in D_f$, primitive functions $w, q, o_i \in O, m$, and predicates $T$ in the definition collectively designated by the tuple $<D, w, q, O, m, T>$ must be constrained so as to make it a terminating definition.

A definition is terminating if for each $d \in D_f$ the sequence of expressions resulting from substitution for forms $f(a)$ (where $a$ is any expression using $I$ which starts with $f(d)$, $d \in D_f$, and next produces $w(f(o_1(d)), \ldots, f(o_m(d)(d)))$, etc. has the properties:

1. It is always possible to evaluate $T(a)$ and if $T(a)$ is false it is always possible to evaluate $m(a)$, and $o_i(a)$ for $1 \leq i \leq m(a)$.
Independent of the order of substitution for the different appearances of the form \( f(a) \) after the same finite number of such substitutions a 'terminal' expression will be obtained in which, for every appearance \( f(a) \), \( a \) is terminal (i.e. \( T(a) \) is true) and \( q(a) \) can be evaluated.

The function \( w \) is defined so as to make it possible to evaluate the terminal expression in any order consistent with its parentheses structure.

The tuples \(<D,w,q,0,1,m,I>\) which satisfy the above constraint are members of the set \( V \). The set of definitions of form \( I \) which satisfy these constraints constitute the recursive scheme \( F(V) \).

This form of definition often arises in practice as an initial solution to an algorithm design problem, particularly when the problem can be viewed as requiring an enumeration or an enumeration followed by a selection (search). The examples of recursive definitions in \( F(V) \) given below arose from adopting such a point of view. Their structure can be easily seen by evaluating them for some small initial values of their arguments.

Examples:

Ex. 1.1 If \( f(X) \) is to be the set of all \( n \) bit binary numbers (let \( N \) be the set of positive integers), then:

\[
X \in \{<a,n>|a \text{ a string of 0's and 1's}, n \in N\}
\]

\[
f(X) = f(a,n) = (a) \quad \text{if } n = 0
\]

\[
f(X) = f(a,n) = f(a/<0>,n-1) \cup f(a/<1>,n-1) \quad \text{if } r
\]

where \( / \) is string catenation, and \( \cup \) set union.

\( X \) initially \( \in \{<a,n>|n \in N\} \).

Then ex. \( f(\lambda,2) = f(<0>,1) \cup f(<1>,1) = (f(<0,0>,0) \cup f(<0,1>,0)) \cup f(<1>,1) \)

\[= \{<00>, f(<0,1>,0) \cup f(<1>,1) = \text{ etc.} \]
Ex. 1.2 If \( f(X) \) is the set of all permutations of the first \( n \) integers, 

\[
X \in \{ \langle n, a \rangle | n \in \mathbb{N}, a \text{ is a string of positive integers} \}
\]

\[
f(X) = f(n, a) = \{a\} \quad \text{if } n = 0
\]

and if \( p = |a| = \) the length of \( a \) then

\[
f(X) = f(n, a) = f(n-1, a[a_0+n]) \cup \ldots \cup f(n-1, a[a_p+n]) \text{ if } n > 0
\]

where \( a[a_i+n] \) is an inserting function; i.e.

if \( a = \langle a_1, \ldots, a_p \rangle \) then \( a[a_i+n] \) is the result of inserting the

integer \( n \) after component \( a_i \) in \( a \) or is \( \langle a_1, \ldots, a_{i-1}, n, a_{i+1}, \ldots, a_p \rangle \).

\( X \) is initially \( \epsilon \) \( \{ \langle n, \lambda \rangle | n \in \mathbb{N} \} \)

Then ex. \( f(2, \lambda) = f(1, \langle 2 \rangle) = f(0, \langle 12 \rangle) \cup f(0, \langle 21 \rangle) = \{ \langle 12 \rangle \} \cup f(0, 21) \)

Ex. 1.3 \( f(X) \) is the string of moves (each a pair of numbers \( <a, b> \)

meaning move a disc from pin \( a \) to pin \( b \)) necessary to optimally

solve the, now classical, Tower of Hanoi puzzle. To move \( n \)

discs initially on pin 1 to pin 2:

\[
X \in \{ \langle <x, y, z>, n \rangle | <x, y, z> \text{ is a permutation of } \langle 1, 2, 3 \rangle, n \in \mathbb{N} \}
\]

\[
f(\langle <x, y, z>, n \rangle) = \langle x, y \rangle \quad \text{if } n = 1
\]

\[
f(\langle <x, y, z>, n \rangle) = f(\langle <x, z, y>, n-1 \rangle) \cup f(\langle <x, y, z>, 1 \rangle)
\]

\[
f(\langle <x, y, z>, n \rangle, n-1 \rangle) \quad \text{if } n > 1
\]

\( X \) is initially \( \epsilon \) \( \{ \langle 1, 2, 3 \rangle, n \rangle | n \in \mathbb{N} \} \)

Algorithms to Implement Definitions in \( F(V) \) which are Efficient in Use of Memory

An 'algorithm scheme' defining a set of algorithms is defined in a manner

analogous to that used in defining a recursive scheme like \( F(V) \). In this paper

algorithm schemes generally will involve standard assignment and conditional

statements using the same unspecified set of data-structures \( D \), primitive

functions \( w, q, 0, m \) and predicate \( T \)(designated by the tuple \( \langle D, w, q, 0, m, T \rangle \)
used in defining the recursive scheme \( F(V) \). If we constrain the selection of tuples to be a member of a set \( V \), the set of algorithms thus defined is designated \( S(V) \) and a particular algorithm \( c \in S(V) \), corresponding to a tuple \( v \in V \) is designated \( S(v) \). The recursive and algorithm scheme \( F(V) \) and \( S(V) \) are equivalent iff for each \( v \in V \), \( F(v) \) is equivalent to \( S(v) \). A recursive function definition \( F(v) \) and an algorithm \( S(v) \) are equivalent if with domain \( D \) in \( v \), for every \( d \in D \), the value of \( f(d) \) as computed with recursive definition \( F(v) \) = value of the result of running the algorithm \( S(v) \) with \( d \in D \) as its initial value.

The main purpose of this paper is to show that for a set \( V' \) built from \( V \) by constraining the function \( w \) to be 'associative' and the set of functions \( O \) to have an 'inverse', there is an algorithm scheme \( S(V') \) equivalent to \( F(V') \) which is particularly efficient in its use of memory. The algorithm scheme available when these conditions are satisfied is given in figure 2.2. The algorithm scheme \( S(V') \) is given in terms of the data-structures, primitive functions and transformations of these primitive functions (inverse of \( O \) for example) which are immediately available under the assumption of the existence of an 'inverse', that appear in the equivalent recursive scheme \( F(V') \).

For many of the recursive definitions in the class \( F(V') \), the equivalent member of the class \( S(V') \) - which can be obtained mechanically from the recursive definition is the 'good' algorithm usually used to realize that definition. Thus corresponding to example 1.1, the algorithm obtained by instantiation of that particular \( D, w, q, \varphi, m \) and \( T \) in \( S(V') \) is one in which:

---

2 These terms are defined in section 2. An inverse operation plays a similar role in [6]. Our 'inverse', however, is different, having been independently developed [7,8] in combination with associativity to delineate another class of definition with efficient implementations. The result is in Theorem 2.2.
First a string of \( n \) 0's is formed and outputted - being the first binary number produced, then, because the rightmost symbol in the string is a 0 it is changed to a 1 and the result outputted. In general, the algorithm remembers the last binary number formed and outputted, say \( X \). The next binary number is formed by a scan of the bits of \( X \) starting with the rightmost bit, and changing them by the following scheme. Let \( b \) be the bit under scrutiny - if \( b \) is a 0 it is changed to a 1 and the result is the next binary number to be outputted - if it is a 1 it is changed to a 0, \( b \) becomes the bit in \( X \) one position to the right of the current \( b \) and the scrutiny is repeated. When the leftmost bit of a number \( X \) becomes \( b \) and that bit \( = 1 \) then the process terminated. In summary this algorithm for producing all \( n \)-bit binary numbers, consists simply in 'adding 1' to produce successive members of the set. It is the 'good' algorithm for producing the set. It keeps in memory only the last number produced thus using an amount of storage roughly equal to that required to hold the argument of \( f \) in its recursive definition. This is characteristic of all the algorithms in \( S(V') \) in relation to the equivalent member of \( F(V') \) and is the 'memory efficiency' mentioned.

In a similar way, the algorithm for example 1.2 obtained by instantiation of the primitives that appear in the recursive definition in example 2.2 produces one permutation at a time. A permutation is produced from the previous permutation by interchange of adjacent terms. This again is the 'good' algorithm for generating permutations.

Creating an Inverse

In examples 1.1 and 1.2, the given O-functions had an inverse - in example 1.3 the O-function as given does not have an inverse and thus the
In a recursive definition, \( f \in F(V) \), the 0-function does not have an inverse - a simple transformation of \( f \) to an equivalent definition, say \( f' \), involving an 0-function having an inverse can always be found in \( F(V') \). Thus \( f' \) will have an equivalent in \( S(V') \). This new definition \( f' \) is equivalent to \( f \) in the sense that to each argument \( d \) of \( f \) there is a 'simply' computed argument \( d' \) of \( f' \) such that \( f'(d') = f(d) \). Using this transformation, an equivalent definition to that of example 1.3 will be given subsequently, whose equivalent algorithm in \( S(V') \) will produce the moves necessary to solve the Tower of Hanoi problem - one at a time, the only temporary memory necessary being that for a record of the previous move and its number.

**Memory Efficiency**

In the standard compiler implementation of a recursive definition of the form of I, that definition is taken to describe a procedure which calls itself. The procedure uses a stack to temporarily remember, amongst other things the set of arguments (the data structure) associated with the call. The size to which the stack grows varies and depends on the depth of the calls. In general, if the definition is non-linear, i.e. has more than 1 call of the defined function on the right, then the arguments of the \( w \) functions will have to be stacked also. When the memory efficient algorithm, to be described here, is applicable then both of these stacks can be eliminated. Instead only 1 copy of the argument of the calling function will be saved. All other temporary memory uses in the algorithm are comparable to those in the standard implementation. It will be possible to eliminate the need for these stacks for any definition of form I, provided, only, as we have said that the \( w \) function is associative and the 0 functions have a uniform inverse.

Although the 'memory efficient' algorithms of \( S(V') \) are honestly so for the most part, the nature of the memory efficiency can be misleading. The implementing algorithm available when \( w \) is 'associative' and the 0-function has an 'inverse' is efficient in the sense that the memory required is usually of the order of the largest storage required for the argument (also called a data structure) of \( f \) which arises if \( f \) is evaluated by successive substitutions.

Usually this largest data-structure for which memory need be provided...
requires a small amount of memory relative to the total of all data-structures produced during the implementation of the definition for a given initial data-structure - ex. of the order of a single member of a set when a set is being enumerated. Even when the 'inverse' does not exist it can be incorporated as previously noted, leaving the 'memory efficiency' notion still viable. However there is another way of obtaining a 'memory efficient' equivalent algorithm which is deceiving.

This technique involves obtaining a technically correct equivalent recursive definition of \( f \), say \( f' \), having only one occurrence of \( f' \) on the right, but in compensation involving much larger data structures \( X' \) and complex function \( o'_i \) than the corresponding \( X \) and \( o_i \) of \( f \). That is, for each definition of form I there is an equivalent definition of the form:

\[
\begin{align*}
\text{II} & \quad \begin{cases} 
  f'(X') = q'(X') & \text{if } T(X') \\
  f'(X') = w(f'(o'(X'))) & \text{if } T(X') \\
\end{cases} \\
\text{Initially, } X' \in D_{f'}
\end{align*}
\]

By equivalent, we mean that there is a 1-1 correspondence \( g \) between \( D_f \) and \( D_{f'} \), so that for each \( d \in D \):

\[ f(d) = f'(g(d)) \]

If \( f' \) has an inverse then it can be realized in the same memory efficient manner as other definitions in \( F(V') \) and if not it can easily be modified so as to have one while still keeping the result in the form of II. Memory efficiency, however, means that the memory requirement will not exceed the size of the largest data-structure which arises as an argument of \( f \) during evaluation of \( f' \). But in this equivalent definition that

---

4 The two classical ways this can be done are by constructing a general breadth-first or depth-first algorithm to implement the recursive definition of form I and then equivalently giving these as recursive definitions,
data-structure is typically much larger (often exponentially) than that which could arise in the original definition.

The term 'memory efficiency' as used here then requires caution in its application.

2. MEMORY EFFICIENT IMPLEMENTATIONS

The first part of this section, thru page 16, is largely devoted to material which is probably familiar. This is done inorder to develop the definitions of a number of terms which are used later in this section. Altho the concepts are familiar the terms we use may not always be so.

In any case, the material in these preliminary pages can easily be skipped and only referred to to pick up definitions of terms used later, without losing the main point of the paper.
Definition of Standard Recursive Scheme F:

Consider the set $F'$ of all functions $f$ that can be defined as follows:

Def. 2.1

\[
\begin{align*}
  f(x) &= q(x) & \text{if } T(x) \\
  f(x) &= w(f(o_1(x)), \ldots, f(o_m(x))) & \text{if } T(x) \\
  \text{initially } x \in D_f
\end{align*}
\]

where the primitive functions and predicates which are used in the definition are weakly constrained as to the nature and extent of their domains and ranges. $D_f$ is the set of initial data-structures and may be any set. Other sets must be included in some of the domains of some of the primitive functions. These other sets are defined recursively, using the primitive functions. First these sets are named and their relation to the primitive functions given, then they are defined.

- $m$ is a function whose domain must include the set $\Delta_f$ and whose range is the positive integers $\geq 1$. $m(x) \geq 1$ for all $x \in \Delta_f$.

- $O$ is a set of functions $\{o_1, o_2, \ldots\}$.
  
  The domain of $O$ must include the set $\delta^1$. $\Delta_f$ is the union of all the domains of the functions in $o_1$ and is called the domain of the $O$-function.
  
  The range of $o_1$ must include the set $p^1$.
  
  The union of the sets $p^1$ of all the functions in $O$ is the range of the $O$-function and is called $P_f$.

- $T$ is a predicate whose domain includes $D_f \cup P_f$. Its range is $\{\text{true}, \text{false}\}$.

- $q$ is a function whose domain must include $Q_f$. Its range may be any set, say $W_f$. 

\( W \) is a function whose range is called \( W_x \) and whose domain must include 
\[ U_y \cup u \]

The sets named above are defined as follows (the subscript \( f \) is dropped where it is not essential):

\[ \Delta^1 = \{d|d \in D \text{ and } T(d)\}; \text{ and for } j > 1 \]
\[ \Delta^j = \{o_i(X)|X \in \Delta^{j-1} \text{ and } i \leq m(X) \text{ and } T(o_i(X))\} \]
\[ \Delta = \bigcup_{i=1}^{\infty} \Delta^i \]

The set \( \delta^i \) of \( o_i \in \Delta \) is:
\[ \delta^i = \{X|X \in \Delta \text{ and } i \leq m(X)\} \]

The range of \( \Delta \) is:
\[ p = \{o_i(X)|X \in \Delta \text{ and } i \leq m(X)\} \]

The range \( p^i \) of \( o_i \in \Delta \) is:
\[ p^i = \{o_i(X)|X \in \delta^i\} \]

The set of terminal data-structures \( Q \) is:
\[ Q = P - \Delta \]

The set \( W \) is defined as follows:
\[ W^1 = \{w(x_1, \ldots, x_n)|x_1 \in w_x, n \text{ is a positive integer } N\}; \text{ for } j > 1, \]
\[ W^j = \{w(x_1, \ldots, x_n)|x_1 \in W^k, k(j); n \in N\} \]
\[ W = \bigcup_{i=1}^{\infty} W^i \]

If in addition to being a member of the set \( F \), a recursive definition is terminating as defined below it is a member of the set \( F \). We need some preliminary definitions.

If \( <i_1, \ldots, i_n> = 1 \) is a sequence of integers then \( o_{<i_1, \ldots, i_n>}(X) = o_{i_n}(X) \).

\( o_i(X) \) is an abbreviation for \( o_{i_n}(o_{i_2}(o_{i_1}(X)) \ldots); o_\lambda(X) = X. \)
A length 1 sequence of integers \( <i_1> \) is applicable to a data-structure \( X \in \Delta_f \) if \( i_1 \leq m(X) \). A length \( n \) sequence of integers \( <i_1, \ldots, i_n> \) is applicable to a data-structure \( X \) if \( <i_1, \ldots, i_{n-1}> \) is applicable to \( X \) and \( <i_n> \) is applicable to \( o_{<i_1, \ldots, i_{n-1}>}(X) \).

\( f(X) \) is the set of all integer sequences applicable to \( X \in \Delta_f \).

\( f \) is terminating iff \( \forall d \in D \), \( d \in D \) implies \( T(d) \) is finite.

Note that if \( f(X) \) is finite it cannot contain an infinite sequence, because it always contains all prefixes of any sequence it contains.

This completes our definition of \( f \). Next we give some simple consequences of the definition which will be used later. First, the substitutionally solvable property that \( d \in D \), \( T(d) \) is finite can be extended to any \( X \in \Delta_f \). This is done in lemmas 2.1 and 2.2.

**Simple Properties of \( f \in F \)**

**Lemma 2.1:** If \( f \in F \) and \( X \in \Delta_f \) then \( \exists \) an integer sequence \( I \in f(d) \) and \( \exists \) a data-structure \( d \in D \) such that \( o_{I}(d) = X \).

**Proof:** If \( X \in \Delta_f \) then obviously there exists some \( c \) (at least 1) such that \( X \in \Delta^c \). The lemma is proven by induction on the sets \( \Delta^c \). Assuming there is a length \( k-1 \) sequence \( I_Y \) for each data-structure \( Y \in \Delta^{k-1} \) and \( d \in D \) such that \( o_{I_Y}(d) = Y \). Then it follows, by definition of \( \Delta^k \) that if \( X \in \Delta^k \) then \( X = o_k(Y) \) for some \( i \leq m(Y) \), and \( Y \in \Delta^{k-1} \). Thus \( X = o_i(o_{I_Y}(d)) = o_{<I>}(d) \). Since also \( D = \Delta^1 \), and \( o_{I_Y}(d) = d \) for each \( d \in D \), the proof is complete.

**Lemma 2.2:** If \( f \in F \) and \( X \in \Delta_f \), then \( f(X) \) is finite.

**Proof:** From the previous lemma the data-structure \( X = o_I(d) \) for some \( d \in D \) and integer sequence \( I \). Therefore \( f(d) \geq \) the

\* \( F \) as defined here is the same as \( F(V) \) as defined in the Introduction.
set consisting of $I$ concatenated with each member of $I(X)$. Thus if $I(X)$ is not finite, $I(d)$ cannot be finite but this contradicts the condition that $f \in F$ is substitutionally solvable.

Another consequence of the definition of $F$ is that the data-structures in $\Delta_f$ can be usefully ordered in another, almost reverse, manner than the ordering by membership in the subsets $\Delta^j$. In most of the subsequent inductive proofs, induction will be carried out on this ordering.

Ordering the Data-Structures in $\Delta$ (Remoteness):

For any function $f$ in $F$:

We say a data-structure $X$ in $\Delta_f \cup Q_f$ is of remoteness $0$ (or is terminal) if $X \in Q_f$.

We say a data-structure $X$ in $\Delta_f \cup Q_f$ is of remoteness $n$ if:

1. $\exists i : i \leq m(X)$ and $o_i(X)$ is of remoteness $n-1$ and
2. $\forall i : ism(X)$ implies $o_i(X)$ is of remoteness $n-k$ and $k \geq 1$.

Lemma 2.3: If $f \in F$, then there is a function $r$ with domain $\Delta_f \cup Q_f$ such that if $X \in \Delta_f \cup Q_f$ then $r(X) = \text{the remoteness of } X$.

Proof: For each $X \in \Delta_f \cup Q_f$ let $r(X)$ be the maximum of the length of all the sequences in $I(X)$. For each $X \in \Delta_f \cup Q_f$, $X$ is of

6 Alternately this can be phrased 'of remoteness $< n$'.
remoteness \( r(X) \). This is shown by induction. If \( T(X) \) then \( T(X) \) is empty and \( r(X) = 0 \). Assume that if \( r(X) < n \), \( X \) is of remoteness \( r(X) \). Let \( r(X) = n \), i.e. there is a longest sequence of length \( n \), say \( I = <i_1, \ldots, i_n> \) in \( T(X) \). Let \( o_i(X) = Y \). Then \( I' = <i_2, \ldots, i_n> \) is in \( T(Y) \). Furthermore, no sequence applicable to \( Y \) is longer than \( I' \) because otherwise \( I \) could not have been a longest sequence in \( T(X) \). So \( r(Y) = n-1 \) and \( Y \) is of remoteness \( r(Y) = n-1 \). Therefore, since \( o_i(X) = Y \) and for all \( j \neq i \), \( j < m(X) \), \( r(o_j(X)) \leq n-1 \), \( X \) is of remoteness \( r(X) = n \) by definition of remoteness.

Properties of \( f \in F \) Sufficient for Memory Efficient Implementations

An efficient implementation becomes available when the recursive definition \( f \in F \) has some special properties. These properties are now defined.

**Associativity:** Associativity has the usual meaning here. The function \( w \) is associative if:

\[
 w(a_1, a_2, \ldots, a_m) = w(w(a_1, a_2), a_3, \ldots, a_m) \quad \text{for} \quad m \geq 3
\]

\( w = \) minimum, sum, catenation and union provide examples of \( w \)-functions with this property. In each case one can compute \( w(a_1, \ldots, a_m) \) as follows:

\[
\begin{align*}
  & X = X \\
  \text{For } i = 1 \text{ to } m \\
  & Y = w(X, a_i) \\
  & X = Y \\
\end{align*}
\]

End
thus requiring at any one time memory for at most 2 copies of the result of $w(a_1, \ldots, a_j), j \leq m$. If $w$ is the function minimum, this memory does not increase on the number, but only on the value of its arguments, $a_i$. If $w$ is catenation, sum, or union the memory required will increase, albeit at different rates, with the number of arguments. There is, however, a significant difference in use of the memory, between a computation of catenation and of union. To obtain catenate $(a,b)$, $b$ needs only be attached at the end of $a$. To obtain the union $(a,b)$, $a$ must be searched for an occurrence of a member of $b$. If $a$ represents the result of a previous computation then in the union case it is necessary to re-access this memory whereas this is not necessary in the catenation case. This is an important consideration because memory that is not re-accessed can be located in areas of memory (disc) which need not be easy to access (as is core). The temporary memory requirements for the implementation of a function then do not depend on the usual mathematical properties of that function only, but also depend on the means available for accessing the memory. Nevertheless, for compactness our results are given in terms of the usual mathematical properties—so caution is needed in their interpretation.

**Uniform Inverse:**

Consider a set of functions $H = \{h_1, \ldots, h_M\}$. Let $D_i$ be the domain over which $h_i$ is defined and let $R_i$ be the corresponding range of $h_i$.

Then we will say $D = \bigcup_{i=1}^{M} D_i$ is the domain of $H$ and $R = \bigcup_{i=1}^{M} R_i$ is its range.

The set of functions $H$ is said to have a uniform inverse on the domain

---

7 It is also true that there may be some advantage in time efficiency in one grouping of the arguments of $w$ over another though both give the same result when $w$ is associative. An example of such a function is merge, i.e. $\text{merge}(a_1, \ldots, a_m)$ in which $a_i$ are each finite sorted sets of numbers.
domain $D$ if:

1. Every $h_i \in H$ has an inverse and
2. $R_i \cap R_j = \emptyset$ for every $R_i \neq R_j$ in $R$.

If $H$ has an uniform inverse then it is easy to see that the following two 'uniform inverse' functions on $R$ exist for $r \in R_i \subset R$.

1. $H^{-1}(r) = d \in D$ such that $H_i(d) = r$
2. $i_H(r) = i$ the index of the range $R_i$ of which $r$ is a member.

A recursive definition, $f \in F$, has a uniform inverse if the set of functions $O_i \in O$ in $f$ has a uniform inverse.

For a given function set $O$ it is possible that none, one, or two of the pair $H^{-1}, i_H$ exist. Despite the fact that the uniform inverse is a strong condition it does often occur. Furthermore when it doesn't, there is always a strongly equivalent definition which does have a uniform inverse. This is shown after a short degression required to define strong equivalence.
Equivalence of Recursive Definitions:
Consider two definitions in F:

(1) \( f \) on domain \( D \)
\[
\begin{align*}
    f(X) &= q(X) & \text{if } T(X) \\
    f(X) &= w(f(o_1(X)), \ldots, f(o_m(X))) & \text{if } \overline{T}(X)
\end{align*}
\]
Initially \( X = d \in D \)

(2) \( g \) on domain \( D' \)
\[
\begin{align*}
    g(X') &= q'(X') & \text{if } T'(X') \\
    g(X') &= w'(g(o_1'(X')), \ldots, g(o_m'(X'))) & \text{if } \overline{T'}(X')
\end{align*}
\]
Initially \( X' = d' \in D' \)

If there is a 1-1 correspondence between \( D \) and \( D' \) such that whenever \( d \in D \) and \( d' \in D \) are two corresponding data-structures \( f(d) = g(d') \) then the two definitions are equivalent. The above correspondence may be extended to one between \( \Delta_f \) and \( \Delta_g \) with \( \delta \in \Delta_f \) corresponding to \( \delta' \in \Delta_g \) by having \( o_1(\delta) \) correspond to \( o_1'(\delta') \) whenever \( \delta \) corresponds to \( \delta' \) and \( o_1(\delta) \) and \( o_1'(\delta') \) are both defined. This is called a structural correspondence. If in addition to such a structural correspondence of \( \Delta_f \) to \( \Delta_g \) the following conditions hold

(1) \( T(\delta) = T'(\delta') \)
(2) \( q(\delta) = q'(\delta') \) if \( T(\delta) \) (and \( T'(\delta') \))
(3) \( m(\delta) = m'(\delta') \) if \( \overline{T}(\delta) \) (and \( \overline{T}'(\delta') \))
(4) \( w = w' \)

then \( f \) and \( g \) are strongly equivalent.
Strong equivalence of two definitions implies that they not only give the same results but also require the same number of substitutions in their evaluation for corresponding initial arguments.

As an example of a strong equivalence, consider the two functions $f$ and $g$ each in $F$:

\[(1) \begin{cases} f(X) = q(X) & \text{if } T(X) \\ f(X) = w(f(o_1(X)), \ldots, f(o_m(X))) & \text{if } T'(X) \\ \text{initially } X = d \in D \end{cases}\]

\[(2) \begin{cases} a) \ g(X,Y) = q(X) & \text{if } T(X) \\ b) \ g(X,Y) = w(g(o_1(X),h_1(Y)), \ldots, g(o_m(X),h_m(Y))) & \text{if } T'(X) \\ \text{initially } <X,Y> = <d,y_d> \in D' \text{ with } d \in D \text{ and } y_d \in \text{a set } \mathcal{H} \\ \mathcal{H} = \{h_1, \ldots, h_n\} \text{ is a set of primitive functions} \end{cases}\]

Let data-structure $d \in D$ correspond to $<d,y_d> \in D'$. Extend this correspondence to one between $\Delta_f$ and $\Delta_g$ by letting $o_i(X) \in \Delta_f$ correspond to $<o_i(X), h_i(Y)> \in \Delta_g$ whenever $X \in \Delta_f$ corresponds to $<X,Y> \in \Delta_g$ and $T(X)$ and $i \leq m(X)$. For example if $d \in D$ and $o_i(d)$ is defined then it corresponds to $<o_i(d), h_i(y_d)> \in \Delta_g$.

For each member of $\Delta_f$ this correspondence defines a corresponding member of $\Delta_g$. This follows because every member in $\Delta_f$ is either in $D$, for which the correspondence is given explicitly, or it $= o_i(X)$ for $X \in \Delta_f$ and $o_i$ is defined and $T(X)$, in which case the correspondence to a member of $\Delta_g$ is given since $o_i(X,Y)$'s existence just depends on $X$, because $m'(X,Y) = m(X)$, $T'(X) = T(X)$.

Conditions (1) through (4) are obviously satisfied for this correspondence in the above definitions. Furthermore, the function $g(X,Y)$ is
independent of \( Y \), its second argument. This is shown inductively as follows. Directly from the definition (2a) we see that \( g(X,Y) \) is independent of \( Y \) when \((X,Y)\) is of remoteness 0. Its being of remoteness 0 is also independent of \( Y \). Referring to (2b), if it is assumed that each term \( g(o_i(X),h_i(Y)) \) appearing on the right is independent of its second argument then it follows certainly that \( g(X,Y) \) on the left of (2b) is independent of \( Y \). If the argument on the left side of (2b) is of remoteness \( n \) from terminal then all the arguments of terms on the right are of remoteness \(< n \) from terminal. Thus the inductive argument is completed concluding that \( g(X,Y) \) is independent of \( Y \) if \( X \) and thus if \((X,Y)\) is of remoteness 0,1,2, ..., \( n \).

Thus definition (2) can be rewritten removing \( Y \) which with \( f \) replacing \( g \) is the same as (1). Therefore:

**Lemma 2.5** \( g \) and \( f \) above are strongly equivalent

Since the value of \( g(X,Y) \) is independent of \( Y \) it may seem silly to ever construct such a definition, with a 'redundant' \( Y \), to replace \( f \), or alternatively that such a redundant \( Y \) would arise inadvertently in \( g \) to be removed by replacement with the equivalent \( f \). The following theorem, however, demonstrates that such 'redundant' additions can be of considerable use.
Theorem 2.1: For any recursive definitions \( f \) in \( F \) there is a strongly equivalent definition in \( F \) which has a uniform inverse.

Proof: If \( f \) already has a uniform inverse it serves as its own strongly equivalent definition. If not the following definition serves that purpose. Referring to Def. 2.1 for the definition of \( f \), the following function \( g \) defined in terms of the same sets, primitives and predicates is strongly equivalent to \( f \). (\( \rho = \langle \rho_1, \ldots, \rho_n \rangle \) is a vector which records indices, and \( d \) is the initial data structure.)

\[
\begin{cases}
  g(X, \rho, d) = q(X) & \text{if } T(X) \\
  g(X, \rho, d) = w(o_1(X), \langle 1 \rangle \parallel \rho, d), \ldots, g(o_m(X), \langle m(X) \rangle \parallel b, d) & \text{if } \bar{T}(X) \\
  \text{initially } \langle X, \rho, d \rangle = \langle d, \lambda, d \rangle \text{ with } d \in D.
\end{cases}
\]

\( g \) is strongly equivalent to \( f \) by application of lemma 2.5, with \( Y \) of that lemma corresponding to \( \{ \langle p, d \rangle : p \text{ a sequence of integers, } d \in D \} \), and \( y_d \) corresponding to \( \langle n, d \rangle \) with \( d \in D \). Furthermore \( g \) has a uniform inverse which is given by the following:

\[
\begin{align*}
  i_0(X, \rho, d) &= \rho_1 \\
  o^{-1}(X, \rho, d) &= \langle \rho_2 ( \ldots (o_{n-1} (o_n (d)) ) , \rho[1 \leftarrow \lambda], d >
\end{align*}
\]
The $0^{-1}$ function is quite complex, requiring recreating a sequence of data-structures starting with the initial data structure. In practice one wants to construct a strongly equivalent definition which gives an inverse but entails the creation of an $0^{-1}$ function which is simple. Simpler, hopefully, than that given in the above theorem. This can often be done. If, for example,

Corollary 2.1: For a given recursive definition $f \in F$ there is no uniform inverse, but each function $o_i \in O$ has an inverse $= o^{-1}_i$, then the definition for $g$ given above with the third component $d$ deleted from its arguments will serve with the additional benefit that an alternative simpler definition of $o^{-1}(X,o) = o^{-1}_i(X,o[0^{-1}n])$ can be used.

This corollary can be applied to the 'Tower of Hanoi' definition ex.1.3. In that example, $o_i \in O$ has an inverse for $i = 1$ and $3$ but does not quite have an inverse when $i = 2$:

- $o^{-1}_1(<x,y,z>,n) = <x,z,y>,n+1>$
- $o^{-1}_2(<x,y,z>,n) = <x,y,z>,A$ where $A$ cannot be determined from $<x,y,z>,n$
- $o^{-1}_3(<x,y,z>,n) = <z,y,z>,n+1>$

So first we slightly modify the definition of $f$ so there will be an inverse for $o_2$. Lemma 2.5 justifies this simple modification in which a component $s$ is added to store the quantity $A$ above when $i = 2$, and otherwise to remain equal to 0.
\[
\begin{aligned}
    f'(<<x,y,z>,n>,s) &= <x,y> & \text{if } n = 1 \\
    f'(<<x,y,z>,n>,s) &= f'(<<x,z,y>,n-1>,s) / f'(<<x,y,z>,1>,n) / \\
    \quad & \quad \quad f'(<<z,y,x>,n-1>,s) \text{ if } n > 1 \\
\end{aligned}
\]

initially \((<<x,y,z>,n>,s) = <<1,2,3>,n>,0>, n \in \mathbb{N}\)

Now \(f\) is equivalent to \(f\) in 1.3 and \(o_i\) has an inverse for \(i = 1, 2, \text{ or } 3\).

These inverses are:

\[
\begin{aligned}
    o_1^{-1}(<<x,y,z>,n>,s) &= <<x,z,y>,n+1>,0> \\
    o_2^{-1}(<<x,y,z>,n>,s) &= <<x,y,z>,s>,0> \\
    o_3^{-1}(<<x,y,z>,n>,s) &= <<z,y,x>,n+1>,0> \\
\end{aligned}
\]

Corollary 2 now applies to \(f'\). Its application yields \(g\) below.

(Some unnecessary \(>\)'s and \(<\)'s have been dropped.)

\[
\begin{aligned}
    g(<x,y,z>,n,s,p) &= <x,y> & \text{if } n = 1 \\
    g(<x,y,z>,n,s,p) &= g(<x,z,y>,n-1,s,<1> // p) / g(<x,y,z>,1,n,<2> // p) / \\
    \quad & \quad \quad / g(<z,y,x>,n-1,s,<3> // p) \text{ if } n > 1 \\
\end{aligned}
\]

initially \(<x,y,z>,n,s,p> = <<123>,n,0,1>>

and the uniform inverse is given by

\[
\begin{aligned}
    i_o(<x,y,z>,n,s,p) &= \rho_1 \\
    o_i^{-1}(<<x,y,z>,n,s,p>) &= <o_i^{-1}(<<x,y,z>,n,s>,\rho_{[\rho_1+1]}> \\
\end{aligned}
\]
Implementation of \( f \in F \) with Associativity and Uniform Inverse

We will give a way of implementing any \( f \) in \( F \) which has a uniform inverse and in which \( w \) is associative. The implementation is described by a flowchart containing, as usual, interconnected assignments and decision statements. The expressions in the assignment statements and decisions are compositions involving the primitive functions and predicates \( w, q_i \in O, m, q \) and \( T \) and the inverses \( 0^{-1}, i_0 \) which enter the definition of \( f \in F \). In addition to the above functions from the definition of \( f \), the repertoire of flowchart expression is completed by an add \( 1 \) function, a push and pop and an \( = \) predicate. There is a storage cell \( X \) which is assumed adequate to hold any member in \( A_F \cup Q_F \cup D_F \). Although there is such a push and pop, the list on which they operate, \( V \), can hold at most only 2 members in \( W_F \cup w_F \).

The flowchart which follows describes a computation for each \( d \in D \). It is necessary to give a concrete interpretation of the sense in which a flowchart describes a computation. We imagine a traveler who starts by entering block \((0)\) of the flowchart. The traveler carries out the computation described in that block then, depending on the nature of the block, proceeds to the appropriate next block. The traveler continues following the block instructions and proceeding through the flowchart until \( \text{FINI} \) is reached completing the voyage. The value found in \( V \) when the traveler has completed the voyage is the value computed by the flowchart.
Flowchart: notation and assumptions

In the flowchart we will use the following notation. General:

\( e \) is an expression

\[
\begin{align*}
X \leftarrow e & \quad \text{the value of} \ e \ \text{is assigned to} \ X \\
V \leftarrow \text{PUSH} \ e & \quad \text{the value of} \ e \ \text{is pushed into list} \ V \\
X \leftarrow \text{POP} \ V & \quad \text{the top member of} \ V \ \text{is popped and assigned to} \ X \\
X \leftarrow \text{POP} \ V[n] & \quad \text{the top} \ n \ \text{members of} \ V \ \text{are popped and assigned to} \ X \\
\end{align*}
\]

If \( V \) is a list = \(<v_1, v_2, \ldots, v_n>\) then \( w(V) \) stands for the expression \( w(v_1, v_2, \ldots, v_n) \).

Primitives and their Compositions: (Some of the definitions are extended to \( \mathcal{Q}_f \) to make the flowcharts work if the initial data-structure is terminal.)

<table>
<thead>
<tr>
<th>Flow chart Notation</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| FIRST.KID(X)        | \begin{align*} \circ_1(X) & \text{if} \ X \in \Delta_f \\
| #KIDS(X)            | \begin{align*} m(X) & \text{if} \ X \in \Delta_f; = 1 \text{if} \ X \in \mathcal{Q}_f \\
| X = \text{TERMINAL?} | \begin{align*} T(X) & \text{if} \ X \in \Delta_f \cup \mathcal{Q}_f \\
| PARENT(X)           | \begin{align*} \sigma^{-1}(X) & \text{if} \ X \in \Delta_f; = X \text{if} \ X \in \mathcal{Q}_f \\
| SIB#(X)             | \begin{align*} l_0(X) & \text{if} \ X \in \Delta_f; = 1 \text{if} \ X \in \mathcal{Q}_f \\
| NEXT.SIB(X)         | \begin{align*} \circ_{SIB#}(X)(\text{PARENT}(X)) & \text{if} \ X \in \Delta_f \\
| #SIBS(X)            | \begin{align*} \#\text{KIDS(PARENT}(X)) & \text{if} \ X \in \Delta_f; = 1 \text{if} \ X \in \mathcal{Q}_f \\
\end{align*}

If \( w \) is associative we assume that there is a member \( Q_w \) in the range of \( w \) such that \( w(X, Q_w) = X \) for all \( X \) in the range of \( w \).
Flowchart:

For $f \in F$ and $f$ has a uniform inverse and $w$ is associative.

Figure 2.1
When we say a flowchart implements or realizes an \( f \in F \) we mean that for each \( d \in D \) the evaluation of the function \( f(d) \) is to the value computed by the flowchart with traveler starting at block 0 and \( d \) in the flowchart = to \( d \) in \( f(d) \).

We now present proofs that the given flowchart does implement \( f \in F \) under the appropriate constraints. The proof uses induction on the remoteness of the data-structures in \( \Lambda_f \).

**Theorem 2.2:** If \( f \in F \) and \( f \) has a uniform inverse and \( w \) in the definition of \( f \) is associative then \( f \) is implemented by the flowchart of (figure 2.1).

**Proof:** First we need to show that if \( A \) of remoteness \( n \), block 1 of the flowchart of figure 2.1 is entered with \( A \) in \( X \) and \( B \) in \( V \) then eventually the traveler arrives at 9 with \( A \) still in \( X \) and with \( V \) containing

\[
\{w[w[B,f(o_1(A))],f(o_2(A))], \ldots , f(o_m(A)(A))] = w[B,f(A)]
\]

by associativity

\[
w[B,w[f(o_1(A)), \ldots , f(o_m(A)(A))]] = w[B,f(A)]
\]

We use induction. The case when \( A \) is of remoteness 1 is easily verified by tracing the flowchart through the sequence of blocks \(<1 3 1 2 6 4 5>_{m(A)-1} \text{times and then through } 1 3 1 2 6 4 7 9>\).
Assume \textbf{First} is correct if the remoteness of \( A \) is \(< n \). Now let \( A \) be of remoteness \( n \); \( X \) is \( A \), \( V \) is \( B \) and the traveler is at \( 1 \). The traveler goes to \( 3 \) where \( X \) becomes \( \text{FIRST.KID}(X) = o_1(A) \) and the traveler returns to \( 1 \). Since \( o_1(A) \) is of remoteness \(< n \), the inductive hypothesis applies. Thus the traveler arrives at \( 9 \) with \( X \) being \( o_1(A) \) and \( V = w[B,f(o_1(X))] \).

\( o_1(A) \) cannot be initial because of the inverse so the traveler goes next to \( 4 \). If we assume now that \( 1 = \text{SIB#}(X) = \# \text{SIBS}(X) \) where \( X = o_1(A) \), the traveler will pass through \( 4 \) and \( 5 \) updating \( X \) to contain \( o_2(A) \) and then enter \( 1 \). By inductive hypothesis again the traveler will eventually arrive at \( 9 \) with \( V \) containing:

\[
w[w[B,f(o_1(A))],f(o_2(A))]\]

and \( X \) containing \( o_2(A) \). Assuming without loss in generality that \( p = m(A) \), the traveler will eventually arrive at \( 9 \) after \( p \) repeats of the journey from \( 1 \) to \( 9 \) with \( V \) containing:

\[
... w[w[B,f(o_1(A))],f(o_2(A))], ..., f(o_p(A))] = w[B,w[f(o_1(A)), ..., f(o_p(A))]]
\]

by associativity and \( = w[B,f(A)] \) by definition of \( f(A) \).

\( X \) contains \( o_p(A) \) at this time. So the traveler goes to \( 4 \) where the decision is yes; \( 7 \) is next with \( X \) becoming its \( \text{PARENT} \), i.e. \( \text{PARENT}(o_p(A)) = A \). Thus the traveler arrives at \( 9 \) again with \( X \) containing \( A \), \( V \) still containing \( w[B,f(A)] \). Thus the First result is proven. Now let the
traveler start by entering 0 thus setting X to d and V to B = 0_w. Next the traveler enters 1 with these values in X and V and so by the First result the traveler will eventually arrive at 2 with X containing d and V containing w[0_w, f(d)] = f(d) by definition of 0_w.

As before the proof is for d ∈ D having remoteness ≥ 1 and is verified to include remoteness 0 by tracing the flowchart explicitly for this case.

The necessity for a 'uniform inverse' as opposed to a simple inverse in developing this theorem results from the fact that in the standard form of recursive definition considered here the number of appearances of the defined function symbol f is determined (≡ m(X)) by X the argument of f. This dependence was incorporated so that many common problems could be naturally expressed in that form.

We have not discussed the higher order recursive definitions having nesting on the right - largely because in our experience such definitions rarely occurred in practice. Such definitions are considered in [6]. The techniques given in [6] in combination with those here can be used to extend the above results to higher order recursive definitions not covered in [6].
Application of Theorem 2.2

Before applying theorem 2.5, it will be useful to make some notes on the flowchart of figure 2.1.

Flowchart Notes:

1) In general, X has a number of components; these components will be saved in identified storage locations.
   In any assignment to X in the flowchart, only those locations holding components which are changed need be assigned.
   In any decision on X in the flowchart, only those component locations need be tested which are necessary to secure the decision.

2) Some of the functions in the boxes like NEXT, SIB(X) in <5>, and the decision in <4> are sufficiently complex compositions so that simplifications are often possible.

3) The assignment in box <7> is made when data-structure X has the property that SIB#(X) = #SIB(X).

4) Conversely, the assignment in box <5> is made when SIB#(X) ≠ #SIB(X).

Beyond these generally applicable simplifications, others are applicable when the primitive functions in the flowchart have special properties.

The following is important for our example.

If w is a union and each q(X) produced in <2> will be different from all others produced there, then boxes <2> and <3> may be replaced by one having the assignment OUTPUT = q(X), meaning place q(X) in the next location in an internal storage table or external (paper) table.
Theorem 2.2 applies to many enumeration problems. It produces a 'good' algorithm from an easily justified recursive definition. For example, the theorem applies to Ex 1.1, a recursive definition for enumerating binary numbers. The definition in Ex 1.1 does have an inverse - namely:

\[
\begin{align*}
\text{if } a &= \langle a_1, \ldots, a_p \rangle \\
0^{-1}(a,n) &= \langle\alpha[a_p + \lambda], n+1\rangle \\
1_0(a,n) &= a_p + 1
\end{align*}
\]

Using this inverse and the \(o,w,m,T\) of ex 1.1, we get the following flowchart expression definitions:

\[
\begin{align*}
X &= \langle a, n \rangle \\
\text{FIRST.KID}(\langle a, n \rangle) &= \langle a//\langle 0 \rangle, n+1 \rangle \\
\#KIDS(\langle a,n \rangle) &= 2 \\
\langle a,n \rangle &= \text{TERMINAL? } = n=0? \\
\text{PARENT}(\langle a,n \rangle) &= \langle a[a_p + \lambda], n+1 \rangle \\
\text{SIB#}(\langle a,n \rangle) &= a_p + 1 \\
\#SIBS(\langle a,n \rangle) &= 2 \\
\text{SIB#}(\langle a,n \rangle) &= \#SIBS(\langle a,n \rangle)? = a_p + 1 = 2? \text{ referring to note 2 we we can simplify in this case: } a_p = 1 \\
\text{NEXT.SIB}(\langle a,n \rangle) &= \langle a[a_p + \lambda]//\langle 1 \rangle, n \rangle \text{ if } a_p = 0, \text{ which is all it can equal so simplifying: } \\
&= \langle a[a_p + \langle 1 \rangle], n \rangle \\
\text{also observe, referring to note 1 that } n \text{ is not changed.}
\end{align*}
\]

The flowchart of figure 2 is obtained by inserting these definitions in that of figure 1 as modified according to note 5 which is valid in this case. It is essentially the 'add-one' algorithms described in the introduction.
Similarly, theorem 2.5 can be applied to ex 2.1, the permutation definition. This definition also has an inverse, namely:

if \( \alpha = <a_1, \ldots, a_p> \), and \( x \) = the position (index) of the integer \((n+1)\) within \( \alpha \)

\[ i^{-1}(n, \alpha) = <n+1, \alpha[a_x + \lambda]> \]

Most of the flowchart definitions result from straightforward substitution of the inverse and the given primitive functions. For example:

\[ \text{SIB}(<n, \alpha>) = x \]
\[ \text{JSIB}(<n, \alpha>) = \text{KIDS(PAPENT(<n, \alpha>))} = m(<n+1, \alpha[a_x + \lambda]>) \]
\[ = |\alpha[a_x + \lambda]| + 1 \]
\[ = |\alpha| \]
From which the decision of box 4:

\[ (\text{SIB}(n,a) = \#\text{SIBS}(n,a)) = (x = |a|) \] which by the interpretation of \( x \) is equivalent to:

\[ a_p = (n+1)? \]

Also note that:

\[ \text{NEXT.SIB}(n,a) = a_{x+1}(n+1,a[a_x + \lambda]) \]

\[ = (n, a[a_x + \lambda],[a_x+1 + n+1]) \] which again, by the interpretation of \( x \) is:

\[ (n,a[a_x a_{x+1} + a_{x+1} a_x]) \] representing an interchange of the \( (n+1) \) component in with its right neighbor.

These definitions when put in figure 1 as follows produce an algorithm which stores only one permutation. The next permutation is always produced by interchanging components of the currently stored one. The resultant flowchart is:

\[ \text{figure 3} \]
Note that this flowchart contains the assignment: of the 'position of (n+1) in α' to the variable x implying a search for the integer (n+1) in α. A modification in the argument of the recursive definition can be made which will obviate this search. If we add to the argument a vector β giving the position of (n+1) in α, the 'position of (n+1)' will be available. Thus modifying Ex 1.2:

\[ X = \{<n, α, β>| n \in N; α and β strings of positive integers\} \]

\[ f(<n, α, β)> = \{α\} \quad \text{if } n = 0 \]

\[ f(<n, α, β)> = f(<n-1, α[α_0 + n], <λ>/β>) \cup \ldots \cup f(<n-1, α[α_p + n], <p+1>/β>) \quad \text{if } n > 0 \]

X is initially \( ε \{<n, λ, λ>| n \in N\} \)

The modification still has an inverse and can be implemented by the inclusion of a variable β and following the prescription of theorem 2.5 to obtain the following assignments to that variable in figure 2.1.

\[ β \leftarrow λ \quad \text{in}\ 1 \]

\[ β \leftarrow <1>/β \quad \text{in}\ 2 \]

\[ β \leftarrow β[β_1 \leftarrow λ] \quad \text{in}\ 3 \]

replace x by β_1 in 4 and after the assignment to α put:

\[ β \leftarrow β[β_1 + β_1 + 1] \]

This illustrates how with this approach one can modify an algorithm focusing on the effect of a change in data-structure without ever having to be enmeshed in the control structure of the algorithm.

The Search for the Inverse

We have been building a system which examines a recursive definition for properties identified in this paper as sufficient to allow efficient
implementation. An important part of this system is a program for determining whether the given definition has a uniform inverse. For a recursive definition of the standard form:

Let the data-structure \( X \) be a vector with \( n \) components, i.e.

\[
\text{let } X = \langle x_1, \ldots, x_n \rangle
\]

Let the primitive \( 0 \)-functions be:

\[
o_1(X) = o(I,X) = o(I, x_1, \ldots, x_n) = \langle y_1, \ldots, y_n \rangle
\]

Let \( o[j](I,X) \) be the \( j \)th component of \( o(I,X) \), thus:

\[
y_1 = o[1](I, x_1, \ldots, x_n)
\]

set \( A \):

\[
y_2 = o[2](I, x_1, \ldots, x_n)
\]

\[
\vdots
\]

\[
y_n = o[n](I, x_1, \ldots, x_n)
\]

It is easy to see that the recursive definition has a uniform inverse iff there is a unique solution of set \( A \) for \( I \) and each \( x_j \) as functions of \( y_1 \) through \( y_n \). If the solution for \( I \) is \( \tau(y_1, \ldots, y_n) \) and for \( x_j = t_j(x_1, \ldots, x_n) \), then the uniform \( i_0 \) and \( o^{-1} \) functions in the uniform inverse are given by:

\[
i_0(y_1, \ldots, y_n) = \tau(y_1, \ldots, y_n)
\]

\[
o^{-1}(y_1, \ldots, y_n) = \tau_1(y_1, \ldots, y_n), \tau_2(y_1, \ldots, y_n), \ldots, \tau_n(y_1, \ldots, y_n)
\]

Currently we have implemented a search for a 'simple' uniform inverse.

This is described below.

The first step is to set up the equations of set \( A \) for a given recursive
definition and then to try to obtain a solution by the following simple procedure.

1. First each equation in the set is tested to determine whether that equation by itself can be inverted. Often the right hand side of the equation will be only dependent on some of the variables \( x_1, \ldots, x_n \) and \( I \), and such an inversion will be possible. For example, say \( y \) on the left only depends on one \( x \) on the right:

\[
\begin{align*}
&\text{if } y = x + 1 \text{ then } x = y - 1, \text{ again assuming } y \text{ is a vector then} \\
&\text{if } y = \langle 0 \rangle / x \text{ then } x = y[y_1 + \lambda], \text{ also} \\
&\text{if } y = I \parallel x \text{ then } \begin{cases} x = y[y_1 + \lambda] \\ I = y \end{cases} \\
&\text{if } y = \begin{cases} \langle 0 \rangle / x & \text{if } I = 1 \\ \langle 1 \rangle / x & \text{if } I = 2 \end{cases} \text{ then } \begin{cases} x = y[y_1 + \lambda] \\ I = \begin{cases} 1 & \text{if } x = 0 \\ 2 & \text{if } y = 1 \end{cases} \end{cases}
\end{align*}
\]

2. If there is no simple equation which can thus be inverted, then the conclusion is that a 'simple' uniform inverse is not available. This does not mean there is no uniform inverse, but since the computation of the inverse when it exists will become part of the implementing flowchart - there is some justification in limiting our search to uniform inverses which are relatively simple to compute.

If one or more equations can be inverted, obtaining solutions for some \( x_j \)'s and perhaps for \( I \), the solutions are substituted for those \( x_j \)'s and perhaps \( I \) in the other unsolved equations and the procedure returns to step 1.
applying it to the unsolved equations with their substitutions.

If finally all equations are solved, the 'simple' uniform inverse has been obtained.

Simple as this procedure appears there is still considerable difficulty in determining how and if a simple equation can be inverted when one is dealing with relatively exotic functions such as concatenation, decision, insertion, etc. which arise in actual recursive definitions. At this stage even this simple step is handled heuristically for a limited number of primitive functions with no guarantee that inverses will always be produced when they exist.
References:

1. Aho, Hopcroft, Ullman; The Design and Analysis of Computer Algorithms; Addison-Wesley, 1975; pp 195, 222

2. Darlington, J. and Burstall, R.M.; A System Which Automatically Improves Programs; Proceedings Third International Joint Conference on Artificial Intelligence; Stanford, California, 1973; pp 479-485


7. Paull, M.C.; Formulation and Manipulation of Enumeration Based Algorithms; Research Report SOSAP-TR-4; December, 1973

Appendix I

Summary of Frequently used Notation

If $P$ is a predicate the $\overline{P}$ means not $P$.

$\mathbb{N}$ is the set of all positive integers = \{1,2,...\}

$\mathbb{N}$ is the finite set of integers from 1 to $n$ = \{1,...,$n$\}

If $A$ and $B$ are sets

$A \cup B$ is set union

$A \cap B$ is set intersection

$\overline{A}$ is the complement of $A$

$A - B = A \cap \overline{B}$

$|A|$ = the number of elements in $A$

$<a_1, \ldots, a_n>$ is an ordered set or vector with components

$a_i : i \in \mathbb{N}$ and $a_{i:j}$ represents the subvector $<a_i, a_{i+1}, \ldots, a_j>$; $a_{i:i} = <a_i>$

If $A$ and $B$ are ordered sets $=<a_1, \ldots, a_n>$ and $<b_1, \ldots, b_n>$ respectively

$A // B = <a_1, \ldots, a_n, b_1, \ldots, b_n>$

$\{A\}$ is the set of all components in $A$

If $E$, $x$ and $y$ are each an expression, i.e. a string or ordered set of symbols from a given alphabet, usually satisfying some constraints as to form, then

$E[x + y]$ is the expression that results when each occurrence of $x$ is replaced by $y$ in $E$.

The notation is extended to allow the specification of a number of replacements $E[x + y, z + w]$ is the expression which results when each $x$ is replaced by $y$ and each $z$ by $w$ in $E$. 
A PRINCIPLE USEFUL IN THE DESIGN OF MINIMUM PATH AND OTHER ALGORITHMS

M. C. Paul

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
Abstract

Central to the development of synthesis procedures for "good" algorithms, is the identification of principles with sufficient generality to provide the design basis for a class of algorithms. We need the simplest principles to cover the largest possible classes. This requires the appropriate formulation of the class and the statement of the principle in terms of that class. A principle which applies to such a class is developed here. The class consists of algorithms which solve sets of equations. It includes, for example, an algorithm for the minimum path graph problem. The distinguishing properties of this class are identified. The principle which underlies the timewise efficient algorithms of this class is given and justified. The algorithm thus developed for the minimum path example is compared with well-known alternatives. Finally, the principle is applied to another example: finding the minimum cost association of matrix multiplication, and shown to provide an algorithm having advantages over others previously described.

Key Words and Phrases: algorithm design, minimum path, complexity, equation sets.

CR Categories: 5.24, 5.25, 5.32, 5.42.
Minimum Constant Principle

In this paper we consider a simple principle, called the Minimum Constant Principle, which accounts for the existence of efficient solutions in a certain class of equation sets. Problems such as finding the minimum path in a graph, and the minimum cost order of association in a matrix multiplication can each be formulated as a member of this class.

The principle has an interesting relation to the minimum coefficient principle which is the name we give to the principle used in Dykstra's minimum path algorithm. When applied to the minimum path problem, in fact, both yield virtually identical results. Their usefulness in application to the minimal cost association of matrix multiplication problem, however, is quite different.

Consider the variables $X_1, \ldots, X_n$ which are to take values in a set $D$ of positive numbers and which are related by the following set of equations:

$$1) \{X_i = \min(T_{i1}, \ldots, T_{ip}, B_i)| 1 < i < n\}$$

in which:

- $T_{ij}$ is a function whose arguments are some subset $S$ of the variables $X_1$ through $X_n$ and whose value is always greater than that of any of those variables.
- $\min(A, B) =$ the smaller of $A$ and $B$
Such an equation set is called minimum-monotonic.

The Minimum Constant Principle is easily derived for such a set of equations.

Take the minimum of both sides of 1):

\[
\min(X_1, \ldots, X_n) = \min(T_{11}, \ldots, T_{1p}, B_1, T_{21}, \ldots, T_{2p}, B_2, \ldots, T_{n1}, \ldots, T_{np}, B_n)
\]

Since \(\min(X_1, \ldots, X_n) = \) some \(X_i\) \((1 \leq i \leq n)\) namely the smallest, and since each \(T_{ij} > \) than at least one \(X_i\) \((1 \leq i \leq n)\)

\[
\min(X_1, \ldots, X_n) \neq T_{ij} \text{ for any } 1 \leq i, j \leq n \text{ thus:}
\]

\[
\min(T_{11}, \ldots, T_{1p}, B_1, T_{21}, \ldots, T_{2p}, B_2, \ldots, T_{n1}, \ldots, T_{np}, B_n) \neq T_{ij} \text{ for any } 1 \leq i, j \leq n
\]

Therefore:

\[
\min(X_1, \ldots, X_n) = \min(B_1, \ldots, B_n)
\]

and if:

\[
\min(X_1, \ldots, X_n) = B_a
\]

then:

\[X_a \geq B_a\]

and since:

\[X_a = \min(T_{a1}, \ldots, T_{ap}, B_a)\]

so:

\[X_a = B_a\]
Therefore for a minimum-monotonic equation set, we have the minimum constant principle:

Theorem 1: \( \text{If } \min(B_1, \ldots, B_n) = B_a \text{ then } X_a = B_a \)

Algorithm Incorporating The Minimum Constant Principle

The principle established by this theorem can be made the basis of an algorithm for solving a set of minimum monotonic equations.

The algorithm is an adaptation of Gaussian Elimination. At each step a new set of equations is formed having one fewer equation than at the previous step and involving one less variable. The choice of the order of elimination is determined by the Minimum Constant Principle. The step will be indicated by the count \( k \) and the variables and constants as they appear in the equations formed in that step will be indexed by \( k \). Thus \( B_1^{(k)} \) is the designation of the constant term in the equation for \( X_1 \) formed in step \( k \). \( U(k) \) is the set of variables which have been used or eliminated in steps 1 through \( k \), \( A(k) \) are those still active.

With \( k \) initially = 1, \( U(0) = \) the empty set \( \text{and } A(0) = N = \{1, 2, \ldots, n\} \), the algorithm to solve for \( X_d \) is given as follows:

Minimum Constant Algorithm:

Loop: Find a \( \min\{B_1^{(k-1)} \mid i \in A(k-1)\} \)

Say it = \( B_1^{(k-1)}(k) \), i.e. the index of the minimum of the constant terms is \( I(k) \).

Record that \( X_{1(k)} \) equals \( B_{1(k)}^{(k-1)} \).

If \( I(k) = d \), then \( X_{1(k)} = X_d \) and the algorithm stops.

This theorem will still hold if \( T_i \) is only required to be greater than at least one variable of which it is a function. However, unlike the property given this property does not necessarily persist when a constant replaces a variable in \( T_i \). This persistence is necessary for consistent use of the principle in the Minimum Constant Algorithm.
terminates; otherwise
Substitute $p_{I(k)}^{(k-1)}$ for all occurrences of $X_{I(k)}$ in each
$I_{I_j}^{(k-1)}$ with $i \in A(k)$ (which is $A(k-1) - \{ I(k) \}$) and $j \leq p$.
Replace the constant term $B_j^{(k-1)}$ in each equation with
the minimum of that term with any new all constant
terms formed as a result of the substitution.
The new constant is designated $B_j^{(k)}$.
The resulting equations, excluding that for $x_{I(k)}$,
form the $k$-th set.
Return to Loop.

If the equations are to be solved for all, rather than for just
one variable, the Loop can be continued $n$ times, with a new variable
value determined each time through the Loop. The check for $I(k) = d$
can be eliminated.

It is easy to show; primarily because we are able always to
substitute a constant rather than an entire expression, involving
variables, in eliminating variables; that this is an $O(n^2)$ algorithm.
This is so in solving in this way for any number of variables.

Application of Minimum Constant Principle to Minimum Path Problems
And Comparison With Alternative Algorithms

The problem of finding the minimum cost path between node $1$ and
node $n$ in a directed graph, $G$, with $n$ nodes $N = \{1, 2, \ldots, n\}$, and
positive non-zero branch costs, can be formulated as a problem of
solving a set of equations. Let \( X_1 \) be interpreted as the minimum cost to reach node \( n \) from node \( i \) (thus \( X_n = 0 \)). Let \( a_{ij} \) be the cost of the branch from node \( i \) to node \( j \), which is a given of the problem. If there is no such branch then \( a_{ij} = \infty \). If a branch exists from \( i \) to \( j \), then \( j \) is an 'outward neighbor' of \( i \), and \( i \) an 'inward neighbor' of \( j \). The following equation set then represents a legitimate set of relations amongst the nodes of \( G \).

\[
\text{PATH1: } \{ X_1 = \min(a_{11}X_1, \ldots, a_{1n}X_n, \infty) | i \in N - \{n\} \} \\
X_n = 0
\]

This expresses the fact that the cost of the minimum path from node \( i \) to node \( n \) is determined by finding the minimum cost from each outward neighboring node of \( i \) adding the cost to reach that neighbor, and then minimizing this sum over all neighbors.

There is also a dual formulation of the problem.

Let \( Y_1 \) be interpreted as the minimum cost to reach node \( i \) from node \( 1 \) (thus \( Y_1 = 0 \)). Let \( a \) be as defined above.

\[
\text{PATH1': } Y_1 = 0 \\
\{ Y_1 = \min(a_{11}Y_1, \ldots, a_{1n}Y_n, \infty) | i \in N - \{1\} \}
\]

This expresses the fact that the cost of the minimum path to a node \( i \) from node \( 1 \) is determined by finding for each inward neighbor of node \( i \) the minimum cost to reach it from node \( 1 \), adding the cost of the branch from that neighbor to \( i \), and then minimizing this sum over all such neighbors.
For a given graph problem the solution for \( X_1 \) in the PATH1 set of equations should equal the solution for \( Y_n \) in the PATH1' formulation.

Either set of equations may be solved by the Minimum Constant Algorithm and is thus of \( o(n^2) \) complexity. (The application of this algorithm to PATH1' will be described in considerable detail shortly.)

A widely favored alternative for solving such minimum path problems is Dykstra's Algorithm. When interpreted as a procedure on a set of equations, a major source of the algorithm's strength is seen to come from the application of the 'Minimum Coefficient Principle'.

An interesting relation exists between the algorithms embedding the Minimum Coefficient and Minimum Constant Principles when respectively applied to a minimum path problem and its dual.

To establish this relation we first will translate Dykstra's Algorithm as it is applied to the minimum path problem formulated as a set of equations. In particular, consider the PATH1 equations.

In Dykstra's Algorithm one keeps a list of the current minimum cost to reach each node \( j \) from the starting node 1. In equation terms one works with an equation for \( X_1 \) which is stepwise developed by substitution for variables on its right. The cost to reach each node from 1 is designated \( a_{ij}^{(k)} \) after the \( k \)th iteration of the algorithm. Initially, \( k = 0 \) and \( a_{ij}^{(0)} \) is the cost associated with the branch \( <i,j> \). [Thus \( a_{ij}^{(0)} = a_{ij} \) in the set of equations formulation of the problem.] Also there is the set \( A(k) \) of all nodes not yet solved.
Initially $A(0) = \{2, \ldots, n\}$. On the kth iteration one finds the minimum of $\{a_{ij}^{(k-1)} | j \in A(k-1)\}$ and determines the index of that term. Call this index $I(k)$. The set of active nodes or indices, $A(k)$, can now be updated. $A(k) = A(k-1) - \{I(k)\}$. Next if $I(k) \neq$ the destination node, n then for each node $j$ which is an outward neighbor of $I(k)$ and in $A(k)$, the cost of branch $<I(k),j>$ is added to the cost of reaching $I(k)$ from 1, $a_{iI(k)}^{(k+1)} + a_{I(k)j}$. Note that $a_{I(k)j}$ as indicated is the coefficient of $X_j$ on the right of the equation for $X_{I(k)}$. For each $j$ this sum is compared with the previous minimum cost to reach node $j$, $a_{ij}^{(k-1)}$, and the minimum of these two becomes the current minimum:

$$a_{ij}^{(k)} = \min[a_{ij}^{(k-1)}, a_{iI(k)} + a_{I(k)j}]$$

This is equivalent to duplicating the $X_{I(k)}$ term on the right of the current $X_1$ equation and then in that duplicated term substituting for the variable $X_{I(k)}$ on the right of the $X_{I(k)}$ equation and gathering terms.

Now the $k+1$st iteration is undertaken.

If $I(k) = n$, the algorithm terminates with the minimum cost of reaching $n$ from the start node 1 ($=X_1$ in the equation) being given by $a_{1I(k)}^{(k-1)} = a_{1n}^{(k-1)}$.

Summary:

- $a_{ij}^{(0)} = a_{ij} = \text{cost of branch from node } i \text{ to node } j$
- $I(k) = \text{minindex}(\{a_{ij}^{(k-1)} | j \in N\})$
  
where \text{minindex}(\{A_1, \ldots, A_n\}) = \text{smallest } j \text{ such that } A_j = \text{minimum}(A_1, \ldots, A_n)$

- $a_{iI(k)}^{(k)} = \min(a_{ij}^{(k-1)}, a_{iI(k)}^{(k-1)} + a_{I(k)j})$
- if $I(k) = n$, answer $= a_{1I(k)}$
In this algorithm, the fact that one can always substitute for the variable on the right of the current \(X_1\) equation which has the smallest coefficient and thus arrive at the solution when \(X_n\) on the right has the smallest coefficient is referred to as the Minimum Coefficient Principle. This principle seems to require that the equations be minimum monotonic as defined on page 1 and further that the terms \(T_{ij}\) in that definition be restricted to a single variable.

For comparison consider the application of the Minimum Constant Algorithm to the minimum path problem. We will apply this algorithm to the dual of the problem to which we applied Dykstra's Algorithm — namely to its formulation in PATHI' . For comparison we will try to develop this application of the algorithm in the same notation used above for Dykstra Algorithm.

In order to show the relation of this algorithm to the previous one the initial step has to be separated from the remainder of the algorithm. \(B_1\) is the constant term in the \(i\)-th equation.

Initially \(B_1 = 0\) for \(i = 1\); \(B_i = \infty\) for \(2 \leq i \leq n\)

\[\text{So } \minindex\{B_i | i \in N\} = 1.\]

\[Y_1 = 0.\]

The result of substituting 0 for \(Y_1\) in the terms containing \(Y_1\) in the equation for \(Y_1\), \(2 \leq i \leq n\), is to make that term equal to \(0 + a_{11} = a_{11}\). This constant is minimized with the existing constant term, which is \(\infty\), in each of these equations giving a new constant term designated \(B_1^{(0)} = a_{11}^{(0)}\) in the \(i\)-th equation.
After this step, which is not counted, the set of equations to be solved for $Y_n$ then is:

\[
Y_2 = \min(a_{22}+Y_2, a_{32}+Y_3, \ldots, a_{n2}+Y_n, a_{12}^{(0)})
\]

\[
Y_3 = \min(a_{23}+Y_2, \ldots, a_{33}+Y_3, a_{13}^{(0)})
\]

\[
\vdots
\]

\[
Y_n = \min(a_{2n}+Y_2, \ldots, a_{nn}+Y_n, a_{1n}^{(0)})
\]

Let $A(0) = \{2, \ldots, n\}$, and $B_1^{(0)}$, also called in this case $a_{11}^{(0)} = a_{11}$ (as in the application of the Minimum Coefficient Principle). Start with $k = 1$.

On the $k$-th iteration the Minimum Constant Algorithm calls for determining the minimum index of all the constant terms, i.e., of $B_1^{(k-1)} = a_{11}^{(k-1)}\ i \in A(k-1)$. Let that index = $I(k)$. Then, because of theorem 1:

\[
Y_{I(k)} = B_{I(k)}^{(k-1)} = a_{I(k)}^{(k-1)}
\]

\[
A(k) = A(k-1) - I(k)
\]

If $I(k) \neq n$, the destination node, then at the next part of the $k$-th iteration $a_{1I(k)}^{(k-1)}$ is substituted for each occurrence of $Y_{I(k)}$ in all equations for $Y_j$ with $j \in A(k)$. The term involving $Y_{I(k)}$ in the $j$-th equation then becomes $a_{1I(k)}^{(k-1)} + a_{I(k)}^{(k-1)}$. The minimum of this and the current constant term $a_{1j}^{(k-1)}$ in the $j$-th equation becomes the new constant term, i.e.,

\[
B^{(k)} = a_{1j}^{(k)} = \min[a_{1j}^{(k-1)}, a_{1I(k)}^{(k-1)} + a_{I(k)}^{(k-1)}]
\]

Now the $k+1$st iteration may be started.
If \( I(k) = n \), the destination node, then the algorithm terminates with \( Y_n = a_{1I(k)}^{(k-1)} = a_{\text{in}}^{(k-1)} = \text{the answer.} \)

After the initial step, this algorithm may be summarized by the same set of relations as given in the Summary for Dykstra's Algorithm. Therefore in application to the problem of finding the cost of the minimum path in a positive weighted digraph, we have shown that the Minimum Constant Algorithm applied to one formulation of the problem follows a virtually identical course to that of Dykstra's Algorithm applied to the dual of that formulation.

These two minimum path algorithms may therefore be considered really the same though the principles on which they are based differ considerably.

So far we have considered only the cost of the minimum path and not the computation of the path itself. That computation for Dykstra's algorithm is well known. For completeness we will briefly sketch the analogous computation for the Minimum Constant Algorithm based on its application to \( \text{PATH}1' \).

The path can be computed by a kind of 'Hansel and Gretel' principle. Whenever the constant term of an equation say the \( j \)-th is changed during the \( k \)-th iteration this occurs because \( a_{1I(k)}^{(k-1)} + a_{I(k)j}^{(k-1)} < a_{ij}^{(k-1)} \). When that happens the value of \( Y_j \) is, as far as is known up to this iteration, directly dependent on \( Y_{I(k)} \) as it occurs on the right side of the \( Y_j \) equation. A vector \( \text{NXT} \) with \( n \) entries may be kept with \( \text{NXT}[j] \) set to \( I(k) \) to record each occurrence of the event described in the previous sentence. When the computation of the cost of the minimum path, \( Y_n \), is completed the minimum path will be
computable from this trail through the right side of the equations in
the NXT vector. That path is \[<1, \ldots, \text{NXT}(k), k, \ldots, \text{NXT}(\text{NXT}(n)), \text{NXT}(n), n>\].

Another Application of Minimal Constant Principle

The Minimum Constant Principle will now be applied to a problem discussed in reference 1 as an illustration of the 'dynamic programming' approach to algorithm design. The problem is to develop an algorithm for deciding which of the equivalent ways of associating the multiplication of a set of \(n\) matrices will result in the fewest number of multiplications of matrix components. For example, if \(M_1\) and \(M_3\) are both 3 by 7 matrices and \(M_2\) a 7 by 3 matrix then the multiplication \(M_1 \times M_2 \times M_3\) can be performed in the order given by the following association \(((M_1 \times M_2) \times M_3)\) with a cost of \(3 \times 7 \times 3 = 63\) component multiplications for \((M_1 \times M_2)\) and then \(3 \times 3 \times 7 = 63\) component multiplications for multiplying the 3 by 3 result of \((M_1 \times M_2)\) with the 3 by 7 matrix \(M_3\). Thus this association costs 126. On the other hand \((M_1 \times (M_2 \times M_3))\) costs 210 multiplications. These calculations use the fact that the cost = the number of component multiplications required to multiply the \(r_a\) by \(c_a\) matrix, \(M_a\), and the \(r_b\) by \(c_b\) matrix, \(M_b\), \(= r_a \times (c_a = r_a) \times c_b\).

In general, we are confronted with the multiplication of \(n\) matrices \(M_1 \times M_2 \times \ldots \times M_n\), with \(M_j\) having the dimensions \(r_j\) by \(c_j\). Let \(X_{ij}\) \((j \geq i)\) be interpreted as the minimum cost, over all possible associations, to multiply \(M_1 \times M_{i+1} \times \ldots \times M_j\) then the relations between the \(X_{ij}\) for a given problem involving \(n\) matrices is:
MATRIX1:

\[ x_{ij} = \begin{cases} 0 & \text{if } j = i \\ \min(a_{ijk} + x_{i,k+1} + x_{i+k+1,j}) & \text{if } j > i \end{cases} \]

where \( a_{ijk} \) is the cost of multiplying the result of \( M_i \times \ldots \times M_{i+k} \) by the result of \( M_{i+k+1} \times \ldots \times M_j = r_{i+k} \times \ldots \times r_{i+k+1} \times c_j \).

This relation represents the fact that the multiplication \( M_i \times M_{i+1} \times M_{i+2} \times \ldots \times M_{j-1} \times M_j \) can be finally done with one of the following associations:

\[
(M_{i+1} \times M_{i+2} \times \ldots \times M_{j-1} \times M_j)
\]

The 'dynamic programming' solution of reference 1 to this problem formulated as a procedure for solving the MATRIX equations is given as follows.

The \( x_{ij} \)'s are arranged according to the difference \( j-i = \Delta \). For \( \Delta = 0 \), \( x_{ij} = 0 \). For \( \Delta = 1 \), all \( x_{ij} \)'s only depend on \( x_{ij} \)'s for which \( \Delta = 0 \), and may be solved immediately. In general all \( x_{ij} \) with \( \Delta = d \) depend only on \( x_{ij} \)'s with \( \Delta < d \), and thus may be solved when these are known. Then the computation can be summarized:

On the \( d \)-th iteration for every \( x_{ij} \) with \( n \geq j \geq i \geq 1 \) and \( j-i = d \), compute \( x_{ij} = \min(a_{ijk} + x_{i,k+1} + x_{i+k+1,j}) \). The last iteration is for \( d = n-1 \). \( x_{in} \) is the answer.
This algorithm involves computing the value of \( n-d \) variables \( X_{ij} \) with \( j-i = d \). The computation of each variable \( X_{ij} \) with \( j-i = d \) requires taking a minimum of \( d \) terms. The computation time for each term is independent of the number of matrices being multiplied. It depends only on the size of the matrices. Let it = \( K \). Thus the maximum time is given by:

\[
K = \Sigma(n-d) \cdot d \text{ which is } o(n^3)
\]

Note that when the computation is done this way the average time = the maximum time.

The minimum cost can also be obtained by the application of the Minimum Constant Principle to the set of equations - MATRIX. An example will illustrate this approach and highlight its differences from the previous approach. Consider the example of multiplying matrices:

\[
M_1 \times M_2 \times M_3
\]

with \( M_1 \) and \( M_3 \) being 3 by 7 matrices and \( M_2 \) being a 7 by 3 matrix. The equations for this case are:

\[
\begin{align*}
X_{13} &= \min(a_{130} + X_{11} + X_{23}, a_{131} + X_{12} + X_{33}, \infty) \\
X_{12} &= \min(a_{120} + X_{11} + X_{22}, \infty) \\
X_{23} &= \min(a_{230} + X_{22} + X_{33}, \infty) \\
X_{11} &= 0 \\
X_{22} &= 0 \\
X_{33} &= 0
\end{align*}
\]

where \( M_1 \) is a \( r_1 \) by \( c_1 \) matrix and \( a_{ijk} = r_1 \times c_{i+k} \times c_{j} \).
The minimum constant is 0. \(X_{11}, X_{22},\) and \(X_{33}\) have this value. Substituting for \(X_{11}\) first leaves the minimum constant = 0, with \(X_{22}\) and \(X_{33}\) still having this value. \(X_{22}\) and \(X_{33}\) are then similarly substituted for after which the remaining equations will be:

\[
\begin{align*}
X_{13} &= \min(a_{130} + X_{23}, a_{131} + X_{12}, \infty) \\
X_{12} &= \min(a_{120}) = 63 \\
X_{23} &= \min(a_{230}) = 147
\end{align*}
\]

The minimum constant now is 63 with \(X_{12}\) already having that value. Substituting for \(X_{12}\) next then leaves:

\[
\begin{align*}
X_{13} &= \min(a_{130} + X_{23}, a_{131} + 63) \\
&= \min(a_{130} + X_{23}, 42 + 63) \\
&= \min(a_{130} + X_{23}, 126) \\
X_{23} &= 147
\end{align*}
\]

The minimum constant now is 126. It is the constant term in the \(X_{13}\) equation so, by the minimum constant principle, \(X_{13} = 126\) and this is the answer.

This answer was obtained without ever substituting for \(X_{23}\). In the 'dynamic programming' algorithm this would have had to have been done. On the other hand, in this algorithm it is necessary to find the minimum of all the constants, once for each of the substitutions, and to keep a record of some partially solved equations.

A matrix multiplication algorithm incorporating the Minimum Constant Principle simulating the above process will now be given. There is a matrix \(M\). An entry matrix \(M(i,j,k)\) indicates the the current state of \(X_{1, i+k}\) and \(X_{i+k+1, j}\), the two variables which appear in the \(k\)-th term on the right of the \(X_{ij}\) equation. The significance
If the entries in \( M(i,j,k) \) are given as follows:

\[
M(i,j,k) = \begin{cases} 
0 & \text{if neither } X_{i,i+k} \text{ nor } X_{i+k+1,j} \\
1 & \text{if either } X_{i,i+k} \text{ or } X_{i+k+1,j} 
\end{cases}
\]

Whenever a value for some variable \( X_{ab} \) becomes known, all \( X_{ij} \) equations which have \( X_{ab} \) on the right are effected. That effect is recorded by updating the matrix \( M \) as follows:

\[
M(a,j,a-b) := 1 \text{ for } n \geq j \geq b \\
M(i,b,a-i-1) := 1 \text{ for } a \geq i \geq 1
\]

if those entries were 0.

If an entry \( M(i,j,k) \) already = 1 is to be updated, then \( X_{i,i+k} + X_{i+k+1,j} + a_{ijk} \) is computed using values stored in a second matrix \( CON \). This is easily done since \( CON[i,j] \) holds the current constant term of the equation for \( X_{ij} \), and under the conditions stated, \( X_{i,i+k} \) and \( X_{i+k+1,j} \) must both be in \( CON \). Furthermore, \( a_{ijk} \) depends only on the given dimensions of the matrices involved. Next this quantity \( CON(i,i+k) + CON(i+k+1,j) + a_{ijk} \) is compared with the current value of \( CON(i,j) \). The minimum becomes the new value of \( CON(i,j) \), i.e.:

\[
CON(i,j,k) := \min(a_{ijk} + CON(i,i+k) + \\
CON(i+k+1,j), CON(i,j))
\]

In order to identify the constants over which the minimum is still to be taken, all those entries in \( CON \) which are neither \( \infty \) (their initial value), nor have already served as a minimum constant are linked together.
The number of rows of the \( i \)th matrix to be multiplied is in \( R[i] \) and the number of its columns in \( C[i] \).

Minimum Constant Algorithm for Minimum Cost of Associating Matrix Multiplications:

Initialize:

\[
M = 0
\]

\[
CON =
\]

while \( A, B \neq 1, n \)

Find the minimum of all linked entries in constant matrix \( CON \).

Put these indices in \( A \) and \( B \).

Remove \( CON[A, B] \) from the linked list.

for \( J = B+1 \) to \( N \)

if \( M[A, J, B-A] = 1 \) then \( UPDATE[A, J, B-A] \)

else \( M[A, J, B-A] := 1 \)

end

for \( I = 1 \) to \( A-1 \)

if \( M[I, B, A-I-1] = 1 \) then \( UPDATE[I, B, K] \)

else \( M[I, B, K] := 1 \)

end

end

procedure \( UPDATE[I, J, K] \)

\[
CON[I, J] := \min[R[I]xC[I+K]xC[J]+CON[I, I+k]+CON[I+K+1, J], CON[I, J]]
\]

end
We have not considered the actual order of association which would achieve the minimum cost computed by the previous algorithm. As in the minimum path example, the actual association could be inferred by using the Hansel and Gretel Principle. This would involve incorporating into the algorithm the facility to record which term on the right of each equation that used solved in finally solving for $x_{ij}$. This information would be sufficient to reconstruct the optimal association of the matrix.

It is easy to see that this algorithm, like the previous one, is $o(n^3)$ for the maximum time despite requiring a minimum operation not required there. Its advantage is in being able to produce a solution in considerably less than that maximum time, as illustrated by the example.

Though the above algorithm is of interest on its own, one of our objectives is to compare it with one which embeds the minimum coefficient principle. There is difficulty in doing so however. Each term on the right of an equation can involve up to two variables. Therefore after having identified the minimum coefficient it may not be clear which of two variables is to be substituted for. Also — after substitution and rearrangement of the right-side of an equation into a maximum of a set of terms, each being a sum of variables and a constant — the same variable may appear in more than one term. Furthermore, to show the relation of the Minimum Constant, and Minimum Coefficient Principles in the case of the minimum path problem it was necessary to consider dual formulations of that problem. Here it is not clear what a dual formulation would be.
Though it may still be possible to generalize the Minimum Coefficient Principle so that it is applicable to this problem, it appears that any equivalence with the algorithm above thus established must be of a substantially different character than that established for the minimum path problem.

References:

THE MIN-MAX BRANCH IN A GRAPH--AN APPLICATION OF THE MINIMUM CONSTANT PRINCIPLE

M. C. Paull

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
THE MIN-MAX BRANCH IN A GRAPH - AN
APPLICATION OF THE MINIMUM CONSTANT PRINCIPLE

In a previous paper [1] we described a principle which formed the basis of a 'good' algorithm for solving certain equation sets. This is the Minimum Constant Principle. In this paper an extension of that principle is given, and, as an example, applied to the problem of finding the minimum of all branches which are themselves each maximum on some path between two specified nodes in a given graph.

In general, the (extended) Minimum Constant Principle is applicable to the solution of equation sets forming a class we call (extended) Minimum-Monotonic. Subsequently we drop 'extended' with the understanding that all terms refer to their modified definitions as given here.

A Minimum-Monotonic equation set has the form:

\[ \{ x_i = \min(T_{i1}, \ldots, T_{in}; B_i) | i \in N \} \]

where \( N = \{1, 2, \ldots, n\} \), \( x_i \), \( i \in N \) are variables; \( B_i \), \( i \in N \) are constants; for each \( j \), \( T_{ij} \) is a function of a subset of the variables, \( x_i, i \in N \). Over the entire range of legitimate values of its variables the value of \( T_{ij} \) is \( \geq \) the value of any of its variables. (Our earlier definition [1] was more restrictive having a \( > \) in place of \( \geq \) above.)

[1] refers to the 1-th reference listed at the end of the paper.
An algorithm will now be described which gives a solution to such a set of equations. A solution is a set of values for the variable $X_i$, $1 \leq i \leq n$. When these values are substituted for the corresponding variables in $I$ and the result evaluated, equal values will appear on the left and right of each equal sign. The solution produced by the algorithm may not be the only one. Other sets of $X_i$ values will also give equality in $I$. This solution will, however, be the maximum one. In no other solution will any variable have a higher value than it has in this one.

The algorithm will transform the set of equations, initially, in its 0-th version in the form $I$, in a number of similar steps, through a number of versions, until for all $1 \leq i \leq n$ the equations with $X_i$ on the left has only a constant on its right.

Algorithm 1 (Minimum Constant Algorithm)

1. $U(1) = N = \{1, 2, \ldots, n\}$
2. do for $k = 1$ to $n$
   3. Find the minimum of all constant terms in the equations for $X_i$, $i \in U(k)$.
   4. Let $I(k)$ be the index of this minimum constant term.
   5. Let $X_i(k) = B_i(k-1)$ replace the equation for $X_i(k)$ in the $k$-th version of the equation set.
   6. Substitute $B_i(k-1)$ for all occurrences of $X_i(k)$ on the right of equations in the $(k-1)$-st version of the equation set — gather terms on the right leaving one constant term designated $B_i(k)$ on the right of each equation. These new equations together with any $k-1$ equations unaffected by these changes become the $k$-th
equation set, with their constant terms designated $B^{(k)}_i$, $i \in \mathbb{N}$.

Set $U(k)$ to $U(k)-1(k)$.

Note that in the final set of equations:

$$x_i(j) \leq x_i(j+1) \text{ for } 1 \leq j \leq n$$

This algorithm involves the following two operations of:

1) Substituting a constant into a term $T_{ij}$ then evaluating that term; then, if the evaluation gives a constant, taking the minimum of that term and the existing constant term.

This must be done at most $\sum_{j=1}^{n-1} (n-j) = n(n-1)/2$ times.

Assuming each part of this operation takes a constant time independent of $n$, the cost of the worst case is proportional to this time.

2) Taking the minimum of $n$, then of $n-1$ constants, etc. These require at most $n-1$, $n-2$, etc operations, costing at most a total of $n(n-1)/2$ basic minimum operations.

So the algorithm is $O(n^2)$, which is good, particularly if, as claimed, the values of $X_i$ computed by the algorithm do in fact satisfy the given equation set. This remains to be verified. The given set of equations may not even have a unique solution. Nevertheless, as will be shown, assigning $x_i(k)$ the value $B^{(k-1)}_{i(k)}$ by the above algorithm will always give the maximum solution. Of all the values assigned to $X_i$ in all possible solutions, the value assigned by this algorithm will be the largest.
Theorem: Algorithm1 gives a maximal solution to the equation set I.

Proof: The proof is by induction on the steps of Algorithm1.

Assume that if for each k ∈ N, $B_{k-1}^{(k)}$ is substituted in all equations of (I) for each occurrence on the right of $x_{k(I)}$ and evaluated—that then, at least for $i = 1, 2, \ldots, j-1$, the equations for $x_{I(1)}, x_{I(2)}, \ldots, x_{I(j-1)}$ will respectively have $B_{0}^{(0)}, B_{1}^{(1)}, \ldots, B_{j-1}^{(j-1)}$ on the right, and that no other solution could assign these variables greater values.

This is true for $j-1 = 1$ because the equation for $x_{I(1)}$ from I is:

$$x_{I(1)} = \min(T_{I(1)}^{(1)}, \ldots, T_{I(I)}^{(1)}, B_{I(1)}^{(1)})$$

Assume substitution is made for each variable in each of the terms $T_{I(1)}(j)$ above. Since (I) is minimum monotonic each $T_{I(1)}(j)$ is the value of every variable of which it is a function. Since further $B_{I(1)}^{(0)}$ is the smallest constant term by definition and the constant $B_{j}$ term gives a lower bound on $x_{j}$, no variable value is less than $x_{I(1)}^{(0)} = B_{I(1)}^{(1)}$. Thus the right side of (1) must evaluate to $B_{I(1)}^{(0)}$. Furthermore no larger value for $x_{I(1)}$ could ever be assigned in any legitimate solution of I since by

$$x_{I(1)}^{(0)} \leq B_{I(1)}^{(1)}$$

Now consider the general case. From (I) in its initial form take the equation for $x_{I(j)}$:

$$x_{I(j)} = \min(T_{I(j)}^{(1)}, \ldots, T_{I(I)}^{(j)}, B_{I(j)}^{(0)})$$

After substitution for $x_{I(1)}^{(1)}, x_{I(2)}^{(1)}, \ldots, x_{I(j-1)}^{(1)}$ on the right of an evaluation of this equation (2) becomes:

$$x_{I(j)}^{(1)} = \min(T_{1}^{(j-1)}, T_{2}^{(j-1)}, \ldots, T_{m_{I(j)}}^{(j-1)}, B_{I(j)}^{(1)})$$
where the terms $T_k^{(j-1)}$ contains only variables selected from:

\[ \{ X_1(1), X(1), \ldots, X_{I(n)} \} \]

Attention to the algorithm shows that this is the form of the equation for $X_{II}(j)$ in the $(j-1)$-st version of the equation set. It has already undergone substitution and evaluation on its right for $X_{II}(1), \ldots, X_{II}(j-1)$.

Further substitutions in the terms on the right of (2') for $X_{II}(j+r), r \geq 0$, can only give values for these terms $\geq$ to the value $X_{II}(j)$ by minimum monotonicity and the fact that $X_{II}(j+r) \geq X_{II}(j)$ for all $r \geq 0$. Thus upon substitution for all variables in (2'):

\[ X_{II}(j) = B_{II}(j-1) \]

Furthermore, under the assumption that none of $X_{II}(1) = B_{II}(1), \ldots, X_{II}(j-1) = B_{II}(j-1)$ could be any larger and since substituting these in (2) gives (2') then $X_{II}(j)$ can not be any larger than a value satisfying (2'). But according to (2'), $X_{II}(j) \leq B_{II}(j)$. So $B_{II}(j)$ is the largest possible value of $X_{II}(j)$.

The Minimum-Constant Principle is the proposition that a minimum monotonic equation set can be solved by the above algorithm which involves using the minimum of all constant terms for variables not yet evaluated to make the next variable evaluation.

The algorithm embedding the Minimum Constant Principle is simple, but deciding whether it is applicable is not always nearly so simple. The remainder of the paper is devoted to an example of a problem to
which the algorithm is applicable. The bulk of this remaining space is needed to show that the Minimum Constant Principle is really applicable.

An Example of an Application of the Minimum Constant Principle

Beyond showing that a solution to a problem must satisfy a set of Minimum Monotonic equations, we need to show that the solution desired is in fact the maximum-solution before the Minimum Constant Algorithm can be claimed to be applicable. If satisfying the equation set is necessary to the solution and the equation set can be shown to have only one solution then clearly that solution is the maximum and satisfaction of the set is sufficient. Thus the Maximum Constant algorithm is applicable. It was to such special cases that this algorithm was earlier shown applicable [1]. Minimum path graph problems, and minimum cost association of matrix multiplication problems provided examples there.

Consider now the problem of finding the min-max cost of paths between two specified nodes in a digraph \( G \). The min-max cost from nodes \( i \) to \( j \) is the minimum cost of all maximal branches on paths from nodes \( i \) to \( j \). The maximal branch on a path from \( i \) to \( j \) is the one with the largest cost of all the branches on that path. This problem has found application in medical diagnosis programs. I learned of it through [2] and discussion with A. Walker.
A necessary condition which must be satisfied by the min-max costs from nodes i to j in a given digraph is that: (1) those min-max costs must satisfy a certain set of minimum monotonic equations. However, in general, such an equation set will have many solutions. For applicability, then, it also must be shown that: (2) the maximum solution is the one sought. Both of these points will be demonstrated.

Necessary Condition:

Corresponding to any digraph G with nodes 1 to n, there is a set of n equations, E, in the variables $X_1$ through $X_n$ such that the cost of the min-max branch from node i to node j in G when substituted for $X_j$ in E will satisfy E.

Let G with nodes $= \{1, 2, \ldots, n = N\}$ be such a digraph. Consider the quantity obtained by first finding the maximum cost branch on each path from node i to node n, and then choosing the minimum of these. Call this quantity $X_i$, or the min-max cost from i to n. Let $c_{ij}$ be the cost of the branch from node i to node j if there is such a branch. If not $c_{ij} = \infty$, $B_i = \infty$ if i $\neq$ n, $B_n = 0$. (The original graph G may always be replaced by one having a branch between every pair of nodes; those branches not appearing in the original being given an $\infty$ cost; all others having their original costs.) Then the following set of relations in which $XY = \text{maximum}(X,Y)$, $XY = \text{minimum}(X,Y)$, must hold.

\[ E: \{X_1 = (c_{11}X_1)\_L (c_{12}X_2)\_L \ldots (c_{1n}X_n)\_L B_1 \mid i \in N \}\]

A more detailed justification that this set E is necessary is now
given in which for all nodes \( i \), \( p_i \) is a path from node \( i \) to node \( n \). \( P_1 \) is all such paths. For any two nodes \( i, k \), \( p_{ik} \) is a path starting with branch \( <i,k> \) and going on to \( n \). \( P_{ik} \) is all such paths. \( i^j \) designates the \( j \)-th outward neighbor of node \( i \).

Proof of necessity of \( E \):

A path from \( i \) to \( n \) must be composed of \( <i, i^j> \) followed by some \( p_i^j \) for some neighbor \( i^j \) of node \( i \). Every such path \( p_i^j \) has a branch of cost \( c_{i^j} \) on it. The min-max cost of all paths in \( P_i^j \), going directly from \( i \) to \( i^j \) and then onto \( n \), must then be \( \geq c_{i^j} \). On the other hand, \( P_i^j \) including as it does the suffix \( P_i^j \) must have a min-max cost \( \geq X_i^j \). Thus it follows that for each \( j \) the min-max of \( P_i^1 = c_{i}X_i^j \). The min-max path cost \( P_i^1 \) is the minimum of the min-max branch cost on all paths \( p_i^j \) over all neighbors \( i^j \) of \( i \), and is thus clearly given by the above equations, \( E \).

Next, the fact that the desired solution to this set of equations is the maximum one will be demonstrated. This is the part of the demonstration which, though simple, is uncomfortably long.

Sufficient Condition:

If \( E \) is the set of equations constructed by the above considerations from \( G \), then the maximum solution of \( E \) for \( X \) is the min-max cost from node \( i \) to node \( n \) in \( G \).

This will be shown by considering two basic transformations equally applicable to a min-max equation set \( E \) and its corresponding graph \( G \).
The first of these transformations, $T_1$, involves removing one of the variables appearing on the left of an equation in $E$ from all its right-side appearances in that equation. The corresponding graph transformation involves the removal of a self-loop.

The second transformation $T_2$ of an equation set $E$ is obtained by substitution for a variable appearance, say $X_a$, on the right of an equation for say $X_b$ with the right side of the equation for $X_a$, and the gathering of terms so as to put the result back in the same form as the original equation. ($X_a \neq X_b$)

Assume that a transformation is applied to an equation set (and corresponding graph) $E(G)$ to obtain $E'(G')$. We will show that if that transformation is $T_1$, then:

1. The min-max cost between any pair of nodes, $i$ and $j$ in $G$ and $G'$ are the same.
2a. Every solution of $E'$ is also a solution of $E$.
2b. Every solution of $E$ is \( \leq \) to every solution of $E'$.

If that transformation is $T_2$, then:

1. The solutions of $E'$ and $E$ are identical. The min-max path costs are identical in $G$ and $G'$.

As long as only these transformations of $E(G)$ to $E'(G')$ are used the maximum solution of $E(G)$ will remain a solution of $E'(G')$. Thus when a series of these transformations is shown to lead to an $E(G)$ with a unique solution, that solution is guaranteed to be a maximum solution.
T1: Loop removal:

Transformation $T_1$ consists of removing a self-loop (a branch from a node to itself).

First, (1) we will show that:

**Lemma 1.1:** Application of $T_1$ to a graph $G$ leaves a resultant graph $G'$ having all the same min-max costs.

**proof:** Assume that $G$ has a self-loop on node $a$. Let the paths from $i$ to $j$ in $G$ be partitioned into the sets $P_1$ and $P_2$ so that $P_1$ contains all those paths in which the self-loop branch $<a,a>$ appears at least once. Then the set of all paths from $i$ to $j$ in $G'$ is given by $P'_1 \cup P_2$, in which $P'_1$ is such that for each path $p_1$ in $P_1$ there is a path $p'_1$ in $P'_1$ which is like $p_1$ except that all branches $<a,a>$ are removed. But all such paths in $P'_1$ are then paths in $G$ in which no branch $<a,a>$ appears and thus are in $P_2$. So the set of all paths from $i$ to $j$ in $G'$ is $P_2$. Now it will be shown that a min-max cost branch of $G$ from $i$ to $j$ must be in one of the paths of $P_2$. From this it follows that it will be the same min-max cost as that of $G'$.

To show that a min-max cost branch from $i$ to $j$ in $G$ is on a path in $P_2$ we need some definitions. Let $\text{maxbc}(p)$ be the cost of the maximum cost branch on path $p$. If $p_1$ and $p_2$ are paths we say $p_1 \sqsupset p_2$ when $p_2$ consists of a subset of the branches of $p_1$. If $p_1 \sqsupset p_2$ then obviously $\text{maxbc}(p_1) \geq \text{maxbc}(p_2)$. Clearly by definition of $P_1$ and $P_2$ for each path $p_1 \in P_1$ there is a path $p_2 \in P_2$ with $p_1 \sqsupset p_2$ then for
each value $\text{maxbc}(p_1)$ there is a path in $P_2$ whose value
$\text{maxbc}(p_2) < \text{maxbc}(p_1)$ and consequently:
\[
\min_{P \in P_2} (\text{maxbc}(p)) < \min_{P \in P_2} (\text{maxbc}(p))
\]
or:
\[
\min_{P \in P_1 \cup P_2} (\text{maxbc}(p)) = \min_{P \in P_2} (\text{maxbc}(p))
\]
Since $\min(\text{maxbc}(p))$ is the min-max cost from $i$ to $j$ in $G$,
that min-max cost involves only the paths in $P_2$ as asserted.

So transformation $T_1$, loop removal, does not effect the min-max
cost from $i$ to $j$. Next it is necessary to show (2a) that:

**Lemma 1.2a:** The set of equations, $E'$, corresponding to $G$
will have solutions each of which will also satisfy $E$, the
set of equations corresponding to $G$.

**proof:** In equation terms the transformation $T_1$ involves a change of a
recursive equation, i.e. one in which the same variable
appears on the left and right. Assume that the variable $x_a$
involved in such a recursive equation and $T_1$ is applied to that
$x_a$ equation then, with $\beta$ having an expression involving
minimums of maximums of constants and variables:

\[
x_a = (c_{aa} \sum x_a) + \beta
\]
in $E$ becomes:

\[
x_a = \beta
\]
in $E'$. All other equations go into $E'$ intact.
Consider any solution to \( E' \). Let \( X_a \) have the value \( v(X_a) \) in this solution. When these values are substituted into an expression \( \beta \) its value is designated \( v(\beta) \). Let us see how these values work in the \( E \) equations. For those equations \( X_j = e_j \), which go over unchanged into \( E' \), evaluation yields \( X_j = v(e_j) \) in both \( E' \) and \( E \). For the \( X \) recursive equation of \( E' \) and the corresponding equation of \( E \) however, \( X_a = v(\beta) \) and \( X_a = (c_a \land v(X_a)) \land v(\beta) \) are the respective evaluations. But \( v(X_a) = v(\beta) \) by definition since that is its value in \( E' \) so:

\[
X_a = \begin{cases} \\
(c_a \land v(\beta)) \land v(\beta) & \text{if } c_a \geq v(\beta) \\
\geq v(\beta) & \text{else} \\
v(\beta) & \text{if } c_a < v(\beta)
\end{cases}
\]

Thus any solution of \( E' \) is also a solution of \( E \).

Next it is necessary to show that:

Lemma 1.2b: For any solution \( S \) of \( E \) which is not a solution of \( E' \), there is another solution \( S' \) which satisfies both \( E \) and \( E' \) in which each variable has at least as large a value as it has in \( S \).

Proof: Assume that the \( T \) transformation from \( E \) to \( E' \) corresponds to the removal of a self-loop on node \( a \) of the corresponding graph, i.e. recursion in the equation for \( X_a \). Thus the difference between \( E \) and \( E' \) is that the first has the equation

\[
X_a = (c_a \land X_a) \land \beta,
\]

whereas the second has \( X_a = \beta \). For each of the
other variables the same equation is in E and E'. Now let \( X_i \)
have the value \( w(X_i) \) in a solution of E and an expression \( \beta \) in
the variables \( X_i \) have the value \( w(\beta) \). In particular:

\[
X_a = (c_{X_a} \land w(X_a)) \land w(\beta) = w(X_a)
\]

if \( c_{X_a} > w(X_a) \) then

\[
X_a = c_{X_a} \land w(\beta) = w(X_a) = w(\beta)
\]

if \( c_{X_a} \leq w(X_a) \) then

\[
X_a = w(X_a) \land w(\beta) = w(X_a)
\]

In general then in E:

\[
(1) \quad w(X_a) \leq w(\beta)
\]

If these same values \( w(X_i) \) are substituted in \( E' \) for \( X_i \) we get:

\[
(2) \quad X_a = w(\beta) \quad \text{(note \( \beta \) does not contain \( X_a \))}
\]

A difference of solutions for \( X_a \) in E and E' only exists if the
\( < \) holds in (1). Assume that to be the case, i.e.
\( w(X_a) = w(\beta) - \epsilon, \epsilon > 0 \), and consider the following iterative
process on the equation set E'. Substitute on the right of the
equation of E' for each \( X_a \) the value of \( w(X_1) \), also designated
\( X_a^{(0)} \), and evaluate. The resultant value computed for each \( X_a \)
in E' is designated \( X_a^{(1)} \). Note that it follows from the above
discussion that \( X_a^{(1)} \geq X_a^{(0)} \) for all \( 1 \leq i \leq n \). Generally designate
by \( X_a^{(j)} \) the value of \( X_a \) which results when \( X_a^{(j-1)} \) is
substituted on the right of the equations of E' and the
resultant expressions evaluated. Then it is easy to see that,
because of the properties of the minimum monotonic equations
each \( X_a^{(j)} \geq X_a^{(j-1)} \). Because of the min-max nature of these
equations also the value that any particular variable can
assume in these evaluations is confined to a member of the set:

\[
\{w(X_a) | 1 \leq i \leq n\} \cup \{c_{X_a} | 1 \leq i \leq n, j \leq n\} \cup \{b_i | 1 \leq i \leq n\}
\]
This is so because the right side of these equations is always the minimum of maximums of these values and thus must equal one of them. For some finite \( j \) then \( x_1^{(j)} \) will equal \( x_1^{(j-1)} \) for all \( i \in N \), since their values are chosen from a finite set while always either increasing or remaining unchanged with an increase in \( j \). Clearly for this value of \( j \), \( \{ x_1^{(j)} \}_{i \in N} \) is a solution to \( E' \), and it is a solution in which \( x_1^{(j)} \geq x_1^{(0)} = w(x_1) \) by transitivity and thus is greater than the solution to \( E \).

This completes the demonstrations of the properties of \( T_1 \).

Next consider transformation \( T_2 \). This transformation is first defined precisely.

Consider a set of min-max equations \( E \) like those on page 7. The equation for \( x_a \) has the form:

\[
x_a = \left[ c_{a1}x_1 \right] L \left[ c_{a2}x_2 \right] L \cdots L \left[ c_{an}x_n \right] L B^n
\]

Consider the substitution of the right-side of this equation for the \( x_a \) appearance on the right of another equation for say \( x_b \), \( b \neq a \).

(1) \( x_b = \left[ c_{b1}x_1 \right] L \cdots L \left[ c_{ba}x_a \right] L \cdots L \left[ c_{bn}x_n \right] L B^b \)

(1') \( x_b = \left[ (c_{b1}L(c_{ba}c_{a1})x_1) \right] L \cdots L \left[ (c_{bn}L(c_{ba}c_{an})x_n) \right] L B^b \)

Let \( E' \) be the same as \( E \) with the exception that the equation for \( x_b \) is replaced by (1') above with its coefficients evaluated. Then the substitution and evaluation which produced \( E' \) from \( E \) is
transformation $T_2$.

If the original set $E$ corresponds to graph $G$, the $G'$ corresponding to $E'$ above, created by the transformation $T_2$, is clearly $G$ with the cost of the branch from $b$ to $j$ changed to $c_{bj}(c_{ba}c_{aj})$ if $j \neq a$, and with the cost of the branch from $b$ to $a$ set to $r_3$. The change from $G$ to $G'$ is illustrated below.

The claim now is that any solution of $E$ is a solution of $E'$ and vice versa. Thus the min-max path cost in $G$ and $G'$ will be the same.

Lemma 2.1: If $E'$ results from $E$ by transformation $T_2$, then $E$ and $E'$ have the same solutions.

Proof: $E$ and $E'$ differ only in their respective equation for $X_b$.

Suppose in a solution of $E$ the value of $X_1$ is $v(X_1)$, $1 \leq i \leq n$.

Then for each equation except that for $X_b$ these same values will when substituted on the right in $E'$ give the same result as in $E$ for the corresponding equations. We can see by the equality of $(1)$ and $(1')$ for all variable values, that substituting $v(X_1)$ for $X_1$ on the right will result in $X_b$ in $E'$ being equivalent to the following $(1)$ form equation:

$$X_b = [c_{b1}v(X_1)]L \ldots L[c_{ba}v(X_1)]L \ldots L[c_{an}v(X_1)]L \ldots L[c_{bn}v(X_1)]L \ldots L[c_{bn}v(X_1)]L \ldots L[c_{bn}v(X_1)]L \ldots L[c_{bn}v(X_1)]L$$
Because the underlined subexpression is the right side of the
equation for $X_a$ with $v(x_1)$ substituted for all variables, its
value is $v(X_a)$:

$$X_b = [c_{b_1} v(x_1)]^L \ldots [c_{b_a} v(x_a)]^L \ldots [c_{b_n} v(x_n)]^{L_B} b$$

which has the same right side as $X_b$ in $E$ when evaluated. So
in $E'$:

$$X_b = v(X_b)$$

On the other hand, suppose in a solution for $E'$, $X_1$ is given
the value $w(x_1)$. Substituting these values on the right of
all equations in $E$ except $X_b$ will give the same values for all
variables in $E$ except $X_b$. For $X_b$ in $E$ we have:

$$X_b = [c_{b_1} w(x_1)]^L \ldots [c_{b_a} w(x_a)]^L \ldots [c_{b_n} w(x_n)]^{L_B} b$$

but in $E'$, again because of equality of $(1)$ and $(1')$ and the
fact that $(c_{b_1} w(x_1)]^L \ldots [c_{b_n} w(x_n)]^{L_B} a$) is the evaluated
right-hand of $X_a$ in $E'$, $X_b$ will have the same evaluated
right-hand as given above for $E$. Thus $w(X_b)$ will be the value
of $X_b$ in $E$ also.

We already know that amongst the solutions to $E$ is the min-max
path costs for the corresponding graph $G$.

Consider now a series of applications of $T_1$ and $T_2$ starting with
an initial graph $G$ and corresponding equation set $E$. Let these yield:

$$E(G) = E_1(G_1) \rightarrow E_2(G_2) \rightarrow \ldots \rightarrow E_n(G_n)$$

In which $E_1(G_1) \rightarrow E_{i+1}(G_{i+1})$ indicates that $T_1$ or $T_2$ were used to get
$E_{i+1}$ from $E_i$. Assume further that in $E$ there are only constants on
the right of each equation, then:

**Theorem 2:** (1) evaluation of the right-sides of \( E \) for \( x_i \)'s give the min-max costs in \( G \) from node 1 to node \( n \) and (2) the solutions in \( E \) will be maximum solutions of \( E \).

**Proof:** (1) follows from the following inductive argument:

Assume that amongst the solutions to \( E_1 \) are the min-max path costs of \( G \), and that the min-max cost of \( G_1 \) is equal to the min-max cost of \( G \). If \( E_{i+1} \) is obtained by \( T_2 \) from \( E_1 \) then by lemma 2.1 it will have the same solutions as \( E_1 \) and so will also include the min-max costs of \( G \) and \( G_{i+1} \) will have the same min-max costs as \( G \). If \( G_{i+1} \) is obtained by \( T_1 \) from \( G_1 \) then since by lemma 1.1 \( G_{i+1} \) has the same min-max path costs as \( G \), it will have the same such costs as \( G \).

(2) follows with a similar argument from the fact that under transformation of \( E_1 \) to \( E_{i+1} \) by \( T_1 \), although solutions may be lost they are all by (lemma 1.2b) smaller than the remaining ones. And under \( T_2 \) no solutions are lost as stated by lemma 1.1. Therefore, when only one solution remains it must be the maximum solution to \( E \).

Now there are a number of orders in which the transformations \( T_1 \) and \( T_2 \) can be applied to obtain the sequence of equation sets \( E = E_1, \ldots, E_n \) with \( E_n \) having only constants on the right. We give one such order of application which is analogous to the process used in Gaussian Elimination.
Algorithm 2

E ← Equation Set
j ← 1
i ← 1

T1: if (in E₁ the equation for Xₖ has Xₖ on the right, i.e. is recursive) then (remove that recursion by applying T₁; i ← i+1)
   if (j=n) then DONE
   k ← 1

T2: if (on the right of the equation for Xₖ+₁ Xₖ appears in E₁)
   then (use T₂ to substitute with the right-side of Xₖ equation on the right of the Xₖ+₁ equation thus eliminating Xₖ from the right of Xₖ+₁ equation; i ← i+1)
   if (j+k=n) then (j ← j+1; go to T₁)
   k ← k+1
   go to T2

Each of the above steps can obviously be carried out. As a result, variables are continually eliminated from the right of successive equation sets in the sequence E = E₁, ..., Eₙ. In going from Eₖ to Eₖ+₁ variable Xₖ is eliminated from the equation for itself by T₁ if necessary, and then from the equations for Xₖ, Xₖ+₁, ..., Xₙ by T₂. This is done for j = 1 to n-1, finally leaving Eₙ, the final equation set with no variables, only constants, on the right of each equation.
The point of this entire development is that the maximal solution to the equation set $E$ corresponding to graph $G$ gives the min-max paths from node $i$ to node $n$ in $G$. Therefore, Algorithm1—incorporating the minimum cost principle which is more efficient than Algorithm2—can be used to get the min-max path cost.

Final Note

The above properties of transformations $T_1$ and $T_2$ were shown to be applicable to the min-max path set of equations. In fact, these properties must apply to a large class of such equation sets, including the min-max set as one member. Virtually any equation set has the properties requires of $T_2$. The properties of $T_1$ on the otherhand will only be true for a constrained class of equation sets. For this larger class of equation sets Algorithm2 would surely be applicable to get the required maximal solution. Furthermore, if the particular equation set satisfies the conditions for application of the minimum constant principle also, then the more efficient Algorithm1 would be applicable.

The class to which the desireable properties apply appears to have considerable similarity with the closed semi-ring as defined in section 5.6 of [3]. We are currently at work on its delineation.

Transformation $T_2$ maintains its properties for virtually any equation set. $T_1$ on the other hand will only maintain its properties under restrictive conditions.
References:

[1] Paull, M.C.; A principle Useful In The Design Of Minimum Path And Other Algorithms; To be published.

[2] Ng, Shuey; Walker, A.; Max-Min Chaining Of Weighted Assertions Is Loop-Free; Internal Report CEM-TR-73; Dept. of Computer Science, Rutgers - The State University of New Jersey, 1977

APPROACHES TO AUTOMATIC PROGRAM GENERATION

S. V. Levy

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
The object of this research has been to gain insight into the problem-solving process in general and the programming process in particular. We chose, for our vehicle, a restricted class of problems - the generation of programs for determining order characteristics of a set of numbers (selection, sorting, etc.).

There are three levels of knowledge which we used in the generation of 'good' programs; knowledge about programming, knowledge about sorting and comparing, and knowledge about numbers and their ordering properties.

Our earliest work considered the general subject of generating algorithms for arranging numbers on the basis of numerical comparisons. A specification of the desired arrangement of the data in the output sequence serves as input to the "automatic programmer" which generates a program to rearrange the input data into that order. It is necessary to specify a rule for manipulating the data, and a rule for determining that the program has successfully completed its task. The first rule, since it is almost completely concerned with order on the data, is easy to specify - the second rule, the stopping condition, turned out to be considerably more complex.

Rule 1 is, in essence, Compare two inputs; if they don't satisfy the definition of the output condition, manipulate them so that they do; if they do satisfy the definition, repeat the process with another element. This rule is necessarily vague at this point. We shall see, below, how it might actually be affected in practice.

Rule 2 is considerably more complex in its interpretation and could easily produce substantial payoffs in the running time of the programs produced by the system.

The simplest algorithms which we produce use only the space in which the inputs are presented (plus perhaps a fixed size workspace, independent of the size of the input set). More sophisticated algorithms, which are able to make use of the transitivity and other
properties of the order relation on real numbers seem in our experience to make use of a space which is dependent on the size of the input set. In order to derive these more sophisticated variants, it is necessary that the automatic programmer include some type of theorem generator. Some simple examples of output specifications and the programs we expect and hope to produce will make this last section more clear.

Examples

Output specification 1: (i) \([B_i \preceq B_{i+1}]\).
where \([B_i]\) \(i=1,2,\ldots, n\) is the output set

In this case, the program that would be produced is obvious - at least as far as manipulating the data. The part of the program which generates a stopping condition is less obvious. The algorithm consists of comparing the first two inputs: if the larger is already on the right, leave the elements in their present order and compare elements 2 and 3. Continue this until all elements are compared, then go back and make another pass through the input set. This is repeated until all the inputs are placed in their correct position.

How can the program decide that all the data are ordered? Several methods come to mind. Fortunately in this case the most obvious method works - when the algorithm makes a complete pass through the input data but performs no interchanges, the data is ordered. In order to know this has occurred it is necessary for the algorithm to keep track of the number of exchanges on each pass, or at least whether or not an exchange has been made. An alternate stopping condition which results in a generally poorer algorithm (at least as measured by running time) makes use of the fact that after \(\frac{n(n-1)}{2}\) comparisons every element will have been compared to every other element, and if it can be shown that the algorithm always moves in the correct direction, then it will always be complete after that many comparisons.
Propagated to its proper place.

Since the largest not-as-yet-ordered input is

This program is easily recognizable as bubble sort or

After step 4:

```
IF p(N-I) = Z
    p = p+1
```

After step 2 and right after Z:

Before step 0:

```
   p = 0
```

End

After step 4:

```
IF p = 1, goto 5
```

At step 2 before b(I+1) = a(I)

To complete the program we add the following steps:

ultimately the termination of the algorithm.

of the program which handles the repeated passes through the inputs and

a single pass through the inputs--we have yet to put in the section

The program so far includes only the body which manipulates the data for

```
   4 IF I < N goto 1
   4 goto 4
   1+I = I
   (I)V = (I+1)V
   (I+1)V = (I)B
   2 goto 4
   1+I = I
   (I)V = (I+1)V
   2 goto 5
```

```
1 IF V(I+1) goto 2
   0 goto 0
   0 = I
   1 = I
```

The Program

-3-
Output specification 2: \((i)(i \leq j \rightarrow B_i \leq B_j)\)

In this case again the data manipulation part of the program is obvious, but unless some cleverness is introduced into the termination decision, the algorithm produced will be very inefficient. Essentially the algorithm takes 1 element at a time, compares it to all the others, and places it between the set of elements greater than it and the set smaller than it. This is clearly insertion, and on each pass one element is put into its final place. Consequently the algorithm terminates after \(N\) passes.

The Program

1. \(I=1\) we place \(A(I)\) at each pass
   \(K=1\) It is placed in position \(K\)
   \(J=1\)

2. If \(J \neq I\) goto 3
   \(J=J+1\)
   If \(J > N\) goto 5

3. If \(A(I) < A(J)\) goto 4
   \(K=K+1\)

4. \(J=J+1\)
   If \(J < N\) goto 2

5. \(B(K)=A(I)\)
   \(I=I+1\)
   If \(I \leq N\) goto 1
End

In addition to the program generator which seems very straightforward it would be necessary to produce some sort of generator of stopping conditions. This would seem to be a very complex part of the system. It would be concerned with proving such properties of candidate programs as they are moving in the correct direction (e.g., it is never necessary to repeat an exchange of two data which are once exchanged by the algorithm). This property together with a bound on the number of
We have a great deal of experience with the design of certain classes of algorithms. For this reason, we choose as one domain of study the non-adaptive comparator.

Although it seems clear that any truly useful automatic program would not permit work to different programs already being worked by many others,

trying to create a full-dom system would divorce too much of the

thinking of the system from the system.

Thus far, there were some sort of theorems proved which knew something

possible interferenced could provide a hint on and proof of termination.
algorithms for \(2^k + i\) inputs \((i < 2^k)\); the Batcher odd-even merge (Batcher, 1968) strategy; and the splitting strategy which we describe below.

These algorithms have as their basic operation a comparison between data stored in two memory locations followed by the assignment of the smaller of the two inputs to one location and the larger to another. The algorithms are called non-adaptive because the sequence of operations is totally independent of the contents of the storage locations, but is completely determined at the start of the algorithm. The algorithms can also be carried out by networks of 'comparators' (a comparator is a 2 input - 2 output computational device which operates on numbers as inputs and produces as outputs the minimum of the inputs at one (designated) output and the maximum at the other output). The networks are formed by connecting outputs of comparators to inputs of other comparators in an obvious (loop-free) way.

There have been several studies of networks for sorting (Levy, Pauli, 1969; Batcher, 1968; Van Voorhis, 1972; Knuth, 1973) which have appeared in the literature--we shall ignore the general sorting problem here and instead focus on the following problem:

The selection problem - to find the \(i^{th}\) largest input
(Note that when we look for the \(i^{th}\) largest, we always assume that \(i < \frac{n}{2}\). If this is not the case, then we would look for the \((n-1)^{th}\) smallest. The transformation between the algorithms is obvious.)

The goals of this research are to learn to design programs, not networks; and we do not treat this as a network design problem. We are interested in the question of how one goes from the specification of a problem to the specification of an algorithm to solve that problem.

It should be obvious that non-adaptive algorithms (NAA - from here on), in general, prove to be less efficient than unrestricted programs to compute the same functions (i.e., the NAAs require a greater number of comparisons).

To find either the largest or the smallest input (out of \(n\)) requires \(n-1\) comparisons in both the NAA and the program case. This is easily shown to be optimal - and realizable. However, to find the \(2^{nd}\) element requires \(n+\lfloor \log n \rfloor - 2\) comparisons in the unrestricted case and \(2n-3\) in the NAA case. The former is optimal and depends on the algorithm locating the first element and
In network notation we can represent this operation by a single comparator

\[
\begin{align*}
\max (x, y) & \rightarrow f_L \\
\min (x, y) & \rightarrow f_L \\
\text{which is interpreted as}
\end{align*}
\]

\[
(M_L, L_L) \rightarrow (f_L, L_L)
\]

The basic operation of the NN may be represented as

comparisons.

We shall show below to find the third input in fewer than \(3n-6 + n = 3n-2n+5\) steps, whereas \(n = 2n+5\). Therefore, the (n - 2) = \((n-1) - 1)\) elements. In order to find the second element can be done by the NN is to repeat the first pass, that is, find the smallest. This is easy to do with a program, but not possible in an NN. The best that can be done by the NN is to repeat the first pass, that is, find the smallest. Therefore, it is necessary to remember the first and second elements. If the algorithm is necessary to remember the first element. In order to perform which, best, to the first element, the first time around, the second had to be looking for the second element only among the (at most) [Log] inputs.
An example of a minimal NAA to find the second largest of eight inputs is:

1) \((L_1, L_2) \rightarrow \text{Comp} (L_1, L_2)\)

2) \((L_1, L_3) \rightarrow \text{Comp} (L_1, L_3)\)

3) \((L_1, L_4) \rightarrow \text{Comp} (L_1, L_4)\)

4) \((L_1, L_5) \rightarrow \text{Comp} (L_1, L_5)\)

5) \((L_1, L_6) \rightarrow \text{Comp} (L_1, L_6)\)

6) \((L_1, L_7) \rightarrow \text{Comp} (L_1, L_7)\)

7) \((L_1, L_8) \rightarrow \text{Comp} (L_1, L_8)\)

8) \((L_2, L_3) \rightarrow \text{Comp} (L_2, L_3)\)

9) \((L_2, L_4) \rightarrow \text{Comp} (L_2, L_4)\)

10) \((L_2, L_5) \rightarrow \text{Comp} (L_2, L_5)\)

11) \((L_2, L_6) \rightarrow \text{Comp} (L_2, L_6)\)

12) \((L_2, L_7) \rightarrow \text{Comp} (L_2, L_7)\)

13) \((L_2, L_8) \rightarrow \text{Comp} (L_2, L_8)\)

and the answer appears in location \(L_2\).

This algorithm can also be represented by the following comparator network.

The Levy-Paul algebra (Levy, Paul, 1968) for sorting algorithms consists of two binary operators: \((\text{Minimum})\) and \((\text{maximum})\). The algebra is isomorphic to a Boolean algebra without negation. Consequently, the set of functions computed by NAA's is isomorphic to the set of positive Boolean functions. As a corollary, the functions computed by an NAA are completely determined by what they do on the input n-tuples of 0's and 1's. (This tells us, for example, that to test an NAA purported to sort \(n\) inputs, it is only necessary to test...
3) Compute \( u_q, \ldots, z_q \).
\[
(1) \quad u_q, \ldots, z_q
\]
\[
(2) \quad u_q, \ldots, z_q
\]

2) Sort the two sets
\[
\begin{align*}
\{ u_1, \ldots, z_1 \} &= \mathbf{A} \\
\{ u_2, \ldots, z_2 \} &= \mathbf{B}
\end{align*}
\]

1) Break the inputs into 2 sets of size \( n \) each.

For finding the \( n \) largest numbers out of a set of \( 2n \) numbers, an immediate consequence of the theorem is an algorithm.

Theorem: \( a^f \), \( b^f \), \( c^f \), \( d^f \), \( e^f \), \( f^f \) cannot be as large as \( C^f \).

Therefore, \( a^f \), \( b^f \), \( c^f \), \( d^f \), \( e^f \), \( f^f \) are dependent on whether \( a^f \) or \( b^f \) is the maximum, and \( f^f \) is the minimum.

(2) \( a^f \), \( b^f \), \( c^f \), \( d^f \), \( e^f \), \( f^f \) can be as large as \( a^f \), \( b^f \), \( c^f \), \( d^f \), \( e^f \), \( f^f \).

Proof: (1) \( a^f + f^f + 1 \) (the greatest of \( a^f \), \( b^f \), \( c^f \), \( d^f \), \( e^f \), \( f^f \)) is greater than or equal to \( a^f + f^f + 1 \) and \( f^f + 1 \) is that is greatest with \( a^f \), \( b^f \), \( c^f \), \( d^f \), \( e^f \), \( f^f \) when \( a^f + f^f + 1 \) \( b^f + f^f + 1 \) \( c^f + f^f + 1 \) \( d^f + f^f + 1 \) \( e^f + f^f + 1 \) \( f^f + f^f + 1 \) are ordered.

We call this in inputs when completely ordered (as \( a^f > \cdots > z^f \) where \( u^f_1, \ldots, z^f_1 \) where there is always a larger number than the previous number).

Theorem (note here that + when used with subscripts is ordinary addition, not concatenation). We introduce a macro which we call \( \text{split} \) which is based on the following.

We get the products of binary variables. Further, we have shown that the function corresponding to these variables is the same as the product of the input tuples of 0's and 1's, and not on all input tuples as

-9-
By our theorem, the max of each of these pairs is $\geq C_{n+1}$; therefore, the set of max's is the n largest inputs.

We shall make much use of this splitting operation in the work which follows.

Another operation which we shall call upon frequently is the operation which makes an $n-1$ sorter from an $n$-sorter, by 'peeling'. Given an $n$-sorter, we can form an $n-1$ sorter by assigning one of our inputs to be the maximum input and eliminating all the comparisons through which it passes.

This operation is much easier to illustrate with a network than a program, but the translation between the two representations is obvious.

**Example:** To construct a 3-sorter from a 4-sorter

- Assume $a$ is the maximum input

The $x$'s mark the comparisons which involve the maximum input, $a$. We can eliminate these comparisons from the 4-sort algorithm yielding the
To form a 5-sorter, peel away 3 inputs

Starting with an optimal 8-sorter

\[
\begin{cases}
\text{Input} \rightarrow \text{Input inserted} \\
5 \text{ sorter} \\
\text{Example 4-sorter} \\
\text{sorter from an n-sorter is to insert the n+1st input} \\
\text{A conventional, although not necessarily optimal, way to design an n+1} \\
\text{resulting 2-sorter}
\end{cases}
\]
The remaining 5-sorter is exactly the same as the five sorter designed by insertion above. Given these constructions and any good general construction for building a \(2^K\)-sorter (e.g., the Batcher odd-even sort merge strategy (Batcher, 1968)), we can now define a macro \(\text{sort}(K)\) which sorts \(K\) inputs.

Thus far we have defined two macros \(\text{sort}(K)\) and \(\text{split}(K)\) which takes two \(K\) input sorted sequences as inputs and splits these sequences into two sets - the \(K\) largest and the \(K\) smallest inputs.

We must also define a pair of macros \(\text{max}(n)\) and \(\text{min}(n)\) which find the maximum and minimum of \(n\) inputs; and a merge macro (again we use, for example, the Batcher merge (Batcher, 1968) which is currently the best merge known.

Our attention has been directed towards the problem of finding the \(i^{\text{th}}\) largest input. We know that for the first or second (from either end) \(n-1\) and \(2n-3\) comparisons, respectively are required. There are no good bounds known for the other outputs. It was suspected that an NAA to find the median would be as complex (require as many comparisons) as an NAA to sort. We will show below that this is not necessarily the case.

Our basic strategy will be to home in on the appropriate output by a series of sorts, splits, merges, maxs and mins, eliminating inputs as possible candidates until we produce the final result.

The cost of the sort and merge macros grows faster than linearly with the size of the input sets. Therefore, it would seem a reasonable tactic to perform sorts and merges on as small sets as possible consistent with our problem. Thus, a 'divide and conquer' strategy is suggested for sorts and merges. This advice suggests that we begin our operation by dividing the input set into smaller groups and then sorting these groups. If we are looking for the \(i^{\text{th}}\) input, and we divide the input set into groups larger than \(i\) and sort them, then any element beyond the \(i^{\text{th}}\) of each group is beyond the \(i^{\text{th}}\) of the whole set and can be disregarded in the later stages of the algorithm. On the other hand, if we divide the input set into sets smaller than, or equal
Example 2: Find (2, 16) the fourth largest of either. (i.e., compute \( \min (4) \))

Here our advice leads us to follow several paths. First we can divide the input set into two sets of equal or nearly equal size. Since the split operation destroys order information, we should divide the input set into approximately equal sets of size larger at that stage of the algorithm. Therefore, as a first step, we sort them, then we cannot eliminate any possible candidates to size 1 and sort them, then we cannot eliminate any possible candidates.
1) two sets of eight or
2) four sets of four or
3) eight sets of two

In each case we start out by sorting. In strategy 1, we eliminate the six smallest inputs from each of the two sorted sets; in strategy 2, we eliminate the two smallest inputs from each of the four sorted sets. In strategy 3, we can eliminate nothing at this point. At the next point, for strategy 1, we can merge the two ordered two-sets and pick the third or split them and take the min of the max set---both alternatives are of the same cost. For strategy 2, we can merge each of the pairs of ordered two-sets which remain and then merge the two smallest from each pair; or we can split the pairs and sort the remaining four candidates. Either strategy is of the same cost. Finally, in strategy 3 nothing can be discarded, so at this point we introduce four pairwise splits which leaves us with eight unordered inputs. Then we can apply the 3 in 8 algorithm (which appears in subsection 5. below).

The three strategies yield the following algorithms.

1) sort 8; sort 8; merge (2,2)
   or sort 8; sort 8; split (2,2); min
2) sort 4; sort 4; sort 4; sort 4; merge (2,2); merge (2,2); merge (2,2)
   or sort 4; sort 4; sort 4; sort 4; split (2,2); split (2,2); sort 4
3) sort 2; sort 2; sort 2; sort 2; sort 2; sort 2; sort 2; sort 2; split (2,2);
   split (2,2); split (2,2); split (2,2); find (3,8).
   where find (3,8) is find 3 in 8 and involves repeating
   the procedure with the new parameters.

Actually algorithm 1 is noticeably more complex than any of the others because of the difficulty of the 8-sorts which require a lot of computation.

At this point we should discuss the relation between splitting and merging. The split operation has the obvious advantage that it allows us to eliminate several candidate answers in a single operation. However, it destroys order information---the outputs are unordered sets---which may have to be re-sorted in the later stages of the algorithm. It may, therefore, pay in some cases to merge sorted sequences rather than to split and re-sort from scratch. In both strategies 1 and 2 above, it appeared to make no difference (from the viewpoint of total number of comparisons) which strategy was adapted.
When looking for the \( T \)th output of a sorting algorithm, we first divide the size of the input array by the number of the output source. It should be nearly as possible to know the size of groups chosen for sorting in the first step. The procedure is as follows:

1. Find the input size of the groups.

2. Sort the input size of the groups.

3. Print the result of the groups.

4. Merge the groups.

If the latter, a merge or a merge of either be a Min (or a merge).

A merge of either in the input size of the groups. Then the number of inputs.

In order to sort groups which are small than large--the complexity of this algorithm depends on the following advantage.

One alternative advantage leads to your alternative algorithms. These are:

- Find (g, 16)
- Print (g, 16)
- Sort (g); print (g)
- Sort (g); print (g)

Operations which follows from our general rules.

This is another example where there is only one reasonable sequence of steps. Our alternative advantage is sort (g); print (g), sort (g), print (g), sort (g); print (g).

And finally take the min of the remaining groups. If your size is on three, then we split on three, disconnect the fourth member of each group of four, then we split on three, disconnect the fourth member of each group of four are obvious candidates. Therefore, we start by sorting the two following our advantage. We need groups of at least three in size--groups.

I) Find (g, 8)
5) The number of strategies to be tried has to do with how \( i \) divides the size of the input set.

It is not clear yet whether merging or splitting is the better strategy (if indeed there is a better strategy). It is hoped that experimenting with computer generated algorithms for large numbers of inputs will provide some insight in that direction. This phase of the research has produced the following strategy for generating 'good' selection algorithms.

A procedure for generating an algorithm to find the \( i^{th} \) out of \( n \):

(Note: that we assume \( i \leq n \)—only the distance from 1 or \( n \) is significant—and the results carry over to the other case by symmetry)

1) Partition the set into as equal as possible parts subject to the constraint that each part be \( \geq i \).

2) For each of these partitionings, sort the partitioned elements,
   i) Try merging the sorted sets (using Batcher odd-even merge)
      discarding (before merging) any elements beyond the \( i^{th} \) in each set.
   ii) Try splitting the sorted sets discarding (before splitting) any
       elements beyond the \( i^{th} \) in each set.

3) In the case that the strategy i has been adopted, continue to merge
   discarding elements beyond the \( i^{th} \) in each set.

4) In the case that strategy ii has been adopted, repeat the procedure
   from step 1 for the resulting set.

5) Compare the complexity for all the algorithms which are generated and
   select the best ones.

This program generator has been programmed in ILISP and runs on the Rutgers 10—it is called NAA. How good is NAA? How good are the programs it generates? It's pretty good, but not optimal, in fact it doesn't even produce programs as good as a clever human programmer. There are two reasons for this deficiency. First we don't in general know any regular procedure for producing optimal sort macros. We don't want to tackle that problem here and in our evaluation factor that part of the problem out by substituting the best (smallest number of comparisons) known sorting NAA into our evaluation procedure. However, NAA failed to produce best algorithms for another reason—it lacked specific domain knowledge. A
Several cases and NaNs produce algorithms which meet those values.

Example: Find (1, 2, 3).

(1) Sort 4
(2) Eliminate 4
(3) Insert 3
(4) Eliminate 1
(5) Insert 2
(6) Eliminate 1

Operation "ELIMINATE" which does not require any comparisons.

Operation "INSERT" which inserts another number into a sorted sequence. It works at a cost of n, whereas the new operation "INSERT" which inserts a new element into a new sequence at a cost of n does not require any comparisons.

"INSERT" inserts the new element into the sequence at the desired position. Then, it sorts the remaining elements into the sorted set one at a time eliminating any duplicates which can not be the desired one as the search begins, and then inserts the remaining elements into the sorted set to form the largest k smaller than 1.

Essentially, this knowledge is encoded onto an alternative approach to sorting.

Known in all the cases we have tried.

Some additional knowledge about the numbers to produce the best algorithms we have written an in-year algorithm that was called NaN which uses consideration as the algorithm progresses.

Order properties of the numbers to eliminate possible candidates from human decisioning these algorithms would take advantage of knowledge of the
In summary, we studied the automatic generation of programs for the solution of a particular class of problems. We succeeded in introducing enough expertise into the program generating procedure to produce extremely good and possibly optimal programs for that class of problems.

References


Van Voorhis, D., 71 Ph.D. Thesis
Stanford University, Stanford, California (1977)
ARCHITECTURE OF COHERENT INFORMATION SYSTEMS: A GENERAL PROBLEM SOLVING SYSTEM

C.V. Srinivasan

Department of Computer Science
Rutgers University
New Brunswick, New Jersey
The Architecture of Coherent Information System: A General Problem Solving System

CHITOOR V. SRINIVASAN, MEMBER, IEEE

Abstract—This paper discusses the architecture of a metasystem, which can be used to generate intelligent information systems for different domains of discourse. It points out the kinds of knowledge accepted by the system, and the way the knowledge is used to do nontrivial problem solving. The organization of the system makes it possible for it to function in the context of a large and expanding data base. The metasystem provides a basis for the definition of the concept of machine understanding in terms of the models that the machine can build in a domain, and the way it can use the models.

Index Terms—General problem solving (GPS), knowledge based systems, metadescription systems (MDS), model based reasoning.

I. INTRODUCTION

Our objective is to create a metasystem which can be used to generate intelligent information systems in different domains of discourse. The metasystem is called the metadescription system (MDS). It has facilities to accept definitions of description schemas and descriptions themselves, of knowledge—about facts, objects, processes, and problem solving—in a domain.

A domain might be a disease system, a piece of mathematics, or computing systems themselves. The description schemas and descriptions of knowledge in a domain specialize the MDS to act as an intelligent information system for the domain. For a domain M, the information system associated with it is called the coherent information system (CIS) of M.

In our research we have two principal concerns. 1) How may one describe knowledge in a domain to a computer; what kinds of knowledge should a system have to exhibit intelligent behaviour; what operational facilities are needed to accept and use such knowledge? 2) How may the computer be made to use given knowledge automatically to solve problems in the domain and answer questions?

The MDS accepts and uses three kinds of knowledge. 1) Structural knowledge pertaining to the form and syntax of descriptions. Descriptions may, of course, be strings of words in some language. The MDS will translate such descriptions to structures within a relational system. The relational system itself may consist of constants, variables, predicate symbols, function symbols, logical operators and quantifiers. The structural knowledge specifies the structure of the relational system used in a domain. 2) Sense knowledge: Logical assertions pertaining to the sense in which structures are interpreted, and constraints on admissible structures beyond those specified in the syntax. 3) Transformational knowledge: This pertains to the knowledge necessary to transform given descriptions of specific objects to new ones, according to specified criteria.

Corresponding to these three levels of knowledge there is a hierarchy of problem solvers, checker-instantiator, theorem prover (TP) and designer, in order of increasing complexity. The checker-instantiator system acts as a sophisticated data management system that establishes, maintains and updates the data base of models of specific objects in a domain in a manner consistent with the structural and sense knowledge. Checker can answer questions pertaining to any of the specific models for which the information is either directly stored in the data base, or is directly derivable by evaluating a given logical assertion in a given context. The TP adds power to the checker in three ways. In certain cases it helps reduce the search effort of checker by giving it advice based on deduced consequences of sense knowledge; where feasible it can warn the checker of impossible situations in the generation and updating of models; it can also determine general truth values of assertions based on the structure and sense knowledge. The designer adds further power to the system by enabling the system to plan courses of actions using given action primitives (transformation rules) in a manner consistent with the facts of a problem. This hierarchy imposes a very useful classification of system facilities, and gives the system a considerable flexibility.

The descriptive language of a domain is itself specified in terms of the model definitions in the domain. Language analysis is thus looked at as a model building process. Most importantly, the model definitions in a domain may include definitions of problem solving states (PSS), relevant to the domain. The PSS may provide facilities to summarize the problem solving experience of the system. This summary may be used to intelligently guide the problem solver.

This work on MDS and CIS systems may be thought of essentially as a further extension of the trend started by REF-ARP [1], [2], QA4 [3], POPS [4], STRIPS [5], [6].
Best Available Copy
and Planner [7]. Its problem solving activity uses 
means-end analysis, a concept originally introduced in 
general problem-solving (GPS) [8], and function in-
voicing schemes based on goals, introduced by Plan-
er. CI systems have both the flexibility of Planner-like 
systems, and model-based reasoning abilities of a GPS-
like system. The entire system depends on the way de-
scriptive data structures are organized in a given do-
main. However, the availability of data-structure and 
model definition facilities, and a separate data manage-
ment system makes it possible to completely isolate the 
data-structure and data base details from the problem 
solving programs. This makes it possible to conceive of 
the metasystem, the MDS, to create CI systems for dif-
ferent domains. It seems reasonable then, if the classes 
of possible models of objects in a domain could be de-
scribed to a computer then, in principle, the computer 
should be able to make use of the descriptions for prob-
lem solving and language understanding in the domain.
In CI systems we show how 1) classes of models can be 
defined and 2) how the definitions could be used for 
language analysis and problem solving in the domain.

The principal contributions of the proposed architec-
ture are:

1) a facility to use large data bases;
2) a stratification of knowledge in a domain and the 
ability to use a highly flexible descriptive mechanism to 
describe objects and problems in a domain; the possibil-
ity of describing knowledge in a domain in a system ad-
apted to a computer;
3) the definition of the descriptive language itself in 
terms of the models the system can build in a domain; and
4) the possibility of specializing the MDS to operate 
efficiently as a problem solving system in a domain of 
discourse.

The MDS is now being implemented in Interlisp. 
Some parts of it (see Section 10) are now ready. This 
paper is, therefore, a report on work currently in pro-
gress. It introduces the principal architectural concepts 
of MDS and CI systems in the context of an example, 
the missionaries and cannibals problem (M&C) [9].

The structure of checker and designer is explained. The 
operation of the TP is discussed in [10].

II. AN OVERVIEW OF THE SYSTEM 
ARCHITECTURE

A. Templates and Their Instantiations

1) The Templates: The concept of templates, the de-
vises used to specify structural knowledge is central to 
the entire system architecture. Templates classify 
objects in a domain into objects of different kinds and 

The pair of relation symbols (occupants, occupants) 
for example, are inverses of each other in the sense that 
in instances of PLACE and PEOPLE the relations (PLACE 
occupants PEOPLE) and (PEOPLE occupants of PLACE) 
will always appear together in the database of models. 
PEOPLE is just a list of PERSONS. An instance of type 
classification occurs in the PERSON template. A PER-
SON can be a MISSIONARY or CANNIBAL. In MDS type 
classification always reflects distinctions in the way 
objects are used. The templates thus specify the struc-
ture of the relational system for a domain; the relation 
symbols to be used in the description of various kinds of 
objects in the domain, and the kinds of objects that a 
relation symbol may relate.

Given such templates, one may use the instantiator to 
create descriptions, which are instances of the tem-
plates. Such instances might be specified to the system 
in some external language, which is translated to the 
internal representation in the relational system. Or, the 
system itself might generate an instance of a template 
when called upon to do so. In either case, to complete 
the instantiation of a template, all the relation symbols 
defined for the template should be assigned values. 
These values will be specific instances of objects within 
the database.

Thus for the M&C problem one may create instances 
of PLACE's called RHANKI and RHANK2, a VEHICLE 
called BOAT, and as many MISSIONARIES and CANNI-
BAL's as necessary. Each PERSON will be the occupant 
of some PLACE and the VEHICLE itself will be at one of 
the PLACE's. We have not, however, introduced any of 
the conditions of the problem. Not all instantiations of 
the templates of the M&C problem would represent 
legal situations. The necessary additional constraints 
are introduced by the sense knowledge. Every relation 
symbol in a template may have a consistency condition 
(CC) associated with it. CC1 in Table 1 is associated 
with the symbol "occupants". It says that the CANNI-
BAL's at a PLACE cannot outnumber the missionaries. 
The symbol "!" in CC1 refers to the current instance of 
PLACE at which the CC might be evaluated. It is 
called the anchor. (PEOPLE X) stands for "(VX) (X is 
PEOPLE)". All CCs have the form: "(! r P(X))" where 
"!" is the anchor, r is a relation symbol occurring in the 
template associated with "!", and P(X) is some logical 
predicate. The predicate P(X) is said to be anchored at 
the (template, relation symbol) pair. Thus, the predi-
 cate in [CC1] is anchored at (PLACE, occupants).

In [CC1] notice that "(! occupants X)" itself is a 
term in its predicate. This has the following signifi-
cance: For a PLACE like, say RHANK1, if the system is 
told to set (RHANK1 occupants y) for some y, it would

There are three missionaries and three cannibals on one bank of a 
river. They want to go to the other bank. There is only one boat avail-
able. It can carry only two people at a time. The cannibals at a shore 
should not outnumber the missionaries at the same shore. Find a way 
of transporting them.
first construct the combined list of existing occupants of RBANK1 and y, and then verify the predicate. CC’s of this kind are called declarative CC’s, as opposed to the other kind, called imperative CC’s, like, say (for a hypothetical template PERSON1)

[CS1] (*) sibling ((PERSON X),(NOT (X is *))

(X child of father of *))

[CS1] may be used to find the siblings of a PERSON in terms of the child of and father of relation symbols. The checker is used to evaluate CC’s. We shall discuss the evaluator in Section II-B.

The significant points to be noted about CC’s are the following.

1) The knowledge represented by the CC’s is of a different kind from the structural knowledge, specified by the templates.

2) Each CC is specifically associated with a particular relation symbol. A relation symbol, say “likes”, might be quite different in the context (HUMAN likes SOMETHING), from (CATTLE likes SOMETHING). A CC is invoked and interpreted only within the particular local context of its anchor, within the overall structure of descriptions.

3) The logic of the CC’s is highly dependent on the structures specified by templates. Also, for a given system of templates there may be more than one way of choosing and anchoring the CC’s. Further, for a given domain, there will undoubtedly be several ways of defining the templates and its associated CC’s. These different definitions will correspond to different ways of representing the knowledge in the domain. The MDS provides facilities to experiment with different choices.

At present we have no formal guidelines to make these choices intelligently. The particular choices made in a domain will have an effect on system efficiency.

2) Instantiation of Templates: We shall call an instance of a template as the model of the object instantiated. Thus, the model of RBANK1 will be an instance of PLACE. Every triplet (x r y) (where r is a relation symbol) appearing in the model x should be dimensionally consistent. That is, for some templates M and T, where x is an instance of M and y is an instance of T, either (M r T) occurs in M, or (T r T) occurs in T, where r is the inverse of r. There are a few relation symbols which are system wide, like template of, name of, elements of, arguments of, etc., which can appear with all instances in the data base, and need not be defined in the templates.

The model of RBANK1 will be a vector of five pointers, say (P1, P0, P0, P0, P0) corresponding to the relations template of, name, elements of, occupants and position of, respectively. P1 will point to a pair (P1, P1), where P1 points to the PLACE template, and P1 to possibly local conditions (LC’s) associated with RBANK1. P0 will, of course, point to “RBANK1”. Let r be any one of the remaining relations: P1 will point to a quintuple of the form (#, P2, P2, P2, P2), called the descriptor unit of P1 (or r). The elements of the descriptor unit are the following.

**Descriptor Unit**

P1 Pointer to y such that (RBANK1 r y) is true, or pointer to list (y) such that (RBANK1 r z) is true for every z ∈ y. We shall write this as (RBANK1 r (y)).

P2 Pointer to list (y) such that for every z ∈ y, (NOT (RBANK1 r z)) is true.

P3 Local conditions on values of (RBANK1 r). P3 To transformation rules (TR’s) local to (RBANK1 r), called LTR’s.

The number of elements in the list, set or tuple pointed to by P1.

Every P1 will have an inverse, say P1, which will point back to RBANK1, P1 is the same as P1. The inverse of (P1, P2) will be (P1, P2), where P1 is the same as P1 (i for instance); P1 will point to RBANK1 from PLACE template.

A pointer in a model can have one of four values: not stored (NS), not enough information (NEI), NUL, or an address (or value). Initially all pointers in a model are set to NEI. A list, set or tuple will have NEI as an element if it is incomplete. Templates thus specify the data structures of models in a domain. They provide the basic framework for the organization of domain dependent knowledge. They also play a major role in the specification and use of problem solving programs in a domain, as we shall see in Sections II-B and II-C.

There are about fifteen different kinds of templates in the MDS. Variations in the structure of descriptions may be specified by defining, what are called variable
templates. Exceptions to the CC's may be specified by associating local conditions (LC's) with specific instances of templates. An LC may be a conjunctive LC (CLC) or a disjunctive LC (DLC). A model should satisfy ((CC ∧ CLC) ∨ DLC) at each one of its relation symbols. Similarly, transformation rules (TR's) for changing a model may be local to a model (LTR) or may be associated with templates themselves.

In addition to the CC's associated with pairs (M, r), where M is a template and r is a relation symbol, r may also have properties (PR) defined for it, which apply to all occurrences r within the relational system. A typical such property is the transitivity property. For a model m, the PR's are used to identify objects y, such that (m r y) is true, but is not stored in the data base.

All problem solving programs communicate with the data base via the instantiator and checker. Templates also provide a way of classifying and storing the CC's, LC's, TR's, and LTR's. Every CC (LC) is anchored at (m, r) where m is a template (or model) and r is a relation symbol (relation) defined on m. The DON list of a CC (LC, TR, or LTR) is the list of (m, r) on which it depends. The DRT list of a pair (m, r) is the list (m, r) of pairs depending on (m, r). So also, the DET list of a TR (LTR) is the list of (m, r) which are affected by the TR (LTR). The DON and DET lists are stored with each CC, LC, TR and LTR. Also, every pair (m, r) will have a pointer to its associated CC, LC, TR, and LTR.

B. Evaluation of Consistency Conditions

1) The Logic of Checker: The evaluation of CC's, LC's and PR's will involve searching of the models in data base. The conditions themselves specify the search paths. The anchor "*" may be used to optimize this search. One can write small efficient programs to evaluate these conditions. (Alternatively one may compile each CC and PR individually.) The checker is the interpreter for CC's, LC's, and PR's. It uses the G function (G for Get) of the instantiator to retrieve objects from data base. G does the following:

(Q1) G(x r ?) = {y | (x r y)} (This may include NEI), or NIL or NEI.
(Q2) G(x r y) = YES, NIL or NEI
(Q3) G(x ? y) = p, NIL

where p is a relation path (r1, r2, ..., rk) joining X and Y (the shortest one). G just looks up the data base using the templates. If the answer is NS, NEI, or if NEI is included in the answer, then G will invoke the checker to evaluate the associated CC's, LC's, and PR's. This evaluation may cause the NEI to be removed and possibly add new elements to (y).

The checker operates on a three-valued logic system having truth values T (TRUE), F (FALSE), and ? (NEI). The logic of checker is shown in Table II. For a given predicate P, besides returning its logical value (one of T, ?, F), the checker also returns seven other quantities, all of which will be subexpressions of P. These are explained below.

In all the following functions assume that \( \phi \) is a particular valuation of the variables in \( P \). \( \phi(P) = T, F, or ? \).

\[ TR_\phi(P) \quad \text{True residue of } P \text{ that caused } \phi(P) = T. \]
\[ FR_\phi(P) \quad \text{False residue of } P \text{ that caused } \phi(P) = F. \]
\[ R_\phi(P) \quad \text{Residue of } P \text{ that satisfied } \phi(P) = ?. \]
\[ TP_\phi(P) \quad \text{True part of } P \text{ that evaluated to } T. \phi(P) \text{ itself may be } T, F, \text{ or } ?. \]
\[ FP_\phi(P) \quad \text{False part of } P \text{ that evaluated to } F. \phi(P) \text{ itself may be } T, F, \text{ or } ?. \]
\[ NTP_\phi(P) \quad \text{Not true part of } P \text{ that evaluated to } F. \phi(P) \text{ itself may be } T, F, \text{ or } ?. \]
\[ NFP_\phi(P) \quad \text{Not false part of } P \text{ that evaluated to } T. \phi(P) \text{ itself may be } T, F, \text{ or } ?. \]

The various residues and parts of \( P \) are used in the planning phase of designer to summarize past experiences of the system. In the TP the residues are used to guide the problem solving search. The properties of these functions that make them useful are given below. The logic of these functions is shown in Table II.

Let \( \phi \) and \( \psi \) denote valuations of the variables in \( P \). Assume in what follows, that a certain task \( k \) was done either by the TP or designer, and the outcome of the task \( k \) depended on \( P \). Also, at the time \( k \) was attempted, \( \psi \) was the valuation of the variables in \( P \). At a later time, a new valuation \( \phi \) of the variables \( P \) was obtained, and the system had to decide whether to attempt \( k \) for the new valuation.

1) \((VP)(V\psi)(V\psi)(\psi(TR_\psi(P)) \rightarrow \psi(P)).\)

If \( \psi(TR_\psi(P)) \) is true; then try \( k \) again.

2) \((VP)(V\psi)(\psi(\neg \psi(FR_\psi(P)) \rightarrow \neg \psi(P)).\)

Do not try \( k \) if \( \neg \psi(FR_\psi(P)) = F. \)

3) \((VP)(V\psi)(\exists \psi)\{(\psi(NTP_\psi(P)))(\psi(TP_\psi(P)) \rightarrow \psi(P)).\)

A goal \( k \) could not be reached for valuation \( \psi \) because \( \neg \psi(P) = F. \) Then try and find a \( \psi \) for which \( \psi(NTP_\psi(P)) \) is true. This is used to break up a goal into subgoals, in means-end analysis.

4) \((VP)(V\psi)(\exists \psi)\{(\psi(R_\psi(P)))\psi(TP_\psi(P)) \rightarrow \psi(P)).\)

Here goal \( k \) could not be reached because \( \neg \psi(P) = ?, \) and the unknown parts of \( P \) are given by \( R_\psi(P) \). Then find a \( \psi \) (build new objects) for which \( \psi(R_\psi(P)) \) and \( \psi(TP_\psi(P)) \) are true. This is used in means-end analysis, and in the TP.

2) What Checker and Instantiator Can Do; What More is Needed: The checker and instantiator together act as a fairly sophisticated data base management system. The checker makes sure that data entered into the data base are consistent and also keeps track of what additional data are needed to complete the descriptions
TABLE II
Logic of Checker

<table>
<thead>
<tr>
<th>Literal $x$ denotes the value of $x.$</th>
<th>$x$</th>
<th>$T_X(x)$</th>
<th>$F_X(x)$</th>
<th>$B_X(x)$</th>
<th>$T_P(x)$</th>
<th>$P_X(x)$</th>
<th>$NTP_X(x)$</th>
<th>$NFP_X(x)$</th>
<th>$A$</th>
<th>$T$</th>
<th>$F$</th>
<th>$V$</th>
<th>$T$</th>
<th>$F$</th>
<th>$P_X$</th>
<th>$N_P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$x$</td>
<td>$T$</td>
<td>$T$</td>
<td>$x$</td>
<td>$T$</td>
<td>$T$</td>
<td>$x$</td>
<td>$x$</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$x$</td>
<td>$T$</td>
<td>$T$</td>
<td>$x$</td>
<td>$x$</td>
<td>$?</td>
<td>$?</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$?</td>
<td>$?</td>
<td>$?</td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$F$</td>
<td>$x$</td>
<td>$x$</td>
<td>$F$</td>
<td>$F$</td>
<td>$F$</td>
<td>$C$</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$?</td>
<td>$?</td>
<td>$F</td>
<td></td>
</tr>
</tbody>
</table>

Propositions: $P, Q$. Let $X$ denote one of $TR, FR, R, TP, FP, NTP, NFP$. Then

$$[x(P \land Q) \iff x(P \lor Q)]$$

is true.

Also $\land$ and $\lor$ are symmetric:

$$x(P \land Q) = x(Q \land P); \quad x(P \lor Q) = x(Q \lor P).$$

The various functions are defined below for $(P \land Q)$.

<table>
<thead>
<tr>
<th>$(P \land Q)$</th>
<th>$(T, T)$</th>
<th>$(T, F)$</th>
<th>$(T, F)$</th>
<th>$(T, F)$</th>
<th>$(F, F)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TR$</td>
<td>$[TR(q) \land TR(q)]$</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
<td>$F$</td>
</tr>
<tr>
<td>$FR$</td>
<td>$\land$</td>
<td>$T$</td>
<td>$F$</td>
<td>$FR(q)$</td>
<td>$FR(q)$</td>
</tr>
<tr>
<td>$R$</td>
<td>$T$</td>
<td>$R(q)$</td>
<td>$F$</td>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$TP$</td>
<td>$[TP(q) \land TP(q)]$</td>
<td>$TP(q)$</td>
<td>$TP(q)$</td>
<td>$T$</td>
<td>$F$</td>
</tr>
<tr>
<td>$FP$</td>
<td>$T$</td>
<td>$\land$</td>
<td>$TP(q)$</td>
<td>$TP(q)$</td>
<td>$F$</td>
</tr>
<tr>
<td>$NTP$</td>
<td>$T$</td>
<td>$NTP(q)$</td>
<td>$NTP(q)$</td>
<td>$[NTP(q) \land NTP(q)]$</td>
<td>$NTP(q)$</td>
</tr>
<tr>
<td>$NFP$</td>
<td>$[NFP(q) \land NFP(q)]$</td>
<td>$NFP(q)$</td>
<td>$[NFP(q) \land NFP(q)]$</td>
<td>$NFP(q)$</td>
<td>$NFP(q)$</td>
</tr>
</tbody>
</table>

of objects with respect to the templates. The templates describe the structure of the data base for the domain. The checker uses this structure to guide the instantiator to create and retrieve items in the data base selectively.

The limitations of the checker arise in the automatic guidance it can provide in the updating process. The checker has facilities to interpret individual CC's and to recognize the relation symbols whose value in the data base might be affected as a result of a change made at one place in the data base. Checker keeps track of the relation symbol, by cataloging the relation symbols in terms of their appearances in the various CC's. In general, a change in the value of one relation symbol might propagate through the data base to a series of other relation symbol values. As long as any given instance of the value of a relation symbol does not repeat itself in this series, checker will have no problems. It can execute the series of necessary changes without ever having to go back to a value that it had previously changed within the sequence.

Checker simply performs search in the data base, and logical combinations of search. It has only simple facilities to keep track of alternate choices in search paths, and choices in possible valuations of relation symbols. Also, checker can handle only constants as possible valuations for relation symbols. When the number of alternatives is large or when loops occur in an updating chain, the checker, if left to run will keep assigning new values to the relation symbols involved until a consistent set of valuations is obtained, or until all known possibilities are exhausted. The only choices it can generate are those that are already available in the data base, or those that may be obtained by evaluating specific consistency conditions in specific local contexts. It does not have the capability to deduce logical consequences and make use of them to find contradictions where possible. To do this general theorem proving capability is necessary. The essential difference between the checker and a TP is the following. Whereas the checker can assign as values to relation symbols only specific constants in the data base, the TP can assign as values, variables with specified logical properties. The TP can carry with it the logical properties assigned to variables and use them in making new assignments as it goes along. Resolution based theorem proving systems have this capability built into the unification algorithm (see Nilsson, 1971).
In MDS the checker will invoke the TP whenever it does not find enough information in the data base to evaluate a CC at a particular anchor, or whenever the validity of an assertion is to be proven universally; not merely with respect to the facts known about the specific objects in the data base. The checker will call the TP also when it recognizes a loop in an updating chain.

The deduction process and the control structure of the TP in MDS is different from that of a resolution based system (see [10]).

B. The Dynamic Aspects of Modeling: The Transformation Rules and Their Interpretation

1) The Primitives: There are about 20 primitives that enable one to do programming in a backtracking environment. The primitives are classified as shown in Fig. 1(a). The ECP's environmental control primitives ECP's in Fig. 1(a) are used to establish a control environment (environ) within a scope. The execution of functions within the scope are affected by it. See Table III for a description of the ECP's. The sequential control primitives (SCP's) like GO, CONJ, etc. There are seven active primitives, GOAL, ASSERT, DELETE, BIND, IFDON, TRY, and TRY. The execution sequence for the GOAL and other active commands are shown in Fig. 1(b) and (c). GOAL invokes appropriate definitions from data base, and does “means-end” analysis when necessary. ASSERT and DELETE issue I and D commands to the instantiator, when successful. All primitives, other than the control primitives, may have CANDO, IFDON, and TRY functions associated with them. A primitive can be executed only if its associated CANDO's are satisfied. If a primitive fails then one may try its associated TRY functions. If a primitive is successful then its associated IFDON's should be executed. Only if the IFDON's are also successfully completed may the primitive return success to its parent. Let us follow the operation with an example.

2) Interpretation of the Active Primitives: The syntax of the various active primitives is shown in Table IV. The designer is the interpreter for the primitives. Consider, for example, the (dimension) (see syntax of (dimension) in Table IV) of the GOAL function TR1 in Table V.

\[
\begin{align*}
(\text{GOAL} \quad (Q \text{ occupants } X)) \\
\text{bindings} \\
\text{fn-clause}
\end{align*}
\]

The function call that will cause this TR1 to be invoked is

\[
\begin{align*}
(\text{PEOPLE } X) \quad (\text{PLACE } P \quad Q) \quad (P \text{ occupants } X) \\
\text{fn-clause} \\
(\text{GOAL} \quad (Q \text{ occupants } X)) \\
\text{bindings} \\
\end{align*}
\]

Let us follow the interpretation of this function call, as specified in Fig. 1(b).

a) Find Possible Bindings: The checker is used to bind variables in a (dimension) statement. We shall assume that the (proposition) in the GOAL clause is always in disjunctive normal form. In the above case X will be bound to (M1 M2 M3 C1 C2 C3). If the checker returns NE1, or a loop is encountered then the TP may be invoked to complete the bindings. Unless the IDB clause (see Table III for an explanation of the IDB clause) is present the TP will create new objects, if necessary to complete the bindings.

b) Find Initial Conditions: This is done by checking whether the GOAL is already satisfied in the data base for specific bindings of variables. In the case of our example, this will bring out the fact, (RBANK1 occupants (M1 M2 M3, C1 C2 C3)). This will cause the following invocation pattern to be built:

\[
(\text{GOAL} \quad (X3 \text{ occupants } X1)) \\
\text{fn-clause}
\]

Let \( b \) be the (bindings) and \( g \) the (proposition) of the (fn-clause). The canonical form of an invocation pattern (also a (dimension) after binding the variables) is

\[
(\text{bound quantifiers}) \quad (b, g; \forall \rightarrow (b, g))
\]

where each \( b_i \) and \( g_i \) is a conjunction of terms, possibly with OPNL, IPND, IDB or * clauses. Let \( D \) be an invocation pattern and \( D_i \) any (dimension) in the data base. \( D_i \) and \( D \) are said to match, \( (D_i \rightarrow D) \) if there exist bindings for the variables in \( D_i \) such that for some \( b_j \) in \( D_i \) and \( b_k \) in \( D \) (\( b_j \rightarrow b_k \)) and \( (g_j \rightarrow g_k) \), and in addition \( b_i \) is true in the data base. The \( b_i \) will be the initial conditions. The invocation process will retrieve all \( D_i \) that match \( D \).

If no such functions, \( D_i \), are available then the designer will force the GOAL by issuing the appropriate ASSERT and DELETE commands. These will cause their associated CANDO's to be executed. If the CANDO's succeed then the corresponding I and D commands will be tried. This will cause the associated CC's to be evaluated at the given bindings of the variables, say \( \psi \). If the CC's are not satisfied then the not true part of \( \psi \) (NTP\( \psi \)) and true part (TP\( \psi \)) will be issued as subgoals. If the CANDO's are not satisfied then the goal will be abandoned.

In general, both the binding and invocations processes will return more than one possible course of action. In both these cases the problem solver needs to be guided intelligently in making its choices. The DESIGNER has some built-in facilities for intelligent selection of choices from a set of alternatives. The problem solving state (PSS) provides this guidance. This is discussed in the next section.

3) The PSS: The PSS itself is defined by templates. The PSS template is shown in Table VI. This table is self-explanatory. Every time the designer invokes a function or executes a (fn-call) it will create an instance
GOAL FUNCTION CALL

Create a PSS instance, pss. 1.) Record the call and enter all the appropriate information in pss linking it to other PSS instances.

Find all possible variable bindings
Enter bindings in pss.

- No < ANY BINDINGS LEFT? > No
  Yes

Choose the next one. Use pss to make the choice. Enter choice in pss.
Find initial and terminal conditions.

< IS GOAL ALREADY SATISFIED IN THE DATA BASE? > Yes --> SUCCESS

No

Build invocation pattern. Invoke all implied functions.
Update pss.

< FIND ANY (MORE) FUNCTIONS? > No -->
Yes

Choose next function using pss
< ANY CANDO'S? > No -->
Yes

Generate sub-goals go into subgoal search mode do means end analysis.

RETURN FAILURE

RETURN SUCCESS

EXECUTE FUNCTION

The flow-chart (6b) is used for this purpose.

Fig. 1. Classification of primitives and their execution sequences.
ACTION FUNCTION CALL (NOT A GOAL FN.) CREATE A PSS INSTANCE, PSS, TO RECORD THE CALL AND ENTER ALL APPROPRIATE INFORMATION IN PSS, LINKING IT TO OTHER PSS INSTANCES.

FIND ALL POSSIBLE VARIABLE BINDINGS
Enter bindings in PSS.

IF FUNCTION IS ASSERT OR DELETE THEN CHECK DATA BASE:
< IS STATEMENT ALREADY TRUE?  NO>

RETURN SUCCESS
(Update PSS)

IF FUNCTION IS ASSERT OR DELETE THEN CHECK DATA BASE:
< IS STATEMENT ALREADY TRUE?  NO>

NO < ANY CANDO'S? >

RETURN SUCCESS
(Update PSS)

IF FUNCTION IS ASSERT OR DELETE THEN CHECK DATA BASE:
< IS STATEMENT ALREADY TRUE?  NO>

NO < ANY CANDO'S? >

RETURN SUCCESS
(Update PSS)

Fig. 1. Continued.

of PSS corresponding to the function. The network of all such PSS instances is the problem solving protocol. The CC's associated with the PSS template provide the necessary guidance to DESIGNER. Of particular interest are the CC's associated with the bindings and alternates (see Table VI) relations. Let us call these [CCB] and [CCF], respectively. These CC's will specify the choices of current bindings and current function. Two important notions that make this possible are the notions of similarity of two PSS instances, and cc summary of a PSS instance.

cc summary—[CCS]: A CCS is a record of evaluations of CC's, branching conditions, CANDO conditions and binding conditions, made during the tenure of a PSS instance. For each sequence of conditions evaluated, the CC summary will contain the TRUE RESIDUE's of the conditions evaluated if the condition evaluated to TRUE, the FALSE RESIDUE, NOT TRUE PART and TRUE PART if the condition evaluated to FALSE, the RESIDUE if the condition evaluated to NEI. It will also have the outcome (fn state) of the PSS instance in which the condition was evaluated, and specific variable bindings if any in terms of the kinds and types of objects used. All variable bindings in the CC summary of a PSS will be specified in terms of the variables that appear in the bindings of the PSS. The concept will become clear in the example considered below. The consistency condition [CCB] uses CC summaries.

The general rule is:pick for bindings the same kind and type of objects that previously succeeded in similar PSS instances; do not pick the kind and type of objects that previously failed. Use cc summaries to check whether a chosen binding is likely to succeed. If no bindings could be picked by the above rules, then pick arbitrarily.

To define the notion of similarity of two PSS states
we need some additional concepts. Let \( k \) be an arbitrary PSS instance defined as follows:

**h**: \((\text{dimension } D_k) \) (bindings \( B_k \)) (initial state \( I_k \)) (alternates \( A_k \)) (fn state \( S_k \)) (environments \( E_k \)) (cc summary \( CCS_k \)) (history \( H_k \)) (type \( T_k \))

**v_k** = \( [X_1, X_2, \ldots, X_n] \) be the variables appearing in the (dimension), \( D_k \). Let \( D_k \) itself be \( (Q_k, P_k, F_k) \) where \( Q_k \) is the (quantifiers) of \( D_k \), \( P_k \) its (proposi-
TABLE V
The Transformation Rules for the M&C Problem

\[(\text{TR1})\] Goal Function Definition. It says, "Pick up some people \(Y\), possibly as many as the vehicle will hold, load the vehicle with \(Y\), and take them to \(Q\)." "Proc" in the statementaborates procedures. It is similar to Proc in Lisp, but differs from Proc in the sense that Proc has backtracking.

\[\left\{\left(\text{people} \ X\right) \left(\text{place} \ P \ Q\right) \left(\text{P occupants} \ X\right) \left(\text{Goal} \left(\text{q occupants} \ X\right)\right)\right\}\]

\[\left(\text{repeat} \left(\text{q occupants} \ X\right)\right)\]

\[\left(\text{proc} \left(\text{some vehicle} \ V\right) \left(\text{some people} \ Y\right) \left(\text{p

1. \(\text{assert} \left(\text{v occupants} \ Y\right) \left(\text{not} \left(\text{p occupants} \ Y\right)\right)\right)\)
2. \(\text{assert} \left(\text{v position} \ Y\right) \left(\text{not} \left(\text{v position} \ P\right)\right)\)
3. \(\text{assert} \left(\text{v occupants are occupants of} \ Q\right) \left(\text{not} \left(\text{v occupants} \ Y\right)\right)\)

\[(\text{TR2})\] Cando Clause for Loading a Vehicle. It says, "Unload the vehicle first. If there are more people than the vehicle can hold then drop some and try again. If the vehicle is not at the same place as the people are, then bring it to where they are. Each try statement is preceded by the conditions on whose failure the try would be attempted.

\[\left(\text{cando} \left(\text{people} \ X\right) \left(\text{place} \ P \ Q\right) \left(\text{v vehicle} \ V\right) \left(\text{p

1. \(\text{assert} \left(\text{v position} \ P\right)\)
2. \(\left(\text{people} \ Z\right) \left(\text{p occupants} \ Z\right) \left(\text{not} \left(\text{v occupants} \ Z\right)\right)\)
3. \(\left(\text{v occupants are occupants of} \ V\right)\)
\[\left(\text{try} \left(\text{some people} \ Y\right) \left(\text{v among} \ X\right) \left(\text{assert} \left(\text{v occupants} \ Y\right) \left(\text{not} \left(\text{p occupants} \ Y\right)\right)\right)\)
\[\left(\text{if} \left(\text{and} \left(\text{assert} \left(\text{v occupants} \ X\right) \left(\text{not} \left(\text{p occupants} \ X\right)\right)\right)\right)\)

\[(\text{TR3})\] Cando Clause Associated with Bringing a Vehicle from a Place \(P\) to \(Q\). It says, "Get a Pilot and Take the Vehicle. If Pilot cannot be removed get some one to go with him."

\[\left(\text{cando} \left(\text{vehicle} \ V\right) \left(\text{place} \ P \ Q\right) \left(v\ right position) \left(\text{assert} \left(\text{v position} \ Q\right) \left(\text{not} \left(\text{v position} \ P\right)\right)\right)\)

1. \(\left(\text{some person} \ Y\right) \left(v\ right pilot \ Y\right) \left(\text{v\ right\ position} \ Y\right)\)

\[\left(\text{try} \left(\text{some person} \ Z\right) \left(v\ right pilot \ Z\right) \left(\text{p occupants} \ Z\right) \left(\text{assert} \left(\text{v\ right occupants} \ Z\right) \left(\text{not} \left(\text{p\ right occupants} \ Z\right)\right)\right)\)

\[\left(\text{try} \left(\text{some person} \ X \ Y\right) \left(v\ right\ pilot \ X \ Y\right) \left(\text{p occupants} \ X \ Y\right) \left(\text{assert} \left(\text{v\ right occupants} \ X \ Y\right) \left(\text{not} \left(\text{p\ right occupants} \ X \ Y\right)\right)\right)\)

TABLE VI
PSS Template

<table>
<thead>
<tr>
<th>(dimension)</th>
<th>DIMK</th>
<th>&quot;dimension&quot; of the function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(init. state)</td>
<td>MODEL STATE</td>
<td>Initial State of Model</td>
</tr>
<tr>
<td>(bindings)</td>
<td>BINDINGS</td>
<td>All possible variable bindings and current bindings</td>
</tr>
<tr>
<td>(alternates)</td>
<td>PSSL</td>
<td>PSS instances of possible alternates; also the currently active function</td>
</tr>
<tr>
<td>(cc-summary)</td>
<td>CSSL</td>
<td>Summary of CC's evaluated during the active tenure of in</td>
</tr>
<tr>
<td>(fn-state)</td>
<td>FNSATE</td>
<td>Function States: SUCCESS, FAILURE, SUSPENDED</td>
</tr>
<tr>
<td>(environment)</td>
<td>ENVIRON</td>
<td>The Control environment</td>
</tr>
<tr>
<td>(history)</td>
<td>PSSH</td>
<td>Previous instances of PSS with the same dimension</td>
</tr>
<tr>
<td>(type)</td>
<td>PSSTYPE</td>
<td>Any useful type classification of PSS</td>
</tr>
<tr>
<td>(final-state)</td>
<td>MODELSATE</td>
<td>Model changes done by the PSS instance</td>
</tr>
<tr>
<td>(successor)</td>
<td>PSSS</td>
<td>Possible successor functions</td>
</tr>
<tr>
<td>(predecessor)</td>
<td>PSSS</td>
<td>List of parent functions</td>
</tr>
<tr>
<td>(conditions)</td>
<td>COBOS</td>
<td>Conditions appearing in REPEAT, COND and other such statements that caused the current PSS to be invoked</td>
</tr>
</tbody>
</table>

The remaining text discusses the transformation rules for the M&C problem, including clauses for loading and unloading vehicles, bringing a vehicle from one place to another, and the PSS template. It also defines a function invocation process and discusses the structure of PSS instances.
Also, the type of $K$ is equal to the type of $J$.

2) they have the same control environment $E_K = E_J$; and

3) $V_K, N_K$ satisfy one or more of the cc summaries in $J$, for some binding $\beta_j \in B_j$. Thus, $V_j, N_j$ satisfy one or more cc summaries in $K$ for some binding $\beta_j \in B_j$.

$K$ is identical to $J$ if $D_K = D_J$, $T_K = T_J$, $E_K = E_J$, $B_K = B_J$, and $N_K = N_J$.

To understand how these work let us consider the solution of the M&C problem.

1) M&C Problem Solution: The sequence of possible function calls is shown in Fig. 2. Each box in Fig. 2 is labeled to indicate its correspondence to the functions in Table V. The boxes are numbered 1 through 15. For a box with number $i$, let $K_i$ denote its associated PSS instance.

Suppose we are at the beginning, and are at box 3 in Fig. 1. Then the following sequence of actions might happen (follow arrows in order):

$$(Y \leftrightarrow (M1, M2)) \rightarrow (Y \leftrightarrow (M, M)) \rightarrow (Y \leftrightarrow (M, M)), (P \leftrightarrow PLACE)$$

The indicated I and D commands are returned by the assert function, TR1-1 (see Table V). This will cause [CC4] and [CC1] to be evaluated, and the following cc summaries to be returned to box 3 (since box 3 is still active):

$[CC4]:[K_4, 0]$, $[CC1]:[K_4, 0]$.

Here $K_4$ is the PSS instance at which the evaluation took place. The variable bindings are indicated only in terms of the kinds and types of objects used. $T$ and $F$ are the CC evaluation outcomes, and $Fail$ is the outcome of $K_4$. Notice that moving these two cc summaries to $K_3$ makes it still possible for $K_3$ to use these, because the variables $P$ and $V$ have in $K_4$ the same bindings as they do in $K_3$ and none of the terms appearing in the CC's have changed in value between $K_3$ and $K_4$.

The failure of $K_4$ brings us back to $K_3$. Now the choice of next bindings to be tried will be guided by [CCB]. Either a $(M,C)$ or a $(C, C')$ will succeed. A more interesting case is the following. Now suppose that $(M1, C1)$ are already on RBANK2. This would have caused the following series of successful ee evaluations:

$[CC4']:[K_4', 0]$, $[CC1']:[K_4', 0]$.

All these cc summaries would now be available at box 3 of Fig. 2, since PROC would have been still active during the whole course of events.

After this, PROC will be reinvoked because IFIE will have been still active. The new instances of PSS for boxes 3 and 4 will be created. These will be similar to the previously created instances. The BOAT is now at RBANK2. The CANDO clause, box 5, (statement TR2 in Table V) will now cause the BOAT to be brought back with a pilot, who in this case will have been the missionary $M1$. This time when a new pair of people are picked from RBANK1, the system will already check for the satisfaction of the successful path of CC executions, depicted by the summaries $[CC4']$, $[CC1']$, $[CC2]$, $[CCS2]$. Picking another $(M, C)$ will in this case fail; $(C, C)$ will succeed in satisfying the cc summaries. Thus, with anticipation the system will pick the right candidates likely to lead to success. From here on the availability of the cc summaries, and the guidance provided by [CCB] will enable the system to always pick the right candidates. The following solution will be obtained.

**RBANK1**

| Step 1 | (M1, M2, M3, C1, C2, C3) |
| Step 2 | (M2, M3, C2, C3) |
| Step 3 | (M1, M2, M3) |
| Step 4 | (C1, M3) |
| Step 5 | (C1, M3, M1, C2) |
| Step 6 | (C1, C2) |
| Step 7 | (C1, C2, C3) |
| Step 8 | (C1) |
| Step 9 | (C1, C2) |
| Step 10 | 0 |

**RBANK2**

| Step 1 | (M1, M2, M3, C1, C2, C3) |
| Step 2 | (M2, M3, C2, C3) |
| Step 3 | (M1, M2, M3) |
| Step 4 | (C1, M3) |
| Step 5 | (C1, M3, C1, C2) |
| Step 6 | (C1, M2, M3) |
| Step 7 | (C1, M2, C3) |
| Step 8 | (C1, M1, M2, M3) |
| Step 9 | (C3, M1, M2, M3) |
| Step 10 | (M1, M2, M3, C1, C2, C3) |

Step 5 in the above solution is caused by box 14 in Fig. 2.

5) Summary: Thus, the designer provides the high-level control structure necessary to pass on to the checker the right CC's to be evaluated, and to the instantiate, the right model changes to be done. The designer programs themselves are independent of the descriptive data structures used. Again the templates and instantiate provide a desirable isolation. The PSS itself may be changed for different domains of discourse, or different problem types. In this sense, the templates and the rules of transformation, together with the PSS specialize the MDS to a given problem, or a given do-
I. CONCLUDING REMARKS

We have introduced the basic concepts of CI systems and MDS. The CI systems provide a basis for the definition of the concept of machine understanding in terms of models that a machine is capable of building in a domain, and the way the models are used. The understanding exhibited at the problem solving level of checker is relatively simple understanding. A deeper level of understanding is exhibited in the kinds of problems that the TP can solve (see [10]). At the level of designer the level of understanding is very sophisticated. The system is able to plan and build procedures to solve problems.

In this paper we have discussed only a part of the problem solving aspects of the system; the workings of the checker and designer.

We are proposing the use of $DL(D)$, $T(D)$, and $K(D)$ to transfer domain dependent descriptive knowledge to a computer. We have briefly indicated how such descriptive knowledge could be used to solve problems in a domain automatically.

The specification of $DL(D)$, $T(D)$, and $K(D)$ in a domain will, of course, require a very good understanding of the concepts and problems in a domain. There are several domains where, at present, such understanding is available. The MDS provides a way of transferring this understanding to a computer.

There is much work to be done to make the MDS a
viable system. It is necessary to develop a working system first. We are presently involved in this task.

REFERENCES


PAS-II: An Interactive Task-Free Version of an Automatic Protocol Analysis System

DONALD A. WATERMAN AND ALLEN NEWELL, FELLOW, IEEE

Abstract—PAS-II, a computer program which represents a generalized version of an automatic protocol system (PAS-I) is described. PAS-II is a task-free, interactive, modular data analysis system for inferring the information processes used by a human from his verbal behavior while solving a problem. The output of the program is a problem behavior graph: a description of the subject’s changing knowledge state during problem solving. As an example of system operation the PAS-II analysis of a short cryptarithmetic protocol is presented.

Index Terms—Cryptarithmetic, hypothesis formation, model building, natural language processing, problem space, production system, protocol analysis.

I. INTRODUCTION

AUTOMATIC protocol analysis is a joint effort by man and machine to infer from the record of the time course of a subject’s behavior, the underlying information processes. As developed [5], it usually refers to the verbalizations of a subject solving some problem under instructions to think out loud. Protocol analysis designates the full range of activities engaged in by the psychologist when working with protocols: description of the subject’s behavior according to an hypothesized model, induction of new rules, derivation of consequences from a model in the context of specific data, and measurement of adequacy of a model. The initial focus of our work has been behavior description in terms of information processes, given an hypothesized general model (the so-called problem space in which the subject operates).

The PAS-I system [14], [15] was our first attempt at automatic protocol analysis. This is a fully automatic, noninterative, specialized system designed to analyze cryptarithmetic protocols and produce as output a problem behavior graph (PBG) describing the subject’s search through a posited problem space. The protocol analysis is represented as a sequence of processing stages that eventually transform the raw protocol into a problem behavior graph. At each stage rules are applied which effect a transformation of the data. The organization of PAS-I is shown in Fig. 1.

Chitoor V. Srinivasan (M'63) was born in Cuddapah, India, on November 6, 1933. He received the B.S. degree from Madras University, Madras, India, in 1953, the D.M.I.T. degree in electronics from the Madras Institute of Technology, Madras, in 1956, and the M.S. and D.Eng.Sc. degrees in electrical engineering from Columbia University, New York, NY, both in 1963.

From 1956 to 1959 he was at the Tata Institute of Fundamental Research, Bombay, India. From 1962 to 1969 he was at the RCA Laboratories, Princeton, NJ. Presently he is with the Department of Computer Science, Rutgers University, New Brunswick, NJ.

Dr. Srinivasan is a member of the Association for Computing Machinery.
PROGRAMMING OVER A KNOWLEDGE BASE: THE BASIS FOR AUTOMATIC PROGRAMMING *

C.V. Srinivasan
Department of Computer Science, Hill Center
Rutgers University, New Brunswick, N.J. 08903

Abstract: This paper introduces the notion of using a highly flexible general problem solving system as the basis for developing domain dependent automatic programming systems, that can actively and intelligently assist its users to formulate problems and develop programs in the domain. The system is called the Meta Description System. It is being currently implemented in LISP 1.6. The system accepts definitions of description schemas for describing KNOWLEDGE in a domain, and uses these schemas to specialize itself as an efficient problem solver in the domain. It also has the capability to accept definitions of a language of discourse for a domain and have the users communicate with it in the specified language.

1. Introduction

Our objective is to create an automatic programming (AP) system that can actively and intelligently assist users to solve problems in a domain. A domain might be as complex as the design of a computing system, or it might be the diagnosis and treatment of a disease system; it might be a piece of mathematics or psychology. Our mode of operation will be the following:

Suppose one wanted to create an automatic programming system for a domain D. Then one would first specify to our system some core knowledge in the domain D. This would consist of schemas specifying how objects in the domain D are described and their descriptions represented in the data base (these schemas are called description schemas); descriptions of specific objects in the domain satisfying the given schemas (we shall refer to these as instances (or models) of objects in the domain); rules specifying how given descriptions of objects in a domain may be transformed to new ones satisfying given criteria; and possibly also strategies for problem solving in the domain D. These specifications of knowledge in the domain will cause the system to create a data base, called the Coherent Data Base for the domain D, CDB(D). The system will, of course, assist the user in setting up the CDB(D), by looking for inconsistencies, seeking out missing information, and where necessary itself supplying the missing information. The CDB(D) constitutes a knowledge base over which all domain dependent programming in the domain D will take place. As the system is used, its knowledge base will continue to expand. The system itself will use this knowledge base automatically, to intelligently assist its users to solve problems in the domain D.

* This work was supported by grant #: DAHCIS-73-G-6, from ARPA.
The system is called the meta-description system (MDS)*. It is a meta system in the sense that it accepts definitions of description schemas (in terms of devices called templates and sense definitions), for a domain D, and uses these schemas to specialize itself in an active way to solve problems efficiently in domain D. This specialization occurs in three ways:

(i) In the data structures used to represent descriptions of (models of) objects in domain D,

(ii) In the problems solving control structures used for the domain D, and

(iii) In the way problem solving experiences in the domain are summarized and later automatically used for self-improvement.

The architecture of the MDS allows for fundamental structural changes to take place in the system, to efficiently utilize the available domain dependent knowledge. The MDS is thus a general problem solving system that can specialize itself to perform efficiently in a given domain. At every point in its operation the MDS can automatically make full use of its knowledge base to actively and intelligently assist its users. We shall refer to programming in the context of the MDS as programming over a knowledge base.

There are several new concepts in the architecture of the MDS. Usually general problem solving systems have a way of imposing their own will on everything around them. They would demand that data be represented in certain ways, they might demand that problems be stated only in certain ways, and they often resist strongly interference with their problem solving procedures--do not take to advice easily. The limitations caused by these were well recognized early in the game. The trend towards the development of programming languages like PLANNER [Hewitt (1972)] and CONNIVER [McDermott (1973)] was, in fact, a response to overcome this limitation. These language systems do not fix a priori any problem solving scheme. They let the designer specify schemes and strategies for given domains. In doing this, however, they do not provide any automatic and intelligent problem solving help to the users. It is the programmers' responsibility to specify and develop all the problem

*Since 1971 August, our work on the MDS has been partially supported by a grant from NIH, Grant No. RR643.
solving means within the confines of the given control structure of the language system. Whereas, intelligent programming systems may be created by using these languages, they do not by themselves provide for, what we call, programming over a knowledge base.

The general problem solvers in the MDS are very flexible and obliging ones: They do not demand that data be represented one way or another, and also more importantly they do not impose any a priori chosen search strategy for problem solving. Representations for the descriptions (models) of objects in a domain \( D \), follow the dictates of the description schemas for the domain, and not the dictates of the problem solvers. More importantly, the problem solving protocols—summaries of the system's problem solving experiences—may themselves be treated as objects in the domain \( D \), with their own associated description schemas. We shall refer to these as the problem solving schemas for domain \( D \), \( P: S(D) \). \( PSS(D) \) will again be specified in terms of devices called templates and sense definitions. For a domain \( D \), its \( PSS(D) \) will specify the problem solving control structures and search strategies. Instances of \( PSS(D) \) (models of problem solving experiences) may then be used by the problem solvers in the same way as any other data in the Coherent Data Base. The \( PSS(D) \) may be so defined that the problem solver improves itself by using the models of prior experiences. How is this all done? The full answer to this question is necessarily a complex one. We shall here illustrate the operation of the MDS with a small example, chosen from Balzer's paper [Balzer (1973)]. We shall use this example to introduce the basic conventions of the MDS, its operational characteristics, its logical processes, and to show how it does problem solving. Later, in section 3 we shall comment further on the MDS and compare it with other works in Automatic Programming, to place it in perspective with the other works.

We are at present in the early stages of implementation of MDS. We expect to complete the implementation of all of its facilities in about two years. The data base management part of the MDS is expected to be ready in the Spring of 1974. The implementation, so far, has been in LISPl.6. We expect to convert the existing system to INTER-LISP and continue further work in a TENEX system.
2) An Example

2.1) Specification of the Description Schema

Briefly, the problem is the following:

"A PERSON is HAPPY if he/she has a COMPATIBLE MARRIAGE, or is RICH. A MARRIAGE is COMPATIBLE if the COUPLE has a common hobby, and the wife is not more than 5 YEARS older than the husband. A PERSON is RICH if he/she is worth more than a million DOLLARS. Make JOHN happy."

The words in capitals in the above statement are the objects of the domain of this problem, which we shall now describe to our system. The underlined words will appear as relation names in our system. We begin by telling the system how to describe PERSON, HAPPY, COMPATIBLE, MARRIAGE, RICH, COUPLE, YEAR and DOLLAR. In effect we shall say that a PERSON is an individual (node, in contrast to a list or tuple) with a name, who has an age, some hobbies, a worth, some attribute, a sex, an emotion, a marriage, and may have a wife, or a husband (spouse). The template for this is shown below. The flag RN associated with the PERSON template indicates that a PERSON template is a regular node, i.e. a node with a name. Thus, every instance of PERSON will have a name in the CDB.

```
((PERSON RN)  
 (age (YEARS TI) )  (Worth (DOLLARS TI) )  
 (hobbies (HOBBIES $L) ) (loves (PEOPLE $L) )  
 (marriage (MARRIAGE $N) ) (sex (SEX RN) )  
 (attributes (ATBTL $L), SENSE1)  
 (spouse $) PERSON, SENSE2)  
 ((wife $) PERSON, SENSE3)  
 ((husband $) PERSON, SENSE4)  
 (emotion (EMOTION RN), SENSES))
```

Let us follow the other definitions in the PERSON template. PERSON calls other templates like YEARS, MARRIAGE, etc. via relation names like age, hobbies, etc. YEARS has been declared as a Terminal Integer (TI) template. Every instance of YEARS is an integer with dimension YEARS. Similarly, DOLLARS is also a terminal integer. Notice that in the CDB, two integers, say 8 YEARS and 1000 DOLLARS, will be recorded as objects with different dimensions. HOBBIES is a dummy list ($L) template. Every instance of HOBBIES is a list (say, a
a list of ACTIVITIES), and not every instance of HOBBIES need have a name in the CDB. MARRIAGE is, similarly, a dummy node ($N$) template. Not every instance of MARRIAGE will have a name in the CDB. The attributes of a PERSON should be an instance of (ATBTL $L$), and so also, a PERSON's emotion should be an instance of (EMOTION RN), and the sex of a PERSON is an instance of (SEX RN). We shall choose not to associate any relation names with the EMOTION and SEX templates. Since both of these are regular templates, instances of these in the CDB will just be descriptive names like SAD, HAPPY, and MALE, FEMALE etc.

The relations spouse, wife and husband are defined by the sense definitions SENSE2, SENSE3 and SENSE4. The $\$N$ flag associated with these relations indicates that their values can always be computed from the sense definitions, and thus need not be stored in the CDB for any instance of PERSON. The EMOTION of a PERSON is defined by SENSE5, but we require that its value be stored in the CDB for every PERSON. We shall later see how the sense definitions are specified. Let us now complete the definitions of the other templates.

$$
\begin{align*}
((\text{MARRIAGE } \$N)(\text{partners } (\text{COUPLE } \$L) \text{SENSE6}) (\text{quality } (\text{MQUAL RN} \text{SENSE7}))) \\
((\text{COUPLE } \$L)(\text{elem } (\text{2 PERSON}))) \\
((\text{ATBTL } \$L)(\text{elem } (\text{ATBT RN}))) \\
((\text{HOBBIES } \$L)(\text{elem } (\text{ACTIVITIES RN}))) \\
((\text{PEOPLE } \$L)(\text{elem } \text{PERSON}))
\end{align*}
$$

COUPLE is constrained to be a list of exactly two PERSONs. HOBBIES is a list of an arbitrary number of ACTIVITIES, where each ACTIVITIES is a regular node. The above templates define the description structure of objects in the domain of our problem. Let us now create some of the descriptive names. In the commands below "IT" stands for "Instantiate Template", and "DR" stands for "Delete Relation". In the CDB, "?" denotes an unknown. A "?" in a list indicates that the list may contain additional elements. The following instantiations are now done:

\begin{align*}
\text{IT(SEX MALE)} & \quad \text{IT(SEX FEMALE)} \\
\text{IT(EMOTION HAPPY)} & \quad \text{IT(EMOTION SAD)} \quad \text{IT(EMOTION BLAH)} \\
\text{IT(ACTIVITIES GARDENING)} & \quad \text{IT(ACTIVITIES PROGRAMMING)} \\
\text{IT(ATBT RICH)} & \quad \text{IT(ATBT POOR)} \quad \text{IT(ATBT ORDINARY)} \quad \text{IT(ATBT THIEF)} \\
\text{IT(MQUAL COMPATIBLE)} & \quad \text{IT(MQUAL LOUSY)}
\end{align*}

For SEX the CDB will now have: (SEX instance (MALE FEMALE ?)),

where "instance" is a system relation, and (MALE FEMALE ?) is a list. If SEX is constrained to have only two instances, we may now indicate this by simply removing the ? from the (MALE FEMALE ?) list. In the CDB, new elements may be introduced in a list or a set only if the list or set contains a ?. So, we may now issue DR(SEX instance ?). Just to see what happens, let us now also instantiate a PERSON, called JOHN: IT(PERSON JOHN). This would, of course, cause (PERSON instance (JOHN ?)) to be created in the CDB, and JOHN itself will have the following structure associated with it.

The model of JOHN is a tuple consisting of pointers, defining the various relations associated with JOHN. The name of the model is JOHN, it is an instance of PERSON, and it points to JOHN's age, worth, hobbies, etc. The pointers appear in the model in the same order as their associated relations appear in the template PERSON. Relation symbols in the template with $ flags do not have associated pointers in the model. Initially all the unknown relation values are set to ?. We have now created JOHN about whom we know nothing, except that JOHN is an instance of PERSON. In the CDB if (x r (y₁ y₂ ... yₙ)) is true (i.e. the model of x points to the list (y₁ y₂ ... yₙ) for the relation symbol r) then it is interpreted as (x r y₁)(x r y₂)...(x r yₙ). Thus (JOHN attributes (RICH THIEF)) would mean (JOHN attributes RICH) and (JOHN attributes THIEF) are both true. Also, for every (x r y) in the CDB, the CDB will also contain (y r⁻¹ x), where r⁻¹ is the inverse of r. That is, if the model of x points to y for the relation symbol r, then the model of y will point back to x for the relation symbol r⁻¹. Let us now take a look at the sense definitions. SENSE₁ is given below. It is associated with (PERSON attributes). In the definition of SENSE₁ read "!" as the "current instance of PERSON's", and read "(x is RICH)" as "(x EQ RICH)" ("is" has the status of EQ in LISP). We shall also use "is" together with relation symbols to improve readability wherever convenient.
SENSE1: (PERSON attributes)

\[\begin{align*}
((\text{ATBT } x) & \iff (*! \text{ worth is } \geq 1000000)) \\
((x \text{ is POOR}) & \iff (*! \text{ worth is } \leq 1000)) \\
((x \text{ is ORDINARY}) & \iff \neg(x \text{ is RICH}) \land \neg(x \text{ is POOR})) \\
(*! \text{ attributes THIEF}) & \implies (x \text{ is THIEF}))
\end{align*}\]

Here \(((*! \text{ attributes THIEF}) \implies (x \text{ is THIEF}))\) is interpreted as "If THIEF is declared to be an attribute of *! then \((x \text{ is THIEF})\)." Thus, to set up \((\text{JOHN attributes THIEF})\) someone should declare IR(\text{JOHN attributes THIEF}), where "IR" stands for "Instantiate Relation". Let us refer to this sense definition by SENSE1(*!, x) indicating that it has two arguments: One is the current instance of PERSON at which it is being evaluated, and the other, x, is the attribute for which it is desired to know whether (*! attribute x) is true or not. Every sense definition is thus a function of exactly two arguments, one of which is always *!. *! is called the anchor of a sense definition.

If SENSE1(*!) is issued then the system will attempt to find all the attributes of *! that satisfy SENSE1. If none could be found then it will return ?. In the evaluation of SENSE1(*!), (*! attributes THIEF) will be true if it is so indicated, already in the database. In the evaluation of SENSE(*!, x), (*! attributes THIEF) is true if x is bound to THIEF. Thus, in the definition of SENSE1, (*! attributes THIEF) has a special status, since the SENSE1 itself defines (*! attributes).

As the reader might have already guessed the sense definitions are evaluated over a three-valued logic system, T, ? and NIL; T dominates ?, and ? dominates NIL. The other sense definitions are shown below:

SENSE2: (PERSON spouse): \(((\text{PERSON } x) \mid \neg(*! \text{ is } x)(*! \text{ marriage.partners } x))\).

This definition also says "the list of ALL PERSONs x such that ...", but the MDS will interpret this as "THE PERSON x such that ...", because the template for PERSON says that the spouse of a PERSON is a PERSON and not a list of PERSONs. The spouse of a PERSON is distinct from the PERSON and is the PERSON's "marriage.partners". The "," here indicates concatenation of relations. It corresponds to a relation path in the CDB.

SENSE3: (PERSON wife): \(((\text{PERSON } x) \mid (x \text{ sex is FEMALE})(*! \text{ spouse } x))\)
SENSE4: (PERSON husband): \(((\text{PERSON } x) \mid (x \text{ sex is MALE})(*! \text{ spouse } x))\)
SENSE5 (PERSON emotion):  ((EMOTION x) | \\
(x is HAPPY) <-> ((!* attributes RICH) V (*! marriage.quality COMPATIBLE)) \\
(*! attributes THIEF))\\n((x is SAD) <-> (*! attributes POOR) V (*! marriage.quality LOUSY))\\n((x is BLAH) <-> ~(*! emotion HAPPY) V (*! emotion SAD)))

SENSE6 (MARRIAGE partners):  ((PERSON x y) | (x sex is MALE)(y sex is FEMALE)\\n(x loves y)(y loves x)\\n(*! marriage of x)(*! marriage of y))

Notice that in SENSE6 "!*" stands for "the current instance of MARRIAGE", and "marriage of" is used as the inverse of "marriage". Notice also that the PERSON template specifies that the marriage of a PERSON is unique, since MARRIAGE is a node template. This precludes a PERSON from having more than one marriage.

SENSE7 (MARRIAGE quality):  ((MQUAL x) | (SOME PERSON y)(SOME ACTIVITIES z)\\n((x is COMPATIBLE) <-> (y spouse.age. < (PLUS (y age) 5))(y hobbies z)\\n(y spouse.hobbies z))\\n((x is LOUSY) <-> ~(*! quality COMPATIBLE)))

This completes the definition of the description schema for the domain of our example. The system uses the sense definitions to keep track of the interactions among the various relations. Thus, (PERSON sex) is used in the definition of (PERSON wife), (PERSON husband) and (MARRIAGE partners). The MDS will, therefore, set up

DETL(PERSON sex) = ((PERSON wife) (PERSON husband) (MARRIAGE partners))

It does seem reasonable that a PERSON's sex should determine the PERSON's wife, husband and MARRIAGE partners. The DETL's associated with the various (template, relation) pairs in our example are shown below:

DETL(PERSON age) = ((MARRIAGE quality))
DETL(PERSON worth) = ((PERSON attributes))
DETL(PERSON hobbies) = ((MARRIAGE quality))
DETL(PERSON loves) = ((MARRIAGE partners))
DETL(PERSON marriage) = ((MARRIAGE partners)(PERSON spouse)(PERSON emotion)\\n(MARRIAGE quality))
DETL(PERSON sex) = ((PERSON marriage)(PERSON wife)(PERSON husband))
DETL(PERSON attributes) = ((PERSON emotion))
DETL(PERSON spouse) = ((PERSON wife)(PERSON husband)(MARRIAGE quality))
DETL(PERSON emotion) = NIL.

Suppose we now wanted to say the following: "If you wanted to make a PERSON RICH then make him rob a BANK for 1000000 dollars. Also, if you succeed in doing this then make the PERSON a THIEF." Let us assume that the template
for BANK already exists, and also a function of two arguments, called ROBBANK has been already defined. The above procedure may then be declared to the MDS as a transformation rule, as follows:

TR1: ((PERSON x)(GOAL(x attributes RICH)) (((SOME BANK B)(ROBBANK x B)
     (ASSERT(x worth is 1000000))(IFDON (ASSERT(x attributes THIEF))))).

The IFDON clause is activated only if both ROBBANK and ASSERT are successfully completed. The entire function is said to be successful if the IFDON clause completes successfully. Transformation rules like this operate in a backtracking environment. In response to (GOAL(JOHN attributes RICH)) the MDS will invoke TR1, if JOHN is not already RICH in the CDB. Our objective is now to make (GOAL(JOHN emotion HAPPY)). Before we see what might happen in response to this command, let us first consider how the description schema so far given is used by the MDS to establish and control the CDB for the domain, and how the CDB is itself used for problem solving.

2. The Data Management System and the Problem Solvers.

MDS has a hierarchy of three problem solvers: CHECKER-INSTANTIATOR (CHIN), THEOREM PROVER (TMPR) and DESIGNER. The CHECKER evaluates sense definitions in three valued logic system, and the INSTANTIATOR sets up, updates, deletes and retrieves data in the CDB in accordance with the rules specified by the templates. These two together constitute the data management system of the MDS.

The CHECKER evaluates sense definitions always modulo the objects in the CDB. Thus all the quantifiers in a sense definition become bounded quantifiers. Since each sense definition has an anchor, *, the CHECKER uses it to begin its search over the data base. In fact the sense definitions may be written carefully to make this search efficient. There are basically two kinds of sense definitions: imperative ones and declarative ones. Let $S_{T_r}^*(x)$ be the sense definition associated with template T and relation symbol $r$. Then $S_{T_r}^*(x)$ is imperative if $(x r x) \iff S_{T_r}^*(x)$. Imperative $S_{T_r}^*$'s may be used to find $(x | (x r x))$. If $(x r x) \Rightarrow S_{T_r}^*(x)$, then $S_{T_r}^*$ cannot be used to find $(x | (x r x))$. It can, however, be used to find a superset of the relation $r$, or given a $(x r x)$ it can be used to find out whether it is TRUE, ?, or NIL. To force the CHECKER to look for this declared x, we shall write declarative definitions of this kind as: $S_{T_r}^*(x) = (x r x)S_{T_r}^*(x)$. In this case $S_{T_r}^*(x)$ is either ? or whatever is stored in the CDB.
Besides returning the truth value of a $S_{T,r}$, the CHECKER also will return certain subexpressions of $S_{T,r}$, called residues. Let $a$ be a particular anchor. If $S_{T,r}(a,x) = T$, then the true residue of $S_{T,r}(a,x)$ is the part of $S_{T,r}(a,x)$ that caused it to be true (the support of the condition). Similarly, if the condition is $?$, then the residue is the part of the condition that evaluated to $?$. And, if $S_{T,r}(a,x)$ is NIL then the false residue of $S_{T,r}(a,x)$ will be the part of it that evaluated to NIL. Let us consider a small example.

Let $P = (x_1 \lor x_2)(\neg x_1 \lor x_3)$. Then for various valuations of $x_1$, $x_2$ and $x_3$ the various residues would be as shown below:

<table>
<thead>
<tr>
<th>valuation $\phi$</th>
<th>$\phi(P)$</th>
<th>$TR_{\phi}(P)$</th>
<th>$R_{\phi}(P)$</th>
<th>$FR_{\phi}(P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$ $T$ $T$</td>
<td>$T$</td>
<td>$(x_1 \lor x_2)x_3$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$ $T$ $?$</td>
<td>$?$</td>
<td>$?$</td>
<td>$x_3$</td>
<td>$?$</td>
</tr>
<tr>
<td>$T$ $T$ NIL</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
<td>$(\neg x_1 \lor x_3)$</td>
</tr>
<tr>
<td>$?$ $? $ $T$</td>
<td>$T$</td>
<td>$x_1 x_3$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$NIL$ $? $ $? $</td>
<td>$?$</td>
<td>$?$</td>
<td>$x_2$</td>
<td>$?$</td>
</tr>
<tr>
<td>$NIL$ $NIL$ $T$</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
<td>$(x_1 \lor x_2)$</td>
</tr>
</tbody>
</table>

These residues are used in various ways in problem solving. The residues are used by the TMPR to construct new objects that satisfy given conditions. Both the residues and false residues are together (called the Not True Part) used by the DESIGNER for means-end analysis. All the residues are used by the CHECKER to speed up the data base updating process: If $(x r y)$ is to be changed to $(x r z)$ then the CHECKER will check all the residues associated with every $(Y_i, r_i)$ in the DETL($x,r$). Only if the residue changes value [i.e. a true residue or a residue evaluates to NIL, or a false residue evaluates to $T$] should the CHECKER evaluate the parent sense definition. In a

* $TR_{\phi}(P)$ is the True residue of $P$ for valuation $\phi$, and similarly we have $FR_{\phi}(P)$ and $R_{\phi}(P)$. 
problem solving process the residues are also used to summarize the problem solving experience: If an action succeeded, the associated true residues will then explain the reasons for success, if it failed then the associated false residues say why the failure occurred. Summaries of these residues may then be used, with appropriate generalizations, for guiding the problem solver subsequently when "similar" problem solving situations arise. We shall briefly see the use of residues in the discussion of our example in section 3.

The INSTANTIATOR will complete an IR(x r y) [Instantiate Relation] command only if no contradiction arises in $S_{X,IR}(x,z)$, and among all the conditions associated with the DETL(x,r). There is also an IRN(x r y) command, which will set ~(x r y) true in CDB, if possible. Corresponding to IR and IRN we also have DR and DRN (D for Delete) and JR and JRN (J for Justify). Where an IR command will not accept (x r y) if a contradiction arose in DETL(x,r), JR (and similarly JRN) will attempt to modify the relations in DETL(x,r) appropriately (if possible) and thus attempt to justify the given (x r y).

The sense definitions act as the gate keepers of the CDB, making sure that nothing illegal happens. The CDB is thus always kept contradiction free. However, as discussed in Srinivasan 1973b, because of the three valued logical system, there might exist hidden contradictions (contradictions arising because of incomplete knowledge) in the CDB. An assertion in a domain is true, if and only if models can be built in the CDB to satisfy the assertion.

This feature of the CDB is used by TMPR (as discussed in Srinivasan 1973b) to find proofs of assertions in a domain. The TMPR provides the control structure to the CHIN system, to direct it appropriately, to build models to satisfy an assertion, if such models are possible. If models do not exist then it will discover a contradiction. In the model building process the TMPR uses the residues generated by the CHIN system, to guide itself. The theorem proving process in TMPR develops proofs by synthesis. It introduces a new approach to theorem proving.

The DESIGNER is used to do means-end analysis, to invoke the appropriate transformation rules, like (TR1), to reach a goal, and to interpret the transformations. The DESIGNER may use the CHIN and TMPR systems to find (or build) the appropriate objects in the CDB to accomplish a given task.

To do intelligent problem solving in a domain both the TMPR and DESIGNER should be able to appropriately summarize a problem state, and their own past experiences, and use these effectively to search the solution space (the goal-subgoal tree) for a given problem. For a given domain, the description
schemas for describing the states of the DESIGNER and TMPR may themselves be again specified by templates and sense definitions. Let DS (Designer State) and TPS (Theorem Prover State) be the templates with associated sense definitions, that specify the respective states of the problem solvers for a domain D. Every time the DESIGNER invokes a function (note that the DESIGNER can invoke the TMPR itself as a function) it will create a new instance of DS to describe the problem state associated with the invoked function. The DS itself might be as follows:

((DS $N)
  (fn-called FND): Some unique way of identifying the called function
  (initl-state ICOND, SENSEI) Some way of specifying the initial state in CDB, and possibly other problem conditions that caused the invocation of the function.
  (bindings BNDGS, SENSEB): The list of all possible bindings available in the CDB for the arguments of the invoked function (actually its closure), for the current invocation. In general, there might be more then one possible binding. Let us assume that BNDGS also flags the currently chosen bindings. SENSEB might specify how to choose the current binding.
  (subgoals SUBGLS ,SENSEG): The list of DS-instances corresponding to all available subgoals, if such subgoals exist. SENSEG might specify how to choose one from among the list.
  (sensessummaries SSM, SENSES): Summaries (usually made out of the residues) of all sense definitions evaluated during the active tenure (i.e. before the DS-instance is closed up as having been successful or a failure) of the DS-instance. All these summaries will appear in terms of the variables appearing in the invoked function. The template SSM might itself be domain dependent. The sense definitions associated with the SSM template might provide ways of analyzing and summarizing all the residues obtained in a DS-instance. One may include in these sense definitions, domain specific evaluation functions, if any.

The sense summaries will generally fall into two classes: Those associated with the successful completion of the invoked function (SUCSSSM) and those associated with its failure or suspension of the function (FAILSSSM). Notice that for each DS-instance, its associated sense summaries would specify the special cases in which the invoked function either succeeded or failed. When new invocations of the same (or similar, in a suitably defined sense, again possibly specific to a given domain) functions occur, these special cases
might first be checked to avoid repeating errors, and to choose, if possible
the correct course. Conditions for examining such sense summaries might appear
in SENSEB and SENSEG given above. To facilitate such checking, DS may also
have,

**history** **DSL** **SENSEH**: List of all other DS-instances of the same
(or similar) function. The reader should note that the instantiations of the
relations in a DS-instance will itself cause the associated sense definitions
to be invoked, and their evaluations will produce residues which might them-
selves cause an entirely new sub-problem solving activity to take place.

**final state** **FSTATE** **SENSEF**: The changes performed in the CDB
during the tenure of the DS-instance.

**fn-state** **FNS** **SENSE?**: The state in which the DS-instance was
finally closed: success, failure, or suspended.

**successor DS** **SENSFS)** successor DS-instance, if any.

The relations given above are typical of what one might want, in order to
meaningfully describe the state of the DESIGNER. The important concept to notice
is that different such DS templates might be defined for different domains.
The network of instantiations of the DS template, generated during the course
of a problem solving process, would constitute the problem solving protocol.
This protocol will not only contain a trace of changes done on the items in
CDB, but it also will document the reasons why certain courses of actions were
taken, and certain others abandoned. For each DS-instance, the program schema,

\[ [(ICOND) \land (SUCSSM) \rightarrow FSTATE] \land \\
[(ICOND) \land (FAILSSM) \rightarrow ICOND] \]

in effect summarizes the effect of the DS-instance: For the given initial
state and conditions summarized in the SUCSSM, the DS-instance leads to the
indicated FSTATE (final-state), and for the same ICOND and FAILSSM the ICOND
is left unchanged. This program schema may now be used to translate a protocol
to a program. One would, of course, choose only the DS-instances appearing
in the successful execution path of the protocol. Each such program will
correspond to a special case of the invoked function.

The template for the theorem proving state may also be similarly used to guide the
theorem prover intelligently in a domain dependent way. It is worth mentioning
here certain important features:

In MDS the problem solvers, in fact, generate a description of what they do
as they solve a problem. In fact the problem solving process is itself simply
the process of describing what the MDS is doing. These descriptions are, how-
ever, generated in a highly domain dependent way. Again the notion of specialization comes in. Most importantly, summaries of the problem solving protocols may be made in the form of canned programs, with characteristic conditions for their invocation. These canned programs may later be called whenever appropriate. In this sense the MDS can constantly learn and improve itself. Also, clearly, it is being used to do automatic programming in a non-trivial way.

The operations of the TMPR and DESIGNER are discussed in fair amount of detail in Srinivasan 1973b and 1973a. Let us now get back to our example.

3. **Making JOHN Happy**

The DESIGNER receives the goal \((\text{GOAL}(\text{JOHN emotion HAPPY}))\). First it checks the CDB to see if JOHN is already HAPPY. Then it searches its repertoire of transformation rules to see whether there exists a transformation to make a PERSON happy, since JOHN is a PERSON. It does not find any. So it simply issues JR(\((\text{JOHN emotion HAPPY})\), (JR for Justify Relation), to the INSTANTIATOR. The INSTANTIATOR, of course, calls the CHECKER, which now evaluates \(\text{SENSES}\) (see page 8 ). Since none of JOHN's properties are known, the condition evaluates to \(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_,\) and the CHECKER returns to the INSTANTIATOR the following residue, \((\text{RI}):\)

\[
\text{(RI)}: (\text{PERSON JOHN})
\]

\[
((\text{JOHN attributes RICH}) \lor (\text{JOHN marriage-quality COMPATIBLE}))
\]

\[
(\neg(\text{JOHN attribute THIEF}) \land \neg(\text{JOHN attribute POOR}) \land (\text{JOHN marriage-quality LOUSY})\].
\]

The INSTANTIATOR now passes this on to the TMPR, since it has the JR command. It is now the TMPR's job to create new objects satisfying \((\text{RI})\). Let us suppose it first sets up the goal

\[
(\text{GOAL}(\text{JOHN attributes RICH})).
\]

When the DESIGNER gets this, it finds the transformation rule \((\text{TRI})\) (see page 9 ). Now, if the bank robbery and the following ASSERT statements are both successfully completed in TRI, then JOHN will be RICH. However, now the IFDON statement should also be executed. This makes JOHN a THIEF and hence, not HAPPY (this violates \((\text{RI})\)). This approach should therefore be abandoned. During this process all the sense summaries and problem conditions would have been recorded in the various DS-instances created by the problem solvers.
Now, the next possibility is to get JOHN married, and make the marriage compatible. So create a MARRIAGE for JOHN. To complete this marriage, another partner (a woman) has to be found according to SENSE6 (see page 8). The woman has to love JOHN. Also, since the marriage quality should be compatible, the woman should not be more than 5 years older than John and also should share a hobby with John (From SENSE7). What is the age of JOHN? What are his hobbies? There are no sense definitions associated with a PERSON's age and hobby. So, ask the user.

Now, if permitted, the TMPR can create a new FEMALE PERSON with the appropriate properties (to love John, be not more than 5 years older than John, and share a hobby with John), and marry John off in order to make him happy. If the MDS is advised not to create women like that (this condition can be imposed by advising the MDS that only the available resources in the CDB may be used to solve the problem), then the system will now merely put John as being happy, and associate with his happiness the residue (RI) as a condition. Later, when more properties of John becomes available, the system will check whether the conditions on John's happiness are being satisfied. (But, of course, bachelor John could well land in a LOUSY marriage later on and loose his happiness!)

If the job had been successfully completed (by creating a woman), then essentially, the program generated from the protocol would say,

Making a PERSON happy:

(PERSON properties unknown)

(ASK FOR PERSON's age)

(ASK FOR PERSON's hobby)

(CREATE PERSON's marriage, and make it compatible
by creating an appropriate partner for PERSON).

in some suitable programming language (which could, of course, be the command language of the INSTANTIATOR, together with some facility to invoke them conditionally). In the execution of this program one may, if desired, entirely suppress the CHECKER. Since, for the conditions satisfying the program, it is known to succeed without creating any contradictions in the CDB. One has to, of course, include within the program the steps for creating a woman such that the marriage is compatible.

Briefly, this illustrates the essential concepts in MDS, the organization
of the MDS and its operational features. The significant innovations are the
following:

(i) The concept of the description-schema, and a system organization where
every aspect of the system's functioning adapts itself to the description schemas.
One may think of these schemas as representation strategies for domain depen-
dent knowledge and problem solving techniques. For a complex domain the cre-
tion of these schemas will itself be a formidable problem. The MDS can help
intelligently in this task.

(ii) The Coherent Data Base operates on a three-valued logical system. As
shown in Srinivasan, 1973b, it is this three-valued logic feature that makes
constructive proofs in a domain possible.

(iii) The problem solving process is itself viewed as a process of descri-
b King what the MDS is doing. From the description of the way a problem is solved
(the description has more information than a program trace) the MDS can generate
a program for solving the problem.

There are several new concepts in the MDS organization. Logical conditions
(the sense definitions) are used in MDS as programs as well as data. (Generally,
in theorem proving systems logical conditions are used only as data.) The struc-
tural organization of descriptions themselves build into the system a lot of
logical constraints. The description structure of an object is used to classify
objects in a domain into objects of different kinds, like PERSON, MARRIAGE,
PEOPLE, etc., in our example. This classification of objects in a domain into
objects of different kinds, later aids the system in summarizing its own prob-
lem solving experiences; it is thus capable of generalizing what happens to JOHN
as what might in general happen to a PERSON with certain properties.

The realization of the MDS as a working system will significantly advance
the art of AI as well as the art of automatic programming. In the next section
we shall attempt to place the MDS work in perspective, within the spectrum of
automatic programming systems, as viewed from the point of view of a specific
classification schema.

4. MDS and automatic programming

Every AP system should, of course be capable of providing the elementary
clerical and syntactic help that one would expect of any good programming system.
In addition, we require that it understand in a well-defined sense, what a user
is saying in a discourse with the system. It should, for example, be able to resolve ambiguities of specification to the extent possible; and it should be able to identify the missing pieces of information in a given context of discourse, and seek out to obtain them. Also, whenever necessary, it should call on available canned programs, or generate by itself the necessary programs, to solve the problems it encounters during the course of its interaction with the user. Ideally, it should be able to improve itself by experience. The MDS can satisfy all these requirements.

To do all this in a given domain, the AP system should not only have some expectations on the nature of knowledge—facts, conjectures and procedures—in the domain, but it should also be capable of automatically invoking the appropriate pieces of information within a given context, and use them correctly. The kinds of problem solving facilities necessary to create such a system exist in systems like STRIPS, and the programming facilities necessary to create such systems are available in CONNIVER and PLANNER like systems. STRIPS uses a general theorem prover for problem solving and is thus restricted in its scope of applications. There is no notion of domain specific specialization in STRIPS. The CONNIVER and PLANNER like system enable one to create highly specialized domain specific problem solvers. But the programmer has the responsibility to build all the problem solving systems. In MDS, we find a general problem solving facility that can be specialized to specific domains. We have, truly, the concept of programming over a knowledge base.

For our purposes here I shall classify the existing works in AP-systems as shown below. I should hasten to point out that the classification given here is not intended to be complete; on the contrary it is meant to reflect one man's biased opinions. A survey of the AP-systems appears in Balzer [1972].

AP-EFFORTS:

[1A] Systems with general problem solving.
[1A1] Those dominated by the problem solver.
These have no capacity for domain specific specialization. Leading example is STRIPS [Fikes 1972]. Some of the other examples appear in Darlington [1973], Manna and Waldinger [1971], Luckham & Buchanan [1973].

[1A2] MDS-Type: The general problem solver is driven by the representation strategies chosen for a domain. There is a strong sense of domain specific specialization.
Programming Systems.

Leading examples are PLANNER [Hewitt 1972], CONNIVER [McDermott 1973]. Useful to create domain specific intelligent systems. But do not have problem solving features to provide intelligent guide for automatic programming.

Systems that include a lot of domain specific knowledge

Made of a collection of canned programs which expertly encapsulate domain specific knowledge.

With an intelligent interface to cleverly select the appropriate functions in response to problem conditions.

EX: NONE.

With no such intelligent interface. EX: MAXSYM

Made of special purpose synthesis routines that produce highly optimized code for given classes of problems in restricted domains.

EX: Wilkens [1973, included in the Appendix].

Guard, J, [1972].

A set of programming conventions--as for example in structured programming--together with a well organized, automated clerical systems to document programs and guide users in debugging.


In our project the work of Welsch [1973 a b] falls in this category.

Proving properties of Programs

General Approach


Programs to improve programs

General Approach

Darlington & Burstal [1973].

Restricted Approach

Marvin Paull's work in our project falls in this category.

Paull [1973].

In this schema we are placing the MDS in a class by itself. In our discussions of the MDS here we have ignored the problem of design of a language of discourse for communicating with the MDS in a domain. We have, in fact, identified a way of defining a language to the MDS in terms of a mapping from the linguistic units in a language (lexical items and phrases) to items in the description-schema of a domain. The language understanding process is viewed as a process of translation from utterances in a language to models in the CDB. This model building process may use the full problem solving power...
of the MDS to effect the translation process. We shall report our findings in this area in subsequent reports.

We believe that the MDS can bring the full powers of a general problem solving system to the services of common computer users in different domains of discourse, each communicating with the machine in a language appropriate to the domain.

**Acknowledgement.**

The discussions I had with Balzer were very useful in clarifying many of the concepts presented here. It is a pleasure to acknowledge this help.
REFERENCES:


THE BLIND HAND PROBLEM

T. Hsu

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
THE BLIND HAND PROBLEM

By

Tau Hau

December 1976

ABSTRACT: Three kinds of representations: the higher order logic in Darlington's system, the AESTRIPS in Sacerdoti's system and Meta Description System in Srinivasan's system are investigated using the blind hand problem. The advantages and disadvantages of each representation are discussed.

I. INTRODUCTION

In this short note, three kinds of representations: the higher order logic in Darlington's system, the AESTRIPS in Sacerdoti's system and Meta Description System (MDS) in Srinivasan's system are investigated using the blind hand problem. Some difficulties of each representation are discussed.

The blind hand problem can be stated as follows:

There are two places termed "here" and "there" and a mechanical hand with three actions, namely "pickup", which causes a randomly selected object at the place of the hand to be held; "let-o", which results in the hand being empty; and "co", which moves the the hand to a place. In the initial state so there are red things and only red things "here", and the goal is the condition that at least one red

*This work was supported by grant from the Advanced Research Agency (GRANT NO.: DANCIS-73-66), of the Government of the United States of America.

*Dept. of Computer Science, Rutgers University, New Brunswick, New Jersey 08903
thing is "there". The location of the hand in state s0 is unknown. The problem is to design a sequence of actions to achieve state in which the goal condition holds.

II. THE HIGHER ORDER LOGIC

The Blind Hand example in Darlington's paper (1971) has been solved by using a second order logic theorem prover. Darlington has stated several advantages of this representation: first, it is a very neat representation. Especially the frame problem can be easily solved by adding some extra rules as 'frame axioms'. Secondly, it has a shorter proof than the first order logic and most strategies used in the first order logic, e.g. set of support, restrictive strategy, can also be used in the higher order logic. In addition to using the unification algorithm of the first order logic, a more powerful unification algorithm called 'f-matching mode' is invoked whenever the system fails to find the most general unifier by applying the first order unification algorithm. Thirdly, the plan can be automatically built by the mechanical theorem prover when the given goal is proved.

The main task of solving this problem is to automatically maintain consistency of the information in the system, i.e. automatically update the information by system. For example, in the blind hand problem, when the mechanical hand goes to a new place, then whatever was held by the mechanical hand will also change to the new place. In the second order logic we can write this as:

$$f(\text{thingsat}(x1, r0(x1,s1))) = f(\text{thingsat}(x1,s1) \cup \text{thingsheld}(s1))$$

Which says the things at place x1 after the mechanical hand 'goes to
x in state s will be union of the things original at x and things held by hand in state s. Moreover, all the properties of the things at x after action 'go' are the same as that of things under union operation. (f is the function variable ranging over properties of sets.)

In the above rule, lots things have been said: first, this rule includes the set concept, e.g. the expression thingsat(x,s) will return a set which contains all the things at x in state s, it is very convenient to have this kind of operator; STRIPS is weak in this feature. secondly, it describes what is changed when action 'go' is invoked. Finally, it says what is unchanged.

A special feature of this problem is the 'nondeterministic feature'. The action 'PICKUP' deals with random choice. It causes some uncertainty of a plan which makes the problem difficult.

The final plan that Darlington generated is:

\[ \text{go}(\text{there, pickup}(\text{go}(\text{here, letgo}(\text{so})))) \]

This solution will not guarantee to work if initially the hand is 'here' and holds a thing which is not red. The correct plan should be

\[ \text{go}(\text{there, pickup}(\text{go}(\text{here, letgo}(\text{go}(\text{there, s0})))))) \]

This defect arises because there is no 'AT' predicate in the Darlington's system, hence it is needed to use 'go(here,s)' to express implicitly where the location of the hand is. Otherwise this special situation may conflict with the rule on 'go', and the initial condition which says that there are only red things 'here'.
III. ABOUT ASTOPRIPS

(1). Introduction

In ASTOPRIPS, an action is represented in the form of three lists, i.e. a precondition list, an addition list and a deletion list. Each list contains a set of well-formed formulas (wff) and each wff in the precondition list has been assigned some criticality value by the system, according to the importance and the difficulty of the wff comparing with some user predefined criticality value of some atomic wffs. Using these lists, it avoids using the situation variable, since it assumes that the truth values of only the assertions in the add list and delete list will change. When the system attempts to build a plan, it will try to accomplish the wff which has the highest criticality value first. The difficulty of satisfying a goal increases with the criticality value. This biases the system towards rejecting unfeasible plans, resulting in a smaller planning space and hopefully a more efficient system. Also, to maintain efficiency the system tries to avoid using negated predicates. An example of this occurs in the use of two predicates 'STATUS(x, CLOSE)' and 'STATUS(x, OPEN)' instead of 'STATUS(x, CLOSE)' and 'NOT (STATUS(x, CLOSE))'. To enable the system to assign criticality values properly to the wff, the system needs an extra axiom in the world model:

\[(\forall x) \text{STATUS}(x, \text{CLOSE}) \iff \neg \text{NOT (STATUS}(x, \text{OPEN}))\]

(2). The modified version of the blind hand problem
The original blind hand problem will be discussed in the section 3. The modified blind hand problem will be described in terms of the representation of A*STRIPS first. The only difference between these two versions is in the action 'pickup'. In the modified version, instead of random pick-up, we could specify the thing we want to pick up.

We shall first define the following predicates:

\[
\begin{align*}
\text{AT}(x, y): & \text{ True, if } x \text{ is at place } y; \text{ false, otherwise.} \\
\text{HELD}(x): & \text{ True, if } x \text{ is held by the hand; false, otherwise.} \\
\text{NOTHELD}: & \text{ True, if the hand does not hold anything; false, otherwise.} \\
\text{RED}(x): & \text{ True, if } x \text{ is red; false, otherwise.} \\
\text{TYPE}(x,y): & \text{ True, if } x \text{ has the type } y; \text{ false, otherwise.}
\end{align*}
\]

For efficiency, we defined both \text{HELD}(x) and \text{NOTHELD}. The extra axiom: (ALL x)\text{NOT(HELD(x))} \leftrightarrow \text{NOTHELD} , is also defined for the sake of completeness. The operators are defined as follows:

\[
\begin{align*}
\text{GO}(x): & \text{ The hand goes to place } x. \\
\text{Preconditions: } & \text{AT(hand, } 1), \text{TYPE( } 1, \text{PLACE)}, \text{TYPE}(x, \text{PLACE}) \\
\text{add list: } & \text{AT(hand, x)} \\
\text{delete list: } & \text{AT(hand, } 1)
\end{align*}
\]

\[
\begin{align*}
\text{MOV}(obj,x): & \text{ Move \text{OBJECT obj to PLACE x.}} \\
\text{Preconditions: } & \text{AT(hand, } 1), \text{AT(obj, } 1), \text{HELD(obj), TYPE( } 1, \text{PLACE),} \\
& \text{TYPE}(x, \text{PLACE}), \text{TYPE(obj, OBJECT),} \\
\text{add list: } & \text{AT(hand, x)}, \text{*AT(obj,x)} \\
\text{delete list: } & \text{AT(hand, } 1), \text{AT(obj, } 1)
\end{align*}
\]

(where * denotes the predicate which is the main purpose for applying this operator.)

\[
\begin{align*}
\text{PICKUP}(x): & \text{ The hand picks up the object } x. \\
\text{Preconditions: } & \text{NOTHELD, AT(hand, } 1), \text{AT(x, } 1), \text{TYPE(x, OBJECT),}
\end{align*}
\]
LETGO: Release anything held by the hand.
Preconditions: HELD($1)
add list: NOTHELD
Delete list: HELD($1)

For the initial conditions, we want to say that there are only red things at 'here'. In the second order logic, we could write like:

\[
\text{NOT[INTERSECTION(anyof(thinsat(here,s0)),redthinrs)]=0}
\]

In APSTRIPS, since it does not include the set concept and set operators, we may write some thing like:

\[
(\text{EXIST } x)[\text{TYPE}(x,\text{OBJECT}) \text{ AND } AT(x,\text{here})] \text{ AND } \left( (\text{ALL } x)(\text{TYPE}(x,\text{OBJECT}) \text{ AND } AT(x,\text{here}) \rightarrow \text{RED}(x)) \right)
\]

There is still one problem. Since the right part of the conjunction is only true for the initial conditions, we have to introduce a time parameter. In Darlington's system, this time information is embedded in the state variable. For simplicity, let's just define the initial world model with all the instances which satisfied the initial condition as follows:

\[
\begin{align*}
\text{TYPE}(\text{here},\text{PLACE}) & \quad \text{AT(obj1,here)} \\
\text{TYPE}(\text{there},\text{PLACE}) & \quad \text{AT(FAND,there)} \\
\text{TYPE}(\text{obj1},\text{OBJ}) & \quad \text{RED(obj1)} \\
\text{TYPE}(\text{obj2},\text{OBJ}) & \quad \text{HELD(obj2)}
\end{align*}
\]
Besides the lack of the set concept in APSTRIPS, it is usually also necessary to avoid disjunction and negation in wffs. The use of negation in wff would cause the system to check the entire data base every time, causing a considerable waste of time. Using the disjunction in wff will potentially cause backtracking, and it is not clear yet how to assign the criticality value to each predicate and how to proceed with the control flows in the disjunctive form.

The criticality value is assigned, first, to the predicates which can not be changed by any operators. In our case, the predicates with the highest criticality would be TYPE and RED. Let then both have the criticality value 6. Then, since accomplishing the predicate HELD requires the truth value, true, for the predicate AT in its precondition list; HELD is assigned a higher criticality value than AT. Let us assign the value 4 for HELD and 2 for AT. Because we have the axiom:

\[(\forall x)\neg \text{HELD}(x)\iff \text{NOTHELD}\]

NOTHELD and HELD will have the same criticality value, i.e. 4.

The goal of this problem is:

\[(\exists x)[ \text{TYPE}(x,\text{OBJECT}) \land \text{AT}(x,\text{there}) \land \text{RED}(x)]\]

First, we instantiate a Skolem dummy variable, say v, to delete the existential quantifier, so we have

\[\text{TYPE}(v,\text{OBJECT}) \land \text{AT}(v,\text{there}) \land \text{RED}(v)\]
as the new goal. Now we are going to accomplish the new goal by binding \( x \) to some value. We will first try to accomplish the predicates which have the highest criticality value, i.e., \( \text{TYPE and RED} \). Hence by knowing \( \text{obj1} \) is red, we bind \( x \) to \( \text{obj1} \) immediately. Then, we try to accomplish \( \text{AT(obj1,there)} \) which is not true in our initial world model. The only operator to accomplish this goal is \( \text{MOVE} \). We need to invoke \( \text{MOVE(obj1,there)} \) which has the precondition list:

\[
\text{AT(hand,$1$), AT(obj1,$1$), HELD(obj1), TYPE($1$,PLACE), TYPE(there,PLACE), TYPE(obj1,OBJECT)}
\]

The predicates with the highest criticality value, \( 6 \), are \( \text{TYPE($1$,PLACE)}, \text{TYPE(there,PLACE)} \) and \( \text{TYPE(obj1,OBJECT)} \). If we bind \( \$1 \) to 'here', then they are all satisfied in the space of criticality value \( 6 \). Next, we try to accomplish the predicate with the criticality value \( 4 \), i.e., \( \text{HELD(obj1)} \). After searching all the operators, the only possible operator, \( \text{PICKUP(obj1)} \), is invoked. The precondition of \( \text{PICKUP} \) is

\[
\text{NOTHELD,AT(hand,$1$),AT(obj1,$1$),TYPE(obj1,OBJECT)}
\]

where \( \text{TYPE(obj1,OBJECT)} \) is satisfied already. So we try to accomplish the predicate \( \text{NOTHELD} \), which in turn will require operator \( \text{LETGO} \). By binding \( \$1 \) to \( \text{obj2} \) in the precondition of \( \text{LETGO} \), we have completed the space of criticality equal to \( 4 \). For the space of criticality equal to \( 2 \), the following predicates which are left over from the previously applied operators have to be satisfied:
From operator PICKUP, we have: \( AT(\text{hand}, \$1), AT(\text{obj} 1, ^*1) \) from operator
MOVE, we have: \( AT(\text{hand}, \text{here}), AT(\text{obj} 1, \text{here}) \)

The easiest way to satisfy all of these is binding \$1 to "here", and invoking the operator GO(here), which has the precondition:

\[ AT(\text{hand}, \$1), \text{TYPE}(\$1, \text{PLACE}), \text{TYPE}(\text{here}, \text{PLACE}) \]

This can be satisfied by substituting \$1 to "there". Then all the remaining predicates can be satisfied automatically without invoking any more operators. Therefore, we have built the following plan:

LETGO, GO(here), PICKUP(obj1), MOVE(obj1, there)

where the order of LETGO and GO(here) is not important.

In the above process for building plans in APSTRIPS, we could see some interesting points:

1). Feint guiding by the criticality values of predicates, the system bound \$x to obj1 immediately and correctly in the first try. It will not be mislead by the fact that obj2 is already at place "there", even though it satisfies a part of the goal.

2). The frame problem occurs in the operator GO. When the hand goes to \$x, we do not know whether the hand is holding something or not. If the hand is holding something, then this thing must also change to the new place. Hence an extra rule which interacts with the action 'GO'
is needed in the world model:

\[(\forall s)(\forall x)(\forall v)(\exists z)(\text{Held}(x, s) \land \text{At}(x, v, s) \rightarrow \\
\text{At}(x, z, \text{RESULT}(\text{GO}(z, s))) \land \neg(\text{At}(x, v, \text{RESULT}(\text{GO}(z, s))))\]

where a situation variable \(s\) is attached to all the predicates and the function \(\text{RESULT}\) is used to map an action to a situation [reference to McCarthy and Hayes]. Using the situation variable increases the complexity of the control process. It seems that a better approach is to add a wff to the add list of the action \(\text{GO}\) as follows:

\[\text{Held}(y) \rightarrow \text{At}(y, x) \land \neg(\text{At}(y, s))\]

Then build a special control for it.

We also need an extra rule to tell the system that if something is being held by the hand, then this thing is at the same place as the hand is, i.e.,

\[(\forall x)(\forall y)[(\text{Held}(x) \land \text{At}(\text{hand}, y)) \rightarrow \text{At}(x, y)]\]

Hence, in the world model there are usually quite a few axioms hanging around. Each time when an action is done or some predicates are being updated, we need to check through all of these axioms. A heavy price is paid for this in loss of efficiency.

3). One good point in delaying the substitution of the unknown variable is to avoid the backtracking. In the above example, while invoking the operator \(\text{PICKUP}\), we did not bind \(s1\) in the space of criticality 4, since we did not have any information for binding at
that point. We wait until the space of criticality 2, where we have a
good reason to bind $1$ to "here".

(3). The original blind hand problem

Now, we consider the original blind hand problem in which the
touch randomly picks up an object. Thus, it does not need an argument
for PICKUP. The new PICKUP will look like:

PICKUP
Pred: NOTHELD, (EXIST x)(AT(x,$1$) & TYPE(x,OBJECT)), AT(hand, $1$)
  add: HELD(x) where x is one of objects at place $1$.
  del: NOTHELD

An ad-hoc way to solve this problem is using the same plan we had
in the modified version repeatedly removing one object from "here" to
"there" until no objects at place "here". Since initially there is a
red object at place "here", therefore there is a red object at place
"there" in the final condition.

In order to solve it in a more formal way, we need to use the set
concept and set operator in APSTRIPS and to tell the system some
heuristics for guiding the control flow.

First, we want the system to know that the only way to guarantee
that the hand will always pick up a red thing at some place is by
making sure that there are only red things at that place. We could
write this as an extra axiom with a situation variable, s, and
function, RESULT, as mentioned in the last section.
The next thing we want the system to know is that one way to guarantee that there are always only red things \textit{"here"} is never let the hand hold anything when it goes to \textit{"here"}. Since initially there are only red things \textit{"here"}. Thus we have another rule:

\begin{align*}
(\text{ALL } s) & (\text{NOTHELD}(s) \rightarrow \\
(\text{ALL } x)[\text{AT}(x, \text{here}, \text{RESULT}(\text{GO(here,s)))} \rightarrow \text{RED}(x, \text{RESULT}(\text{GO(here,s))})])
\end{align*}

The control structure of these two rules is quite complicated. An easier way is to attach these kinds of strategy rules to the related actions, e.g., \textsc{PICKUP} and \textsc{GO}, so that the situation variable can be omitted. This can not be done in the current \textsc{KRIPS} representation scheme. Another difficulty is how to define the properties of the argument of \textsc{FIELD}, which deals with the random choice from a set, in the add list of action \textsc{PICKUP}.

The solution of the original blind hand problem can be generated in a way similar to the procedure in the last section. Because of the two extra strategy rules, the hand must be empty before it goes to \textit{"here"}. Since we have a rule:

\begin{align*}
(\text{ALL } x)(\text{ALL } y)[(\text{FIELD}(x) \& \text{AT}(\text{hand},y)) \rightarrow \text{AT}(x,y)]
\end{align*}

This implies initially either the hand is not at place \textit{"here"} or the hand is holding a red object in order to satisfy the initial condition. Therefore the hand does not have to go to somewhere else to empty it. Thus, the plan we have will look like:

\textsc{LETGO, GO(here), PICKUP, MOVE(there)}
The FSTIPS example in Sacerdoti's paper that describes how the system can build a plan for a robot to push a box from one room to another points out a few other weaknesses in the system:

1). For one action 'GO' in the robot problem, it has three different type 'GO'es, namely: go to object bx 'GOTO(bx)', go to door dx 'GOTO(dx)', and go to coordinate location (x,y) 'GOTO(x,y)'. Moreover, it has an action in order to go through the door dx into room rx called 'GOTHRUDED(dx,rx). All these 'GO'es change the location of an object, but each may cause some different side effects: some will cause the object to be in a new room, some will cause the object to be next to the another object, and some will result in a change of the location of an object. For a specific interested outcome, the choice of an action is determined by the explicit recognition of the kind of side effects that are desired. Similarly for the action push, they have push box bx to box by 'PUSHF(bx,by)', push bx to door dx 'PUSHD(bx,dx)', push bx to coordinate location (x,y) 'PUSHL(bx,x,y)', push bx through door dx into room rx 'PUSHTHRUDED(bx,dx,rx)'. This kind of representation appears to be too heavily specialized to the particular, stylized example. It is not a scheme that one might adapt in general, for representation of actions.

2). It needs some rules to enable the system to completely define the criticality value. Sometimes such rules do not make too much sense for us; for example, it has

\[(\forall x)[PUSHAPHL(x) \rightarrow TYPE(x, \text{OBJECT})]\]
which seems quite hard to be defined completely by a user.

3). It has no inverse relation available, hence when we delete NEXT(X,Y), we have also to delete NEXT(Y,X). It is very inconvenient and the system needs the extra storage to record both predicates NEXT(x,y) and NEXT(y,x).

4). For each action, before execution, it needs to do lots of type checking since usually the type predicate has the highest criticality value. It spends quite a bit time on that.

IV. MDS

(1). The Domain Definition of the Blind Hand Problem

In MDS, many of the above problems are completely avoided. (In this section, we assume that the reader has a basic knowledge of MDS.)

The domain definition of the blind hand problem will look as follows:

\[ \text{(TDN: OBJECT} \ (\text{isat } \ \text{(PLACE X)}) \ \text{locationof CC1)} \]
\[ \text{(color } \ \text{(COLOR X)} \ \text{colorof)} \]
\[ \text{(heldby } \ \text{HAND X) holding)} \]

Notice that the relation flag 'RN', Regular Node, tells the system an object can only be at one place, only have one color and only be held by one hand. Here 'locationof' and 'heldby' are defined as the names of inverses of relations 'isat' and 'holding', respectively.

\[ \text{CC1-OBJECT-isat:} \]
\[ \text{(QSCC: } \ \text{(QUOTE } \ ((\text{PLACE X)} )) \]
\[ \text{(((ALL HAND H) NOT(H holding \ X)) (0 isat X))} \]
\[ \text{OR} \]
TH Planning Problem

Here is a consistency condition that specifies the condition which an object will have to satisfy to be at a certain location. CC1 says that if the object is not held by any hand, then the location of that object could be any location asserted by the user or the system. But, if it is held by a hand, then the location of that object is the same as the location of the hand.

\[ ((\text{SOME HAND H})(\text{H holding } O)(\text{X location of } H)) \]

\text{OBJECT isat}

CC1

\text{ Here is a consistency condition that specifies the condition which an object will have to satisfy to be at a certain location. CC1 says that if the object is not held by any hand, then the location of that object could be any location asserted by the user or the system. But, if it is held by a hand, then the location of that object is the same as the location of the hand.}

\[ \text{(TDN: HAND (isat (PLACE RH) location of)} \]  
\text{(holding (OBJECT RH) held by CC2 TR1))} \]

\[ \text{CC2-HAND-holding:} \]
\[ \text{(CS2C: (QUOTE ((OBJECT X) \}  
\text{(h holding x)(x isat:location of}) \]
\text{HAND holding)}

\[ \text{TR1-HAND-holding:} \]
\[ \text{(OSTR: (QUOTE (((T ?) (DCONF \}
\text{(((SOME OBJECT X)(X is CLDVAL)))} 
\text{(IR (OLDVAL isat 0 isat))))} 
\text{(((SOME OBJECT X)(X is NEWVAL)))} 
\text{(IR (NEWVAL isat:flag 1))))))} \]
\text{HAND holding)}

CC2 says that if a hand is holding an object, then the object and the hand must be at the same place. The anchored transformation rule is invoked only for \text{ASSERT} or \text{IR} (Instantiate Relation) commands. If \( \text{(ASSERT (h holding b))} \) is successful (i.e. \( h \) was assigned value of \( (h holding) \)), then the CC evaluation would result in the truth value \( T \) or \( ? \). In this case, \( b \) would be the \text{NEWVAL} of \text{TR1}, and \text{TR1} will execute \( \text{(IR (NEWVAL isat:flag 1))} \), i.e. set the \text{flag} of relation 'isat' of the \text{object} \( b \) to '1'. Then the \text{flag} is 1 the value of the relation is not stored in the model space, but is computed, everytime it is needed. If one asserted \( \text{(IR (h holding b))} \) when \( h \) was initially holding \( b \), the \( b \) would be the \text{CLDVAL} of \text{TR1}, and in this case the \text{NEWVAL} will be \( ? \). Also, the CC2 would have returned the truth value \( ? \). Thus, \text{TR1} will execute \( \text{(IR (b isat (h isat)))} \) which will reset the \text{flag} of \( (b \text{isat}) \) back to 0, and assign the location of the hand \( h \), as the new location of \( b \).

\[ \text{The inverse relations are defined automatically by the system. Thus we get:} \]
\[ \text{(TDN: (COLOR BK) (colorof (OBJECTS SL) color))} \]
\[ \text{(TDN: (OBJECTS SL) (ELEMD OBJECT))} \]
Notice that many objects may have the same color, so the system automatically creates a new template, called 'OBJECTS', which is the collection of 'OBJECT's. For the similar reason, 'OFJOBRAND' is also created. The names for these new templates are declared by the user.

The CC's and TR's in the above domain are used by NDS to establish and maintain a consistent model space for the domain. In the case of the PLIND HAND domain, such a model space will contain specific instances of HANDs, PLACEs, OBJECTs and COLORs, and relations that relate these instances as per constraints specified by the template and consistency conditions. For a discussion of the way NDS uses the above CC's and TR to maintain consistency in the model space see Srinivasan [February 1976]. We shall discuss below only the aspects of NDS operations relevant to our example.

(2). Some comments on conventions in NDS

Let h be an instance of HAND in the model space, p a PLACE, t an OBJECT and c a COLOR. Let us consider the constraint CC1-OBJECT-isat. This constraint has the following form:

\[ ((\text{PLACE } X) \land P(c X)) \]

where \( P(c X) \) is a predicate expression with two free variables: \( c \) and \( X \). \( X \) is called the set-variable, and \( c \) is called the anchor of the CC. For an OBJECT, \( b \), if one asserts, \( (b \text{ isat } p) \), then NDS would bind the anchor variable \( c \) to \( b \)—the anchor variable is always bound to
the current instance at which an assertion is being made -- and the
set variable X to p, and evaluate the predicate \( P(p, X) \) in CC1. If the
predicate is satisfied then the assertion will be accepted.

Predicates like \( P(p, X) \) are evaluated in the 3VP model space in
3-valued logic: True, Unknown (?) and NIL. For example, if the
set variable X in \( P(p, X) \) is unknown, then the truth value of 
'\((\text{isat } X)\) appearing in CC1 will be hypothesized to be unknown in the
evaluation of CC1. In this case, if there is a HAND h, such that h
is holding b, then during the evaluation process of CC1, X will set
bound to the location of h, as a result of the expression: 
'\((((\text{SOME HAND } h)(h \text{ holding } b)(X \text{ locationof } h)))\)'. In this case the evaluation
of CC1 will return the location of h as its value. Also, in this
case, if one asserted that \((b \text{ isat } c)\) for a PLACE c, that is different
from p, then predicate in CC1, namely ' \((((\text{ALL HAND } h)(\text{NOT}(h \text{ holding } c))(c \text{ isat } X)) \text{ OR } (((\text{SOME HAND } h)(h \text{ holding } ?)(X \text{ locationof } h)))))\) will evaluate to NIL. If there is no HAND holding b, then CC1 will
accept any assertion of the form \((b \text{ isat } c)\) for any c. The reader may
similarly examine the interpretation of CC2-HAND-holding. In general,
if CC\([X r]\) is the consistency condition associated with the relation r
of a template X, then in 3VP CC\([X r]\) is evaluated as a function of two
arguments: \( CC(X r)(C Y) \), where C is the anchor variable and Y is the
set variable. The evaluation of \( CC(X r)(C Y) \) will return the bindings
for Y together with the truth value of the predicate in CC\([X r]\). This
truth value may, of course be, T(True), ? (Unknown) or NIL(False). In
general, CC\([X r]\) has the following interpretation:

\[ (C r Y) \leftrightarrow CC(X r)(C Y). \]
The model space will not accept assertion that produce contradictions in the CC's defined for a domain.

The transformation rules like the rule TR1-HAND-holding in the PLIND HAND domain are used in NDS to perform the side-effects that may be caused as a result of accepting an assertion into the model space. The specific side effects may of course depend on the truth value produced by the CC evaluation, of the CC's associated with a transformation rule. In the PLIND HAND domain TR1-HAND-holding will be invoked by NDS after CC2-HAND-holding is evaluated. Depending upon the truth value returned by CC2-HAND-holding the action prescribed in the rule are executed, as discussed before.

After completing the domain definition, the NDS will automatically build the DON-LISTS and DET-LISTS. Let CC[\(\lambda \, r\)] be the CC at \((X \, r)\). Then the DON-LIST of \((X \, r)\) is the list of all anchors \((Y \, m)\), such that \((Y \, m)\) occurs in CC[\(\lambda \, r\)]. In other words, the evaluation of the CC[\(\lambda \, r\)] will call for the value of \((y \, m)\) for some or all instances \(y\) of \(Y\), in the model space. For every \((Y \, m)\) that occurs in the DON-LIST of \((X \, r)\), the anchor \((X \, r)\) itself will occur in the DET-LIST of \((Y \, m)\). This has the following interpretation:

Let \(y\) be any instance of \(Y\), and let \([x]\) be the set of all instances of \(X\) in a model space. Then, every time the value of \((v \, x)\) is changed, in order to maintain the consistency of the model space, it may be necessary to check the CC's at every \((x \, r)\), for every \(x\) in \([x]\). Of course, for a particular \(v\), only a subset of \([x]\) may depend on the value of \((y \, m)\). To identify this subset, we shall associate with the DET-LIST entry \((X \, r)\), a constraint of the form \(((X \, x)\, r)\).
The PLIND HAND PROBLEM ...................... T. Hsu Dec. 1976 Page 20

(X)). Constraints of this kind are called FILTERS in UDS. If a filter is available at (Y m) then, when (v n) is asserted for a particular instance y of Y, the CC's at the anchors (x r) will be checked only for the objects x in ((X \ x) \ P(\ x)).

In the PLIND HAND domain, there are only two CC's, CC1-OBJECT-is-at and CC2-HAND-holding. So only (OBJECT isat) and (HAND holding) have the DOF-LISTS. For example, the DOF-LIST of (OBJECT isat) is '(((OBJECT isat) (HAND holding) (PLACE locationof)))'. The DOF-LIST of (HAND holding) is '(((OBJECT isat) (HAND holding) (PLACE locationof)))'. (PLACE locationof) occurs in the both DOF-LISTS of (OBJECT isat) and (HAND holding). Thus, the DET-LIST of (PLACE locationof) will have both the anchors (HAND holding) and (OBJECT isat), with associated filters. The DET-LIST and DOF-LIST generated by UDS for the PLIND HAND domain are shown below:

[A1]:
The DOF-LIST of (OBJECT isat) is:

(((HAND holding) (OBJECT isat) (PLACE locationof)))

The DET-LIST of (OBJECT isat) is:
At DET-anchor (HAND holding):

(((HAND Y) \ (S isat:locationof Y)))

-------------------------------

[A2]:
The DOF-LIST of (HAND holding) is:

(((HAND holding) (OBJECT isat) (PLACE locationof)))

The DET-LIST of (HAND holding) is:
At DET-anchor (OBJECT isat):

(((OBJECT Y) \ (E holding Y)))

-------------------------------

[A3]:
No DOF-LIST for (PLACE locationof).

The DET-LIST of (PLACE locationof) is:
1). At DET-anchor (OBJECT isat):
((OBJECT Y) ((? locationof:holding Y))
  OR (? locationof Y))

2). AT DET-anchor (HAND holding):
  ((HAND Y) (? locationof Y))

--------
[A4]:
No DON-LIST and DET-LIST for (COLOR colorof)

--------
[A5]:
No DON-LIST and DET-LIST for (OBJECT color)

--------
[A6]:
No DON-LIST for (OBJECT heldby)

The DET-LIST of (OBJECT heldby) is:
  At DET-anchor (HAND holding):
    ((HAND Y) (? heldby Y))

--------
[A7]:
No DON-LIST and DEL-LIST for (HAND isat)

-----------------------

It is instructive to examine the above DON-LISTS and DET-LISTS with reference to what happens when a hand, h, holding an object b, moves from one place to another: i.e. when one makes an assertion (h
isat q) for a place, q, when initially (h isat p) was true. The following operations will result in HDS:

The system will focus attention on the relation (h isat), (? locationof) and (b locationof). It will delete h from the value of (b locationof) and insert h in (a locationof), while at the same time substituting a for p in (h isat). This in effect is the new desired configuration. (h isat) has no DET-LIST. However, (PLACE locationof) has two DET-LIST entries: one is (OBJECT isat) with the filter shown in [A3] above; the other is (HAND holding), also with an associated
filter. These DET-LIST will be activated when (p locationof) and (o locationof) are changed.

The DET-LIST entry, (OPJECT isat) with its associated filter, demands that when (p locationof) is changed, all the objects satisfying the condition:

\[
((\text{OPJECT Y}) \land (p \text{ locationof:holding Y}) \lor (p \text{ locationof Y}))
\]

should now be examined, at (Y isat).

Thus, normally, when a hand, h, moves the location of the object held by the hand will get examined and updated if necessary. However, in our model, for the object, say b, held by h, (b isat:'flat') is '1'. Thus, the location of b will always be computed using CC1-OBJ-GEI-isat, every time it is needed. Therefore, there is really no need to examine the location of an object held by h when h is moved. Recognizing this fact, one may now associate with the DET-LIST anchor (OBJ-JECT isat) at, (PLACE locationof) an additional NULL filter, saying:

\[
((\text{OBJ-JECT X}) \land \text{NIL}).
\]

In this case, every time (PLACE locationof) is updated for any place p, no DET-LIST interactions will take place with (x isat) for any object x. This is done in KDS by set filter (CSFILTER) command below:

\[
\text{CSFILTER} : \left[ \left( \text{OBJ-JECT X} \right) \land \text{NIL} \right] \\
\text{(PLACE locationof)} (\text{OBJ-JECT isat})
\]
Thus, as the hand, h, is moved the system will update the location of h, without having to examine interactions with any of the locations of objects in the model space. The frame interactions are identified in NDS via the DET-LIST mechanisms. This enables NDS to identify inconsistencies, if any, in an updating process. The filter mechanism provides a way of controlling the combinatorial explosion that may result in general, in frame interactions of this kind. We have in the PLIND HAND domain an extreme case of the use of filter, where the filter is set to NIL. In general, one may associate a variety of filters to selectively control the frame interactions.

(3). The Solution of the Blind Hand Problem

In the statement and solution of the PLIND HAND problem, we will see below how NDS uses the above domain definition. In NDS, to solve this problem it is not even necessary to define separate actions, like 'GO', 'MOVE', 'PICKUP', etc. The following single transformation rule is enough:

\[
\text{CONSTR}: \text{MOVEOFJ}(X Y Z)\\
\begin{align*}
((\text{OBJECT } X) &\text{ (PLACE } Y Z) (X \text{ isat } Y) (\text{GOAL } (X \text{ isat } Z))) \\
((\text{SOME HAND } H) &\text{ (ASSERT } (H \text{ holding } X))\\
&\text{ (ASSERT } (H \text{ isat } Z))\\
&\text{ (ASSERT } (\text{NOT}(H \text{ holding } Y))))
\end{align*}
\]

Here the first line defines the name and the arguments of this transformation rule, MOVEOFJ(X Y Z); the second line is called the 'dimension' of the transformation rule which states how the arguments are bound and what the goal is; the rest of the rule is the body, which states how the goal is accomplished. This rule will be involved,
whenever there is a need to change the location of an object. The algorithm for changing is simple: set some hand to hold the object, change the location of the hand, and let the hand stop holding the object. All the necessary frame interactions that are needed to maintain the consistency of the model space while executing those actions are automatically inferred from the domain definition.

The statement of the problem would simply be:

\[
\text{((SOME OBJECT O) (O color red) (GOAL (O isat there)))}
\]

In response to this input, the DESIGNER will first enquire the MDS data base for the current value of each instance of OBJECT. If the current condition satisfies the goal, then would return 'SUCCESS'.

Let us assume that initially the model space contains the following objects and relations:

- OBJECTS: obj1, obj2, ..., obj10
- PLACES: here, there
- HAND: hand1
- COLOR: red
- RELATIONs: (here locationof (obj1, obj2, ..., obj6))
  (there locationof (obj7, obj8, ..., obj10))
  ((obj1, obj2, ..., obj6) color red)

Let the initial value of (hand1 isat) is ?.

From the input goal statement, the system first finds the set of objects such that the color of each element is red. Then it checks the location of each object in the set. If any one in that set is at place 'there', as mentioned before 'SUCCESS' is returned. Otherwise an 'invocation pattern' is generated. The 'invocation pattern' is used by MDS to invoke transformation rules that might be appropriate
to reach the goal. In our case, the model space does not satisfy the
goal. As a result of the initial examination of the model space the
system would have identified the following relevant bindings and
conditions:

\[(\text{OBJECT} \ 0): \ 0 \leftarrow (\text{ONEOF}(\text{obj}1, \ \text{obj}2, \ \ldots, \ \text{obj}6))\]
\[(\text{PLACE} \ P): \ P \leftarrow \text{here}\]

Initial Condition: \((\text{here locationof} (\text{obj}1, \ \text{obj}2, \ \ldots, \ \text{obj}6))\)
Goal Condition: \((\text{PLACE} \ C): \ C \leftarrow \text{there}; (C \ \text{locationof} \ C)\)

The generated invocation pattern would be:

\[((\text{OBJECT} \ 0)(\text{PLACE} \ P \ 0)(P \ \text{locationof} \ 0)(\text{GOAL} (C \ \text{locationof} \ C)))\]

Using this invocation pattern NOS would invoke the transformation
rule, \text{MOVEOPJ} defined above. The bindings shown above will be used in
the execution of the transformation rule, to bind the local variables
of the rule. \((\text{ONEOF}(\text{obj}1, \ \text{obj}2, \ \ldots, \ \text{obj}6))\) will cause one of the
indicated objects to be bound to \(X\). Let \(X \leftarrow \text{obj}1\) be the initial
choice; \(Y \leftarrow \text{here}\), and \(Z \leftarrow \text{there}\). The predicate '(\text{HERE} \ \text{PLACE} \ P)' in
the body of the rule will cause a hand from the model space to be
selected. Notice that there could be more than one hand in the model
space. Then an arbitrary choice will be made. In our case, of
course, hand1 will be chosen, resulting in \(l \leftarrow \text{hand}1\). Having done
all the bindings, the actions: '(\text{ASSERT} (\text{hand}1 \ \text{holding} \ \text{obj}1)),
(\text{ASSERT} (\text{hand}1 \ \text{isat} \ \text{there})), (\text{ASSERT} (\text{NOT}(\text{hand}1 \ \text{holding} \ \text{obj}1)))' will
be initiated in sequence. The assertion of '(\text{hand}1 \ \text{holding} \ \text{obj}1)' will
cause \text{CC2-HAND-holding} to be evaluated. Since, the location of
hand1 is unknown, \text{CC2} will evaluate to ?, and the residue '(\text{obj}1
\ \text{isat:locationof} \ \text{hand}1)' will be returned. This will cause the system
to make the hypothesis '(\text{here locationof} \ \text{hand}1)'
assertion. If initially, hand1 was 'there' then, of course, cc2 would have evaluated to NIL with the false residue '(obj1 is at:locationof hand1)'. In this case, if more hands are available in the system, then the system will choose another hand. However, while choosing another hand, h, it will make sure that the false residue '(obj1 is at:locationof h)' is not again violated. Thus, the system would have already learnt from its first mistake and avoid the mistake in subsequent trials. If no other hand is available then the above false residue may be used to set up a new subgoal, namely (GOAL (here locationof hand1)). In our case, (hand1 holding obj1) will succeed, causing TH1-HAND-holding to set (obj1 is at:flag 1), as discussed before.

It should also be noted that in the domain definition, '(HAND holding OBJECT)' indicates that a HAND can hold only one OBJECT. If hand1 was already holding an object b, then the assertion (hand1 holding obj1) will cause the system to remove b from being held by hand1, and introduce obj1 as the NEWVAL of (hand1 holding).

The remaining assertion in MOVEOBJ would now follow and complete the realization of our goal. The important point to note here is that, at the time the problem is stated or at the time the transformation rule is defined, it is not necessary for a user to be aware of domain constraints and frame interactions.

(4) Discussion
Besides the special facilities for controlling the combinatorial explosion of the frame problem mentioned above, the MDS has other distinguished features as follows:

1). The set concept is automatically built into the MDS formalism. There are two layers in MDS for building a knowledge base: the first one is the domain definition layer which defines the syntax and constraints of a domain; the second layer is the instantiation layer which builds the model space by instantiating instances and relations. During the instantiation, the system will automatically check the consistency of the new instance or relation according to the definition of the first layer, then accept it or reject it. Hence, each instance is closely related with its defined type, called "template", in the first layer. In the 3-valued logical system of the MDS model space, both the positive and negative values of elementary relations may be stored. Hence, MDS can efficiently evaluate negations of predicates.

2). The inverse relation updating process is built into the system. In general, if the relation \((x \rightarrow y)\) is defined, then the inverse relation \((y \leftarrow x)\) is automatically defined by the system.

3). Similar to the idea of the criticality value in ATATTIPS, in MDS one can define 'focus lists'. The focus list contains all the important predicates which must be satisfied first while making a new assertion. It helps the system find the correct order to process the control when many predicates must be satisfied at the same time.
4). There is no "state" variable or "time" parameter in DZ or AFSTRIPS. Hence it is very difficult to describe a rule which depends on time; like our initial condition, it is hard to say that "initially, all the objects at 'here' are red".

5). Because of the implementation of POM-LIST and DET-LIST, although the MDS is written in the first order logic, it really has some features of the second order logic. When each particular predicate is instantiated or updated, the system will go through the related POM-LIST and DET-LIST checking the consistency. Notice that this procedure which checks all the related predicates in the model space, in fact, is the same work we described in the frame rule in the second order logic. In addition the "residue" concept in MDS helps maintain the system efficiently. The residue is that subexpression of the CC which supplies the reason why the predicate evaluated to a particular truth value. Hence each time a predicate is updated, the value of each residue which contains that predicate is examined. If it keeps the same value, then it implies that the truth value of the predicate to which the residue belongs is also unchanged. On the other hand, if the truth value of a residue is changed, then re-evaluation of the related CC is necessary. Another distinguished feature of residues is the learning capability in a problem solving context. If a binding had generated 'true' for the truth value of the true-residue, then next time the same binding will be used. But if a binding had generated 'false' for the truth value of the false-residue, then the same binding will not be used again. In this way, the system learns how to bind things correctly and avoid the same wrong binding again.
Another good feature is having the model space of "DS work on a three valued logic. Hence we can say that the truth value of a predicate is unknown, which is very useful in creating the model or solving the problem. For example, in the blind hand problem, a lot of missing informations are involved, but the "DS can still solve the problem according to the available information. When an unknown residue is returned, the system will make the various necessary proper assertions associated with that residue. Since most AI problem solving systems only deal with two valued logic, when the unknown predicate occurs, it will assume that the truth value of that predicate is either true or false which may cause some inconsistency in the model later on. It is tedious to maintain such a system. Especially when a theorem prover is used in the system, the system may derive unexpected wrong results using this inconsistent data.

ACKNOWLEDGEMENTS

I wish to express my gratitude to prof. Chittoor V. Srinivasan who fully explained the various features in HDS to me and helped me solve the blind hand problem in HDS. Moreover, he carefully read the draft and corrected many mistakes in my English. I also wish to thank prof. Katesa S. Sridharan who suggested me to do this study and made the valuable comments and criticisms. Finally, many thanks are due to my good friend, Frank Kavrusik who read the early draft carefully and thoroughly, gave me a good deal of assistance in my writing.

REFERENCE


INTRODUCTION TO THE META DESCRIPTION SYSTEM

C. V. Srinivasan

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
INTRODUCTION TO THE META DESCRIPTION SYSTEM.

by

C.V. Srinivasan.

KEY WORDS: KNOWLEDGE REPRESENTATION, PROBLEM SOLVING, DESCRIPTIVE SYSTEMS.

Abstract:

In this paper we introduce the basic concepts of a knowledge based system called the Meta Description System (MDS). In MDS one first defines the language to be used for describing the knowledge in a domain, and the semantics of the language. Based on this definition MDS builds for itself a model space for the domain and uses the model space in a variety of problem solving activities.
1. INTRODUCTION.

The "problem of representation" in AI systems arises because of the ever present need to work with an incomplete corpus of immediately accessible information. In small well understood domains of reasonable complexity it is often possible to arrive at a "good" decomposition of the domain of knowledge into simpler parts for each one of which the necessary information and its associated processing facilities can be appropriately packaged, with reasonable assurance that all the components would interact harmoniously. As the complexity of the domain increases it becomes necessary to transfer some of the responsibility for packaging knowledge to the system itself. Here we face enormous difficulties. We do not even have a commonly agreed upon view of what the issues of the "problem of representation" are. At one extreme we have the view of procedural encapsulation of knowledge [Winograd 1972, Hewitt 1973, Minsky 1975]. At the other extreme we have the "purely declarative" approaches associated
with the use of general deductive systems, where there is only a weak notion of packaging; the necessary information in a context is available only implicitly. We also have now the middle view: We need both procedural encapsulation and declarative representations of knowledge; we should, therefore, find ways for wiring in both in some common framework of a problem solving control structure [Winograd 1975].

In the organization of these problem solving control structures there are now a few organizational concepts that are here to stay. These should have a role to play in the architecture of any intelligent system: One is the use of "model space" and model based reasoning, the other is the use of some kind of general deductive facility, and the third is procedural encapsulation. The current procedural/declarative controversy seems to be centered mainly around how one might wire in the necessary procedural knowledge and deductive facilities in the model space for a domain of knowledge. We believe that this is a non-issue. We would like to present below a point of view where the issues of "problem of representation" are presented as issues of communication among interacting processes. The basis for this shift of view is briefly the following:

The control structures associated with the model space, with the deductive mechanisms and with the procedure
invocation and execution can all be made independent of domain of knowledge. And, what is more, they can be made to specialize themselves by sharing their experiences with a given corpus of domain knowledge. This domain knowledge might itself have been described in a language of the domain, without reference to the control structures that use it. In this context the effectiveness of the specialization will depend crucially on the nature of the communication that can take place among the interacting process. Our thesis is that these interacting processes should have available to them the full richness of the language of the domain to communicate among themselves. The seeds of the communication problem will then lie in the structure, brevity, effectiveness and focus achieved in this communication. These would depend on the primitives available in the language of the domain and the concepts expressible in the language. The language of logic is too general, cumbersome and non-domain-specific to be useful here. To view the problem in this manner it is essential that one be able to have a view of what the knowledge in a domain is, independent of the model space and the control structures that use it. The framework of the Meta Description System (MDS) encourages the development of this view. The basic outlines of this framework are introduced here. At the moment we are still unable to offer a well reasoned approach that would reduce the "problem of representation" to a viable technical problem. But we believe we have some hopes of
achieving this end.

The central concept in the organization of the Meta Description System (MDS) is this separation that is achieved between the structure and semantics of domain knowledge on the one hand, and the control structures of the model space and problem solvers on the other. In MDS one first specifies the structure and semantics of the language of discourse for a domain. This is the DOMAIN DEFINITION specified in the MDS formalism. Based on the domain definition MDS builds a model space [Srinivasan 1976a] for the domain. This model space is used by a goal directed problem solver, called DESIGNER, as well as a Theorem Prover. These problem solving control structures have the ability to specialize themselves to operate efficiently in the domain, by communicating with each other in the language of the domain. The nature and effectiveness of this communication will depend on the primitives available in the domain language. In the context of MDS one may now experiment with alternate modes of description and investigate their effects on the problem solving efficiency.

In this paper we shall introduce the descriptive formalism of MDS in the context of a simple domain: The domain of MAZE problems. We will discuss the solution of the maze problem in the model space, and in the context of the DESIGNER. In another paper [Srinivasan 1976b] we show how the
same problem is solved by the Theorem Prover in MDS, using the very same definitions and the model space. We shall not discuss here aspects of domain specific specialization. The DESIGNER has a limited capacity to learn and generalize from its interactions with the model space [Srinivasan 1973, 1975a].

2. NATURE OF KNOWLEDGE AND PROCESSORS IN MDS.

MDS accepts three kinds of knowledge --about facts, objects, processes, and problem solving in a domain. The first is the STRUCTURAL knowledge. This pertains to the forms of descriptions of objects in the domain. The second is the SENSE knowledge. This pertains to the semantics associated with the structures. Sense knowledge is specified as predicates in the context of set constructions. The third is the TRANSFORMATIONAL knowledge. This pertains to the knowledge necessary to create new objects in the model space of the domain, and specialized knowledge pertinent to the updating processes in the model space. These rules are of two types: Those that are directly accessed and executed by CHECKER to effect well specified contingent changes in the model space, and those that are invoked on the basis of a pattern directed invocation.

The execution of all the processes is transparent to the system, in the sense that the different components of the
system can pass information to each other about what they are doing, unless one invokes specially built in opaque functions to do specific tasks. The STRUCTURE and SENSE definitions are used by the CHECKER-INstantiator system to create and maintain a consistent model space. The transformation rules are used by DESIGNER to plan and execute sequences of actions that modify the model space to reach desired objectives. The third control structure is that of the Theorem Prover [Srinivasan 1976b]. The TP is used to aid the CHECKER-INstantiator, and the DESIGNER. The fourth control structure is that of the LINGUIST. This may be used to define special user languages specific to a domain. The language understanding process is viewed as one of generating the appropriate structures in the model space in response to utterances in the language. This understanding process may implicitly invoke the full problem solving power of the system. All these control structures are domain independent. They specialize themselves to the domain based on the domain information.

MDS is not yet fully operational. We expect the model space management system [Srinivasan 1976b] to be working in a few months.
3. THE FORMS OF DOMAIN DEFINITION. THE MAZE PROBLEM.

The first part of domain definition is to identify the classes of objects in the domain. MAZE's have NODES, NODE, PATH, MAZEPROBLEM, etc. The description structure for these are specified first. This is shown in Table I. The definition of MAZE says the following: A MAZE is a NODE (in contrast to being a LIST, which is a collection of objects in the domain). The $N$ flag associated with the MAZE specifies this. The $ flag indicates that instances of MAZE need not be named objects in the model space. (For a full discussion of the various descriptive facilities in MDS see Srinivasan 1975). A MAZE has three descriptive relations associated with it: They are "startingnodes", "exit", and "contains". The starting node of a MAZE is an instance of NODES, which is a collection of instances of NODE. The exit of a MAZE is a NODE. This indicates that there is precisely one exit node, since NODE is a $N$ class of object. The MAZE contains NODES (a collection of nodes). The flags CC1, CC2 and CC3 indicate that certain consistency conditions (sense definitions, we shall use the two names interchangeably) are associated with the respective relations. We shall investigate the forms of these later. Similarly, we have the NODES, NODE, PATH, and MAZEPROBLEM schemas. Schemas of this kind are called TEMPLATES in MDS. The flag $L$ of NODES indicate that it is a LIST template.
TABLE I: THE DEFINITION OF THE MAZE DOMAIN.

\[
\begin{align*}
(MAZE \, N) \, &\{\text{startingnodes (NODES $L$) of CC1} \} \\
(\text{exit (NODE $N$) of CC2} \} \\
(\text{contains (NODES $L$) of CC3}) \}.
\end{align*}
\]

CC1: [MAZE startingnodes].
\[
[[\text{(NODE $X$)} \, \, (\text{startingnode $X$}) \, (X \, \text{connected from NIL})] \}.
\]

CC2: [MAZE exit].
\[
[[\text{(NODE $X$)} \, (\text{exit $X$}) \, (X \, \text{connected to NIL})] \}.
\]

CC3: [MAZE contains]
\[
[[\text{(NODE $X$)} \, (\text{startingnode $X$}) \, V \, (\text{exit $X$}) \, V \, (\text{contains $Y$}) \, (Y \, \text{can reach $X$})] \}.
\]

\[
\begin{align*}
(\text{NODES $L$}) \, &\{\text{element (0 * NODE)} \} \\
(\text{connected from $V$) (NODE $N$) connected to} \} \\
(\text{belong to $V$) (MAZE $N$ contains}) \}.
\end{align*}
\]

\[
\begin{align*}
(\text{NODE $N$}) \, &\{\text{connected to (NODES $L$) connected from CC4} \} \\
(\text{can reach $X$) (NODES $L$) can be reached from CC5}) \}.
\end{align*}
\]

CC4: [NODE connected to].
\[
[[\text{(NODE $X$)} \, (\text{connected to $X$}) \, (\text{NOT} (X \, \text{is} \, @))] \}.
\]

CC5: [NODE can reach].
\[
[[\text{(NODE $X$)} \, (\text{connected to $X$})] \}.
\]

\[
\begin{align*}
(\text{PATH $N$}) \, &\{\text{startingnode (NODE $N$) of CC6} \} \\
(\text{tail (PATH $N$) of CC6} \} \\
(\text{endingnode $S$) (NODE $N$) of CC7}) \}.
\end{align*}
\]

CC6: [PATH tail].
\[
[[\text{(PATH $P$)} \, (\text{tail $P$}) \} \\
(\text{P is NIL} \, V \} \\
(\text{(SOME NODE $N$) \} \\
(P \, \text{startingnode $N$} \} \\
(\text{is: connected to } $N$))]) \}.
\]

CC7: [PATH endingnode].
\[
[[\text{(NODE $X$)} \, (\text{going to NIL} \, V \, (\text{startingnode $X$}) \} \\
(\text{(ALL PATH $P$} \} \\
(\text{P has an end $X$})] \}.
\]
Every instance of NODES is a collection of the form \((n_1 \, n_2 \ldots \, n_k)\) where each \(n\) is an instance of NODE. This is indicated by the form "\((\text{elem} \, n \, (0 \, \ast \, \text{NODE}))\)" in the definition of the NODES. 0, \(\ast\) indicates that the lower bound on the number of elements in any instance of NODES is 0, and the upper bound is unlimited. Thus, for an instance of MAZE, say \(m\), its startingnodes might be \((n_1 \, n_2 \ldots \, n_j)\). This will appear in the model space of MDS as an assertion of the form \((m \, \text{startingnodes} \, (n_1 \, n_2 \ldots \, n_j))\). In the model space this is interpreted as: \((m \, \text{startingnode} \, n_1), (m \, \text{startingnode} \, n_2), \ldots,\) and \((m \, \text{startingnode} \, n_j)\) [we shall use singular and plural forms of relations interchangeably as is convenient].

For every relation used in a template, the template also specifies its inverse relation. Thus, the inverse of "contains" is "belongsto", and the inverse of "exit" is "exitof": For a MAZE \(m\), if \((m \, \text{exit} \, n)\) is true for a node, \(n\), then \((n \, \text{exitof} \, m)\) is also true in the model space, and vice versa.

Each consistency condition is of the form

\(((\langle \text{template} \rangle \, X) \, \| \, (P \, \& \, X))\),

where \((P \, \& \, X)\) is called the "predicate" of the CC, \(X\) is called the "set variable" of the CC and \& is called the "current instance" of the CC. It is the instance at which the CC is being evaluated in the model space. The CC may be read uniformly as: "The collection of all instances, \(X\), of
<template> such that \((P @ X)\) is true." Thus, CCl says that the starting nodes of a MAZE is the collection of all NODEs such that, if \(X\) is given as the starting node, \("(\@ startingnode X)" appearing in CCl is to be read in this manner), then \((X \text{ connectedfrom NIL})\) is true. We shall explain this convention further below. \((X \text{ connectedfrom NIL})\) means that there is no node from which \(X\) is connected to (connected to and connected from are inverses of each other). If the necessary and sufficient conditions are known for the definition of a relation, then the predicate of the CC, \((P @ X)\), will be such that, for a relation, \(r\), at which the CC is defined,

\[
(\@ r y) \iff (P @ y).
\]

However, if only the necessary condition is known for a relation to be true, then one would have a predicate, \(Q\), such that,

\[
(\@ r y) \implies (Q @ y)
\]

In this case we modify the predicate of the CC as shown in below:

\[
(\@ r y) \iff (\@ r y)(Q @ y).
\]

Thus, for ALL CC's the if and only if condition is true.

All quantifications appearing in CC's will range only over specified classes of objects in the domain. Thus, we write in CC7 \(((\all PATH P)(\@ tail P) \implies (P endingnode X))\), to indicate \(((\all P)(P \text{ instanceof PATH}) \& (\@ tail P) \implies (P \text{ endingnode X}))\). Generally, between adjacent predicates of the form \((x r y)(p r l q)\) implicit & is assumed. In general, \(((\all
$\forall x) P(x)$) is interpreted as \((\forall x)(x \text{ instanceof } X) \Rightarrow P(x)\). Similarly, \(((\exists x) P(x))\) is interpreted as \(((\exists x)(x \text{ instanceof } X) \& P(x))\). CC6 thus says that for a given PATH, $P$, $P$ can be the tail of a PATH if $P$ is NIL or for some NODE, $N$, it is true that $(P \text{ startingnode } N)$ and $@ \text{ startingnode is connectedto } N$. The phrase "($@$ tail $P)$" indicates that $P$ should be specified by an external agent. $(P \text{ is NIL})$ is interpreted as "$P$ is identically equal to NIL." CC3 defines the nodes contained by a MAZE inductively in terms of the startingnodes of a MAZE. The flag $@X$ is associated with the relation "canreach" in the NODE template. By convention, $X$ here indicates that the relation is transitive; the $@$ indicates that the value of this relation is never stored in the model space. Every time it is called for it is computed using the CC, CC5. CC5 specifies that the instance of NODES reached by a NODE is precisely the same as the NODES to which it is connectedto. However, since canreach has been declared to be transitive the model space will return always the transitive closure of the relation. Notice that a PATH has been defined to be anything starting with a node, containing a tail which is itself a PATH. The endingnode of a PATH is not stored in the model space. CC7 specifies that the ending node is the same as the starting node iff the tail of the PATH is NIL, else the endingnode is the same as the endingnode of the tail.
Even though the value returned by the CC's (Sense definitions) is always viewed as collections, the model space would give the returned value the proper interpretation based on the template definition associated with the CC. Thus, in case of the ending node of a PATH, from the template definition it is known that the ending has got to be an unique NODE.

The template for MAZEPROBLEM is defined in Table II.

**TABLE II: THE MAZEPROBLEM.**

\[
\text{((MAZEPROBLEM } \text{SN})(\text{startingnode (NODE } \text{SN}) \text{ startingnodeof CC8)} \\
\text{(solution (MAZEPROBLEM } \text{SN}) \text{ solutionof CC9}}
\]

\[
\text{TR9)} \\
\text{(maze (MAZE } \text{SN) mazeof}).
\]

CC8: [MAZEPROBLEM startingnode]
\[
\text{[(NODE X) (@ startingnode X) } \\
\text{(@ maze:contains X)}].
\]

CC9: [MAZEPROBLEM solution]
\[
\text{[(MAZEPROBLEM MP) (@ startingnode:is:exitof:mazeof @) } \\
\text{(MP is NIL) V } \\
\text{((SOME NODE N) } \\
\text{(N canreach:exitof:mazeof @) } \\
\text{(@ startingnode:is:connectedto N) } \\
\text{(MP startingnode N)}].
\]

TR9: [MAZEPROBLEM solution]
\[
\text{[(IFUNKNOWN (((SOME NODE n) } \\
\text{(@ startingnode:is:connectedto: n) } \\
\text{(n canreach:exitof:mazeof @)) } \\
\text{(BIND MP (CREATE MAZEPROBLEM } \\
\text{(startingnode n) } \\
\text{(maze (@ maze))))} \\
\text{(ASSERT (@ solution MP))} \\
\text{(ASSERT (MP solution)]}.}
\]

The MDS model space works in three valued logic, T, ?, (unknown), and NIL, T ?> >NIL. The CHECKER is used to evaluate CC's in the model space. CHECKER has no authority to
change the model space while evaluating the CC's. It can only poll and check, giving the proper interpretation for the quantifiers. In the case of the solution of the MAZEPROBLEM we have both a CC and a TR (transformation rule). TR9 will be invoked by the CHECKER of the model space if CC9 evaluated to UNKNOWN. This would be the case if no appropriate instances of MAZEPROBLEM are available in the model space. By convention TR9 will have three arguments available to it: The current instance, $\theta$, at which the associated CC was evaluated; the set variable, MP, of the CC; and the so called RESIDUES, if any (see Srinivasan 1976a for a definition of residues). Residues are predicates describing the reasons for the success, failure or the unknown value of the associated CC. They will always be sub-expressions of the CC, with specific bindings for the bound variables of the CC. TR9 looks for some node, $n$, such that $\theta$ startingnode is connected to $n$, and $n$ can reach the exit of the maze of the given mazeproblem. If it succeeds, then it creates a new instance of MAZEPROBLEM and assigns this as the solution of $\theta$. It then goes ahead and further asserts that the solution of the new mazeproblem should now be found. In this manner, if a MAZEPROBLEM is started off with a startingnode, and an assertion is made to find its solution, the CHECKER control structure can by itself find the solution.

Another approach for solving a maze would be by the use of the DESIGNER. The DESIGNER transformation rule for solving
a maze is shown Table III. The above rule will be invoked if one makes the assertion, for some known node, n, in the maze currently in the model space, \[((\text{SOME} \ \text{PATH} \ P)(P \ \text{startingnode} \ n)(\text{GOAL} \ (P \ \text{endingnode} \ e))).\]

**TABLE III: THE SOLVEMAZE RULE.**

SOLVEMAZE[m n e].

\[((\text{MAZE} \ n)(\text{NODE} \ n \ e)(m \ \text{contains} \ n)(m \ \text{exit} \ e)
\ (\text{SOME} \ \text{PATH} \ P)
\ (\text{GOAL} \ (P \ \text{startingnode} \ n)(P \ \text{endingnode} \ e)))\]

This is the Header for the Rule. The rule is invoked by pattern matching with this header.

\[(\text{BIND} \ P \ (\text{CREATE} \ \text{PATH} \ (\text{startingnode} \ n)))\]
\[(\text{DCOND} \ ((n \ \text{is} \ e)(\text{ASSERT} \ (P \ \text{tail} \ NIL)))\]
\[((n \ \text{canreach} \ e)
\ ((\text{SOME} \ \text{NODE} \ D)(n \ \text{connectedto} \ D)
\ (D \ \text{canreach} \ e)
\ (\text{ASSERT} \ (P \ \text{tail} \ \text{SOLVEMAZE} \ m \ D \ e)).\]

4. CONCLUDING REMARKS.

The basic elements of the descriptive formalism of MDS were introduced here in the context of a very simple example. A detailed discussion of the language of Transformation Rules and the DESIGNER processes appears elsewhere [Srinivasan 1975a]. An interesting aspect of the organization of the MDS model space [Srinivasan 1976a] is that the same model space is used by a variety of problem solving control structures: DESIGNER and the Theorem Prover [Srinivasan 1976b]. This is made possible because of the way the problem solving control structures communicate with the model space [Srinivasan...
5. ACKNOWLEDGEMENTS.

During the course of the development of the concepts in MDS, I have had several discussions with many of my colleagues. Discussions with Dr. Bertram Bruce, and Dr. Robert Balzer were particularly valuable. Prof. Sridharan participated in my class on Knowledge Based systems. Explaining the modeling concepts of MDS to Prof. Sridharan and describing parts of BELIEVER (a psychological modelling system) in the MDS formalism had been a particularly valuable experience. My students John Ng, Joel Irwin and Tau Hsu are all involved in the implementation of MDS. I am thankful to Prof. Adrian Walker, Ng and Hsu for carefully reading through this manuscript and for constructive suggestions for revisions.

6. REFERENCES:


Winograd, T: [1975]: Frame representations and the declarative/procedural controversy, in Representation and Understanding, Bobrow & Collins (Ed), Academic Press.


[1975b]: A formalism to define the structure of knowledge, RUCBM-TR-51, Department of Computer Science, Rutgers University, New Brunswick, N.J.

[1976a]: "The model space of the meta description system", Department of Computer Science Report, SOSAP-TR-19, Rutgers University.


Irwin, J. & ...: [1975b]: The description of CASNET in MDS., RUCBM-TR-49, Department of Computer Science.
THEOREM PROVING IN THE META DESCRIPTION SYSTEM

C. V. Srinivasan

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
THEOREM PROVING IN THE META DESCRIPTION SYSTEM.

by

C.V.Srinivasan.

KEY WORDS: THEOREM PROVING, MODEL SPACE, GENTZEN's SYSTEM.

Abstract:

In this paper we introduce a way of using the natural deduction system of Gentzen to do theorem proving in the context of a model space. The Theorem Prover actively uses the model space to test and generate hypothesis and guide itself. The model space itself is defined in the context of the descriptive formalism of the Meta Description System.
THEOREM PROVING IN THE META DESCRIPTION SYSTEM. (*1)

by

C.V. Srinivasan. (*2)

1. INTRODUCTION.

The purpose of this paper is to introduce the basic concepts used in the organization of the Theorem Prover (TP) in the Meta Description System (MDS) [Srinivasan 1973, 1975a,b, 1976a,b]. We follow essentially Beth's Semantic Tableaux [Beth 1959] approach. The TP seeks to construct a counter example, and in doing this it makes use of the model space of MDS, not only to test and generate hypotheses, but also to keep track of the various cases encountered in the theorem proving process and to actually build the representations for the counterexample or solutions, as the case may be. Our emphasis in this paper is on the kinds of interaction and communication that take place between the theorem proving control structure and the model space of MDS. We shall attempt to exhibit this interaction in the context of a simple example: the solution of maze problems. We assume that the reader is familiar with the concepts and organization

---

*1. This work was supported by grants from both the National Institute of Health (grant number RR-643), and the Advanced Research Projects Agency (Grant number DAHCIS-73-G6), of the Government of the United States of America.

*2 Department of Computer Science, Hill Center, Rutgers University, New Brunswick, N.J. 08903.
of the model space in MDS [Srinivasan 1976b]. We shall also assume that the reader is familiar with the Gentzen's [Kanger 1963] system of logic and its use in Beth's Semantic Tableaux approach to Theorem Proving. For the definition of the problem domain we shall use the MAZE DEFINITION introduced in [Srinivasan 1976a]. In [Srinivasan 1976c] we shall discuss possible application of the MDS theorem prover, to prove Cantor's theorem in Set Theory.

We believe, the most significant aspect of the TP organization introduced here is the separation that is achieved between the model space for a domain and its associated control structures, and the control structure of the Theorem Prover itself. The two control structures communicate with each other in the language defined for the domain, within the descriptive formalism of MDS.

The rules that govern the operation of the Theorem Prover are summarized in Tables IIa-c. The interpretation of these rules in the context of the TP control structure is described in Table V. The application of these rules will become clear in the discussion of the example.

2. BASIC DEFINITIONS AND CONVENTIONS.

The basic assumptions and conventions of the formalism of SEQUENTS and the concept of a Theorem Proving State, TP-State, are explained in Table I. The TP rules, shown in Tables
IIa-c, specify transformations on the sequents. Each rule of transformation has two parts, one above the line and the other below the line. The rule specifies that in any TP-state, a sequent with the form shown below the line may be replaced by sequent(s) of the form shown above the line.

**TABLE I: THE SEQUENTS AND THE THEOREM PROVING STATE.**

**SEQUENTS:** Each sequent is a string of the form

(P1)(P2)...(Pk) => (Q1)(Q2)...(Qm),

where P, Q are arbitrary predicates (in mini-scope form). The string (P1)(P2)...(Pk), may be interpreted as a conjunction of the predicates, (P1), (P2), ..., and (Pk). The string (Q1)(Q2)...(Qm), may be interpreted as a disjunction of the predicates, (Q1), (Q2), ..., and (Qm). We shall use the symbol, $S$, to denote arbitrary, possibly null, sequences of predicates of this form. Thus, the general form of sequent is: "$S_1 \Rightarrow S_2$".

**THEOREM PROVING STATE:** The TP-STATE is a set of sequents. We shall use ";" to separate the members of this set, and typically represent a TP-state by a string of the form: SEQ1;SEQ2;...;SEQk, where each "SEQ" is a sequent. The TP-state "\( S \Rightarrow S \)" asserts that $S$ is a theorem. The TP-state "\( S \Rightarrow \)" asserts that $\neg S$ (the negation of $S$) is a theorem.

There are three kinds of rules: PROPOSITIONAL rules, SUBSTITUTION rules, and QUANTIFIER rules. The propositional rules are derived from tautologies. Each rule has a label of the form "\( \Rightarrow c \)" or "\( c \Rightarrow \)" associated with it. It is convenient to think of a rule \( \Rightarrow c \), as a rule for the elimination of the symbol, $c$, (which could be a connective, or a quantifier symbol) from the right hand side of a sequent.
Similarly, a rule of the form \((c \rightarrow)\) is used to eliminate, \(c\), from the left hand side of a sequent. The order of application of these rules is shown in Table III. The SUBSTITUTION rules and the QUANTIFIER rules shown here are slightly different from the usual ones (See Kanger 1963). In the case of a substitution, the predicate \((x=y)\) is not maintained in the sequents of the TP-State, after the substitution [See Table IIb]. In the case of the quantified expressions, the expressions themselves are not maintained in the sequents of the TP-state after substituting for the bound variables by their respective Eigen Variables (or Eigen Terms) [See Table IIC]. These differences from the usual forms of these rules arise because of the way the rules are used, in the context of the MDS model space. The equality predicate gets incorporated into the model space and thus need not be maintained in the TP-state. In the case of the quantified expressions, the expressions themselves are always available in the model space, and are accessed when needed. Thus, they need not be maintained in the TP-state. At any instance the TP-state represents only a partial state of the theorem proving process. The total state of the process will include both the TP-state and the state of the entire model space. The ever present model space is always, implicitly on the left hand side of every sequent in the TP-state. These considerations will become clear in the discussion of the example.
TABLE II: RULES OF TRANSFORMATION OF SEQUENTS.

(a). PROPOSITIONAL RULES:

\[
\begin{align*}
(\rightarrow \neg) &: S_1 \Rightarrow S_2(P)S_3 \quad (\neg \rightarrow) &: S_1(Q)S_2 \Rightarrow S_3 \\
S_1(\neg P)S_2 \Rightarrow S_3 & \quad S_1 \Rightarrow S_2(\neg Q)S_3 \\
(\rightarrow \land) &: S_1 \Rightarrow S_2(Q_1)S_3; S_1 \Rightarrow S_2(Q_2)S_3 \\
& \quad \quad \quad S_1 \Rightarrow S_2((Q_1)\land (Q_2))S_3 \\
(V \rightarrow) &: S_1(P_1)S_2 \Rightarrow S_3; S_1(P_2)S_2 \Rightarrow S_3 \\
& \quad \quad \quad S_1((P_1)\lor (P_2))S_2 \Rightarrow S_3 \\
(\rightarrow \rightarrow) &: S_1(Q_1) \Rightarrow S_2(Q_2)S_3 \\
& \quad \quad \quad S_1 \Rightarrow S_2((Q_1)\rightarrow (Q_2))S_3 \\
(\rightarrow \rightarrow) &: (P_2)S_1 \Rightarrow S_2; S_1 \Rightarrow (P_1)S_2 \\
& \quad \quad \quad ((P_1)\rightarrow (P_2))S_1 \Rightarrow S_2 \\
(\leftrightarrow \rightarrow) &: S_1(P_1)(P_2)S_2 \Rightarrow S_3; S_1S_2 \Rightarrow (P_1)(P_2)S_3 \\
& \quad \quad \quad S_1((P_1)\leftrightarrow (P_2))S_2 \Rightarrow S_3 \\
(\rightarrow \leftrightarrow) &: S_1(Q_1) \Rightarrow S_2(Q_2)S_3; S_1(Q_2) \Rightarrow S_2(Q_1)S_3 \\
& \quad \quad \quad S_1 \Rightarrow S_2((Q_1)\leftrightarrow (Q_2))S_3
\end{align*}
\]

(b). SUBSTITUTION RULES.

\[
\begin{align*}
(= \rightarrow) &: S_1[x:y]S_2[x:y] \Rightarrow S_3[x:y] \\
& \quad \quad \quad S_1(x=y)S_2 \Rightarrow S_3
\end{align*}
\]

The notation "S[x:y]" is to be read as: "All occurrences of \(x\) in \(S\) are substituted by \(y\)." We have the axiom, "\(\rightarrow .S_1(x=x)S_2\)."

We also have the variant of the above \((= \rightarrow)\)' rule, where "\((y=x)\)" occurs in the sequent below the line, instead of "\((x=y)\)" shown above.

(c). QUANTIFIER RULES.

NOTE: ((\text{SOME \(T_1\) \(x\)P(x)}) is logically equivalent to ((\text{THERE-EXISTS \(x\) \(T_1\) instance \(x\)} \& P(x)). Similarly, ((\text{ALL \(T_1\) \(x\)P(x)}), or simply, ((\text{T1 x)P(x)} is logically equivalent to ((\text{ALL x})\((T_1\) instance x)\(\rightarrow P(x)\)).
Existential Generalization.

\[(\Rightarrow \text{SOME}): \quad S1 \Rightarrow S2(Q[x:z])S3 \]

\[S1 \Rightarrow S2((\text{SOME T1 } x)Q(x))S3 \]

Universal Generalization.

\[(\text{ALL } \Rightarrow): \quad S1(P[x:z])S2 \Rightarrow S3 \]

\[S1((\text{ALL T1 } x)P(x))S2 \Rightarrow S3 \]

The variable, \(z\), should be a \textit{NEW} variable, not used earlier in the TP-process. We shall call it \textit{EIGEN TERM} of type \(T1\). Its potential values range over all the instances of \(T1\) and eigen variables of type \(T1\). In effect, \(z\) has the status of a LISP unbound variable whose potential range of values are known. If \((z1 \ z2 \ ... \ \ zk)\) are the potential values of \(z\), then make note that \(z = (\text{ONEOF } (z1 \ z2 \ ... \ \ zk))\).

Existential Instantiation.

\[(\text{SOME } \Rightarrow): \quad S1(P[x:q])S2 \Rightarrow S3 \]

\[S1((\text{SOME T1 } x)P(x))S2 \Rightarrow S3 \]

Universal Instantiation.

\[(\Rightarrow \text{ALL}): \quad S1 \Rightarrow S2(Q[x:q])S3 \]

\[S1 \Rightarrow S2((\text{ALL T1 } x)Q(x))S3 \]

Here \(q\) is a new instance of \(T1\). We shall call it an \textit{EIGEN VARIABLE} of type \(T1\). \(q\) can be potentially equal to any one of the Eigen Terms of type \(T1\) or eigen variables of type \(T1\) created in the TP-process, prior to the creation of \(q\).

3. THE CONTROL STRUCTURE.

The control structure for the theorem prover, is shown in Table III. This table is best understood in the context of the example discussed in the next section. The significant aspects of the table are discussed below.

All the applicable propositional, equality and quantifier
rules are applied first. As a result of this process the sequents in the TP-state might have several occurrences of elementary predicates of the form \((x \, r \, y)\) (*3). In each predicate of the form \((x \, r \, y)\), \(x\) and \(y\) might be either eigen variables (i.e. constants in the model space), or eigen terms (variables in the model space with possibly specified ranges of potential values). These elementary predicates are asserted into the model space, using the THASSERT function, as described in step 2 of Table III.

In each THASSERT the model space will be used to actively seek for possible assignments of values for the eigen terms, that result in a contradiction. We have two kinds of contradiction: The WEAK contradiction is a contradiction with respect to the model space. In this case a predicate in the THASSERT could not be accepted by the model space for any of the possible assignments of values to its eigen terms. The STRONG contradictions is the usual concept of contradiction used in theorem proving: There are predicates, \((x \, r \, y)\) and \(-(u \, r \, v)\) in a THASSERT, and value assignments, say \(p\) for \(x\) and \(u\), and \(q\) for \(y\) and \(v\). If such valuations exist then the THASSERT will find them. In each THASSERT we first look for strong contradictions. If no strong contradictions exist then we

---

*3 We shall talk about only binary predicates in this paper. In MDS there are facilities available to consider n-ary predicates for any \(n > 0\). Also, one may have function symbols occurring in the first order expressions. Examples of theorems with function symbols in them are discussed in [Srinivasan 1976c].
look for weak contradictions. We shall say that a TP-state contradicts the model space if there is a strong or weak contradiction in every sequent of the TP-state for some possible, but common choice of values (i.e., the same eigen term occurring in two different sequents should have the same value chosen for it in both sequents), for the eigen terms in the TP-state. Notice that the contradiction is defined with respect to the model space. We are making the assumption here that our domain definitions are such that if there is a contradiction with respect to the model space then in the TP-process it will eventually show up as a strong contradiction — i.e., the model space is complete and consistent. A basis for this assumption is the residue theorem, mentioned below. If there are inconsistencies in the domain definition then it is impossible to predict what might happen. The collection of THASSERTs is used to find valuations for the eigen terms in the TP-state, that will produce a contradiction with the model space in every sequent in the TP-state, if such a contradiction is possible.

For each eigen term its possible range of values will be separated into two parts: Those that do not produce a contradiction in any of the THASSERTs, and those that produce a contradiction in at least one THASSERT. If no contradiction is encountered for any of the possible assignments of values for the eigen terms in a THASSERT, then the appropriate UNKNOWN residue (See Srinivasan 1976b) is returned to the
TP-state, if one existed. The unknown residue is substituted for the predicate associated with it, in the appropriate sequents as explained in Table III.

For a predicate, \( (P) \), with or without quantifiers, the unknown residue exists only if \( (P) \) has truth value, \(? (UNKNOWN)\), for the given valuations of the terms in the predicate. The residue then is the part of the predicate \( (P) \) that are UNKNOWN in the model space. One may view these as the conditions under which a given assertion to the model space can become true. In building models using the residues we make use of the RESIDUE THEOREM discussed in [Srinivasan 1976b]: Let \( V \) be the valuation of the terms in \( (P) \), and let \( R[V](P) \) be the residue of \( (P) \) for the valuation \( V \). Let \( PV \) be the predicate \( P \) with the known valuations in \( V \) kept fixed. Then the residue theorem says that \( R[V](P) \leftrightarrow (PV) \). That is any model that is valid for \( R[V](P) \) is valid also for \( (P) \). Thus, any model that contradicts \( R[V](P) \) would also contradict \( (P) \).
TABLE III: CONTROL STRUCTURE FOR THE THEOREM PROVER.

Steps 1 through 4 constitute a STAGE in the TP-Process. Each stage begins with a TP-state and ends with (possibly) a new TP-state.

1. First apply ALL applicable propositional rules, equality rules and quantifier rules. For each new Eigen Term, z, keep note of the possible range of values that the eigen term can have. The ONEOF function is used for this purpose. For each application of an Instantiation rule do (CREATE T1 v) -- T1 here is the type of v -- if the model space permits the creation of new instances of T1. The symbol v is a new symbol not used previously in the TP-process. If the creation of new instances is prohibited in the model space, then create a new eigen variable say v, and set

\[ v = \text{ONEOF } \langle \text{the instances of T1 in the model space. These instances are not to include any of the previously generated eigen variables in the TP process.} \rangle. \]

Please note that v is still treated here as an eigen variable, and NOT as an eigen term. The ONEOF function is used only for existential instantiations. If v is the result of an universal instantiation then mark v by the label ALL, else mark it by the label SOME.

2. For each sequent in the TP-state do the following: Let \( p_1, p_2, ..., p_n \) be all the predicates of the form \((x \, r \, y)\) on the left hand side of a sequent in the TP-state, and let \( q_1, q_2, ..., q_m \) be the similar predicates on the right hand side of the same sequent. Do

\[ \text{A: } [\text{THASSERT } p_1 \, p_2 \, ... \, p_n \, \neg q_1 \, \neg q_2 \, ... \, \neg q_m]. \]

This might cause these elementary predicates to be moved into the model space. Let \( A_1, A_2, ..., A_k \) be all the assertions made to the model space, in this manner. Let \( P_1, P_2, ..., P_k \) be the strings of all the predicates, respectively in \( A_1, A_2, ..., A_k \). Let \( \text{SEQ1, SEQ2, ..., SEQk} \) be respectively the sequents with which the assertions are associated. A THASSERT, \( A_i \), is used to update the model space only if it does not contradict any of the HYPOTHESIS (see (c) below) generated by the TP-process. If a THASSERT does \( \neq \) cause a contradiction in the model space, but contradicts a hypothesis, then we shall not update the model space. The associated sequent will be left in the TP-state unchanged.

One of the following three can now happen for each such THASSERT, \( A_i \):

(a) All the predicates in \( P_i \) are accepted by the model space. Some of them are unconditionally accepted by the model
space. In this case delete the accepted elementary predicates in $\Pi$ from the sequent $\text{SEQ}_i$. If as a result of this, $\text{SEQ}_i$, becomes NULL (both its left and right sides become empty), then undo the THASSERT, $A_i$. The predicates in $\Pi$ are now candidates for a new HYPOTHESIS. Some of the predicates are accepted by the model space conditionally. An UNKNOWN-residue (See Srinivasan 1976b for the definition of residues) is returned for each such conditionally accepted predicate. In this case, substitute each such predicate in $\text{SEQ}_i$ by its associated residue.

(b) One or more of the predicates is NOT accepted by the model space. In this case simply delete the sequent from the TP-state and make note that a contradiction has occurred. Keep note of the assignments of values to the eigen terms that caused the contradiction.

(c) After doing all the THASSERTs and the associated updates to the sequents, see whether there are any NULL sequents in the TP-state. If $\text{SEQ}_i, \text{SEQ}_j, \ldots, \text{SEQ}_t$ are all NULL then delete them from the TP-state, and generate the following hypothesis:

$$\text{HYP: } \neg (\Pi \lor \Pi_j \lor \ldots \lor \Pi_t),$$

where $\Pi, \Pi_j, \ldots, \Pi_t$ are respectively the predicates appearing in the assertions $A_i, A_j, \ldots, A_t$, associated with the sequents $\text{SEQ}_i, \text{SEQ}_j, \ldots, \text{SEQ}_t$.

3. If the TP-state is empty, and if a contradiction has occurred then the theorem is proven. The solution may now be extracted from the model in the model space. The solution is found by assigning the proper values for the eigen terms. We shall not discuss here the solution extraction process.

If no contradiction has occurred then GO AND CONSULT USER about what to do next. We shall discuss some of the strategies appropriate for this situation in an ensuing paper [Srinivasan 1976c].

4. If the TP-state is not empty then go to step 1.
When a predicate is asserted into the model space it initiates a rather complex process. First the constraints associated with the predicate and the constraints in all its dependent predicates are evaluated. If there are also TRANSFORMATION RULES (See Srinivasan 1976b, 1975b) associated with any of the relations asserted into the model space they will also get evaluated and all the appropriate "side effects" will be taken care of. In the case of an assertion \((x \ r \ y)\), where both \(x\) and \(y\) are eigen variables (i.e. \(x\) and \(y\) are constants already in the model space), this assertion may actually cause the models in the model space to change. The appropriate residues will then be returned to the TP-state. In the case of eigen terms, we shall associate with each eigen term its range of possible values: Those leading to a contradiction as well as those that do not lead to a contradiction at a given stage of the TP-process. For an eigen term, \(y\), of type, say \(Y\), we will maintain as possible values only the instances of \(Y\). We shall write this as: \(y = \text{ONEOF}\{y_1 \ y_2 \ ... \ y_n; z_1 \ z_2 \ ... \ z_m\}\), where the \(z\)'s cause contradiction. We shall use the notation \((x \ r)\) to denote the collection of all \(y\) such that \((x \ r \ y)\) is true in the model space and at times write \(y = (\text{ONEOF}(x \ r))\) to indicate that the possible values of \(y\) are precisely those that satisfy \((x \ r)\).

The assertions into the model space perform four functions: (a) Recognize contradictions, if any, (b) delimit the scopes of the relevant eigen terms if possible,
(c) generate HYPOTHESIS to guide the TP in its operations (we shall discuss this in greater detail in section 5), and
(d) return to the TP-state additional constraints relevant to the proof of the assertion in the form of UNKNOWN residues—logical expressions whose truth values could not be as yet determined in the model space, because the needed information is not yet available. With these preliminary comments we may now consider the solution of the maze problem.

4. THE MAZE PROBLEM.

To understand the definitions given in Table IV please see [Srinivasan 1976a]. To follow the discussion below it is essential that the reader understand Table IV and the concept of residues [Srinivasan 1976b]. Let us assume that an instance of MAZE has been already created in the model space and the connectivity of all its nodes have been specified; no new nodes might be created in the model space. For some particular node, n, of this maze we wish to prove the assertion [all] of Table V. We shall comment on the proof shown in Table V in the next section. This table is mostly self explanatory.
TABLE IV: THE DEFINITION OF THE MAZE DOMAIN.

\[
((\text{MAZE } $N)) (\text{startingnodes } (\text{NODES } $L) \text{ startingnodesof CC1})
\]
\[
(\text{exit } (\text{NODE } $N) \text{ exitof CC2})
\]
\[
(\text{contains } (\text{NODES } $L) \text{ belongto CC3})].
\]

CC1: \[\text{[MAZE startingnodes]}.\]
\[
((\text{NODE } X)| (@ \text{ startingnode X}) (X \text{ connectedfrom NIL})].
\]

CC2: \[\text{[MAZE exit]}.\]
\[
((\text{NODE } X)| (@ \text{ exit X}) (X \text{ connectedto NIL})].
\]

CC3: \[\text{[MAZE contains]}\]
\[
((\text{NODE } X)| (@ \text{ startingnode X}) V (@ \text{ exit X}) V
\]
\[
((\text{SOME NODE Y})(@ \text{ contains Y}) (Y \text{ canreach X})].
\]

\[
((\text{NODES } $L)) \text{ elemdn } (0 * \text{ NODE}))
\]
\[
((\text{connectedfrom } V) (\text{NODE } $N) \text{ connectedto})
\]
\[
((\text{belongto } V) (\text{MAZE } $N) \text{ contains})].
\]

\[
((\text{NODE } $N)) \text{ connectedto } (\text{NODES } $L) \text{ connectedfrom CC4}
\]
\[
((\text{canreach } $X) (\text{NODES } $L) \text{ canbereachedfrom CC5}).
\]

CC4: \[\text{[NODE connectedto]}.\]
\[
((\text{NODE } X)| (@ \text{ connectedto X}) (\text{NOT}(X \text{ is } @))].
\]

CC5: \[\text{[NODE canreach]}.\]
\[
((\text{NODE } X)| (@ \text{ connectedto X})].
\]

\[
((\text{PATH } $N)) \text{ startingnode } (\text{NODE } $N) \text{ startingnodeof}
\]
\[
(\text{tail } (\text{PATH } $N) \text{ tailof CC6})
\]
\[
((\text{endingnode } $) (\text{NODE } $N) \text{ endingnodeof CC7})].
\]

CC6: \[\text{[PATH tail]}.\]
\[
((\text{PATH } P)| (@ \text{ tail P})
\]
\[
((P \text{ is NIL}) V
\]
\[
((\text{SOME NODE N})
\]
\[
(P \text{ startingnode N})
\]
\[
(@ \text{ startingnode:is:connectedto N})].
\]

CC7: \[\text{[PATH endingnode]}.\]
\[
((\text{NODE } X)| (@ \text{ tail NIL}) \leftrightarrow (@ \text{ startingnode X})
\]
\[
((\text{ALL PATH } P)
\]
\[
(@ \text{ tail P}) \rightarrow (P \text{ endingnode X})].
\]
TABLE V: STATEMENT AND SOLUTION OF THE MAZE PROBLEM.

The nodes n and e below are assumed to be constants, already in the model space.

[a1]. \[\Rightarrow ((\text{SOME PATH } P) \\
(P \text{ startingnode } n) \Rightarrow \\
(P \text{ endingnode } e))\]

Apply \((\Rightarrow \text{SOME})\) rule. No prior instance of PATH exists. Hence, do \((\text{CREATE PATH } p)\) and do \([P:p]\) for the right hand side of the sequent.

[a2]. \[\Rightarrow ((p \text{ startingnode } n) \Rightarrow \\
(p \text{ endingnode } e))\]

By \((\Rightarrow \neg\Rightarrow)\)

[a3]. \((p \text{ startingnode } n) \Rightarrow (p \text{ endingnode } e)\).

[THASSERT \((p \text{ startingnode } n) \neg (p \text{ endingnode } e)\)].

[a4]. \[\Rightarrow \{\text{Residue of } (p \text{ endingnode } e)\} \]

which is \[\Rightarrow ((p \text{ tail } NIL) \leftrightarrow \\
(p \text{ startingnode } e)) \& ((\text{ALL PATH } P) \\
(p \text{ tail } P) \Rightarrow (P \text{ endingnode } e))\].

By \((\Rightarrow \&)\) and \((\Rightarrow \leftarrow\Rightarrow)\),

[a5]. \((p \text{ tail } NIL) \Rightarrow (p \text{ startingnode } e)\); \n\((p \text{ startingnode } e) \Rightarrow (p \text{ tail } NIL)\); \n\[\Rightarrow ((\text{ALL PATH } P) \\
(p \text{ tail } P) \Rightarrow (P \text{ endingnode } e))\];

By \((\Rightarrow \text{ALL}), \text{(CREATE PATH } pl)\), and do \([P:pl]\) for the last sequent above, and do \((\Rightarrow \neg\Rightarrow)\) to get \((p \text{ tail } pl) \Rightarrow (pl \text{ endingnode } e)\).

[THASSERT \((p \text{ tail } NIL) \neg (p \text{ startingnode } e)\)],
[THASSERT \((p \text{ tail } NIL) (p \text{ startingnode } e)\)],
[THASSERT \((p \text{ tail } pl) \neg (pl \text{ endingnode } e)\)].

If \(n = e\), then the earlier assertion \((p \text{ startingnode } n)\) will contradict with \((p \text{ startingnode } e)\). In the second THASSERT, \((p \text{ tail } NIL)\) will contradict with \((p \text{ tail } NIL)\) of the first THASSERT. Similarly, \((p \text{ tail } pl)\) of the third THASSERT also will contradict \((p \text{ tail } NIL)\). This will terminate the proof. If \(n\) is not equal to \(e\), then the first THASSERT will generate the following hypothesis (since sequent becomes NULL as a result of the THASSERTs):
HYPl: "((p tail NIL)" (p startingnode e))

Notice that the above hypothesis is a special case of the condition CC6, associated with [PATH tail]. The TP-process has now discovered a property of the domain as a result of its interactions with the model space. It should be mentioned that one should be extremely lucky to discover properties in this manner! The ability to discover properties like this is dependent on the kinds of domain descriptions that have been given to the system. The second THASSERT will produce a contradiction in the model space, since (p startingnode n) is already true in the model space. The last THASSERT will produce,

[a8]. [Residue of (p tail pl)] =>
[Residue of (pl endingnode e)].
which is

(SOME NODE N)
(pl startingnode N)(n connectedto N)

=>(((p tail NIL)<->
(pl startingnode e))&
((ALL PATH P)
(pl tail P)->(P endingnode e).

Notice that the phrase "(p startingnode:is:connectedto N)" occurs in CC6 from which the residue on the left hand side above was obtained. However, (p startingnode) is already known to be n in the model space as a result of the assertion in [a3]. Thus, (p startingnode) gets substituted by n in the residue. We now do (SOME =>), (=> &), (=> ALL) and (=> ->). Since the creation of new nodes has been blocked, it is now not possible to CREATE a new node for (SOME =>) rule application. Therefore, an eigen variable, vl, is created and set equal to,

vl=(ONEOF <the nodes in the model space>).

A value for this vl has to be found from the model space. Notice, however, that vl does not have the status of an eigen term, in the TP process. That is, in the THASSERT process vl cannot be set equal to another eigen variable. Now, (CREATE PATH p2) and do [P:p2]. Then doing all the propositional rules, we get:

[a9], (pl startingnode vl)(n connectedto vl)(pl tail NIL)

=>(pl startingnode e);
(pl startingnode vl)(n connectedto vl)

=>(pl startingnode e)
(pl startingnode e)

=>(p2 endingnode e).
This leads to,

\[ \text{THASSERT (pl startingnode } vl \text{) (n connectedto } vl \text{) (pl tail NIL)} \]
\[ \text{ "(pl startingnode e)]}, \]
\[ \text{THASSERT (pl startingnode } vl \text{) (n connectedto } vl \text{) (pl startingnode e)" (pl tail NIL)}], \]
\[ \text{THASSERT (pl startingnode } vl \text{) (n connectedto } vl \text{) (pl tail p2) " (p2 endingnode e)}. \]

The assertion of \((n \text{ connectedto } vl)\) would limit the scope of possible values of \(vl\) to those nodes that are known to be connected to \(n\), in the model space. If \(vl\) contains \(e\), then \(vl\) can be assigned the value \(e\), and this would produce a contradiction in the first THASSERT. Maintaining the same assignment for \(vl\) would also produce contradictions in the second and third THASSERTs above. In the case of the second ASSERT \("(pl tail NIL)\) will not be accepted, since as a result of the first assertion we now would have \((pl \text{ tail NIL})\) in the model space. Similarly, in the case of the third ASSERT \((pl \text{ tail p2})\) will not be accepted. This will thus terminate the proof. The PATH \(p\) in the model space is the solution.

If \(vl\) does not contain \(e\), then the first THASSERT will cause a NULL sequent. Its predicates will become candidates for a new hypothesis. The second THASSERT will cause a contradiction, since \((pl \text{ startingnode } e)\) cannot be accepted. It would violate CC6. In the third ASSERT we will have,

\[ [a10]. \text{ [Residue of (pl tail p2)]} \]
\[ \Rightarrow [\text{Residue of (p2 endingnode } e)]. \]

which would exactly be the same as \([a8]\) with the substitutions: of \(p2\) for \(pl\) and \(pl\) for \(p\), and the process \([a8]\) through \([a10]\) will be repeated. This will iterate until a path is reached whose starting node could be \(e\). In the case we would have in the model space the following:

\((p \text{ startingnode } n) (pl \text{ startingnode } vl)(p2 \text{ startingnode } v2) \]
\[ \ldots (pl \text{ startingnode } vl)(p(i+1) \text{ startingnode } e), \]

for some integer \(i\). Also we would have,

\( vl = \text{(ONEOF (n connectedto}; \ldots)), \]
\( v2 = \text{(ONEOF (v2 connectedto}; \ldots)), \]
\( \ldots \]
\( vi = \text{(ONEOF (v(i-1) connectedto}; \ldots)). \]

The paths that are solutions to the problem may now be extracted from this.
5. DISCUSSION OF THE PROOF.

In this proof we have assumed the simplest situation. We already knew the full description of the maze. If the description of the maze were not available then the above proof might not have terminated. The theorem prover does not have the ability to recognize that some information is missing and could be acquired by consulting with an outside agent. The process will not terminate also in cases were there is no path and there is a loop in which the search is trapped. We could have defined the domain differently to take care of the loop problem: One would introduce a new relation in PATH which identifies all the nodes in the path prior to a given PATH location, and insist that the starting node of a tail should never be one of the nodes already appearing in the prefix of a path. There are facilities in the MDS model space also to indicate situations in the description of an object where a missing piece of information has to be obtained from an outside agent. In the case of a maze one might, for example, specify that the connecting nodes of a given node has to be obtained from an agent. So also, one might specify that the starting node of a PATH should be obtained from outside consultation. In general, for every relation in the various schemas shown in Table IV, if the relation does not have a constraint associated with it, then one might say that to acquire that piece of information one has to consult with an outside agent. This can be done in MDS by associating flags
with the relations [See Irwin, J. & Srinivasan 1975].

The organization described here does not provide a good search strategy to look for the right kinds of bindings for the eigen terms. In general the problem of estimating the possible bindings and choosing the right ones is an unsolvable problem. Usually one just carries along all the available values and does not make any commitment until a contradiction is identified. In the procedure described above we do make commitments on the choice of values based on what the model space accepts. At times this leads to NULL sequents in the TP-state. In the example discussed above we had a lucky set of circumstances: We encountered the hypothesis before we made any commitments on the choice of values for any of the eigen terms. In general, when a sequent becomes NULL one is in a situation where one might have to do some back-tracking, and undo some of the changes previously made in the model space.

An important property of Gentzen's system is, it can run in either direction; there is no information loss. This is not true for the RESOLUTION approach. When a sequent becomes NULL one should back track to the previous TP-state and undo any changes that might have been made in the model space by the THASSERT associated with the sequent corresponding to the NULL sequent. This might cause one to reevaluate the previous TP-state all over again, and change many of the bindings chosen for the eigen terms. We shall not discuss here the organization of the model space and the TP-state that make
such back tracking feasible. Again, if the domain descriptions are right, hopefully, such back tracking would be avoided.

If the TP-state becomes null and no contradiction had been obtained, and possibly also some hypothesis are available, then in certain cases it might still be possible to continue with the proof procedure if the right kinds of new instances of the necessary objects are created in the model space. This might be done by consulting with a user. Also, in these situations, at times, the system can use the so-called "model completion" criteria to continue with the proof. The concept of model completion is discussed in detail in Srinivasan [1976c]. The objective in model completion is to create new objects in the model space necessary to complete all the missing (unknown) information in the model.

The thrust of this work has been so far not in the identification of good search strategies, but in creating a TP control structure that can interact with a model space which has itself been defined in a context totally independent of the TP process. Also, it seems the TP can explain its operations to a user in a natural way.

The procedure presented here is bound to be incomplete, because of the weak requirement for contradiction: The notion of contradiction is dependent on the model space. However, if the domain description is right then we can prove interesting
theorems. We do not use theorem proving as the principal mode of problem solving in MDS. Our objective had been to fulfill an aesthetic theory: If one could describe knowledge to a computer and have the computer understand the knowledge then one should be able to use the knowledge not only in highly specialized ways but also in the context of a general deductive system. The TP is expected to be used in MDS to guide construction of instances of schemas, and to resolve unanticipated conflicts in the updating of the model space. Our ideas on the use of TP in these areas are at the moment still in the developing phase. The central problem is one of using the TP in reasonably efficient ways to help in the updating process.

We expect to have the MDS model space management system working in a few months. The TP is not yet implemented. We are still at the stage of exploration.

ACKNOWLEDGEMENTS:

During the early stages of development of the ideas presented here, in the spring of 75, when we were still learning about Gentzen's system and Beth's Semantic Tableaux I had the benefit of many discussions with Jared Darlington. These discussions were extremely valuable in crystallizing many of my ideas. The discussions I had with my student, Tau Hsu were very useful for identifying the deficiencies in the TP process and correcting some errors.

5. REFERENCES:

Kanger, Stig [1963]: A simplified proof method for elementary logic, In Computer Programming and Formal systems,
Beth, E.W [1959]: The foundations of Mathematics, North Holland publishing company, Amsterdam.


[1975a]: A formalism to define the structure of knowledge, RUCBM-TR-51, Department of Computer Science, Rutgers University, New Brunswick, N.J.

[1975b]: The meta Description System, RUCBM-TR-50, Department of Computer Science.


Irwin, J. & ...[1975]: The description of CASNET in MDS., RUCBM-TR-49, Department of Computer Science.
DESCRIPTION OF CASNET IN MDS

J. Irwin and C. V. Srinivasan

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was partially supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
TABLE OF CONTENTS

1. Introduction.
2. Basic concepts: CASNET and the METHODOLOGY of description in MDS.
3. The DESCRIPTION STRUCTURE of CASNET.
4. Templates and their instantiations.
   4.1: Types of templates and examples of their use in CASNET.
   4.2: The relation flags and their use.
   4.3: The CONSISTENCY CONDITIONS.
      4.3.1: The form of CC's.
      4.3.2: The interpretation of CC's.
5. The MODEL DEFINITION and MODEL INSTANTIATION processes.
   5.1: The Definition of CAUSALMODEL.
   5.2: MODEL INSTANTIATION process.
      5.2.1: The Record Keeping Process.
      5.2.2: The test selection and application.
7. Acknowledgements.

REFERENCES

APPENDIX I: CASNET DEFINITION.
1. Introduction.

This report presents a formal description of the CASNET system of Kulikowski and Weiss, [Weiss, 1972], in the formalism of the Meta Description System (MDS) [Srinivasan ...]. The report is addressed to readers who have some knowledge of the CASNET system. The purpose of the report is to illustrate the use of MDS formalism, its power and flexibility. The description of a system in the MDS formalism not only serves as a precise documentation of what a system is, but also as a program that implements the described system in the context of the processors in MDS. Thus, MDS can not only execute the system described to it, but also answer questions about the described system. The report is written in such a way that readers familiar with CASNET will get a fairly good grasp of the facilities, power and advantages of using the Meta Description System.

2.0: Basic Concepts: CASNET and the METHODOLOGY of description in MDS.

CASNET is a system for modeling disease processes in terms of CAUSAL NETs. The nodes of a causal net would represent the so-called "disease states", and directed links between pairs of states (nodes) would represent the "causality" relation between the states. Thus, \( (s_1 \rightarrow s_2) \), would indicate that the occurrence of the disease state \( s_1 \) at one time, might cause a later occurrence of the disease state \( s_2 \). Both nodes and links in a causal net are weighted
Subjects. The weight of a node is used to determine its state of confirmation, and the weight or a link would represent the strength of causality. These weights are in general vectors. For a given \((s_1 \rightarrow s_2)\) the weights of \(s_1\), \(s_2\) and the link would be required to satisfy prescribed rules of consistency. Thus, for example, if \(s_1\) has the status CONFIRMED, and the strength of the causal link is say 1, then one might require that the status of \(s_2\) should also be CONFIRMED.

We shall refer to the disease model used by the CASNET system as the CAUSALMODEL. To model a disease in CASNET one has to first define the CAUSALMODEL for the disease. The CAUSALMODEL will necessarily include a description of the causal net to be used for the disease. We shall refer to this as the CAUSALNETDEFN. In the MDS formalism we shall say that both the relational forms

\[(\text{CAUSALMODEL} \text{ causalnetdefn} \text{ CAUSALNETDEFN}), \text{ and} \]
\[(\text{CAUSALNETDEFN} \text{ causalnetdefn} \text{ of CAUSALMODEL}), \]

are well defined for the domain of CASNET. For the processors in MDS this would mean,

"For every CAUSALMODEL, the causalnetdefn or the CAUSALMODEL is a CAUSALNETDEFN."

For a given disease its associated CAUSALNETDEFN will specify the states of the disease, the causal links, their weights and constraints. Thus, if a CAUSALMODEL had been defined for GLAUCOMA, then the phrase (GLAUCOMA
causalnetdefn) will produce in MDS the definition of the
CAUSALNETDEFN of GLAUCOMA.

In the CASNET system one may also define
correspondences between the weights of the disease states
and the testable symptoms of a disease process. These
correspondences are then used to construct a causal net
description of a disease process from the set of observed
symptoms of an afflicted patient. In the MDS formalism we
shall indicate this feature by defining the relation forms:

(CAUSALMODEL testdesns TESTDESNS),
(TESTDESNS testdesns of CAUSALMODEL), and
(TESTDESNS elements TESTDESNS).

Here TESTDESNS has been defined to be a collection of
TESTDESNS, an arbitrary number of them. Each TESTDESNS
itself will specify one test and the interpretations for its
results, namely, the correspondences between the test
results and the weights of the disease states. A
CAUSALMODEL may have a collection of such TESTDESNS
associated with it. Again, if the CAUSALMODEL for GLAUCOMA
were available to MDS, the phrase (GLAUCOMA testdesns) would
refer to the collection of all tests defined for the
GLAUCOMA causal model.

The collection of confirmed states in the causal net
description of the disease afflicting a given patient,
together with their associated causal chains, are used in
CASNET to determine the disease diagnosis and possibly also
the necessary therapy. The associations between the causal
net descriptions of a disease process and the disease
diagnosis and therapy are specified in CASNET by
(CAUSALMODEL classifications). In the MDS description of
CASNET we shall introduce the relational forms:

(CAUSALMODEL classifications CLASSDEFS),
(CLASSDEFS classifications of CAUSALMODEL), and
(CLASSDEFS elements CLASSDEF).

Once again, a CAUSALMODEL may have a collection of
CLASSDEFS associated with it. Each CLASSDEF will
introduce a rule of classification associated with a given
disease domain.

The disease states, the causal links, their
associations with disease symptoms, and the correspondences
between an instance of a causal net and the diagnosis and
therapy of the disease represented by the causal net, would
all depend, of course, on the nature of the disease being
modelled. In the conception of the CASNET system there are
two phases of operation: The first is the domain
definition phase. In this phase the CAUSALMODEL for a
disease domain would be defined. The second phase is the
domain execution phase. In this phase, the CAUSALMODEL
definition would be used to generate a causal net
description of an instance of the disease in an afflicted
patient. In the implemented version this is not however
quite true. Besides defining the CAUSALMODEL for a disease
domain a user is required to also write a few domain
specific programs in order to be able to apply the defined CAUSALMODEL to generate causal net descriptions of instances of the disease. These domain specific programs may vary widely from one disease domain to another. Thus, in the case of the CASNET defined for GLAUCOMA the disease is defined only for one eye. Special routines had to be written to consider the disease affliction for both eyes simultaneously. The version described here is close to (**), the implemented version, described in Weiss [1974] (see section 6 for a discussion of what has been left out).

In the MDS description the domain definition process would manifest itself in the following manner: One would first create the causal model definition for a disease domain by creating an instance of the object, called CAUSALMODEL. If the disease domain is, for example, GLAUCOMA, then one may name this instance of CAUSALMODEL, by the name GLAUCOMA. The causal model, GLAUCOMA, would have associated with it, its own specific instances of CAUSALNETDEFN, TESTDESNS, and CLASSDEFNS. Let (GLAUCOMA causalnetdefn) be called, GCASNET. Similarly, let (GLAUCOMA testdesns) be (q1, q2, ..., qN), collectively referred to by GTESTS, and (GLAUCOMA classifications) be GCLASSES = (c1, c2, ..., cM). The domain definition for GLAUCOMA will thus consist of GCASNET, GTESTS and GCLASSES by applying the

** It should be pointed out that it would be relatively easy to modify the MDS description of the CASNET, given here, to automatically take care of disease manifestations in more than one organ of an afflicted organism.
CASNET, GTESTS and GCASSES to particular instances of the JLAUCOMA disease in given patients, one may now create descriptions of the disease process in the patients concerned. We shall refer to these disease descriptions as DISEASEDESNS. To take care of the generation of these DISEASEDESNS we shall introduce the following additional relations to the CAUSALMODEL,

(CAUSALMODEL diseasedesns DISEASEDESNS),
(DISEASEDESNS causalmodel CAUSALMODEL), and
(DISEASEDESNS elements DISEASEDESNS).

Now, every CAUSALMODEL may have a collection of DISEASEDESNS associated with it. The definition of DISEASEDESNS will specify how the (CAUSALMODEL causalnetdefn), (CAUSALMODEL testdesns) and (CAUSALMODEL classifications) would be used to create instances of DISEASEDESNS.

In this preamble we have already introduced the central properties of the CASNET system and the forms their descriptions will take in MDS. The objects like CAUSALMODEL, CAUSALNETDEFN, TESTDESNS, etc., that were introduced above, are called TEMPLATES. The CAUSALMODEL template has four relations defined for it: causalnetdefn, testdesns, classifications and diseasedesns. As with the CAUSALMODEL the structure and components of a CAUSALNETDEFN will be specified by the template for CAUSALNETDEFN. Similar considerations hold for the other relations defined for the CAUSALMODEL.
The formal definition of the CAUSALMODEL template will appear in MDS as shown below (The prefix "TDN:"--Template Definition-- in the description below is the MDS command that is used to define templates):

```
[ TDN: CAUSALMODEL
  (causalnetdefn (IT CAUSALNETDEFN)
  causalnetdefnof)
  (testdesns TESTDESNS testdesnsof)
  (classifications CLASSDEFNS classificationsof)
  (diseasedesns DISEASEDESNS causalmodel)].
```

```
[ TDN: TESTDESNS (ELEM TESTDESN) ]
[ TDN: CLASSDEFNS (ELEM CLASSDEFN) ].
```

Here "(IT CAUSALNETDEFN)" specifies that for each CAUSALMODEL a new instance of CAUSALNETDEFN ought to be created. "IT" is the Instantiate Template command of the MDS system. The word "ELEM" is used to denote "elements". In MDS, the templates with ELEM relation are used to define collections of objects. We shall refer to a template with ELEM relation as a LIST TEMPLATE. Thus, TESTDESNS and CLASSDEFNS above are list templates. A template like, CAUSALMODEL (that does not contain the ELEM relation) is called a NODE TEMPLATE. Instances of NODE templates are individual objects, whereas the instances of LIST templates are collections. We shall later see the use of flags with templates and relations to classify templates and identify variations in their interpretations. The templates given above specify the DESCRIPTION STRUCTURE of the objects they define. In the description structure of CAUSALMODEL, (CAUSALMODEL causalnetdefn) is said to be a DIMENSIONALY
CONSISTENT relational form. The template called by (CAUSALMODEL causalnetdefn) is CAUSALNETDEFN. In the case of (CAUSALMODEL testdesns) the called template is TESTDESNS, which is a collection of TESTDESN. In situations like this, we shall say that both TESTDESNS and TESTDESN are the called templates of the anchor. We shall refer to dimensionally consistent relational forms like (CAUSALMODEL causalnetdefn) as the ANCHORS of CASNET. The description of an instance of CAUSALMODEL will be said to be complete only when all its four anchors have been instantiated. Generally speaking, the instantiation of an anchor might call for an instance of the template called by the anchor. Thus, the instantiation of (CAUSALMODEL causalnetdefn) might call for an instance of CAUSALNETDEFN. It is, of course, possible that not any instance of CAUSALNETDEFN would be satisfactory. To take care of situations like this, one may associate with the anchor (CAUSALMODEL causalnetdefn) a CONSTRAINT. For an instance of CAUSALMODEL, say X, an instance of CAUSALNETDEFN, say Y, would be accepted as the causalnetdefnof X, only if Y satisfied the constraint of the anchor (CAUSALMODEL causalnetdefn). One may think of the constraint associated with an anchor as defining the semantics of the relation in the anchor. These constraints are specified in MDS in first order logic. We shall refer to these constraints variously as CONSISTENCY CONDITIONS (CC's), or SENSE DEFINITIONS. We shall later discuss in more detail the forms and interpretations of these anchored
Thus, **TEMPLATES** and their associated anchored **CONSTRAINTS** are used in MDS to describe the objects in a domain. The description of the CASNET domain in MDS will consist of a whole lot of different **TEMPLATES** (like, **CAUSALMODEL**, **CAUSALNDEFN**, **TESTDESNS**, **TESIDESN**, **CLASSDEFNS**, **CLASSDEFN**, **STATEDESNS**, **CAUSEDESNS**, **STRATEGIES**, **LIKELIHOOD**, etc.) and constraints associated with their various anchors. These templates and constraints are shown in appendix A. Descriptions of this kind become interesting if the system could use the templates and constraints automatically to create valid instances of the objects so described. Thus, from the template for **CAUSALMODEL** and all its component templates (and components thereof), we would like to be able to create a complete instance of **CAUSALMODEL**. In this instantiation process the system should produce wherever possible the necessary instances of needed objects automatically, and where appropriate it should acquire the necessary descriptions or needed objects from the user (the model builder). If the only information available consisted of templates and constraints, then this kind of instantiation process is in general very difficult to do. No (efficient) general methods exist for constraint satisfaction problems of this type. In MDS automatic instantiations of templates and anchors are made possible by providing additional information to the system in the form of control procedures, called **TRANSFORMATION RULES** (TR's).
we shall later discuss the forms and interpretations of TR's. At this point it suffices to mention that the forms of both the CC's and TR's are such that the processors in MDS can not only evaluate them, but also understand what they do. Thus, if a CONSTRAINT at an anchor is not satisfied by a given object, then MDS would know the reason, way. It may use this reason for the failure, to search for a more suitable candidate. If a transformation rule is invoked to construct a new object or complete an updating process, the MDS can generate from the transformation rule a description of what it expects to accomplish by the application of the rule, and how the rule goes about its process. These two features of MDS are the most significant ones that enable it to act as a general problem solving system that can do non trivial problem solving tasks, by making use of the domain KNOWLEDGE that has been described to it. We may now briefly summarize the concepts introduced so far as follows:

The concept of TEMPLATES and the process of their instantiation are central to the operation of MDS. Each TEMPLATE in a domain is associated with a CLASS of objects in the domain. Thus, in CASNET there is a class of objects, called TESTDESNS. A TEMPLATE is expected to contain in it all the specifications necessary to create instances of the class of objects that it defines. Instances of TESTDESNS will be descriptions of tests that may be applied for a given disease domain. Thus, in GTESIS (tests for GLAUCOMA)
(qt1 qt2 ... qtN), each qt will be an instance of \textit{STATEDESIGN}, and will describe a particular TEST that might be applied to diagnose GLAUCMA.

The information provided by a template may have three parts to it: The \textbf{STRUCTURAL INFORMATION}, specifying what objects in the domain may relate to what other objects and by what relation; \textbf{SENSE INFORMATION} (the \textit{CONSISTENCY CONDITIONS}) specifying constraints on anchors; and \textbf{TRANSFORMATION RULES} specifying the procedures to be used to create instances (or updates of instances) of templates and anchors satisfying all the specified constraints. Thus, the \textbf{CAUSALMODEL} template, together with all the other templates that it calls (and others that the called templates themselves (recursively) call), would contain the information necessary to create instances of \textbf{CAUSALMODEL}s.

In the descriptions above we have shown only the specifications of the \textbf{structural components} of \textbf{CAUSALMODEL}.

An example of anchor definition in a template, with indications of associated CC and TR, occurs in \textbf{(STATEDESIGN presence)} (See Appendix A):

\textbf{(presence (PRESENCE TA) presenceof CCLJ Tk2)}.

In this definition TA is a flag associated with the \textbf{PRESENCE} template, that declares \textit{PRESENCE} to be a "Terminal Atom" template: Every instance of \textit{PRESENCE} should be a terminal atom. For our purposes we shall just create two instances of \textit{PRESENCE}, one called \textit{CONFIRMED} and the other called
Thus, one may say that for an instance of \texttt{STATEDESIGN}, called say \texttt{S}, the value of (\texttt{S presence}) can be either \texttt{CONFIRMED} or \texttt{DENIED} (*). Both \texttt{CONFIRMED} and \texttt{DENIED} are, of course, atoms (in the LISP sense). The labels \texttt{CC13} and \texttt{TR2} respectively refer to the \texttt{CONSTRAINT} and \texttt{FP} anchored at (\texttt{STATEDESIGN} presence). The \texttt{CC} and \texttt{TR} themselves are defined separately, by using the \texttt{QSCC}: (Set \texttt{CC}, the prefix \texttt{q} indicates that this command is part of the knowledge acquisition system of \texttt{MDS}, called \texttt{QUEST}), and \texttt{QSTR}: (Set \texttt{Ta}) commands, as shown in Appendix A. \texttt{CC13} specifies that the presence of a state should be \texttt{CONFIRMED} if its status exceeds a certain threshold, it should be \texttt{DENIED}, if the status is less than the threshold. In the beginning the presence of a state might have the value \texttt{UNKNOWN} (denoted by \texttt{?}, in \texttt{MDS}). If the status of the state changes during the course of an experiment, then its presence will be automatically set in accordance with \texttt{CC13}. However, in certain situations a contradiction might arise: Suppose the presence of the state was to begin with \texttt{CONFIRMED}, and later...

* This is not, however, strictly true. Because of a convention in \texttt{MDS} (to be explained later), the \texttt{CC13} applies to an instance of an instance of \texttt{STATEDESIGN}. An instance of \texttt{STATEDESIGN} would itself be a template. This is indicated by the flag "MN" (Meta Node) associated with \texttt{STATEDESIGN} template (see Appendix A). This flag indicates that every instance of \texttt{STATEDESIGN} should be itself a \texttt{NODE TEMPLATE}. To create a model for a disease domain, \texttt{D}, one may create several instances of \texttt{STATEDESIGN}. These would then be the disease state templates for the domain, \texttt{D}. Instances of these disease state templates will occur in the causal net description of a particular occurrence of the disease, \texttt{D}, in a patient. \texttt{CC13} would then apply to the states occurring in the causal net description of this particular patient's disease.
During the course of a diagnostic experiment the status of the state changed as a result of new evidence, and went below the threshold. This would then cause, a CONTRADICTION, in MDS. The new situation contradicts the existing situation. To resolve such a contradiction, the @R2 associated with the anchor will be invoked. We shall later discuss the operations performed by @R2. One may notice that the CASNET description has only about six transformation rules in all. Also, each Tr-rule is small, and is intended to do specific remedial tasks in well constrained situations. The major repository of CASNET knowledge resides in TEMPLATES and SENSE DEFINITIONS. The general problem solving processors built into MDS enable MDS to accept descriptions of this kind and use them for domain execution, via the instantiation process. The facilities in MDS are quite general; they are not necessarily restricted to the kinds of tasks encountered in CASNET. At the moment we are using MDS also to describe the BELIEVE system of Schmidt and Sridharan [Sridharan 1975]. MDS is a META system in the sense that it can accept descriptions of KNOWLEDGE in any domain and specialize itself to act efficiently in the domain. A fairly detailed exposition of the various facilities in MDS is available in [Srinivasan 1975a]. A formal description of the TEMPLATE definition system, and the various types of TEMPLATES that one may use, are discussed in [Srinivasan 1975b].
After introducing the description structures of the objects in CASNET (i.e. the structure of all the template in CASNET) we shall briefly discuss the forms and uses of various types of templates, the way relation flags are used in MDS to specify variations in the interpretation of anchors, and the forms and interpretations of CC's and TR's in CASNET. These will enable the reader to follow the formal description in Appendix A more closely. We shall follow this with detailed explanations of selected portions of CASNET, with illustrations of their use.

It should be pointed out that the MDS system as described here is not yet fully operational. Only the DOMAIN DEFINITION part of MDS and a small part of the DOMAIN EXECUTION system (the system that interprets templates, instantiates them and evaluates the CC's and TR's) are not operational. The development is not yet at a stage where it can execute the CASNET description, presented here. The purpose of this report is to introduce the MDS facilities to investigators in the area of medical modelling, in the Rutgers Resource.

3.0: THE DESCRIPTION STRUCTURE OF CASNET.

We have already discussed the structural description of CAUSALMODEL shown in figure 1, below. In this section, we shall present the structures of the components of
CAUSALMODEL, that are used in CASNET description. The
depicted structures presented here are intended to convey a general
understanding for what a CASNET is. These structures have
been taken directly from the templates defined for CASNET.
The branches in the diagrams below are labelled with the
relation names used in the templates, and have indications
there on, for the CC's and TR's associated with the various
relations. The two letter labels within [...] appearing as
suffixes of template names, indicate the types of the
respective templates. These are explained in greater detail
in section 4. For the moment the reader should keep in mind
the following:

MN: Every instance is itself a NODE template.
PN: NODE template. Every instance should have a name.
TI: Every instance is an INTEGER.
T#: Every instance is a NUMBER.
TA: Every instance is a LISP ATOM.
TS: Every instance is a STRING.

The structural view presented here is precisely the view
available to the system from the templates.

In the templates shown in Appendix A, the relations at
the various anchors have at times certain flags associated
with them. These flags are not shown in the structures
shown in this section. In discussing informally the
interpretations of the structures, we shall sometimes refer
to the flags associated with the relation in Appendix A. A
detailed discussion of the use of relation flags appears in
section 4.2.
The structure of CAUSALNETDEFN is shown in Figure 2. Instances of CAUSALNETDEFN need not have names associated with them (it is a $\text{SN}$ template). It has seven components. Notice that one of its components is the CAUSALMODEL. Thus, for each CAUSALMODEL there is exactly one CAUSALNETDEFN and for each CAUSALNETDEFN there is exactly one CAUSALMODEL.

The common threshold defines the threshold that is common to all instances of STATEDESN. This common threshold will be assumed to be the threshold of a $<$STATE$>$ (an instance of STATEDESN), unless a separate threshold had been declared for the state. This condition is specified by $\text{SS67}$, which is the constraint associated with the anchor, (STATEDESN threshold), appearing in figure 2.
FIGURE 2: Structure of CAUSALNET[PLAN]. <STATE> is an instance of STATEDESN.
Principally, each CAUSALNETDEFN contains a collection of STATEDESNS, and CAUSEDESNS. Each CAUSEDESN has a state, and a transition probability. Each CAUSEDESN, \( x \), is in fact, the description of a link of the form shown below:

\[-(x \text{ transitionprob}) \rightarrow (x \text{ state})\]

When an instance of CAUSALNETDEFN is created, the system will prompt the user for supplying its STATEDESNS. The need for this prompting is indicated by associating the prompt flag, \( ! \), with the STATEDESNS relation (we shall discuss the various relation flags and their uses in section 4.2). The starting states, terminal states, etc., of a CAUSALNETDEFN are defined by CC1, CC2, etc. Once the STATEDESNS are defined, then the system will automatically find and assign the starting, terminal and interior states of the CAUSALNETDEFN using the constraint definitions CC1, CC2 and CC3 respectively, if and when they are needed. Similarly, it will use CC8, anchored at (STATEDESNS causedesns), to find the cause descriptions from the definitions of the state descriptions.

The constraint CC4, for designated states, is such that the system can use it only to check whether a given <STATE> is an appropriate candidate for being a designated state; if no candidate state is given then CC4 cannot, by itself, find the appropriate candidates (these features are discussed in greater detail in section 4.3). To find the designated states of a CAUSALNETDEFN, the system will therefore prompt the model builder.
Each STATEDESIGN has eight components. For each instance of STATEDESIGN, called say <STATE>, each of these eight components will be instantiated. The relations, starting weight, causes, and threshold have prompt flags associated with them. Therefore, for an instance of STATEDESIGN, MDS will prompt the user for the values of these relations, which should be respectively, instances of NUMBER, CAUSEDDESNS, and INTEGRER. The descendents of a STATEDESIGN is an instance of STATEDESNS. This will be computed by CC8 from the definitions of the "causes" of the STATEDESIGN.

The relations, status, presence, conflict and likelihood of a STATEDESIGN, have the flag "C" (for CONSTANT) associated with them. Because of this flag, the called templates of the anchors containing these relations are treated as constants. Thus, in an instance, <STATE>, of STATEDESIGN, (<STATE> status) will also be the STATUS template, the same template, called by (STATEDESIGN status). Similarly, we will have (<STATE> presence PRESENCE), (<STATE> conflict CONFLICT) and (<STATE> likelihood LIKELIHOOD).

<STATE> itself will be a template, since STATEDESIGN is a template. In a further instantiation of <STATE>, the anchors of <STATE> that can be again instantiated will be those, whose called objects are templates. In the case of <STATE> the relations, "starting weight", "causes" and "threshold", will call respectively, a number (instance of PROB), an instance of CAUSEDESNS (which will be a collection),
and an integer (instance of THRESHOLD). These called objects are not templates. Therefore, in a further instance of <STATE>, say x, these relations cannot be again instantiated. In situations like this, for an instance of STATEDESIGN, like <STATE>, the values of the relations (<STATE> startingweight), (<STATE> causes), and (<STATE> threshold) will have the following significance:

For all instances, x, of <STATE>, (x startingweight), (x causes) and (x threshold) will be the same as (<STATE> startingweight), (<STATE> causes) and (<STATE> threshold), respectively.

Thus, if <STATE1> and <STATE2> are two instances of STATEDESIGN, these two may have, for example, different startingweights. However, all instances of <STATE1> will have the same startingweight as <STATE1> itself, and similarly with <STATE2>. Now, for both <STATE1> and <STATE2>, it will be true that (<STATE1> status is STATUS) and (<STATE2> status is STATUS). "STATUS" here is a template. Therefore, instances of <STATE1> will have as their status, instances of STATUS. Thus, different instances of <STATE1> may have different statuses. In fact, for instances of a <STATE> the only instantiated relations will be: status, presence, conflict and likelihood. Different instances of a <STATE> may vary in the values they have for these relations.

The remaining structures, the structures of LIKELIHOOD and CONDPROB, shown in Figure 2, are self explanatory. We shall have more to say about the interpretation of these
Structures later in this report. The structures for TESTDESNS, CLASSDEFNS and DISEASEDESNS are shown in figure 3, 4 and 5 respectively. The reader is invited to scan through these before proceeding with the rest of the paper.

For an instance of TESTDESNS, say <TEST>, the anchors for the relation names, summaryquestion, repeatability, cost, confidence, negative determinacy, counter, effects, nexttest, firsttest, negative nexttest, and testtype would all be instantiated to constants (items that are not templates, or items which are templates but which are to be treated as

```
TESTDESNS[ $L ]
|   strategies --> STRATEGIES[$L] --> elements --> STRATEGY[ MN ]
|   INFLUENCE[ MN ] <-- influences -->

elements

TESTDESNS[ MN ]
|   components, CC21 --------- --> TESTDESNS[ $L ]
|   summary question, CC23 --> QUESTION[ TS ]
|   repeatability, CC24 --> YESNO[ TA ]
|   cost, CC25 --> COST[ T# ]
|   confidence, CC26 --> CONFIDENCE[ TI ]
|   negative determinacy, CC27 --> YESNO[ TA ]
|   counter, CC28 --> COUNTER[ TI ]
|   effects, CC29 --> EFFECTS[ $L ] --> elements
|   first test, CC30 --> TESTDESNS[ MN ]
|   next test, CC31 --> TESTDESNS[ MN ]
|   test type, CC32 --> TESTTYPE[ TA ]
|   cost ratio, CC33 --> COSTRATIO[ T# ]
|   test result, CC34, TR1 --> YESNO[ TA ]
|   state
|   application --> APPLICATION[ S ]

STATEDESNS[ MN ]
|   candidate states, CC50 --> STATE>
|   candidate tests, CC51 --> TEST>
|   next choice, CC52 --> TEST>
|   application of, CC65 --> TLSI>
```
FIGURE 3: The structure of TESTDESNS. <TEST> is an instance of TESTDESN.

Thus, when a <TEST> is instantiated, as part of the process of building a DISEASEDESN, only the anchors, (<TEST> costratio), (<TEST> application) and (<TEST> testresult) will be instantiated. In the process of instantiating the testresult the appropriate question for the test will be asked by a function, called ASKQUESTION. The result of the test will then interact with the affected states, and cause a whole chain of changes to take place in the current model of the disease, consistent with the semantics of the pertinent CC's. These changes will be completely guided by the CC's involved, with the TR's providing some of the crucial control structures. We shall examine this process more carefully in section 5.

CLASSDEFNS[$L$]--elements--->CLASSDEFN[$N$]

| -classname,CC15-->CLASSNAME[TA] |
| -firstentry-------ENTRYDEFN[$N$] |

STATES[$MN$]<--CC70,entrystate-|
STATES[$SL$]<--CC17,descendants-|
COMMENTS[$SL$]<--comments-|
ENTRYDEFN[$SN$]<--nextentry-|
ENTRYDEFN[$SN$]<--CC10,lowerentries-|

COMMENTS[$SL$]--elements--->COMMENT[$SN$]

| -diagnosis-->STATEMENT[TS] |
| -therapy------STATEMENT[TS]. |

FIGURE 4: The structure of CLASSDEFNS.
Each CLASSDEFN has a class type, which is a CLASSNAME, and an ENTRYDEFN, which is its first entry. The system will prompt for both of these when a CLASSDEFN is instantiated. Each ENTRYDEFN has an entry state, a collection of descendents (which would be the entry states of the lower entries of the ENTRYDEFN). The comments associated with an ENTRYDEFN would specify the diagnosis and therapy for the entry. The next entry of an ENTRYDEFN is again an ENTRYDEFN. The lower entries are the collection of ENTRYDEFNS, under the closure of the next entry relationship (lower entries is a transitive relationship). It is also reflexive. Hence an ENTRYDEFN is by definition, its own lower entry.

\[\text{DISEASEDESIGN[SN]} \leftarrow \text{elements} \rightarrow \text{DISEASEDESNS[SL]}\]

\[\text{causalnetof} \rightarrow \text{CAUSALMCDEL[HN]} \]
\[\text{date} 
\rightarrow \text{(DATE) } \rightarrow \text{SEX[TA]} \]
\[\text{diseaseof} \rightarrow \text{PERSON[RN]} 
\rightarrow \text{years} \rightarrow \text{YEARS[TI]} \]
\[\text{diagnosis/therapy,CC37} 
\rightarrow \text{COMMENT[SL]} \]
\[\text{topoleveltest} 
\rightarrow \text{TOPLEVELTEST[SN]} \]
\[\text{causalnet} 
\rightarrow \text{CAUSALNET[SN]} \]

\[\text{TSTDESNS[SL]} \leftarrow \text{CC39,currenttests} 
\leftarrow \text{TEST}s \leftarrow \text{CC103,selectedtests} \]
\[\text{TOPLEVELTEST} \leftarrow \text{CC101,descendants} \]
\[\text{TOPLEVELTEST} \leftarrow \text{CC102,ancestors} \]
\[\text{TOPLEVELTEST} \leftarrow \text{CC5,nexttest} \]

\[\text{causalnetof} \rightarrow \text{DISEASEDESIGN[SN]} \]
\[\text{states,CC40} 
\rightarrow \text{STATE}s \]
\[\text{causes,CC41} 
\rightarrow \text{CAUSE}s \]
\[\text{tests,CC10} \rightarrow \text{TST}s \]
\[\text{startingstates,CC42} 
\rightarrow \text{STATE}s \]
\[\text{terminalstates,CC43} 
\rightarrow \text{STATE}s \]
\[\text{interiorstates,CC44} 
\rightarrow \text{STATE}s \]
\[\text{startingstates,CC45} 
\rightarrow \text{STATE}s \]
\[\text{pathways} 
\rightarrow \text{PATHWAY[SN]} \]

\[\text{STATE} \rightarrow \text{startingstate,CC46} \]
\[\text{STATE} \rightarrow \text{components,CC47} \]
\[\text{PATHWAY} \rightarrow \text{nextpathway,CC48} \]
FIGURE 5: The structure of DISEASEDESBN.

Each DISEASEDESBN has a CAUSALMODEL. It is made on the indicated date, supplied by the (DATE) function. The description is the disease of a PERSON, and has a causal net associated with it. The construction of the DISEASEDESBN will begin with the TOPLEVELTEST, which would then on continue the testing process via its own nexttest, which is again an instance of TOPLEVELTEST. Each time the currenttests of a TOPLEVELTEST is instantiated, a collection of possible currenttests will get selected. From among these the "selectedtest" of the TOPLEVELTEST will be chosen. If there is only one currenttest, then obviously this will get chosen as the selectedtest. If more than one currenttest exist then the system will prompt the user for advice about which, among the equally feasible collection of currenttests, should be applied in the existing context. The selected test (or tests) will get instantiated, and as a result will get applied. The application of these <TEST>s will cause the appropriate changes in the CAUSALNET of the DISEASEDESBN. This process will continue till the nexttest is NIL. At this point the diagnosis/therapy will be instantiated. The associated COMMENTS may be printed out.

An instance of CAUSALNET will have to begin with, a complete set of instances of the <STATE>s and <CAUSE>s of the associated CAUSALNETDEFN. The starting states, terminal states, etc. of this instance of CAUSALNET will be determined during
the course of test applications, by the CC's, CC42, CC43 and CC44. The starting states of a CAUSALNET is the collection or starting states from which the largest number of CONFIRMED states in the causalnet are reachable. The PATHWAYS of a CAUSALNET will be the causal chains in the net which have all their states CONFIRMED, and for which, none of the ancestors of their states are DENIED. These PATHWAYS are used to produce the diagnosis/therapy.

This completes the structural description of the CASNET. To understand the details of the structural description as specified by the templates in Appendix A, it is necessary to know the significance of the various template and relation flags, and also the use of the so called function template. To understand the way these templates are used and interpreted it is necessary to know the meaning and interpretations of the CC's and TR's associated with the various anchors. These are all described in the next section.

4.0: Templates and their instantiations.

4.1: Types of templates and examples of their use in CASNET.

By instantiating a TEMPLATE in MDS, one gets an instance of the object specified by the template. A complete instantiation of a template would call for the instantiation of all its anchors. There are various types of templates in MDS. We have already seen the distinction between the NODE template and the LIST template.
CAUSALMODEL is a NODE template, whereas TESTDESN is a LIST template. For convenience we shall require that every instance of CAUSALMODEL should have a NAME. This is indicated by the flag "RN" (Regular Node) associated with the CAUSALMODEL template. If the naming of instances of a template is to be left optional, then the flag "RN" would be associated with a NODE template. Thus, CAUSALNETDEFN is a $N (Dummy Node) template. Not every instance of CAUSALNETDEFN need have a name. An unnamed instance of CAUSALNETDEFN may be accessed only via the CAUSALMODEL to which it is related to. For example, the CAUSALNETDEFN for GLAUCOMA can be accessed only by the phrase (GLAUCOMA causalnetdefn), if the instance of the CAUSALNETDEFN itself was unnamed. Similarly, we may have also REGULAR LIST (RL) and DUMMY LIST ($L) templates. We shall indicate the type of a template by the flag associated with it.

We saw several examples of TERMINAL templates in section 3. Templates like, THRESHOLD (Terminal Integer), P4OB (Terminal Number), STATUS (Terminal Integer), PRESENCE (Terminal Atom), STATEMENT (Terminal String), etc., are all TERMINAL templates. Instances of TERMINAL templates in MDS would be the primitive data types of the system. In MDS, TEMPLATE itself is one of its primitive data types. It is the only instanceable data type. Thus, we can have TERMINAL TEMPLATE templates, whose instances are themselves required to be templates. One may think of a TERMINAL TEMPLATE as specifying the schema for defining a template.
We have encountered several TERMINAL TEMPLATES, the templates STATEDESN, CAUSEDESN, TESTDESN, etc. are special cases of TERMINAL TEMPLATES. They are all TERMINAL NODE templates. Instances of these are required to be themselves NODE TEMPLATES. They could be either REGULAR NODE or DUMMY NODE templates. Thus, in GLAUCOMA, the instance of TESTDESNs, which is a collection of instances of TESTDESN,

\[ \text{TESTS} = (q_{t1}, q_{t2}, \ldots, q_{tN}) \]

each \( q_t \) would itself be a NODE template. It would define a particular test applicable to GLAUCOMA. Instances of \( q_t \) might be used for the diagnosis of particular occurrences of the disease. In the model establishment process, the various terminal templates (like, TESTDESN, CAUSEDESN, STATEDESN, etc.) are used to create descriptions of \(<\text{TEST}>s\), \(<\text{CAUSAL-LINK}>s\), \(<\text{STATE}>s\), etc., that pertain to a given disease domain. Since these descriptions are themselves templates, they may further be instantiated to generate DISEASEDESNs for particular occurrences of the disease.

The few other templates of interest to us in this paper are the UNIVERSAL TEMPLATE, **, and FUNCTION TEMPLATES. Anything can be instance of a UNIVERSAL template. Thus, a CAUSALMODEL, CAUSALNETDEFN, STATEDESN, etc., may all be viewed as being instances of **. We shall see uses of ** in CASNET description. A function template is used to define functions about which MDS would know some properties. The principal properties of a function defined in a function
template are: What the arguments of a function can be, what
its result is and how the arguments and the result relate.
The process of instantiation of a function template is the
same as the evaluation of the function for given arguments.
The result of the instantiation would be the result returned
by the function. As an example of a function template
consider,

[TDN: ADD (FNDEF NUMBER NUMBER NUMBER),

which is the definition of the function template, called
ADD. It has two arguments, both of which are NUMBERS, and
has a result which is also a NUMBER. The last item in the
FNDEF of a function template is always the definition of the
result of the function. In the case of ADD no constraints
have been defined relating the result of the function to its
arguments. This relationship will be established by the
procedure for ADD associated with the ADD function. In this
case MDS itself would not know anything about this
procedure. We have another example of a function template
in SUM defined below:

[TDN: SUM (FNDEF (LISTDT CC99) (** CC60))].

In this case the argument of SUM is an instance of LISTDT (a
LIST data type—LISP LIST—to be contrasted with a LIST
template), that satisfies the constraint CC99. This
constraint specifies that the elements of the LISTDT should
d all be instances of either INTEGER, or NUMBER, or a template
whose template flag is TI or T#. The result can be anything (instance of **) that satisfies the constraint CC60. CC60 specifies that the result is a NUMBER, INTEGER, or an instance of a TI or T# template depending on the elements of the LISTDT argument. If there exists a TI or T# template X, such that all the elements of LISTDT are instances of X, then the result is also an instance of X. If there exists an instance of NUMBER in the LISTDT then the result is a number. Otherwise, the result is an INTEGER. MDS has more information about SUM than it has about ADD. One may use these function templates in the definitions of CC's and Tn's, and may also use them as the called templates in anchors. The function IT (Instantiate Template) is often used as the called template in CASNET description. Thus, in the STATEDES template (see Appendix A), the called template for (STATEDES likelihood) is (IT LIKELIHOOD). This indicates that for an instance of STATEDES called, say STATE1, (STATE1 likelihood) should be a new instance of the LIKELIHOOD template created by (IT LIKELIHOOD). The function template, IT itself has the definition:

```
[TDN: IT (FNDEF (TEMPLATE TEMPLATES) (NAME NAMES)
       (MDNH MDNHS))]
```

The first argument (arg1) of IT can be either a TEMPLATE or TEMPLATES (collection of TEMPLATES). The second one can be, similarly a NAME or a collection of NAMES. The result is a data type called MDNH (Model Definition Header, a pointer), or a collection of MDNH's. Every instance of a
template in MDS will have a model definition header associated with it. The instantiated model will have the name given by the NAME argument. If no NAME is given, or the name NIL is given, then the instantiated model will have no name.

When using a function template as the called template one can use a CC to bind the argument for the particular function call of the function. Thus, the called template for the anchor (APPLICATION nextchoice) in Appendix A, has been defined to be (IT (? CC64)). The argument (? CC64) is to be read as "anything that satisfies CC64", where the constraint CC64 actually specifies the algorithm for the selection of the test for the nextchoice in a diagnosis process.

The various types of templates introduced so far are summarized in Figure 6 below (not all the types of templates available in MDS are shown here).
LIST templates have a special interpretation in MDS. For the CAUSALMODEL, GIAGUCCMA, as we have seen before we have \( \text{GLAUCOMA testdesns} = (gt_1 \, gt_2 \ldots \, gt_N) \). We shall say that \( \text{GLAUCOMA testdesns} (gt_1 \, gt_2 \ldots \, gt_N) \) is true in the data base of MDS. The relation \( \text{GLAUCOMA testdesns} (gt_1 \, gt_2 \ldots \, gt_N) \) is interpreted in MDS as signifying that all the relations \( \text{GLAUCOMA testdesn} \, gt_1 \), \( \text{GLAUCOMA testdesn} \, gt_2 \), \ldots and \( \text{GLAUCOMA testdesn} \, gt_N \) are true. We shall interchangeably use the singular and plural forms of a relation, as in "testdesns" and "testdesn", above). Thus, by convention, relations normally distribute over collections.

### 4.2: The relation flags and their uses

In each anchor of the form \((X \, r)\), where \(X\) is a complex and \(r\) is a relation name, one may associate relation flags to indicate a variety of variations on the interpretation of the relation, \(r\), in the anchor. Only some of those are discussed below. For a more complete description of the relation flags and their uses the reader is referred to [Srinivasan 1975b].

**[Flag !]: The PROMPTING flag (*).**

A flag often used in the CASNET description is the PROMPTING flag, \(!\). For example, we have in the CAUSALMODEL template, the relation definition,

\[
((\text{testdesns} !) \, (\text{TESTDES} \, \&L) \, \text{testdesns})
\]

[See Appendix A]. The exclamation mark is here associated with the relation "testdesns". This has the following
**Significance:**

When an instance of CAUSALMODEL is created, the system would attempt to instantiate every anchor in the CAUSALMODEL template that has the prompting flag associated with it. If a consistency condition (CC), or a transformation rule (TR), is associated with the anchor, and if it were possible to find the called object of the anchor by evaluating the CC and/or TR, then the system would find (or create) the appropriate called object and instantiate the anchor. If no CC or TR is available, or if an appropriate called object for the anchor cannot be found by evaluating the CC or TR, then the system would prompt the user for supplying the appropriate called object for instantiating the anchor.

Thus, in the case of the (CAUSALMODEL testcases), since there is no CC or TR associated with it, the system would prompt the user to supply the appropriate instance of TESTDESNS to be incorporated in the instance of the anchor, (CAUSALMODEL testcases). As the reader may notice, the same prompting convention holds also for the relations "causalnetuefn" and "classifications", in the CAUSALMODEL template. The use of this prompting flag enables one to create a complete CAUSALMODEL for a disease domain by just issuing the command:

```
[IT CAUSALMODEL DISEASE-NAME].
```

The presence of the prompting flag at the various anchors in the templates describing CASNET would then initiate a whole series of enquiries to the model builder to complete the instantiation of the CAUSALMODEL, and the instances of the

---

* The use of the PROMPTING flag convention was suggested by S.S. Sridharan.
templates called by CAUSALMODEL. Thus, the prompting for the anchor (CAUSALMODEL causalnetdefn) would necessitate the instantiation of CAUSALNETDEFN, which would then initiate further promptings to get its own instantiation completed. These prompting will propagate in "depth first" manner through the network of templates beginning at the CAUSALMODEL. An example of the model building process using NDS is discussed in section 5.1.

[Flag D]: DEPTH Flag.

Normally, when an anchor is instantiated, like for example the anchor (LIKELIHOOD inverseweight), for an instance x, of LIKELIHOOD, it would call for an appropriate instance of the called template of the anchor to be supplied. In our case here, an instance of CONDPROB. Let y be the instance of CONDPROB assigned to (x inverseweight). It is, of course, possible that the instance y of CONDPROB, might itself be not completely instantiated--i.e., some of the relations in CONDPROB might not have been instantiated for y. If the anchor (LIKELIHOOD inverseweight) had, however, the D flag (indeed, it does), then the system would automatically attempt to complete the instance y for all the relations defined in CONDPROB. This process of completing the instances would proceed to arbitrary depth, until terminal objects (primitive data types) are reached.

In the case of CONDPROB (see figure 2 in section 3) the anchor (CONDPROB nextprob) calls again CONDPROB. Thus, in this case the D flag would cause a whole series of instances of CONDPROB to be created, until the process is terminated by the CC, CC92, associated with (CONDPROB nextprob). Unless the D flag is carefully used one can get into loops.

[Flag V]: The VARIABLE relation flag.
An example of this flag occurs in the anchor (TESTDESN confidence). The V flag indicates that the relation "confidence" is a variable relation in the TESTDESN template. Not every instance of TESTDESN would have the "confidence" relation defined for it. The CC at the anchor (TESTDESN confidence), CC26, would specify the conditions under which an instance of TESTDESN would have the "confidence" relation defined for it.

[Flag  $]$: The DUMMY Flag.

An example of this occurs at the anchor (TESTDESN costratio). The $ flag associated with this flag indicates that the instances of this anchor are not stored in the database. Every time the costratio of an instance of TESTDESN is required it would be computed by the system using the CC, CC33, associated with the anchor (TESTDESN costratio).

[Flag C]: The CONSTANT Flag (*).

Normally, if the called template of an anchor (X r), is Y, then in an instance, x, of X, (x r) will have as value, an instance, y, of Y. However, if the anchor (X r) has the constant flag, C, associated with it, then in the instance x, the value of (x r) would be simply Y. An example of this occurs at the anchor (STATEDESN status). The called template for this is STATUS. For an instance, <STATE>, of STATEDESN, (<STATE> status) will also be STATUS. Because of the constant flag, one level of instantiation is skipped. Also, the CC and TR associated with (STATEDESN status) will be moved to the level of <STATE>. Thus, the definition of (<STATE> status) would appear as:

(status (STATUS TI) statusof CC11),

where CC11 is the CC associated with (STATEDESN status).

[Flag >]: The INDIRECT Flag.

The indirect flag is used only in TERMINAL TEMPLATEs. An example of this flag occurs in the STATEDESN template in Appendix A, at the anchor (STATEDESN likelihood). The definition of this relation occurs in the STATEDESN template as follows:

----------

* The use of CONSTANT flag was suggested by Joel Irwin.
\[ (((\text{likelihood} \ C>1)) \ \text{(IT LIKELIHOOD NIL)) likelihoodof}) \]

For an instance of \text{STATADESN}, say \text{STATE1}, \text{(STATE1 likelihood)} will be also \text{(IT LIKELIHOOD) (IT is the Instantiate Template command)}. This is because of the \text{C} flag (see \text{Flag C} convention above). The ">" flag indicates that the flag immediately following it should be attached to the instantiated anchor. Thus, in our case \text{(STATE1 likelihood)} will acquire the flag \text{!}, or in other words the likelihood relation definition in the \text{STATE1} template will look like:

\[ (((\text{likelihood}) \ \text{(IT LIKELIHOOD NIL)) likelihoodof}) \]

This would imply that, when \text{STATE1} itself is instantiated, a prompting would occur for the anchor \text{(STATE1 likelihood)}.

[Flag X]: The \text{TRANSITIVITY} flag.

This flag is used to indicate that the relation at an anchor is transitive. An example of this occurs at \text{STATADESN descendants}). The \text{CC} at this anchor, \text{CC0}, specifies that the descendants of a \text{STATADESN}, \text{STATE1}, are precisely the states that are causing \text{STATE1}. However, since the descendants relation is transitive, everytime \text{(STATE1 descendants)} is asked the system will return the transitive closure of what is stored in the data base.

[Flag R]: The \text{REFLEXIVE} flag.

This specifies that the relation at an anchor is reflexive. An example of this occurs at the anchor \text{(ENTRYDEPN lowerentries}). For every instance, \text{E}, of \text{ENTRYDEPN}, \text{(E lowerentries E)} is true.

This completes the discussion of all the relation flags used in CASNET description.

4.3: The \text{CONSISTENCY CONDITIONS}.

4.3.1: The form of \text{CC}’s.
As we have seen before, every anchor may have a CONSTRAINT (CC) and a TRANSFORMATION RULE (TR) associated with it. Conversely, it is true that every CC has an unique anchor associated with it. This is not, however, true for TR's. There can be so called "floating TR's" which are not attached to any anchor. Floating TR's have not been used in the CASNET description. The CC's are stated as expressions defining sets (collections of objects). Let $(X \ r)$ be an arbitrary anchor. Let $Y$ be the called template of $(X \ r)$. Then the CC at $(X \ r)$, $CC[(X \ r)]$, will have the form:

$$\{ (Y \ y) \mid P(\omega \ y) \}.$$ 

This is to be read as: "The collection of all instances, $y$, of the template, $Y$, such that the PREDICATE, $P(\omega \ y)$ is satisfied". Here, the distinguished symbol, $\omega$, refers to the CURRENT INSTANCE of the template, $X$, the template of the anchor of the CC. The CC is evaluated at $\omega$. The predicate in a CC will always have two FREE VARIABLES. We shall refer to $\omega$ as the ANCHOR VARIABLE (or simply the ANCHOR) of the CC, and $y$ as the SET VARIABLE of the CC. The predicate itself is a first order expression using function symbols, relative quantifiers, and logical connectives. We shall refer to the predicate $P(\omega \ y)$ as the SET PREDICATE of the CC. Let us consider a simple example.

Consider the anchor (CAUSALNETDEFSN startingstates). This anchor calls the template STATEDESNS. Thus, for GLAUCOMA, the starting states of its CAUSALNETDEFSN, will be a
an instance of STATEDESN, i.e. a collection of instances of STATEDESN. The CONSTRAINT, CC1, is associated with this anchor. CC1 states,

\[(\text{STATEDESN } S) \land (\text{a statedesn } S) (S \text{ causedby NIL})\].

CC1 specifies that

"The starting states of a CAUSALNETDEFN is the collection of all instances of STATEDESN, \( S \), such that \( S \) is a statedesn for the CAUSALNETDEFN (i.e. \( S \) is a state in the CAUSALNETDEFN), and there are no other states that cause \( S \)."

As before, we shall use the name GCASNET for the causalnetdefn of GLAUCOMA. Then, CC1 will be evaluated at (GCASNET startingstates), and the ANCHOR VARIABLE, \( \omega \), of CC1 will be bound to GCASNET. In CC1 the predicate \( P(\omega S) \) is "(\( \omega \) statedesn \( S \))(\( S \) causedby NIL)". There is an implicit logical AND between the two relations in \( P(\omega S) \). (*) In this CC there are no quantifiers. We shall later see examples of CC's that use quantifiers. In the context of instantiation of an anchor, its associated CC has the following interpretation:

**THE IFF INTERPRETATION OF A CC:**

Let \((X r)\) be an anchor, and let \(Y\) be the called template of \((X r)\). Let the CC at \((X r)\) be \(CC(X r)\). Let \(P(\omega S)\) be the SET PREDICATE or \(CC(X r)\). Then, for an instance, \(y\) or \(Y\), it is true that,

\((\omega Y) \iff P(\omega Y)\).

*We shall generally omit the logical AND sign, \&\, between relational predicates occurring in a CC. The logical symbols one can use in a CC are: \(\lor\) (OR), \(\land\) (AND), \(\neg\) (NOT), \(\rightarrow\) (IMPLIES), \(\iff\) (IFF).
Thus, for (GLAUCOMA startingstates), a candidate state, say \( s \), can be the starting state of GCASNET, if and only if, it is a state of GCASNET, i.e. \((\text{GCASNET startedon } s)\) is true, and no other state causes \( s \), i.e. \((s \text{ causedby } \text{NIL})\) is also true:

\[
(\text{GCASNET startingstate } s) \leftrightarrow (\text{GCASNET startedon } s) \\
(s \text{ causedby } \text{NIL}).
\]

In general, the relational predicates used in the SET PREDICATE will be the dimensionally consistent relations defined for the domain under consideration. These will be typically of the form \((x \; r \; y)\), where \( x \) and \( y \) refer to instances of templates of the domain, or of the form \((x \; r_1:r_2:...:r_n \; y)\), where \( r_1:r_2:...:r_n \) is a RELATION PATH (RELPATH). The relation path above has the following interpretation: Starting from \( x \), if one traversed in the data base the relation path \( r_1:r_2:...:r_n \), in ALL POSSIBLE WAYS, then for EVERY SUCH traversal, \( y \) will be among the objects in the data base reached from \( x \) via the relation path, if \((x \; r_1:r_2:...:r_n \; y)\) is true. Relation predicates of the form \((x \; r \; y)\) may have one of three possible truth values in the MDS data base: TRUE (T), FALSE (NIL) or UNKNOWN (?).

In the case of \((x \; r_1:r_2:...:r_n \; y)\) its truth value will be ? if \( y \) was not among the objects reached from \( x \) via the relation path, for some traversal of the path, and the collection of objects so reached, say \((y_1 \; y_2 ...\)) had ? included in it. The truth value will be NIL if \( y \) or ? were included in \((y_1 \; y_2 ...\)). The SET PREDICATES of
CC's are evaluated in three valued logic, where T dominates ?, ? dominates NIL, and ? = ? (i.e. (T V ?) = T, (? V NIL) = ?). While writing CC's, one may where convenient omit one or more relations from a relation path of the form r1:r2:::rn. The system will find the shortest relation path that is dimensionally consistent for the given (x r1:r2:::rn y). Thus, for the CASNET domain, as described here, the form (s causedby NIL) is not dimensionally consistent, since the "causedby" relation is not defined for STATEDESN and s is an instance of STATEDESN. In CASNET, for STATEDESN, only (STATEDESN causes CAUSEDESN), (CAUSEDESN state STATEDESN), and (CAUSEDESN causedby STATEDESN) are dimensionally consistent. Thus, only a CAUSEDESN may have the relation "causedby" associated with it. In the case of (s causedby NIL) the system will, therefore, automatically interpret it as (s stateof:causedby NIL). Here (s stateof) will be a collection of CAUSEDESNs. For all the CAUSEDESNs in (s stateof) the (causedby NIL) suffix should hold true.

Let us now consider an example of CC that uses quantifiers. In the domain of CASNET there are not many such CC's. A simple one, CC93, occurs at the anchor (CONDPRCB nextprob). This says,
The template for CONDPROB specifies that (CONDPROB nextprob) is (IT CCNDPROB NIL), i.e. a new instance of CONDPROB. Since CONDPROB is a NODE template and (CONDPROB nextprob (IT CCNDPROB)) is dimensionally consistent, the system would know that for any given instance of CONDPROB, say p1, (p1 nextprob) should be a unique instance of another CONDPROB, say p2, provided, of course, CC92 is true. Thus, when (p1 nextprob) is instantiated, the system would automatically generate the new instance p2, and check whether the SET PREDICATE of CC92 is satisfied for the free variables (p1 p2). The anchor here is, of course, p1, and p2 is the set variable. Let us first examine the context of this CC.

As the reader may notice in figure 2, CONDPROB is called by the anchor (LIKELIHOOD inverseweight), where LIKELIHOOD itself is called by STATEDES. Thus for a state, say xyz, its likelihood will be an instance of LIKELIHOOD, say h, and the inverseweight of h will be p1. This is shown by the diagram below:
Best Available Copy
\begin{itemize}
  \item \textbf{causestate} \hspace{1cm} \textbf{inverseweight}
  \item \textbf{effectstate}
\end{itemize}

(all the states, \textit{S}, that are descendants of \textit{xyz}, and have their presence CONFIRMED, with no intervening DENIED states between \textit{xyz} and \textit{S}. In addition, there is no CONDPROB, \textit{p}, other than \textit{p1}, having the same causestate \textit{xyz}, that has S as its effectstate).

The causestate of \textit{p1} will be \textit{xyz} (this is specified by \textit{CC93} of the anchor \textbf{(CONDPROB causestate)}). In \textit{CC92} above, let us substitute \textit{p1} for \textit{a}. This is what will happen if \textit{CC92} is evaluated at \textit{p1}. Also, the set variable will now be \textit{p2}. We wish to find out the truth value of \textit{CC92}(\textit{p1} \textit{p2}). The constraint says the following:

For some descendant, \textit{S}, of \textit{xyz} (notice that \textit{xyz} is the causestate of \textit{p1}) \textit{S} presence is CONFIRMED, with no intervening DENIED states between \textit{xyz} and \textit{S}, and \textit{S} itself is not the effectstate of any CONDPROB, \textit{p}, whose causestate is also \textit{xyz}.

The reader may follow this interpretation of \textit{CC92} by substituting "\textit{xyz} " for every occurrence of "(a causestate:""). If no such state \textit{S}, can be found then \textit{p2} cannot be instantiated as the nextprob of \textit{p1}. Thus, in the \textit{CC} above "(SOME S)" is used in the sense of "(THERE EXISTS S)".
In general the quantified expressions in SET PREDICATES have the following forms and interpretations. The quantified expression

\[ \text{(SOME template}\ X)Q(X) \]

is interpreted as

\[ \text{(THEREEXISTS X)\ ((template instance X) \& Q(X))}, \]

where \( Q(X) \) is any arbitrary predicate expression. The form

\[ \text{(SOME X)Q(X))} \]

is interpreted as \( \text{(SOME ** X)Q(X))} \). The form

\[ \text{(template X)Q(X))} \]

is interpreted as

\[ \text{(ALL X)\ (template instance X) \to Q(X))}. \]

The form \( (X)Q(X) \) is interpreted as \( (** X)Q(\bar{X}) \). Thus, quantifications are always relative to the instances of a template.

4.3.2: The interpretation of CC's.

We have already seen the IFF interpretation of a CC. There is also an IF interpretation. It is possible in many situations that one wishes to specify a SET PREDICATE of the form \( Q(\bar{X} \& s) \), such that, for an anchor \((X \; r)\), its associated \( CC[Y \; r] \), an instance \( x \) of \( X \), and instance \( y \) of the called template \( Y \) of \((X \; r)\), it is true that,
IF INTERPRETATION OF A CC:

\[(x \, r \, y) \rightarrow q(x \, y).\]

Thus, if one asserts \((x \, r \, y)\) then \(q(x \, y)\) should be true. However, if \(q(x \, y)\) is true, it does not necessarily follow that \((x \, r \, y)\) is true. We shall refer to CC's of this type by the name DECLARATIVE CC's. The value of an anchor instance can be checked using a DECLARATIVE CC, only if a candidate value is given to it (declared to it). In the case of IFF interpretation one can often find a value for an anchor instance, if one existed in the data base, by evaluating its associated CC. The CC at the anchor (CAUSALNETDEFN startingstates) is an IFF CC (also called IMPERATIVE CC). In this case, the evaluation of the CC can find all the STATEDESNs in the data base, which are the starting states of a CAUSALNETDEFN.

We shall write the declarative CC, or an anchor \((x \, r)\), in the form,

\[\langle \langle \text{TEMPLATE} \, y \rangle \rangle \rightarrow (x \, r \, y) \rightarrow \neg (\neg y).\]

This is to be read as, "if \(y\) is declared to be the value of \((\neg x \, r)\) then \(q(\neg x \, y)\) should be true. The use of \(y\) as the SRT VARIABLE in CC\([x \, r]\) enables us to write declarative CC's in this form. An example of a declarative CC occurs in the anchor (CAUSALNETDEFN designatedstates), CC4:

\[\langle \langle \text{STATEDESN} \, S \rangle \rangle \rightarrow (\neg \text{designatedstate} \, S) \rightarrow (\neg \text{statedsn} \, S) \rightarrow \neg (\neg \text{startingstate} \, S).\]

One may read this as
"If S is declared to be the designated state of a CAUSALNETDEFN then (w state design S) and (NOT(w starting state S)) should be true."

If no designated state is declared then the evaluation of CC4 will return the value ?.

In MDS the commands used to instantiate a relation are the IR and ASSERT commands. (For the purposes of this report the reader may think of them as synonyms.) (IR (x r y)) would attempt to assign y as the value of (x r). (IR (x r)) would attempt to first find the appropriate y (or collection of y's) such that (x r y) is true, and assign this newly found value as the value of (x r). Clearly, a prerequisite for being able to find the values y, for a given (x r) is that the CC associated with the anchor (x r) be an IFF CC, or there be present a TR that finds the appropriate y. Thus, at the end of defining all the states of a CAUSALNETDEFN, like GCASNET, and all the causal links, one may merely issue the command (IR (GCASNET starting states)) to find and set all the starting states of GCASNET.

The command (IR (x r y)) will succeed only if CC[X r](x y) does not evaluate to NIL, and at ALL the other anchors (z r), the following is true: If (z r) depended on the value of (x r), (*) then CC[Z r](z) (***) also does not evaluate to NIL. While evaluating CC[Z r](z) the value of (x r) will be hypothesized to

* (z r) is said to depend on (x r) if (x r) occurs in the SET PREDICATE of CC[Z r]. That is the the truth value of the set predicate in CC[Z r] depends on the value of (x r).

** Notice that in the evaluation of CC[Z r] only the anchor is being bound to z. The set variable is not being bound. In this case the values of (z r) existing in the data base will be used as bindings for the set variable.
be y. If \( CC[x,r](y) \), or any of the \( CC[z,r](z) \) evaluated to \( MIL \) then \( y \) will not be accepted as the value of \( x \).

Any time a request for the value of \( (x,r) \) is made, and the stored value of \( (x,r) \) in the data base is \textit{UNKNOWN} \( (?) \), the system will automatically evaluate the \( CC \) and \( TR \) associated with the anchor \( (x,r) \), and try to find the \( y \), such that \( (x,r,y) \) is true. If such a \( y \) is found then the stored value of \( (x,r) \) will be updated in the data base. This process of fixing the values of instantiated anchors is used in \textsc{casnet} to instantiate many of the non-prompted instances of anchors, whenever a need for their values is encountered. For example, when the \( CC \) for (\textit{likelihood probability}) is evaluated it would call for the values of the associated instances of (\textit{likelihood forwardweight}) and (\textit{likelihood totalinverseweight}). The values for these two relations will then get instantiated.

This completes our discussion of \( CC \)'s. We shall not discuss here the forms and interpretations of transformation rules. We shall explain them in the next section in the context of their use.

5.0: The \textsc{model definition} and \textsc{model instantiation} Processes.

The structures, constraints and transformations in \textsc{casnet} description embody implicitly all the sequential processes necessary for the instantiation of models, consistent with the specified constraints, and also for answering questions about the \textsc{casnet}, instances of \textsc{disease model}, or instances of
In the context of interactions with a user, MDS can unravel the sequential processes, implicit in the descriptions, that are appropriate for the context, and execute them in the proper manner. In this section we shall illustrate some of the kinds of interactions that can take place, and discuss the processes that are triggered by them. The commands used in the interactions (the IT—Instantiate Template--, IR—Instantiate Relation--, and ASSERI), and the control structures involved in the execution of the commands are general control structures, that are part of the MDS system itself. In the context of the KNOWLEDGE about CASNET, described to MDS, the particular manifestations described below take place. The sequential processes initiated by the commands depend upon the KNOWLEDGE described to MDS. In the context of KNOWLEDGE for a different domain, the sequential processes will be quite different. In this sense MDS is a META SYSTEM. Its operation depends upon the KNOWLEDGE described to it. As part of its instantiation process it can, if necessary invoke its general problem solving machinery. This machinery contains both a THEOREM PROVER, and a GOAL ORIENTED PROBLEM SOLVER (called DESIGNER), that can plan and execute actions. It is this feature, that makes MDS a very powerful system. The general problem solving facilities enable MDS to UNDERSTAND the descriptions given to it (hopefully in a way anlogous to the way, that an intelligent human being would understand). A fairly detailed discussion of the operation of MDS appears in [Srinivasan 1975a].
In the context of CASNET one may think of the sequential processes in terms of the following tasks:

[A]. Generation of CAUSALMODEL for a given disease.
[B]. Using CAUSALMODEL to generate descriptions of disease afflictions in patients. There are three subtasks here:

1. Record Keeping,
2. Application of Tests,
3. Diagnosis and Therapy recommendations.

In this section, we shall discuss the way these tasks are performed by MDS using the description of CASNET.

5.1: The Definition of CAUSALMODEL.

The model definition process is initiated by the system command,

\[ \text{IT CAUSALMODEL disease-name}. \]

This would create an instance of the CAUSALMODEL, and assign the given disease-name to the model. We shall illustrate a part the model building process for a segment of the GLAUCOMA disease (*). Let the disease name be MINIGLAUCOMA. A simulated sample session of the form shown in Figure 7, would be typical of the kinds of user interactions that could take place.

---

* This example is discussed in greater detail in [Weiss 1974], pp 179-188.
[IT CAUSALMODEL MINIGLAUCOMA]
_MINIGLAUCOMA causalnetdefn:statedefsns...
  (Increased-intracocular-pressure
   Cupping-of-optic-disc; nerve-damage
   Loss-of-vision)

_Increased-intracocular-pressure startingweight...0.2
  ... causes...a CAUSEDES
  ... causes:state...Cupping-of-optic-disc;...
  ... causes:transitionprob...0.0
  ... threshold...?
_Cupping-of-optic-disc; nerve-damage
  startingweight...
  ... causes...a CAUSEDES
  ... causes:state...Loss-of-vision
  ... causes:transitionprob_0.0
  ... threshold...?
_Loss-of-vision causes...NIL

  ... threshold...?
_MINIGLAUCOMA commonthreshold...4
  (Increased-intracocular-pressure threshold 4)
  (Cupping-of-optic-disc; nerve-damage threshold 4)
  (Loss-of-vision threshold 4)

  ................
  ................

FIGURE 7: A part of the CAUSALMODEL acquisition process.

Because of the prompt flags associated with "causalnetdefn", "testdefsns", and "classifications" these relations will be instantiated in the order they appear in the template. The instantiation of (MINIGLAUCOMA causalnetdefn) will cause a new instance of CAUSALNETDEFN to be created and assigned as the causalnetdefn of MINIGLAUCOMA. In CAUSALNETDEFN the relations, "statedefsns", "causedefsns" and "commonthreshold" have prompt flags. These relations will now be instantiated in the order shown. This will generate the request
"MINIGLAUCOMA statesesns..."

to the model builder, as shown in figure 7 above. The user might respond with a list of names or <STATE>s, as shown in the figure. In response to this the system will create an instance of STATEDESNS, and assign the given <STATE>s as the elements of the instance of STATEDESNS, and assign the STATEDESNS itself as the (MINIGLAUCOMA statesesns). The elements of STATEDESNS have got to be instances of STATEDESN. Since the indicated <STATES> do not already exist in the data base, the system will create for each <STATE> a new instance of STATEDESN with the given name. Each such new instance of STATEDESN will in turn cause new promptings to be generated, to complete the instantiation the relations in STATEDESN that have the prompting flag. Thus, for increased-intracocular-pressure, its "startingweight", "causes", the "state" and "transitionprob" of the causes, and the "threshold" would be acquired. The threshold is specified in figure 7, to be "?" (UNKNOWN). Similar promptings generated for the other STATES, are shown in the figure. The last prompting in figure 7 is for (MINIGLAUCOMA commonthreshold). This is set to be 4. This will interact with all the instances of the anchor (STATEDESN threshold), because the relation "commonthreshold" appears in the CC, CC67, at the anchor, (STATEDESN threshold), as shown below:

CC67: CC[STATEDESN threshold].

[(Threshold I) | ((> threshold I) V
 (statelesns:commonthreshold I))]

Thus, the instantiation of \((\text{MINIGLAUCOMA command:threshold})\) will cause the system to evaluate the CC67 at the "threshold" of every instance of \(\text{STATEDESNS}\) so far created. This evaluation will now cause the thresholds for all the three instances, shown in figure 7, to be inferred as being 4. The result of this process is printed out to the model builder, as shown in the figure.

In this manner, guided by the prompting flags, strategically placed at the various anchors, the causalnetdefn process will ultimately acquire the model for \(\text{MINIGLAUCOMA}\). At the end of this model definition process, the model builder may, if necessary, delete any of the definitions, and add new ones. Thus, if a new \(\text{STATE}\) is to be added to the \((\text{MINIGLAUCOMA causalnetdefn:state-styles})\) one may simply issue the command \((\text{ASSERT (MINIGLAUCOMA causalnetdefn:state-styles xyz)})\) where \(\text{xyz}\) is the new \(\text{STATE}\) to be added. This will now initiate once again the appropriate interactive session. At any point one may stop the model building process, and start it later. Also, it is not essential that the facts of the model should be presented in the order implied by the collection of prompting flags in \(\text{CASNET}\) description. They may be supplied in any order. The system will, at each state, verify that the facts supplied do not violate any of the constraints. \(\text{MDS}\) uses the descriptions to seek at each point the appropriate new information necessary to complete the model building process. If at any time the specifications violate any of the constraints, it would supply the user the
reasons for the violation and seek modifications. It is also possible that the model building process was terminated by the user prematurely. In this case, if one attempted to use the defined model for instantiating a DISEASEDESIGN, and if in this process one or more of the missing information in the model definition were required, then at the appropriate points of the DISEASEDESIGN instantiation process, MDS could prompt the user for the missing information, update its model definition and proceed with rest of the instantiation process. This kind of operation is made possible because MDS does, in a sense, understand the descriptions given to it. Let us now proceed to examine the model instantiation process.

5.2: MODEL INSTANITIATION process.

5.2.1: The Record Keeping Process.

The model instantiation process may be started by the command,

`(ASSERT (MINIGHLAUCOMA diseasedesign (IT DISEASEDESIGN)))`

if one wanted to create a new instance of DISEASEDESIGN. Else, one might in some way refer to an existing DISEASEDESIGN that might have been previously (may be partially) instantiated. Every time a new instance is created, the first phase of the process would be, what we have called as Record Keeping. This phase would be completed by the precepts:

`'(MINIGHLAUCOMA diseasedesign:causalmodel MINIGHLAUCOMA)`
This is done because, "causalmodel" has been declared to be the name of the inverse of "diseasedesn", in the CAUSALMODEL template.

(MINIGLAUCOMA disease: date 10-12-75)

MINIGLAUCOMA disease: disease of... PERSON John

This would cause either a new PERSON, John, to be created, or pick out the John that already might exist in the data base. If a positively new PERSON is needed, then one might say, for example, NEW PERSON John. In this case the following additional promptings will occur to complete the description of John.

-John sex... MALE
-John age... 68

Notice that the (DISEASEDESN date) did not get prompted, because the (DATE) function supplied the date. This fact is one of the items displayed in the beginning. [The date shown is probably the date at which the MDS implementation would have progressed enough to execute the CASNET described here!]

As a result of this kind of record keeping, one may at a later occasion refer to the disease:sn of a patient, as follows:

[(DISEASEDESN x) | (John disease x) (x date 10-11-76) 
(x causalmodel MINIGLAUCOMA)].

Hopefully no more than one DISEASEDESN would satisfy this request! [One may for example, choose to distinguish between DISEASEDESNS for people with same name, created on the same date, for the same disease, by assigning names to the descriptions.]

5.2.2: The Test Selection and Application Processes.

[a]: The Test Selection Process.
Part 2 of the model instantiation process, which embodies the application of the tests, represents the heart of the process. Based on the structure of the tests as designed by the model builder, a mixture of predetermined and dynamically determined test applications will determine the particular configurations of the causal net of the DISEASEDESNS. We shall briefly examine below the important parts of this process.

For convenience, let us call the DISEASEDESNS, GDESNS. The test application phase will begin with the instantiation of (GDESNS causal net), which is the next relation with prompting flag, in the DISEASEDESNS template. This will cause a new instance of CAUSALNET to be created and set as the causal net of GDESNS. In the CAUSALNET the relations "states" and "causes" will now be instantiated. Let us denote the new instance of CAUSALNET by GNET. (GNET states) will consist of one instance of each STATE in MINIGLAUCOMA. So also, (GNET causes) will consist of one instance of each <CAUSE> in MINIGLAUCOMA. These instantiation processes take place because the called templates for (GNET states) and (GNET causes) are respectively, (IT (? CC40)) and (IT (? CC41)). These two consistency conditions, shown below, bind the arguments of the function calls to IT, to the

\[(\text{MINIGLAUCOMA causalnetdefine:statesesns})\] and
\[(\text{MINIGLAUCOMA causalnetdefine:causedesns})\]

respectively:

\[\text{CC40: [ (TEMPLATES X) | (causalnetdefine:statesesns X) ]}\]
\[\text{CC41: [ (TEMPLATES X) | (causalnetdefine:causedesns X) ]}\]
Those instances of <STATE>s and <CAUSE>s will now form the skeleton of the new diseasedesn to be created for the new patient. In each <STATE> and <CAUSE>, again the relations with prompting flags will get instantiated in the appropriate manner. Once this is done MDS will move on to the next anchor with prompt flag in DISEASEDESN, namely (DISEASEDESN topleveltest). A new instance of TOLEVELTEST will be created, and since the relation "topleveltest" in DISEASEDESN has the D flag, the system will now proceed to complete all the relations in the new instance of TOLEVELTEST. (Let us denote the new instance of TOLEVELTEST by tittest1.) This will result in the following.

TOLEVELTEST has five relations: "currenttests", "nexttest", "selectedtests", "descendants" and "ancestors". The currenttests will get instantiated first. This will cause its associated CC, CC39, to be executed. CC39 is shown figure 8. It may be paraphrased as follows:

The tests to be applied in a given situation of the model instantiation process are selected from the currenttests of the current TOLEVELTEST. The currenttest, M, should not be one of the already selected tests, and in addition several other conditions are required to be satisfied. Let us follow the conditions specified in figure 8.

The following objects in the neighborhood of tittest1 are first picked out:

MINIGLAUCOMA--testdecsns-->N

<table>
<thead>
<tr>
<th>causalmodel</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNET--topleveltest--&gt;tittest1</td>
</tr>
</tbody>
</table>

The variable X in the CC will be bound to MINIGLAUCOMA,
the variable CN to GNET, and N to (MINIGLAUCOMA
testdesns).

There are two cases to consider: One is (ancestor
NIL), (this is the case for tItestI. The other is -(ancestor NIL). This case will apply for the descendents of tItestI. In the case of tItestI, the WIPEOUT test of the MINIGLAUCOMA will be chosen. The WIPEOUT test will be such that it is not the nexttest of any other test, and also, it is not the nextnegativetest of any other test. Once this selection is made, it will be instantiated as the currenttests of tItestI.

If a WIPEOUT test, Q, had just previously been chosen and applied, then we will be in the case -(ancestor NIL). In this case the currenttests will be (Q nexttest) if (Q testresult is YES), else if (Q testresult is NO) then the currenttests will be (Q negativennexttest), if such nexttests exist.
((TESTDESN M))

((TOPELEVELTEST W) ((\ ancestor W) \rightarrow
\neg(W selectedtest M)))

((THE CAUSALMODEL X) (THE TESTDESN S N)
(THE CAUSALNET CN)

((SOME TESTDESN Y)
((\ ancestor Y) V (\ is Y))
(Y topleveltestof CN) (CN causaimodel X))

(M testdesn X) (X testdesns N)
[
(\ ancestor NIL)
(M testtype WIPECUT) (M nexttestof NIL)
(M negativenexttestof NIL)
V
\neg(\ ancestor NIL)
]

((THE ** Q) (\ nexttestof Q) (Q testtype WIPECUT)
((Q testresult YES) (Q nexttest N)
V (Q testresult NO) (Q negativenexttest M)))

((THE STRATEGY S) (N strategy S)
(s influences TESTS)
[
(s is MINCOST) (M elemof (SMIN N cost))
V
(s is MAXWEIGHTMINCOST)
(M affectedstate
(SMAX (** Z) Z)
(s instanceof instanceof STATEDESNS)
(CN state Z)
((SMIN N cost) affectedstate Z)
likelihood:probability)))

(s is MAXWEIGHTCCSTRATIC)
(M elemof (SMAX N costratio))
V
(s is MAXWEIGHT)
(M affectedstate
(SMAX (CN states)
likelihood:probability) ]))

"instance" stands for "immediate instance":
"instance" is a transitive relation, "instance" is not.

FIGURE 8: THE CC for the selection of "currenttests".
If all the WIPER test had been completely applied, we will still be in the case - (\sigma ancestor NIL). In this case the selection of current tests depends on the strategy used for MINIGLAUCOMA. The strategy will be, of course, the strategy of \( s \), the test design MINIGLAUCOMA. If the strategy influences TESTS, (here "TESTS" will be an instance of the INFLUENCE template), then there are four cases to be considered depending upon whether the strategy, \( s \), is MINCOST, MAXWEIGHTMINCOST, MAXWEIGHTCOSTRATIO, or MAXWEIGHT.

The words "SMIN", and "SMAX" in the CC in figure 3, refer to function templates that have been declared to the domain. The definitions of these function templates appear as follows:

\[
\text{TDN: } \text{SMIN} (\text{FNDEF LISTDT (RELPATH CC58))}
\]
\[
\text{TDN: } \text{SMAX} (\text{FNDEF LISTDT (RELPATH CC59))}
\]

The constraints CC58 and CC59 specify that the RELPATH should be such that for all \( X \) in the first argument (arg1) of the function (notice that the first argument is a LISTDT) the RELPATH (Relation Path) should be dimensionally consistent with \( X \). Also, \( (X \leq \text{RELPATH} > : ^{*} ) \) should be an INTEGER or NUMBER (*). The SMIN and SMAX functions pick out all the \( X \)'s (there can be more than one), in the arg1 of the functions, for which \( \text{RELPATH} \) has respectively the MIN and MAX values, among the objects in arg1.

The relation paths chosen in figure 8, for the various strategies and associated applications of SMIN and SMAX functions are: "cost" for MINCOST, tests of

* The relation ",^*" denotes the dimension of a number. For example, the dimension of \((\text{John age})\) would be YEARS, because the age of a PERSON has been defined to be YEARS. However, the dimension of \((\text{John age;}^*\) will be just NUMBER. In the case of collections, the \(^*\) relation is used to refer to the cardinality of the collection.
lowest cost are chosen, "Likelihood:probability" in the case of MAXWEIGHTMINCOST: from the set of tests with minimum cost, a test is picked such that one of its affected states has a weight greater than any other state affected by the other tests. In the case of MAXWEIGHTCOSTRATIO, all tests with the maximum "cost ratio" are chosen, and, in the case of MAXWEIGHT, all tests whose affected states have the maximum likelihood:probability are chosen. In all these cases, it is required that the chosen STRATEGY should influence TESTS.

[It should be noted that the selection criteria specified here are all fixed for the CASNET system at the time of definition of the CASNET system itself.]

After instantiating the current tests, the system would move to the instantiation of the next anchor in TOPLEVELTEST, namely, the anchor with the relation, "selected tests". The selected tests will be the current test, if the current test is unique, else the user will be prompted for the choice of one or more of the current tests. The selection of the tests is controlled by CC103,

CC103: CC[ TOPLEVELTEST selected tests].

[(** X) | ((& current tests: # 1) (& current test X) V (3 selected tests X) (3 current test X))]}

Notice that one of the disjuncts in the above CC would pick the selected tests, if the number of current tests is 1, (3 current tests: # 1). If not, the phrase, "(3 selected tests X)" indicates that the selected test should be specified by an external source. When (TOPLEVELTEST selected tests) is instantiated every selected test will be instantiated. This will cause the selected test to be applied. We shall discuss
the test application process in section 5.2.2[b].

After the application of the selected tests, the "nexttest" of the TOLEVELTEST will be instantiated. This will again be an instance of TOLEVELTEST. Since, the relation "topleveltest" in DISEASEDESN has the D flag associated with it, this new instance of TOLEVELTEST will now have to be again completely instantiated. Thus, the whole process will iterate, until it is terminated by the "currenttests" becoming NIL, for some TOLEVELTEST in the sequence. When this happens, the nexttest also will become NIL, as indicated by CC5.

The next relation of DISEASEDESN to be prompted is "diagnosis/therapy". The instantiation of this anchor is controlled by CC37. This is used to pick out the appropriate actions depending on the PATHWAYS in the CAUSALNET, and the CLASSDEFNS for MINIGLAUCMA. We shall discuss this part of the model instantiation process in section 5.2.2[c].

[b]. The Test Application Process.

As mentioned before, in section 3 each <TEST> has only three relations, "costratio", "testresult" and "application", that are instantiable. The remaining relations in TESTDESIGN are all instantiated to constants. When the selected tests are instantiated, the system will be still under the control of the D flag of the anchor (DISEASEDESN topleveltest). This
will cause all the above three relations in each <TEST> to be instantiated. The first one will be "testresult". This is controlled by CC34 and TR1. CC34 is shown below,

\[
((\text{YESNO } Y) | [ (\text{testresult } Y) (Y \text{ among } \{\text{YES NO ?? NA}\})
\text{ testtype:among } (\text{SQ GROUP WIPEOUT})]
\text{ SOME ** } X (\text{componentof } X)
[ ((X \text{ testtype WIPEOUT}) \rightarrow
((X \text{ testresult NO}) (Y \text{ is NA}) V (Y \text{ is ??}))
V
((X \text{ testtype MC}) \rightarrow
((\text{SOME } U) (X \text{ component } U) \rightarrow (U \text{ is } ?))
((U \text{ testresult } ??) (Y \text{ is } ??) V
(U \text{ testresult YES}) (Y \text{ is NO}))))]
\]

Normally, the testresult should be supplied by an external source. In certain cases, as in the case of earlier applications of WIPEOUT and MC (multiple choice) tests, (*) the result may be determined by earlier test results. In these cases, if a value for Y is supplied from an outside source, it should be consistent with the conditions specified. Let t1 be the current test. By an analysis of the various CC's in CASNET (MDS would perform this analysis) it may be noticed that the following series of interactions would take place whenever the testresult of a test, like t1, is changed:

* The results for combination tests may also be similarly taken care of. Weiss, [Weiss 1974] has defined combination tests as a separate category of tests. These tests are intended to take care of certain kinds of interactions among test results. In the MDS context, the definition of combination tests is the same as the definition of new consistency conditions, pertaining to the way that test results affect states. For this reason, we have not considered combination tests as a separate category of tests in the description presented here.
(t1 testresult) affects, (X status) for all states X, such that (t1 affectedstate is X), we shall write this as,

\[ \text{[TESTDESIGN testresult] interactions:} \]
\[ \{(X \text{ status); \{(STATEDESIGN X) | (affectedstate X)\}\}]. \]

The anchor symbol, \( \& \), here actually refers to a (TESTDESIGN instance:instance). This is because one level of instantiation was skipped by the use of the C flag.

Similarly, the following other interactions may be identified:

\[ \text{[STATEDESIGN status] interactions:} \]
\[ \{(\& presence)\}. \]

\[ \text{[STATEDESIGN presence] interactions:} \]
\[ \{(X \text{ forwardweight);} \]
\[ \{(\text{LIKELIHOOD X}) \}
\[ \{(\& stateof:causes:state:likelihood X)\}]. \]

\[ \{(X \text{ totalweight);} \]
\[ \{(\text{LIKELIHOOD X}) \}
\[ \{(\& causes:state:likelihood X)\}]. \]

\[ \{(X \text{ inverseweight);} \]
\[ \{(\text{LIKELIHOOD X}) \}
\[ \{(\& likelihood X)\}]. \]

\[ \{(X \text{ effectstate);} \{(\text{CONDPROB X}) \]
\[ \{(\& ancestor:causestateof X)\}]. \]

\[ \{(X \text{ nextprob);} \{(\text{CONDPROB X}) \]
\[ \{(\& ancestor:causestateof X)\}]. \]

...etc.

Similarly, one may now peruse through the various interactions caused by \{LIKELIHOOD forwardweight\}, \{LIKELIHOOD totalweight\}, etc., on yet other anchors in the system. For our purposes here it suffices to note that a whole series of interactions and side effects may propagate through the system, every time the testresult of a test is changed. MDS is made aware of this via the definition of the various CC's. For every anchor, MDS will build the interactions list of the
form shown above. These interactions list will be used to
check consistency in every updating process. Everytime an
anchor is changed, MDS can access and check all the other
anchors in the system that are affected by it. A change will
be accepted only if it does not produce a contradiction in any
of the affected anchors. If a contradiction is produced, or
in general, if MDS is not able to find the value for an
anchor, it will consult the TR associated with the anchor.
Let us consider a part of the updating process associated with
the testresult. The transformation rule T1 is associated
with (TESTDESN testresult) (actually this is associated with
(TESTDESN instance:testresult) because of the use of the C
flag). This transformation rule is shown below:

T1: TR[TESTDESN testresult]
   (COND
      ((? testresult ?)
         (ASSERT
           (@ testresult
             (ASKQUESTION (@ summaryquestion))))))
      [ON-CONTRADICTION
       ((X) (a affectedstate X)
         (NOT ((EXISTING (X status)) = (NEW (X status))))))
       (NOT
       (COND
        (NEW (EXISTING (X status)))
        (NEW (X status)))
        (= 0)
        (SOME STATUS S)
        (BIND S (EXISTING (X status)))
        (ASSERT (X conflict: # of S))
        (ASSERT (X status: # ?)))
        (ABS (NEW (X status))) >
        (ABS (EXISTING (X status)))
        (ASSERT (X status (NEW (X status))))
        (COND
         (ABS (X status: #)) >
         (ASS (X conflict: #))
         (NOT (X conflict: # 0))
         (ASSERT (X conflict: # 0)))]
The function Abs in the rule above is the Absolute Value function. DCOND is the conditional statement (DESIGNER COND), it is similar to the LISP COND statement. The EXISTING values used in the rule are the values that are currently present in the model space of CASNET. The NEW values, are the values that we wish to change to. "NET" indicates that BACKTRACKING is to take place while executing the portion within its scope. In general, transformation rules in MDS are executed in a backtracking environment. The above transformation rule may be paraphrased as follows:

If the test result is UNKNOWN then ask the summary question of the test, and assert the answer as the test result.

If a contradiction is obtained in asserting a test result then do the following: For all states X, that are the affected states of the test, if the EXISTING status of the state is not the same as the NEW status, then

If the sum of the NEW and EXISTING status is 0, assert that the conflict of X is equal to the EXISTING status of X, and set the status of X to 0.

If the absolute value of the status of X, is greater than the absolute value of the EXISTING status, then set the status of X to the new status.

If the absolute value of the NEW status of X is greater than the absolute value of the conflict of X, and the conflict is not 0, then set the conflict to 0.

The presence of a state depends on its status. The changes in the status brought about by TR1 might contradict with the existing presence of a state. In this case, the TR associated with (STATEDESIGN presence) will get invoked for taking care of the contradictions. This TR is shown below.
TR2: TR[STATEDESN presence].

NBT

(DCOND
  (¬(EXISING (a presence)) is (NEW (a presence)))
  (ASSERT (a presence (NEW (a presence))))
  ((** S) (a causes:state S)
   (ASSERT (S forwardweight)))
(COND
  ((NEW (a presence CONFIRMED))
   ((** Q) (a causedby:state Q)
    (ASSERT (Q inverseweight))))
  (T
   ((** R) (a causes:state R)
    (ASSERT (R totalweight)))
    (ASSERT (a inverseweight)))))

It not only fixes the new presence of a state but also issues commands to recompute for all states, Q, that causes a, their respective inverseweights (*), if the new value of presence is CONFIRMED. Other similar commands issued by TR2 may be followed by the reader.

The computation of candidate states and candidate tests are controlled by CC50 and CC51. Here again strategies are used, as specified, at the time of model definition. The reader is invited to peruse these constraints, shown in Appendix A.

As mentioned before, the test application process will continue until the next TOPELEVELTEST becomes NIL. At this point the causal net would reflect the full consequences of all the test, as absorbed by the descriptions of the modeling.

* As the reader may notice, the "inverseweight" is not defined directly for a state. It is defined only for (state likelihood). Thus, when (Q inverseweight) is asserted (as in TR2), for a state Q, the system will interpret it as (Q likelihood inverseweight). This kind of interpretation is possible only when there is a unique way of executing the assert.
The next pending relation will be the next relation in DISEASEDESIGN with the prompting flag. This would be (WISTASDESIGN diagnosis/therapy). We shall discuss the processes involved in the instantiation of this relation in the next subsection.

[c]. The generation of diagnosis and therapy.

The process of diagnosis and therapy in CASNET is based on three notions: the notion of the type of the classification table, the notion of most likely starting states, and the notion of admissible pathways from a most likely starting state. At the time of model definition one may specify the classification table type to be SPECIFIC or GENERAL, by instantiating the classtype relation of the CLASSDESIGN template. If the type is SPECIFIC then only the CONFIRMED states will be looked at, when the algorithm of diagnosis and therapy proceeds. If the type is GENERAL, then all undenied states will be looked at.

The diagnosis/therapy is under the control of the CC, CC37, at the anchor (DISEASEDESIGN diagnosis/therapy). We shall briefly discuss this CC. The CC itself is shown below:

```
CC37: CC[ DISEASEDESIGN diagnosis/therapy]:
  :(COMMENT C) [(CLASSDESIGN E) (THE ENTRYDEFN F)]
  (F firstentry:lowerentry 1)
  (F entrystate:instance:presence CONFIRMED)
  (NOT((SOME ENTRYDEFN G)
    (E firstentry:lowerentry 2)
    (F entrystate:descendant:
      entrystate 1 G)
    (G entrystate:instance:
      presence CONFIRMED))
  )
  (((E classtype SPECIFIC) (ENTRYDEFN F))
```
The process of diagnosis/therapy is viewed in CASNET as one of extraction of an entry (an instance of ENTRYDEFN) which is a component of a classification table, a CLASSDEFN. Each CLASSDEFN has a CLASSNAME (like SPECIFIC, GENERAL etc.), and a "firstentry", which is an ENTRYDEFN. An ENTRYDEFN itself has an "entrystate", "descendents", "comments", "nextentry", and the so called "lowerentries". The lowerentries of an ENTRYDEFN is the closure of its "nextentry" relation. The descendents of an ENTRYDEFN is the collection of all the STATEDEFS of its lowerentries.

CC37 selects the "deepest" entry in the classification table in the following sense: The entrystate of the entry must be CONFIRMED; the entrystates associated with the lowerentries must all be not CONFIRMED; if the CLASSDEFN is of type SPECIFIC, then all the entrystates or higher entries must be not DENIED; and finally, all the entrystates of each of the entries, including the deepest entry must all be components of some given admissible PATHWAY.
The notion of admissible PATHWAY depends on the most likely starting states, \( \text{MLstartingstatesof CAUSALNET} \). These states are computed by the function \( \text{MLSTARTINGSTATES} \) in the description shown here. This function call serves as an alternative to implementing the algorithms as part of consistency conditions. Wherever efficiency considerations are important, then the necessary algorithms may be directly implemented as functions in MDS. These functions may be called at the appropriate places in the instantiation process. Function implemented in this manner will always "EXECUTE BLIND". MDS will not be able to monitor the function while it is executing. In the case of the description presented here, the algorithm for the most likely starting states, is thus not anywhere described. The algorithm may be paraphrased as follows:

A starting state is chosen which has more confirmed descendents without intervening denied descendents than any other starting state. If one starting state can explain all the confirmed states (that is all the confirmed states are its descendents) then it by itself is the complete set. If there exists a tie between starting states, then the one with the greatest starting weight is chosen. If no single starting state can explain all the confirmed states, then considering the remaining starting states and the confirmed states not yet explained, the process is repeated until either no confirmed states remain which are not explained, or some confirmed states have no explainable starting states. The set of states that explain the greatest number of confirmed states, then become the most likely starting states.

A state, \( S \), lies on an admissible pathway from a most likely starting state, \( SS \), if it is a descendendent of \( SS \), if there exist no intervening denied states between \( S \) and \( SS \), and
If S has a confirmed descendent with no intervening denied descendents. The PATHWAY template incorporates this notion, via the CC's, CC48, CC49 and CC66, anchored at the relations, startingstates, components and nextpathway, respectively. We shall now leave it to the reader to verify that the CC's shown in Appendix A, do indeed perform as described above. This completes our discussion of the description of CASNET in MDS.

We have touched upon the essential aspects of CASNET and guided the reader through their descriptions in MDS. The reader should be able to follow the rest without much difficulty.
6.0 Concluding Remarks.

The CASNET description given here is based on the discussion presented in [Weiss 1974]. Since then, the CASNET has undergone various modifications. The descriptive formalism presented here would make it easy to modify or extend the system. In any such updating process the MDS itself may be consulted about any of the existing parts of CASNET. It can answer questions about the CASNET, questions pertaining to the various structures, or questions pertaining to the details of the test application and model instantiation processes. Thus, the description of CASNET in MDS can be used also as a documentation of what CASNET is. This has important consequences. This, can for example, make it possible to use CASNET in a teaching or testing mode. A CAUSALMODEL defined in MDS may be used to teach about the disease process, or to check the answers provided by people who are being tested on their knowledge of the disease.

The questions on the CASNET need not necessarily pertain directly to the description of CASNET as presented to MDS, MDS can also answer questions whose answers would have to be inferred from the descriptions of a domain. The Theorem prover in MDS may be used for this purpose. Thus, one obtains flexibility, clarity of expression and versatility. In the early stages of MDS development we anticipate the pay a heavy price in efficiency. We believe, efficient implementation of MDS would be possible, after some experience is gained with an
initial working version.

Not all the concepts in CASNET have been captured by the description shown here. The concepts of counter and combination tests have not been described. The interpretation of unknown responses (??), is left vague. This has been left vague also in the CASNET system of Kulikowsky and Weiss. Should the unknown responses be considered as test results by themselves, or should the tests involved be just set aside for being repeated at a later stage? The disease domain model shown here never uses the repeatability relation defined for TESTDESIGN. But for these minor omissions the description CASNET presented here captures the rest of the system. To illustrate the ease with which modifications and/or additions to CASNET can be made the description of an extended concept of treatment is shown below. The system is modified to pick out the treatment that maximizes some cost criteria.

A new template for TREATMENT is added:

\[
\text{TREATMENT } n
\]

\[
\text{priority (PRIORITY } t\text{) priorityof)}
\]

\[
\text{status (STATUS } t\text{) statusof } c\text{c80)}
\]

\[
\text{treatment (STATEMENT } s\text{) statementof).}
\]

CC80 will be identical to CC11, the CC associated with

\[
\text{STATEDESIGN status). Two changes in the existing template structures are necessary:}
\]

(a). In EFFECT template,

\[
\text{(EFFECT affectedstate STATEDESIGN)}
\]

will become

\[
\text{(EFFECT affectedstate STATEDESIGN/TREATMENT)}
\]

\[
\text{[STATEDESIGN/TREATMENT implies "UNIOF STATEDESIGN or TREATMENT"].}
\]

And,
(COMMENT therapy STATEMENT)

will become

(COMMENT therapy TREATMENT).

The CC37 at the anchor (DISEASE DESN diagnosis/therapy) will have to be changed as follows:

"(F comments:is C)" will become

"(C treatmentof (S MAX ((TREATMENT R))
(TREATMENT S)
((R is S) V
(R status:>=:statusof S)))
priority))"

The transformation rule TR1 need not be changed. The above changes will cause a treatment with maximum priority to be chosen.

It should be noted that if one were to design CASNET in MDS it is most likely that one would not have implemented it as shown here. We have here deliberately restricted ourselves to the description of Weiss's system. The reasoning power of MDS is not used very much in the process of model instantiation. The formalism is used here primarily for the description of an existing program. In our use of MDS as a design tool for the BELIEVER system we begin to see the usefulness of the formalism to express complex structures and their implied use in a variety of problem solving processes. This is discussed in [Sridharan 1975 a,b].

At the moment it is not clear to us, whether the power and facilities available in MDS are, indeed, needed for the kinds of problems encountered in medical modeling. This report should contribute to the making of that decision.
7.0: ACKNOWLEDGEMENTS:

We wish to thank Shalom Weiss and Casimir Kulikowski for their help in explaining to Joel the various aspects of the Casnet implementation. Many of these aspects could not have been otherwise obtained without going through the laborious process of reading the programs.

REFERENCES:

[Weiss 1974]
Weiss, Shalom. "A System for Model Based, Computer Aided Diagnosis and Therapy", Ph.D. Dissertation, Department of Computer Science, Rutgers University, N.J.

[Sridharan 1975]

[Srinivasan, 1975]
Srinivasan, C.V...."A formalism to define the structure of knowledge."
Srinivasan, C.V...."The use of Gentzen's system of logic for Theorema Proving in MDS." In preparation.

[Srinivasan 1973]
APPENDIX I.

CASNET DEFINITION

(1DN: (CAUSALMODEL RN)
(diseasedesns (DISEASEDESNS $L) causalmodel)
((testdesns !) (TESTDESNS $L) testdesnsor)
((causalnetdefn !) (IT CAUSALNETDEFN)
causalnetdefnof)
((classifications !) (CLASSDEFS $L)
classificationsor))

(1DN: (DISEASEDESNS $L)
(ELEMEN (0 * DISEASEDESNS))
(causalmodel (CAUSALMODEL RN) diseasedesns)
((diseaseof V) (PEOPLE $L) disease))

(1DN: (TESTDESNS $L)
(ELEMEN (0 * TESTDESNS))
((testdesnsorof V) (ST1006 $L) testdesns)
((strategies V) (STRATEGIES $L) strategiesor)
((summaryquestion V) (ST1006 $L)
summaryquestionof)
((counter V) (COUNTERS $L) counterci)
((effects V) (EFFECTS $L) effectsor)
((currenttestsof V) (TOLEVELTESTS $L)
currenttests)
((componentsof V) (TESTDESNS $L) components))

(1DN: (CAUSALNETDEFN $N)
(causedesns (CAUSEDESNS $L) causedesnsor CC71)
(startingstates (STATEDESNS $L) startingstatesof CC1)
(interiorstates (STATEDESNS $L) interiorstatesof CC3)
(designatedstates (STATEDESNS $L)
designatedstatesof CC4)
((statdesns !) (STATEDESNS $L) statdesnsorof)
((terminalstates (STATEDESNS $L) terminalstatesof
terminalstates of CC2)
((commonthreshold !) (THRESHOLD T#)
commonthresholdor))

CC71-CAUSALNETDEFN-causedesns
(2SCC:}
(QUOTE
   ((CAUSEDESN C) | (STATEDESN S) (\@ statedesns S)
   (S causes:elem C)))
CAUSALNETDEFN causedesns)

```
C1-CAUSALNETDEFN-startingstates
(QUOTE
   ((STATEDESN S) | (\@ statedesns S)
   (S causesof NIL))
CAUSALNETDEFN startingstates)
```

```
C2-CAUSALNETDEFN-interiorstates
(QUOTE
   ((STATEDESN S) | (\@ statedesns S) ~
   (\@ startingstates S) ~ (\@ terminalstates S)))
CAUSALNETDEFN interiorstates)
```

```
C4-CAUSALNETDEFN-designatedstates
(QUOTE
   ((STATEDESN S) | (\@ statedesns S)
   (S designatedstatesof \@ ~
   (\@ startingstates S)))
CAUSALNETDEFN designatedstates)
```

```
C2-CAUSALNETDEFN-terminalstates
(QUOTE
   ((STATEDESN S) | (\@ statedesns S)
   (S causesof NIL))
CAUSALNETDEFN terminalstates)
```

```
LDN: (CLASSDEFS $L)
(ELEMDN (O * CLASSDEFN))
((classificationsof V) (ST1006 $L)
classifications)
((classtype V) (ST1000 $L) classtypeof))
```

```
LDN: (PEOPLE $L)
(ELEMDN (O * PERSON)))
```

```
LDN: (DISEASEDESN $N)
(dite (DATE) dateof)
((toplpleveltest !D) (IT TOLEVELTEST)
topleveltestof)
(causalmodel (CAUSALMODEL RN) diseasestates)
```
DISEASE

definition in MDS

((diseaseof !d) (person RN) disease)
((diagnosis/therapy !) (comments &l)
diagnosis/therapy of cc37)
((causalnet !) (it causalnet) causalnetof)

cc37-diseasedefn-diagnosis/therapy
(qsc:

QUOTE

((comment c) |

((classdefn e) (the entrydefn i)
(e firstentry:lowerentries f)
(f entrystate:iinstance:presence confirmed)
(-

((some entrydefn g)
(e firstentry:lowerentries g)
(f entrystate:descendants:element:entrystateof g)
(g entrystate:iinstance:presence confirmed)))

(((e classtype specific) (entrydefn h)
(h firstentry:lowerentries h)
(h entrystate:descendants:element:entrystateof f)
(f entrystate:iinstance:presence confirmed))

v

(((e classtype general) (entrydefn i)
(i firstentry:lowerentries i)
(i descendants:element:entrystateof:element f)
(-

(i entrystate:iinstance:presence denied)

((some pathway p) (entrydefn j)
(e firstentry:lowerentries j)
(j lowerentries f)
(j entrystate:componentsof p))

if (comments:is-c)))\)))

DISEASEDEFN diagnosis/therapy)

(tdn: (st1006 &l)
(elemdns (0 * causalmodel) (0 * question)))

(tdn: (strategies &l)
(elemdns (0 * strategy))

((strategies of v) (testdems &l) strategies)

(tdn: (counters &l)
(elemdns (0 * counter)))

(tdn: (facts &l)

(elemdns (0 * fact)))
Z1-LST DEFINITION IN MDS

(ELEMDS (0 * EFFECT))
((effects of V) (TESTDESNS $L) effects)
((how V) (ST1005 $L) reasonfor))

(ADN: (TOPLEVELTESTS $L)
(ELEMDS (0 * TOPLEVELTEST)))

(TDN: (TESTDESNS MN)
((repeatability !V) (YESNO TA) repeatabilityof CC24)
((cost !V) (COST T#) costof CC25)
((confidence !V) (CONFIDENCE TI) confidenceof CC26)
((firsttest !>CV) (TESTDESNS MN) firsttestof CC30)
((nexttest !>CV) (TESTDESNS MN) previoustest CC31)
((negativoextest !>CV) (TESTDESNS MN)
  negativoextestof CC32)
((testtype !) (TESTTYPE RN) testtypeof CC29)
((testresult C>!) (YESNO TA) testresultof CC34)
((testratio $>C) (COSTRATIO T#) costratioof CC33)
((previoustest V) (TESTDESNS $L) nexttest)
((firsttestof V) (TESTDESNS $L) firsttest)
((strategies V) (STRATEGIES $L) strategiesof)
((negativoextestof V) (TESTDESNS $L)
  negativoextest)
((components !) (TESTDESNS $L) componentsof CC21)
((summaryquestion !V) (QUESTION TS)
  summaryquestionof CC33)
((negativoedeterminancy !) (YESNO TA)
  negativoedeterminancyof CC27)
((counter !V) (CCOUNTER TI) counterof CC28)
((effects !V) (EFFECTS $L) effectsof CC29)
((application C>!) (IT APPLICATION)
  applicationof)

CC24-TESTDESNS-repeatability

(QUOTE
  ((YESNO Y) | (~ repeatability Y)
   ((Y is YES) V (Y is NO))
   ((~ testtype WIPEOUT) -> (Y is NO))
   (~ (~ testtype Y C))
   ((Y is NO) -> (~ instance: #: => 1)))))
 TESTDESNS repeatability)

CC25-TESTDESNS-cost

(QUOTE
  ((COST C) |
   ((~ cost C) (~ (~ testtype WIPEOUT)))
   (((~ testtype SQ) ->
   ...
(-
  ((SOME TESTDESIGN M) (M components: elem a)
     (M testtype MC)))))
V
  (((testtype COMBINATION) V
    (0 counter COUNTER))
    (T is 0))))
TESTDESIGN cost)

TESTDESIGN-confidence
QUOTE
  ((CONFIDENCE C) | (0 confidence C)
    ((0 testtype SQ) V (0 testtype COMBINATION) V
     (0 testtype COUNTER) V (0 testtype GROUP)))))
TESTDESIGN confidence

TESTDESIGN-firsttest
QUOTE
  ((TESTDESIGN x) | (0 firsttest x)
    (0 testtype COLLECTION)))
TESTDESIGN firsttest

TESTDESIGN-nexttest
QUOTE
  ((TESTDESIGN x) | (0 nexttest x)
    (0 testtype WIPEOUT)
    ((x testtype WIPEOUT) V (x is NIL))
    V (0 componentof testtype COLLECTION)))
TESTDESIGN nexttest

TESTDESIGN-negativenexttest
QUOTE
  ((TESTDESIGN x) | (0 negativenexttest x)
    (0 testtype WIPEOUT)
    ((x testtype WIPEOUT) V (x is NIL)))
TESTDESIGN negativenexttest

TESTDESIGN-testtype
QUOTE
  ((TESTTYPE x) |
    ((0 components NIL) (0 counter NIL)
      (x is SQ))
    ((0 summaryquestion NIL) (0 firsttest NIL)
      (x is MC))
    ((0 components NIL))
    (0 summaryquestion NIL))
    (0 components NIL))
    (0 cost NIL) (0 is GROUP)) V
      (x is WIPEOUT))
    (0 firsttest NIL) (x is COLLECTION))
    (0 counter: #: 0) (x is COMBINATION))
(\@ counter: #: > 0) (x is COUNTER))))

TESTDESN testtype)

2234-TESTDESN-testresult
(QSCC:
  (QUOTE
   ((YES NO Y) |
   ((\@ testresult Y) (Y elemof (YES NO ?? NA))
    (\@ testtype:elemof (SQ GROUP WIPEOUT))
    (SOME ** X) (\@ components:of X)
    (((X testtype WIPEOUT) -> (X testresult NO)
      (Y is NA) V (Y is ?))
     V
    ((X testtype MC) ->
     (((SOME TESTDESN U) (X components:elem U) -
       (U is \@)
     (((U testresult ??) *(Y is ?)) V
     ((U testresult YES) (Y is NO))))))))

TESTDESN testresult)

2233-TESTDESN-costratio
(QSCC:
  (QUOTE
   ((COSTRATIC C) |
    C is
    (S MAX
     ((COSTRATIO D) | (** M)
      (M instanceof:instanceof STATEDESN)
      (\@ effects:affectedstate M)
      (D is
      (DIVIDE (\@ cost:*)
        (M likelihood:probability: #)))))
   TESTDESN costratio)

2231-TESTDESN-components
(QSCC:
  (QUOTE
   ((TESTDESN M) | (\@ components:elem M)
    (~
     ((\@ testtype SQ) V (\@ testtype COMBINATION) V
      (\@ counter COUNTER)))
    ((\@ testtype MC) -> (M testtype SQ)
     (M cost NIL))
    ((\@ testtype GROUP) ->
     ((M testtype SQ) V (M testtype MC)))
    ((\@ testtype COLLECTION) ->
     ((M testtype MC) V (M testtype GROUP))
    (~ (M nexttest NIL)))
    ((\@ testtype WIPEOUT) ->
     ((M testtype SQ) V (M testtype MC) V
      (M testtype GROUP)))
    (~ (\@ testtype WIPEOUT)) ->
     ((TESTDESN M) (N components:elem M)
     ((N is \@) V (N testtype WIPEOUT)))))))
TESTDESN components)

**223-TESTDESN-summaryquestion**

\[ (QUOTE \]

\[ ((QUESTION Q) | (\w summaryquestion Q)) \]

\[ ((\w testtype SQ) V (\w testtype GROUP) V \]

\[ (\w testtype WIPEOUT))) \]

TESTDESN summaryquestion)

**227-TESTDESN-negativedeterminancy**

\[ (QUOTE \]

\[ ((YESNO Y) | (\w negativedeterminancy Y)) \]

\[ ((\w testtype SQ) V (\w testtype COMBINATION) V \]

\[ (\w testtype COUNTER) V (\w testtype GROUP)) \]

\[ ((Y is YES) V (Y is NO))) \]

TESTDESN negativedeterminancy)

**228-TESTDESN-counter**

\[ (QUOTE \]

\[ ((COUNTER C) | (\w counter C)) \]

\[ ((\w testtype COMBINATION) V \]

\[ (\w counter COUNTER))))) \]

TESTDESN counter)

**229-TESTDESN-effects**

\[ (QUOTE \]

\[ ((EFFECT E) | (\w effects E)) \]

\[ ((\w testtype SQ) V (\w testtype COMBINATION) V \]

\[ (\w counter COUNTER))))) \]

TESTDESN effects)

**211-TESTDESN-testresult**

\[ (QUOTE \]

\[ (((\w testresult ?) \]

\[ (ASSERT \]

\[ (\w testresult \]

\[ (ASKQUESTION \]

\[ (\w summaryquestion)))))) \]

\[ (ON-CONTRACTION \]

\[ ((X) (\w affectedstate X) \]

\[ (= \]

\[ (EXISTING (X status)) = \]

\[ (NEW (X status)))))) \]

\[ (NBT \]

\[ (DCOND \]

\[ (((ADD (EXISTING (X status)) \]

\[ (NEW (X status))) \]

\[ # := 0) \]

\[ (SOME STATUS S) \]
(BIND S
  (EXISTING (X status)))
(ASSERT
  (X conflict: # OF S))
(ASSERT (X status: # 0)))
(((ABS (NEW (X status))) >
  (ABS
   (EXISTING (X status))))
(ASSERT
  (X status
   (NEW (X status))))
(DCOND
  (((ABS (X status: #)) >
    (ABS (X conflict: #))
  ) (X conflict: # 0)))
)

TESTDESN testresult)

(IDN: (CAUSEDESNS $L)
  (ELEMDN (O * CAUSEDESNS))
  ((causesdesns V) (ST1011 $L) causedesns)
  ((causesof V) (STATEDESNS $L) causes))

(IDN: (STATEDESNS $L)
  (ELEMDN (O * STATEDESNS))
  ((startingstatesof V) (CAUSALNETDEFNS $L) startingstates)
  ((statedesns V) (CAUSALNETDEFNS $L) statedesns)
  ((interiorstatesof V) (CAUSALNETDEFNS $L) interiorstates)
  ((designatedstatesof V) (CAUSALNETDEFNS $L) designatedstates)
  ((terminalstatesof V) (CAUSALNETDEFNS $L) terminalstates)
  ((descendantsof V) (ST1012 $L) descendents))

(IDN: (THRESHOLD T#)
  ((threshold V) (STATEDESNS $L) threshold)
  ((commonthreshold V) (CAUSALNETDEFNS $L) commonthreshold))

(IDN: (ST1000 $L)
  (ELEMDNS (O * ENTRYDEFN) (O * CLASSNAME)
   (O * LIKELIHOOD)))

(IDN: (CLASSDEFN $N)
  ((listentry !) (ENTRYDEFN $N) firstentryof)
((classtype !) (CLASSNAME PN) classstypoof CC15))

CLASSDEFN-classtype
((QUOTE

((CLASSNAME C) | (@ classtype C)
 (C elemof (SPECIFIC GENERAL))))
CLASSDEFN classtype)

(APPLICATION $N)
((candidatetests !V) ** candidatetests of CC50)
((candidatetests !V) ** candidatetests of CC51)
((nextchoice !) (IT (? CC52)) nextchoiceof)
((nextapplication !) (IT (? CC65))
nextapplicationof)

APPLICATION-candidatetests
((QUOTE

((** S) | ((testtype GROUP)
 (S iinstanceof:iinstanceof STATEDESIN)
 (THE STRATEGY X)
 (@ applicationof:elemof:strategies:elem X)
 (X influences STATES)
 (((X is GLOBAL) (\$ (S presence DENIED))) V
 (((X is LIKELYHYPOTHESIS) V
 (X is POTENTIALHYPOTHESIS))
 (SOME ** C)
 (0 iinstanceof:iinstanceof STATEDESIN)
 (0 presence CONFIRMED) (S descendent C)
 (~
 ((SOME P)
 (P iinstanceof:iinstanceof STATEDESIN)
 (P presence DENIED) (S descendent P)
 (P descendent 0)))
 (X is LIKELYHYPOTHESIS) \-
 (S

 statesof:mlstartingstates:resu1t:descendentselem S)))))))
APPLICATION candidatestests)

APPLICATION-candidatetests
((QUOTE

((** $) | ((testtype GROUP)
 (S iinstanceof:iinstanceof TESTDESIN)
 (\$ components x) (x testresult ?) (SOME B)
 (B iinstanceof:iinstanceof STATEDESIN)
 (b candidatetest !)
 (\$ effect:effect:state B)
 (x confidence:*>:*of (APS (n status) ,))
APPLICATION candidatetests)

(WSCC:

(QUOTE

(((TESTDES N) )
(((@ testtype GROUP) (THE STRATEGY S) (
@ strategies:elem S) (S influences TESTS)
(THE ** N) (@ candidatetests N)
(((S is MINCOST) (M elemof (SMIN N cost))) V
((S is MAXWEIGHTMINCOST)
(M affectedstate
(SMAX
(((** X) )
(X iinstanceof;iinstanceof
STATEDES N)
(@ candidatestate X)
(X affectedstateof (SMIN N cost)))
likelihood:probability)))
V
((S is MAXWEIGHTCCSTRATIO)
(M elemof (SMAX N ccstratio)))
V
((S is MAXWEIGHT)
(M effects:affectedstate
(SMAX (@ candidatestate)
likelihood:probability))))
V
((@ testtype COLLECTION)
(M iinstanceof;iinstanceof TESTDES N)
(M testresult ?) (@ components M)
(((SOME ** x)
(x iinstanceof;iinstanceof TESTDES N)
(@ components x) (x nexttest M)
(- (x testresult ?)))
V (@ firsttest M))
V (M is NIL))
V
((@ testtype MC)
(M iinstanceof;iinstanceof TESTDES N)
(@ components M) (M testresult ?))
V
(((@ testtype SQ) V (@ testtype COMBINATION)
V (@ testtype COUNTER))
(M is NIL))))

APPLICATION nextchoice argdn 1)

(WSCC:

(QUOTE

(** TT) |
((@ candidatetests:##:< 2) (TT is NIL)) V
(TT is APPLICATION))
APPLICATION nextapplication argdn 1)
(DN: (PEASON RN)
  ((disease V) (DISEASEDESNS NL) disease)
  (age (ASKYEARS) ago) (sex (ASKSIX) sex)
)

(TDN: (TOLEVELTEST AN)
  ((descendent AN) (TOLEVELTEST AN) ancestor: CC191)
  ((descendent AN) (TOLEVELTEST AN) ancestor: CC192)
  (currenttests !) (TESTDESNS NL currenttests of CC19)
  (select test !) (IT (? CC103) selectedtestsof)
  (select test !) (IT (? CC5) previoustest))

(TOPLEVELTEST descendent)

(TOPLEVELTEST ancestor)

(TOPLEVELTEST currenttests)

(TOPLEVELTEST ancestor)

((THE STRATEGY S) (N strategy: S)
 (S influences TESTS)
 (((S is MINCOST)
   (M elemof
    (SHIN (TESTDESNS NN) (Nil elemof N))
    cost))))
V
((S is MAXWEIGHTMINCOST)
(M affectedstate
(SMAX
(* Z) | (Z instanceof instanceof STATEDESIGN)
(CN states Z)
(SMIN
((TESTDESIGN NM) | (NM elemof N))
(cost)
(affectedstate Z))
(likelihood: probability)))
)
V
((S is MAXWEIGHTCOSTRATIO)
(M elemof
(SMAX ((TESTDESIGN NQ) | (NQ elemof N))
costratio)))
V
((S is MAXWEIGHT)
(M affectedstate
(SMAX
((STATEDESIGN ST) | (ST elemof: statesof CN))
(likelihood: probability))))

TOPLEVELTEST currenttests)

1JJK-TOPLEVELTEST-selectedtests-argdn<1>II
(WSCC:
(QUOTE
((** X) | ((3 currenttests: # 1) (3 currenttests: elem X)
V (3 selectedtests X)
(3 currenttests: elem X)))

TOPLEVELTEST selectedtests argdn 1)

1JK-TOPLEVELTEST-nexttest-argdn<1>IT
(WSCC:
(QUOTE
((** TT) | ((3 currenttests NIL) (TT is NIL))
V (TT is TOPLEVELTEST)))

TOPLEVELTEST nexttest argdn 1)

(1DN:
(COMMENTS $L)
(ELEMDN (0 * COMMENT))
((diagnosis/therapy of V) (DISEASEDESNS $L)
(diagnosis/therapy)
((commentsof V) (ENTRYDEFNS $L) comments)

(1DN:
(CAUSALNET $N)
(Causalnetof (DISEASEDESNS $N) causalnet)
(startingstates ** startingstatesof CC42)
(pathways (IT PATHWAY) pathwaysof)
((caus:en !) (IT (? CC41)) causesof)
(terminalstates ** terminalstatesof CC43)
(interiorstates ** interiorstatesof CC44)
(mlstartingstates (KLSTARTINGSTATES (? CC45))
mlstartingstatesof)
((states !) (IT (? CC40)) statesof))

CC42-CAUSALNET-startingstates
(QSCC:
 (QUOTE
   (((** S) | ((S iinstanceof:startingstatesof:elem w) V
   (SOME x)
   (x iinstanceof:iinstanceof STATUSISH)
   (S startingstate x) (x causes:state S)
   (x status DENIED) (= (S status DENIED))))))
  CAUSALNET startingstates)

CC41-CAUSALNET-causes-argdn<1>IT
(QSCC:
 (QUOTE
   (((TEMPLATES X) | (\)
   causalnetof:causalmodel:causalnetdef:causedesns X

  CAUSALNET causes argdn 1))

CC43-CAUSALNET-terminalstates
(QSCC:
 (QUOTE
   (((** S) | (\)
   causalnetof:causalmodel:causalnetdef:terminalstates:instance S

  CAUSALNET terminalstates)

CC44-CAUSALNET-interiorstates
(QSCC:
 (QUOTE
   (((** S) | (\)
   causalnetof:causalmodel:causalnetdef:interiorstates:instance S

  CAUSALNET interiorstates)

CC45-CAUSALNET-mlstartingstates-argdn<1>MLSTARTINGSTATES
(QSCC: (QUOTE ((CAUSALNET C) | (\ is C))) CAUSALNET
mlstartingstates argdn 1))
CAUSALNET-STATES-ARGDN<1>IT
(QUOTE
  (QUOTE
    (@
      causalnetof:causalmodel:causalnetdefn:states<1>ns x
  )))

CAUSALNET states argdn 1)

(IDN: (QUESTION TS)
  ((summary:questionof V) (TESTDESNS $L)
   summary:question))

(IDN: (STRATEGY RN)
  (influences (INFLUENCE RN) influences:of)
  ((influences:byvt V) (ST1003 $L) influences:to:of)))

(IDN: (COUNTER TI)
  ((counter:of V) (TESTDESNS $L) counter))

(IDN: (ST1005 $L)
  (ELENDNS (O * ENTRYDEFN) (O * STATEDESN)
   (O * HOW))))

(IDN: (EFFECT $N)
  ((how !) (HOW TA) reason:for CC36)
  ((affected:state !) (STATEDESN MN)
   affected:state:of)))

EFFECT-how
(QUOTE
  ((HOW H) | (@ how H)
   ((H is CONFIRMED) V (H is DENIED)))))

(IDN: (YESNO TA)
  ((repeat:ability:of V) (TESTDESNS $L) repeat:ability)
  ((negative:u:termin:ancy:of V) (TESTDESNS $L)
    negative:u:termin:ancy))

(IDN: (COST T#)
  ((cost:of V) (TESTDESNS $L) cost))

(IDN: (CONFIDENCE TI)
(((confidenceof V) (TESTDESNS $L confidence))

(IDN: (TESTTYPE RN)
	((testtypeof V) (TESTDESNS $L testtype))

(IDN: (COSTRATIO T#)
	((costrationof V) (TESTDESNS $L costratio))

(IDN: (ST1011 $L)
	(ELEMDN (0 * CAUSALNETDEFN) (0 * PATHWAY)))

(IDN: (CAUSEDESN MN)
	((stateof V) (STATEDESNS MN) stateof)
	((transitionprob !) (PROB T#) transitionprobof CC19)
	((causesof V) (STATEDESNS $L) causes))

CC19-CAUSEDESN-transitionprob

(DSCC:

((QUOTE

	((PROB P) | |

	(~ (state:terminalstatesof:causedesns x)
	(P #>= 0) (P #=< 1))

	V (P # = 0)))

CAUSEDESN transitionprob)

(IDN: (CAUSALNETDEFN $L)
	(ELEMDN (0 * CAUSALNETDEFN)))

(IDN: (ST1012 $L)
	(ELEMDN (0 * STATEDESNS) (0 * ENTRYDEFN)))

(IDN: (STATEDESNS MN)
	((conflict C) (CONFLICT TI) conflictor)
	((presence C) (PRESENCE TA) presenceof CC13 TR2)
	((likelihood C>!) (IT LIKELYHOOD) likelihoodof)
	((affectedstateof V) (AFFECTSTATE V) affectedstateof)
	((entrystateof V) (ST1002 $L) entrystateof)
	((stateof V) (CAUSEDESN $L) state)
	((startingweight !) (PROB T#) startingweightof CC7)
	((descendents X) (STATEDESNS $L) descendentsof CC8)
	((causes !) (CAUSEDESN $L) causesof CC10)
	((threshold !) (THRESHOLD TA) thresholdof CC67)
	((status C) (STATUS TI) statusof CC11))

CC19-STATEDESNS-presence
(defn presence
  (quote
    ((presence P) 1
      ((@ status #:>=:thresholdof a) ->
        (P is confirmed))
      ((@ status #:=< (MINUS (@ threshold #:))) ->
        (P is denied))))

(stateDESn presence)

;;7-stateDESn-starting weight
(stateDESn starting weight)

(stateDESn-descendents
 (stateDESn descendents)

(stateDESn-causes
 (quote
   ((causeDESn C) @ causes C)
   (C causedesnsof:statedesns a)))

(stateDESn causes)

;;7-stateDESn-threshold
(stateDESn threshold)

(stateDESn-status
 (status S) some M
 (some M)
  ((** N) 1
    (N instanceof:instanceof TESTDESN)
    (@ affectedstate:elemente:effects:of N)
    (N testtype SQ)
    ((N testresult YES) V
      ((N testresult NO)
        (N negativedeterminancy YES)))
    cost
    elem M)
  (some M negative determinancy YES)
  ((MINUS (M cost #:)) #of S)
ZAS6LA DEFINITION IN MDS

\[ V(S #:=:\text{cost} \cdot M)) \]

STATEDESN status

\[ \text{IRr-STATEDESN-presentation} \]

\[ (\text{QUOTE} (\text{NBI}) \text{DCOND} \]

\[ ((\text{EXISTING } (\@ \text{presence})) \text{ is} \]

\[ (\text{NEW } (\@ \text{presence})) \]

\[ (\text{ASSERT} \]

\[ (\@ \text{presence} \]

\[ (\text{NEW } (\@ \text{presence}))) \]

\[ ((** S) (\@ \text{causes}: \text{state } S) \]

\[ (\text{ASSERT } (S \text{ forwardweight}))) \]

\[ (\text{DCOND} \]

\[ ((\text{NEW } (\@ \text{presence CONFIRMED})) \]

\[ ((** Q) (\@ \text{causedby}: \text{state } Q) \]

\[ (\text{ASSERT } (Q \text{ inverseweight}))) \]

\[ (T) \]

\[ ((** R) (\@ \text{causes}: \text{state } R) \]

\[ (\text{ASSERT } (R \text{ totalweight}))) \]

\[ (\text{STATEDESN presence}) \]

\[ (\text{IDN:} (\text{ENTRYDEFN } $N) \]

\[ ((\text{descendants } S) (\text{STATEDESN } $L) \text{ descendentsof} \]

\[ CC17) \]

\[ ((\text{comments } !) (\text{COMMENTS } $L) \text{ commentsof}) \]

\[ ((\text{nextentry } !) (\text{ENTRYDEFN } $N) \text{ nextentryof}) \]

\[ ((\text{firstentryof } V) (\text{CLASSDEFNS } $L) \text{ firstentry}) \]

\[ ((\text{nextentryof } V) (\text{ST1000 } $L) \text{ nextentry}) \]

\[ ((\text{lowerentries } $XR) (\text{ENTRYDEFN } $N) \text{ lowerentriesof} \]

\[ CC18) \]

\[ ((\text{lowerentriesof } V) (\text{ENTRYDEFNS } $L) \text{ lowerentries}) \]

\[ ((\text{entystate } !) (\text{STATEDESN } MN) \text{ entystateof } CC70)) \]

\[ CC17-ENTRYDEFN-descendants \]

\[ (\text{QUOTE} \]

\[ ((\text{STATEDESN } S) \]

\[ (\@ \text{lowerentries}: \text{entystate } S))) \]

\[ \text{ENTRYDEFN descendants}) \]

\[ CC18-ENTRYDEFN-lowerentries \]

\[ (\text{QUOTE } ((\text{ENTRYDEFN } R) \]

\[ (\@ \text{nextentry } L))) \]

\[ \text{ENTRYDEFN lowerentries}) \]

\[ CC70-ENTRYDEFN-entystate \]

\[ (\text{QUOTE} \]

\[ ((\text{STATEDESN } S) \]

\[ ((\text{THE } \text{STATEDESN } R)) \]
ZKFL DEFINITION IN MDS

\(((@ \text{nextentryof:entrystate } R) \rightarrow \\
(\text{R descendents:elem } S)))\)

ENTRYDEFN entrystate)

(IDN: (CLASSNAME RN)
((classtypeof V) (CLASSDEFNS $L) classtype))

(IDN: (LIKELYHOOD $N)
(probability (PROB T#) probabilityof CC53)
(forwardweight (FRCB T#) forwardweightof CC54 TR3)
(totalweight (PROB T#) totalweightof CC55 TR4)
((inverseweight D) (IT CCNDFRCB) inverseweightof CC91 TR5)
(totalinverseweight (PROBS $L) totalinverseweightof CC57 TR6))

CC53-LIKELYHOOD-probability

\(\text{QUOTE}
\begin{align*}
\& (\text{PROB P}) & | \\
\& (P #:is & | \\
\& \text{MIN 1} & | \\
\& \text{(MAX (@ forwardweight:#)} & | \\
\& \text{(@ totalinverseweight:#))))))
\end{align*}
\)

LIKELYHOOD probability)

CC54-LIKELYHOOD-forwardweight

\(\text{QUOTE}
\begin{align*}
\& (\text{PROB P}) & | (\text{THE ** X}) \\
\& (X iinstanceof:iinstanceof \text{STATEDESN}) \\
\& (X likelihood \phi) \\
\& (P = & | \\
\& (ADD (X startingweight:#) & | \\
\& (SUM & | \\
\& ((FRCB C) & | (\text{THE ** Y}) \\
\& (Y iinstanceof:iinstanceof \text{CAUSEDESN}) \\
\& (Y state:causes:state X) \\
\& (Y state:presence ?) \\
\& (Q is & | \\
\& (PRODUCT & | \\
\& (Y state:likelihood:forwardweight:#) & | \\
\& (Y transition:prob:)#)))))
\end{align*}
\)

LIKELYHOOD forwardweight)

CC55-LIKELYHOOD-totalweight
CASNET DEFINITION IN MDS

\( \text{CSSC:} \)

\( \text{(QUOTE} \)

\( \text{((PROB P) | (THE \*) X)} \)
\( \text{(X instanceof:instanceof STATEDESN)} \)
\( \text{(X likelihood \@)} \)
\( \text{(P =)} \)
\( \text{(ADD (X startingweight:\#)} \)
\( \text{(SUM} \)
\( \text{((PROB Q) | (THE \*) Y)} \)
\( \text{(Y instanceof:instanceof CAUSEDESN)} \)
\( \text{(Y state:causes:state X)} \)
\( \text{(~ (Y state:presence DENIED))} \)
\( \text{(Q is)} \)
\( \text{(PRODUCT} \)
\( \text{(Y)} \)
\( \text{(state:likelihood:forwardweight:\#)} \)
\( \text{(Y transitionprob:\#))))))))))) \)
\( \text{LIKELYHOOD totalweight)} \)

\( \text{251-LIKELYHOOD-inverseweight} \)

\( \text{CSSC:} \)

\( \text{(QUOTE} \)

\( \text{((CONDPROB C) |} \)
\( \text{((\@ likelihoodof:presence DENIED) \text{~} \)} \)
\( \text{((C is NIL)))} \)
\( \text{LIKELYHOOD inverseweight)} \)

\( \text{251-LIKELYHOOD-totalinverseweight} \)

\( \text{CSSC:} \)

\( \text{(QUOTE} \)

\( \text{((PROB P) |} \)
\( \text{(P is)} \)
\( \text{(S MAX} \)
\( \text{((PROB P) | (CONDPROB C)} \)
\( \text{(C causestate:likelihood \@)} \)
\( \text{(C probability P))} \)
\( \text{NIL)))} \)
\( \text{LIKELYHOOD totalinverseweight)} \)

\( \text{123-LIKELYHOOD-forwardweight} \)

\( \text{CSSC:} \)

\( \text{(QUOTE} \)

\( \text{(NDT} \)
\( \text{(** S) (\@ causes:state S) (S presence ?)} \)
\( \text{(ASSERT (S forwardweight)))} \)
\( \text{(PCOND} \)
\( \text{(~ (\@ presence DENIED))} \)
\( \text{((** Q) (\@ causes:state V)} \)
\( \text{(ASSERT (Q totalweight)))} \)
\( \text{(FORCE (\@ probability)))} \)
\( \text{LIKELYHOOD forwardweight)} \)

\( \text{124-LIKELYHOOD-totalweight} \)

\( \text{CSSC:} \)

\( \text{(QUOTE} \)
(NBT (ASSERT (INVVERSEWEIGHT)))

(DCOND
  ((presence CONFIRMED)
   (((S) (CAUSEDBY:STATE S))
    (ASSERT (S INVVERSEWEIGHT)))))))

LIKELYHOOD totalinverseweight

 Id5-LIKELYHOOD-inverseweight
 (QSTR: (QUOTE (NBT (FORCE (AT PROBABILITY)))))

 TDN: (ENTRYDEFS $L)
       (ELEMDN (0 * ENTRYDEFS))

 TDN: (CONTENT $N)
       ((diagnosis (!)) (STATEMENT T3) diagnosis)
       ((therapy (!)) (STATEMENT T3) therapy)

 TDN: (PATHWAY $N)
       (components ** components of CC49)
       (startingstate ** startingstate of CC48)
       (nextpathway (IT (? CC66)) nextpathwayof)

 CC49-PATHWAY-components
 (QSCC:
  (QUOTE
   (((S) (INSTANCEOF:INSTANCEOF STATEDESN)
      (AT startingstate:descendent S)
      (~
       ((SOMESOME ** x)
        (x INSTANCEOF:INSTANCEOF STATEDESN)
        (3 startingstate:descendent x)
        (x presence DENIED) (x descendent S)))
    (((U) (U INSTANCEOF:INSTANCEOF STATEDESN)
       (AT startingstate:descendent U)
       (U descendent S) (U presence CONFIRMED))))
   PATHWAY components)

 CC49-PATHWAY-startingstate
 (QSCC:
  (QUOTE
   (((S) (INSTANCEOF:INSTANCEOF STATEDESN)
      (AT pathway:startingstates S)
CASUAL DEFINITION IN MDS

\[
\text{((PATHWAY P) (P starting state S) \rightarrow (S is P)))}
\]

PATHWAY starting state

\[\text{PATHWAY-nextpathway-argdn<1>IT}\]

\[\text{(QUOTE)}\]

\[\text{((TEMPLATE P) | (((** S) (@ pathway of ml starting states S) \rightarrow ((SOME PATHWAY Q) (Q starting state S))) (P is NIL)) V (P is PATHWAY))}}\]

\[\text{PATHWAY nextpathway argdn 1)}\]

\[\text{(IDN: (INFLUENCE RN)}}\]
\n\[\text{influenced by (STRATEGY RN) influenced by of)}\]
\[\text{((influences of V) (STRATEGIES $L) influences))}\]

\[\text{(IDN: (ST1003 $L)}}\]
\n\[\text{ELEMENTS (O * INFLUENCE) (O * LIKELYHOOD))}\]

\[\text{(IDN: (HOW TA)}}\]
\n\[\text{((reason for V) (EFFECTS $L) how))}\]

\[\text{(IDN: (PROB T#)}}\]
\n\[\text{((total weight of V) (ST1000 $L) total weight)}\]
\[\text{((forward weight of V) (ST1000 $L) forward weight)}\]
\[\text{((transition probability of V) (CAUSE DESNS $L) transition prob)}\]
\[\text{((starting weight of V) (STATE DESNS $L) starting weight)}\]
\[\text{((probability of V) (ST1001 $L) probability))}\]

\[\text{(TDN: (CONFLICT TI)}}\]
\[\text{((conflict of V) (STATE DESNS $L) conflict))}\]

\[\text{(IDN: (PRESENCE TA)}}\]
\[\text{((presence of V) (STATE DESNS $L) presence))}\]

\[\text{(IDN: (ST1002 $L)}}\]
\[\text{ELEMENTS (O * LIKELYHOOD) (O * ENTROPY))}\]

\[\text{(TDN: (STATUS TI)}}\]
\[\text{((status of V) (STATE DESNS $L) status))}\]

\[\text{(IDN: (CONDPROB $N)}}\]
(causestate ** causestateof CC93)
(effectstate ** effectstateof CC90)
(nextprob (IT CCNDPLOB) nextprob of CC92)
(probability (PROB T#) probability of CC50))

cc93-ccndprob-causestate

(QUOTE
  (** S) |
  ((~ (inverseweightof NIL))
   (inverseweightof:likelihoodof S))
  V (nextprob:causestate S)))

cc90-ccndprob-effectstate

(QUOTE
  (** S) | (causestate:descendants:elem S)
  (S presence CONFIRMED)
  (~
   ((SOME ** Q)
     (causestate:descendants:elem Q)
     (Q descendants S) (Q presence DENIED)))
  (CCNDPROB C)
  ((C causestate:causestateof Q)
   (C effectstate S))
  -> (C is Q)))

cc92-ccndprob-nextprob

(QUOTE
  ((CCNDPROB C?) | (nextprob CP) (SOME ** S)
   (causestate:descendants:elem S)
   (S presence CONFIRMED)
   (~
    ((SOME ** Q)
     (causestate:descendants:elem Q)
     (Q descendants S) (Q presence DENIED)))
    (CCNDPROB C) (causestate:causestateof C)
    (~ (C effectstate S)))
  CCNDPROB nextprob)

cc50-ccndprob-probability

(QUOTE
  ((PROB P) | (forall ** X)
    (X instanceof:instanceof SIATDOESN)
    (causestate X)
    (P =
     (DIVIDE
      (PRODUCT (causestate:totalweight:))
      (SUM
       ((FOR Y) | (**) Z)
       (Y instanceof:instanceof:)))))


CAUSEDES
(@ causestate:causes Z)
(@ causestate:descendentof:stateof Z)
((((@ causestate:stateof Z)
(Y transitionprob Z))
V
((THE CONDPRES C)
(C causestate:stateof Z)
(C effectstate X)
(Y is
(PRODUCT (Z transitionprob:*)
(C probability:*)))))
(X totalweight:*))
CONDPRES probability)

(TDN: (PROBS $L)
(ELEMNS (0 * PROE))
((totalinverseweightof V) (LIKELYHOODS $L)
totalinverseweight))

(TDN: (ST1009 $L)
(ELEMNS (0 * CAUSALNET) (0 * APPLICATION)))

(TDN: (APPLICATIONS $L)
(ELEMNS (0 * APPLICATION)))

(TDN: (CONDPRES $L)
(ELEMNS (0 * CONDPRES)))

(TDN: (STATEMENT TS))

(TDN: (ST1001 $L)
(ELEMNS (0 * CONDPRES) (0 * LIKELYHOOD)))

(TDN: (LIKELYHOODS $L)
(ELEMNS (0 * LIKELYHOOD)))

(TDN: (CLASSNAMES $L)
(ELEMNS (0 * CLASSNAME)))

(TDN: (PATHWAYS $L)
(ELEMNS (0 * PATHWAY)))

(TDN: (CAUSALMODELS $L))
(ELEMDN (0 * CAUSALMODEL)))

(TDN: (CAUSALNETS $1)
 (ELEMDN (0 * CAUSALNET)))

(TDN: (MINUS $F)
 (FNDEF TUPLE ((FLOATP FIXP) NIL)
 ((FLOATP FIXP) NIL)))

(TDN: (ASKQUESTION $F)
 (FNDEF TUPLE ((QUESTION) NIL) ((YESNO) NIL)))

(TDN: (SMIN $F)
 (FNDEF TUPLE ((LISTP) NIL) ((RELPATH) CC56)
 (((*) CC161))))

<SCC: (QUOTE
 (((RELPATH R) | (arg2 R) (X) (arg1:elem X) (DIMNCK X R)))
 SMAX argdn 2))

<SCC: (QUOTE (((** A) | (A elem:eleom:arg1of arg2)) SMAX argdn 3))

(TDN: (SUM $F)
 (FNDEF TUPLE ((LISTP) NIL) ((** CC60)))

<SCC: (QUOTE (((** D) | (arg2 D) (THE T) X))}
(DEFINITION IN MDS

((X) (a arg1:elem X) (X instanceof of x))
(b instanceof of x))

SUM argdn 2)

(IDN: (ASKSEX $F)
(FNDEF TUPLE ((GENDER) NIL) (sexol (PEOPLE $L) sex))

(TDN: (ASKYEARS $F)
(FNDEF TUPLE ((YEARS) NIL))
(ageof (PEOPLE $L) age))

(IDN: (DATE $F)
(FNDEF TUPLE ((STRINGP) NIL))
((dateof V) (DISEASEDESNS $L) date))

(IDN: (MLSTARTINGSTATES $F)
(FNDEF TUPLE ((CAUSALNET) NIL) (** CC47))
((mlstartingstatesof V) (STICCC9 $L)
mlstartingstates))

SCC: MLSTARTINGSTATES<2>

QUOTE
((** S) |
(S instanceof:instanceof STATEDESNS))
MLSTARTINGSTATES argdn 2)

(TDN: (ADD $F)
(FNDEF TUPLE ((FIXP FLOATP) NIL)
((FIXP FLOATP) NIL) ((FIXP FLOATP) NIL))

(TDN: (PRODUCT $F)
(FNDEF TUPLE ((FIXP FLOATP) NIL)
((FIXP FLOATP) NIL) ((FIXP FLOATP) NIL)).

(TDN: (DIVIDE $F)
(FNDEF TUPLE ((FLOATP FIXP) NIL)
((FLOATP FIXP) NIL) ((FLOATP FIXP) NIL))

(TDN: (ABS $F)
(FNDEF TUPLE ((FLOATP) NIL) ((FLOATP) NIL))

(TDN: (MIN $F)
(FNDEF TUPLE ((FIXP FLOATP) NIL)
((FIXP FLOATP) NIL) ((FIXP FLOATP) NIL)))
(TDN: (MAX SF)
  (FNDEF TUPLE ((FIXP FLOATP) NIL)
    ((FIXP FLOATP) NIL) ((FIXP FLOATP) NIL)))

(TDN: (YEARS TII))
(TDN: (GENDER RN))
HEREDITARY - LOCK RESOLUTION: A RESOLUTION REFINEMENT COMBINING A STRONG MODEL STRATEGY WITH LOCK RESOLUTION

D. M. Sandford

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
CONTENTS

Acknowledgments ................................................................. v

Chapter 1

1.0 Orientation ........................................................................... 1
1.1 Conventions and Abbreviations ........................................... 2
1.2 Lock Resolution ..................................................................... 5
1.3 The Model Strategy and Semantic Resolution ....................... 7
1.4 The Intersection of Lock Resolution
    and The Model Strategy ...................................................... 8
1.5 A Completeness Proof of The Model Strategy
    Using Lock Resolution ....................................................... 9

Chapter 2

2.0 Models and Resolution Searches ....................................... 11
2.1 Models for Use with Semantic Strategies:
    A Specific Example .......................................................... 11
2.2 The Connection Between A-Models and Herbrand
    Interpretations ................................................................. 17
2.3 A Defect in The Model Strategy ........................................ 24
ACKNOWLEDGMENTS

I would like to extend sincere appreciation and thanks to Professor C. V. Srinivasan, who provided the initial impetus and continued encouragement for the writing of this report, and to Professor A. Yasuhara, who has read and offered much needed suggestions for improvement in the presentation of this material.
1.0 Orientation

This report assumes the reader is familiar with the techniques of resolution theorem proving (Robinson, 1965) for the first order predicate calculus. This background information is most easily obtained by reading Chapters 6, 7 and 8 of Nilsson, and Chapters 1 through 6 of Chang and Lee, (Nilsson, 1971) (Chang and Lee, 1973). The terminology used in this report is consistent with these two references except where explicitly defined differently.

Chapter 1 of this report sets the context in which to understand the results and viewpoints stated in Chapters 2 and 3. Chapter 2 develops the notion of a model in a somewhat more general framework than is typically done in resolution theorem proving. Chapter 3 states and explains the main result of this report, and this result can be understood independently of the particular views on models stated in Chapter 2.

The main result of this report is that the syntactic resolution strategy known as Lock Resolution, and the semantic resolution strategy known as The Model Strategy can be combined into a single sound and complete resolution refinement strategy. This refinement is called Hereditary-Lock Resolution (HL-Resolution). Although this strategy is a very strong refinement strategy in its present form, it is felt that its primary value lies in future extensions of the method based upon information available in HL-Resolution searches. Such information is usually not available under other resolution strategies.
1.1 Conventions and Abbreviations

1. A resolution step will refer typically to the resolution of just two parent clauses. This corresponds to binary resolution (Chang and Lee, 1973). Factoring will sometimes be considered as implicit factoring, and other times as explicit factoring. Clauses will sometimes be considered as sets of literals, and are written with the literals separated by commas, and with a semicolon to indicate the end of the clause. The same notation will also be used when it is necessary to consider clauses differently (e.g. as lists), and in those contexts where it is not explicitly stated how to consider the clauses, the reader is free to make his own choice.

2. An unsatisfiable set of clauses, $S$, is said to be minimally unsatisfiable iff every proper subset of $S$ is satisfiable.

3. In Chapter 2 the word model is used to signify a set of Herbrand interpretations. If this set contains only a single Herbrand interpretation then it corresponds exactly to what is called a model in the resolution theorem proving literature. Chapters 1 and 3 are best understood by using the word model to signify a single Herbrand interpretation.
4. The phrase "trivial model" will be used loosely to describe a Herbrand interpretation which assigns a truth value to literals based on the literal letter (i.e. the predicate letter plus the negation sign if present), and based on none or relatively few of the terms that appear in that literal. Thus the models for hyperresolution are classed as trivial models.

5. The arrow "⇒" is used to indicate correspondence between constructs in two different languages, with the item on the left being interpreted as the item on the right.

6. The phrase "singly connected" is used in the sense of (Wos et al., 1967).

7. The phrase "normal resolution" is used to indicate unrestricted (i.e. unrefined) resolution in those contexts where it would otherwise be unclear (and be of concern) as to which strategy is being referred to. Similarly for "normal clause" and "normal literal". Likewise a phrase such as "normal lock literals" is used to denote the usual literals used in Lock Resolution.
8. The following abbreviations and notational conventions will be used:

- **CNF**  Conjunctive Normal Form
- **FA.**  The quantifier "for all"
- **HI**   Herbrand interpretation
- **HLR**  Hereditary-Lock Resolution
- **LC (LM)** Language of the clauses (of the model)
- **LIP**  Local Interaction Problem
- **LR**   Lock Resolution
- **M**    Model (usually meaning a collection of Herbrand interpretations), or a model evaluation function
- **SC**   Sentential Calculus
- **SR**   Semantic Resolution
- **TE.**  The quantifier "there exists"
- **TMS**  The Model Strategy
- **TSP**  Term Substitution Problem
- **BOX**  The empty list (or set) of literals
- **-**    subtraction or set difference
- **@**    negation sign

The above listing is reproduced as the last page of this report.
1.2 Lock Resolution

Lock Resolution (Change and Lee, 1973) (Boyer, 1971) is a purely syntactic refinement strategy for unrestricted resolution, in which literals in the input set are assigned integer lock numbers in any arbitrary way. The refinement is to allow two clauses to resolve only on literals, in each clause, which are of the lowest lock number to appear in that clause. The lock number of a literal in a resolvent is the same as the lock number of its parent literal. When factoring, the literal eliminated is the one with the higher lock number. Lock Resolution (LR) is a complete refinement of unrestricted resolution.

An example of an unsatisfiable sentential calculus (SC) clause set with the literal lock numbers written as the second component of an ordered couple, and the sentential letter as the first component, is:

1. \(<A,1>,<B,2>\);
2. \(<C,3>,<\@A,4>\);
3. \(<\@B,5>,<D,8>\);
4. \(<\@C,6>\);
5. \(<\@D,7>,<\@A,8>\);
6. \(<\@D,8>,<A,9>\);
The complete search space under LR for this clause set is (where we write \( i \times j = k \) to mean that clauses numbered \( i \) and \( j \) resolve to give the clause numbered by \( k \)):

\[
\begin{align*}
2 \times 4 &= 7. \quad \langle A, 4 \rangle; \\
1 \times 7 &= 8. \quad \langle B, 2 \rangle; \\
3 \times 8 &= 9. \quad \langle D, 8 \rangle; \\
5 \times 9 &= 10. \quad \langle A, 8 \rangle; \\
6 \times 9 &= 11. \quad \langle A, 9 \rangle; \\
10 \times 1 &= \text{a duplicate of clause 8.} \\
11 \times 7 &= 12. \quad \ast \text{BOX} \ast;
\end{align*}
\]

Lock Resolution is quite efficient when applied to (exactly or nearly) minimally unsatisfiable SC clause sets. This is a result of purely syntactic properties of the reductio ad absurdum approach in CNF. LR is also a strong restriction when applied to first order clause sets because it is almost singly connected. This report does not concern itself with an investigation of the underlying properties of LR, but merely uses the strategy as a basis for constructing a new refinement called HL-Resolution.
1.3 The Model Strategy and Semantic Resolution

Semantic Resolution (Slagle, 1967) and The Model Strategy (Luckham, 1968) are very closely related. Semantic Resolution (SR) involves some predicate letter ordering, and describes its basic resolution step in terms of clashes, and The Model Strategy (TMS) does not. When dealing with resolution search procedures that are (or almost are) singly connected there seems to be little difference between expressing the procedure in terms of clashes as opposed to binary resolutions. The choice in this report is binary resolution. This choice then focuses on the literal ordering in SR as the distinguishing difference between SR and TMS, and SR can be considered a refinement of TMS. The resolution strategy we develop in this report adds a literal ordering to TMS which is somewhat more restrictive than the ordering in SR.

TMS requires that there be a Herbrand interpretation (HI), M, which can be used to evaluate the truth value of clauses. M is called a model. In TMS a clause is true iff every ground instance of the clause is true in M, and a clause is false iff it is not true. TMS is a refinement of unrestricted resolution which does not allow resolution between two clauses that are both true in M. Notice that M can be any HI, and that no HI can satisfy a set of clauses from which *BOX* can be produced by using unrestricted resolution.
1.4 The Intersection of Lock Resolution and The Model Strategy

Lock Resolution and TMS cannot both be applied and preserve completeness. To see this consider the unsatisfiable example clause set of section 1.2. If we choose the Herbrand interpretation, $M = (\theta A, B, \theta C, \theta D)$ as the model, we see that no resolutions can be performed which satisfy both TMS and LR refinements (according to the given lock numbering). In general it is possible to choose, for unsatisfiable sentential clause sets, the model and the lock numbering so as to preserve completeness. However the resulting search is not actually an improvement over what could be done with LR alone. In the case of a first order unsatisfiable set of non-ground clauses the situation is more complex, and the main result of this report concerns itself with exactly this situation.
1.5 **A Completeness Proof of The Model Strategy Using Lock Resolution**

Here we use LR to prove that TMS is a complete strategy. The purpose of this is to orient the reader toward thinking about a simple connection between LR and TMS for ground level clause sets. The basic HL-Resolution strategy, to be presented later, is primarily just an elaboration of this simple connection so as to make it applicable to non-ground level clause sets.

Let $S$ be an unsatisfiable set of general clauses. Then by Herbrand's theorem there exists a finite set of ground clauses, $SG$, which is a set of ground instances of clauses from $S$ (some clauses in $S$ possibly being grounded in several ways) which is truth functionally unsatisfiable. Let $M$ be any HI for $S$. Then $M$ is a HI for $SG$ also. Let there be LT instances of ground literals in $SG$ that are true in $M$, and LF false in $M$. Let the $L = LT + LF$ ground literals of $SG$ be lock numbered from 1 to $L$ with integers, and with the true literals being numbered from 1 to LT, and the false literals from LT + 1 to $L$, with each integer used exactly once.

Then any lock ground refutation, $R'$, of $SG$ with this lock numbering is also a ground refutation under TMS with model $M$. Now we consider $R'$ as a refutation in unrestricted resolution, i.e. we ignore the lock numbers, and we see that $R'$ can be lifted in an obvious way to correspond to a general level refutation, $R$, of $S$. This $R$ will also be a refutation of $S$ satisfying TMS with model $M$. Thus, since LR is complete for an arbitrary assignment of lock numbers, we have shown that TMS is complete for an arbitrary HI, $M$. 

Notice that it may be impossible to assign lock numbers to the general level literals of S so as to make R also a LR refutation. This occurs because a single general level clause in S may represent several distinct ground instances in SG. It may be the case that the assignment of lock numbers to these several ground instances must be made in a manner inconsistent with a single linearly ordered sequences of literals of the general level clause.

The ability to prove the completeness of TMS so easily using LR leads to two immediate considerations:

1. they are probably closely related;

2. LR seems to be at least as strong a refinement as TMS.

Both of these statements are quite true for a minimally unsatisfiable SC problem, which is exactly what the unsatisfiable ground set (SG, above) refutation problem is. When dealing with general clauses with infinite Herbrand universes however, a major (in fact generally the major) problem is in generating the proper type and degree of instantiation of the general clauses so as to cover some unsatisfiable Herbrand ground set. In this task LR, which is a purely syntactic strategy which does not directly take into account any of the arguments of a literal, loses much of its advantage relative to TMS.

After explaining some aspects of some models that could be implemented, a refinement strategy will be developed that combines LR and a stronger form of TMS.
2.0 Models and Resolution Searches

This chapter develops a particular point of view about what constitutes a model to be used in a resolution theorem proving search. This view holds that a model may be specified through a combination of declarative and procedural information, and that the declarative information need not be fully determined at the time that the resolution search begins. This material is presented in sections 2.1 and 2.2 in the form of examples, but no attempt is made to give a formal statement of the concepts involved. The intention is to develop a concept of a model which will be compatible with such strategies as TMS and SR.

Section 2.3 indicates a difficulty in the relationship between a model and the resolution search process (in this case TMS) that uses the model, and is the immediate motivation for Chapter 3, which presents the HL-Resolution strategy.

2.1 Models for Use With Semantic Strategies: A Specific Example

TMS requires that clauses be evaluated according to some model, M. The model will be some HI for the clause set under consideration. If for every ground instance of a clause at least one literal of the clause is true in M, then the clause is true in M. As far as TMS is concerned, it is not necessary to identify which ground literals are true, and which false, nor to identify which ground instances of a clause are true or false. Thus it doesn't matter what method of clause evaluation is used, just so
long as the classification of clauses is the same as would result from using some fixed HI.

Two extreme examples of models that have been used or suggested for semantic resolution strategies are the trivial models and the ground case models.

The trivial models evaluate the truth of a literal based upon the literal being classed in one of a few simplistically recognizable categories. The most extreme case of this is to assume only two categories, e.g. positive and negative instances of a given predicate letter. Such are the models implicit in P1 (and also N1) (Nilsson, 1971) resolution and positive (negative) hyperresolution. Trivial models typically require only a small amount of computational effort in their truth evaluations of clauses.

At the other extreme are the ground case models, which are based on a complete finite list of all the ground instances for each predicate letter which are true, and, usually, a complete list of function values for each ground instance of each function letter. Clearly such an approach requires the use of small domains of individuals. Even with rather small domains, however, the computational effort to evaluate clauses can easily become quite large (Henschen, 1975).

There are intermediate types of models. These models may contain some explicit information in ground instance form, and evaluate the truth of literals and clauses, which have ground
instances not explicitly listed, by some algorithm. In these cases it is the combination of the explicit ground cases plus the algorithm which constitutes the model. Such models will be called algorithmic based models, or A-models.

An example of an A-model will be given for the satisfiable clause set (which we call GAX for Group AXioms).

1. $P(x,y,*((x,y)))$; closure
2. $P(1,x,x)$; left identity
3. $P(x,1,x)$; right identity
4. $P(x,I(x),1)$; right inverse
5. $P(I(x),x,1)$; left inverse
6. $@P(x,y,z),@P(z,u,v),@P(y,u,w),P(x,w,v)$; associativity
7. $@P(x,y,z),@P(x,w,v),@P(y,u,w),P(z,u,v)$; associativity

This clause set could be part of an unsatisfiable clause set containing the negation of a theorem about groups. In that case the interpretation intended in formulating the problem would be:

$P(x,y,z) \Rightarrow x$ and $y$ combined under the group operation yields result $z$.

$*((x,y)) \Rightarrow$ the result of the group operation on $x$ and $y$.

$1 \Rightarrow$ the group operation identity element.

$I(x) \Rightarrow$ the inverse of $x$ for the group operation.
A possible A-model for GAX could be constructed by use of the sentential calculus (SC) with the following interpretation of the elements of the language of GAX:

\[ P(x,y,z) \Rightarrow \text{the statement: } (x \text{ and } y) \iff z \]
\[ *_{(x,y)} \Rightarrow \text{the expression: } x \text{ and } y \]
\[ 1 \Rightarrow \text{the sentential letter: } 1 \]
\[ I(x) \Rightarrow \text{the sentential letter: } 1 \]

In order to be more definite about this for the purposes of this example, we will assume that the model manipulations themselves will be done in CNF format. Then the A-model will be a set of sentential clauses, which contain the ground case information, plus the algorithm. The algorithm contains the translation information for translating clauses into the SC, and a decision procedure for the SC.

The algorithm must translate constructs in the clause set language, LC, into constructs of the model language, LM, in such a way that a truth decision can be made. The following is the way this is done for the specific model we are constructing.

\[ P(x,y,z) \Rightarrow \text{the clause set: } \theta x, \theta y, z; \]
\[ \quad \theta z, x; \]
\[ \quad \theta z, y; \]

\[ \theta P(x,y,z) \Rightarrow \text{the clause set: } \theta x, \theta y, \theta z; \]
\[ \quad x, z; \]
\[ \quad y, z; \]
\[(x,y) \Rightarrow \text{the expression: } x \text{ and } y\]

\[1 \Rightarrow \text{the sentential letter: } 1\]

\[I(x) \Rightarrow \text{the sentential letter: } 1\]

We assume (arbitrarily for the purpose of this example) the truth of the literal "1", and include it in the model as a clause "1;", thereby establishing the only piece of ground case information. In order to evaluate a clause in the model, e.g. the clause GAX.2,

\[
\text{GAX.2 } \quad \text{P}(1,x,x);
\]

we see if the negation of this clause, when translated into the language of the model, is consistent with the ground case clause "1;". If it is consistent, then the negation of this clause has an instance which is true in the model, and therefore the clause (unnegated) has an instance which is false in the model, and the model evaluation is false. On the other hand if the negation is inconsistent, then the model evaluation for the clause is true.

Thus, the negation of GAX.2, namely

\[
\neg(\text{.FA}.x \quad \text{P}(1,x,x))
\]

becomes

\[
\text{.TE}.x \quad \neg\text{P}(1,x,x)
\]
and then translates into the SC clause set:

\[
\begin{align*}
1; \quad & \langle \text{ground case information} \rangle \\
\emptyset 1, \emptyset x, \emptyset x; \quad & \langle \text{SC clauses} \rangle \\
1, x; \quad & \langle \text{associated with} \rangle \\
x, x; \quad & \langle \text{TE}. x \ \emptyset P(1, x, x) \rangle
\end{align*}
\]

This is an inconsistent SC clause set, and so the clause GAX.2 is evaluated as true in this model. This model will evaluate clauses 1, 2, 3, 6 and 7 of GAX as true, and clauses 4 and 5 as false. Thus, e.g., clause GAX.4 is translated as

\[\emptyset (.FA. x \ P(x, I(x), 1))\]

which becomes

\[.TE. x \ \emptyset P(x, I(x), 1)\]

which becomes

\[
\begin{align*}
1; \\
\emptyset x, \emptyset 1, \emptyset 1; \\
x, 1; \\
1, 1;
\end{align*}
\]

which is consistent, giving an evaluation of false for GAX.4 in the model. Notice that the variables, which are now existentially quantified in LM, do not have the quantifier explicitly written in LM. Since the quantifiers have the entire set of sentential clauses in LM as their scope this causes no problems.
2.1 - 2.2

This report treats A-models in an informal fashion since their complete characterization has not yet been accomplished. Because of this several extremely important details concerning the relationship between A-models and the clause sets they are modeling will be ignored in this report.

There are many other SC models that could be developed for GAX by changing the ground case information and by changing the translation mapping from LC to LM. There are also many other types of models than just the SC models, and the next section will mention some of them.

2.2 The Connection Between A-Models and Herbrand Interpretations

The preceding section illustrated some features of a specific A-model. Some features of A-models in general are:

1. The model performs its evaluation based on its algorithm (including the translation from LC to LM), the ground case information (in LM), and the set of literals being evaluated (in LC).

2. The computation is finite, and is comprised of testing the consistency of a set of statements in LM. This requires that the model be a system with a well defined notion of consistency, and that there be a decision procedure which is efficient enough to make it practical in a resolution search process.
3. It is possible to modify the model by changing either the ground case information or the algorithm.

4. Universally quantified variables (in LC) from the clause to be evaluated are all changed to existential constructs (in LM). Note that there are no existential quantifiers in LC.

5. The process of negating a clause to be evaluated eliminates the "OR-ing" and replaces it with an "AND-ing". Thus a clause of \(k\) literals in LC translates to a set of \(k\) or more statements in LM.

Notice that items 4 and 5 above, along with the fact that we are translating and testing individually only one clause at a time from a clause set in CNF strongly structures the task we ask the A-model to accomplish. In particular we do not need to have the notion of universally quantified variables, in LM, whose scope of quantification would be restricted to the statements in LM coming from the clause we are evaluating.

The question arises as to just what HI is actually the one corresponding to the evaluations performed by a particular A-model.

The answer is that in general A-models will be using a set of HI's to evaluate clauses. A clause is evaluated to true iff it is true in each HI in the set. As an example of this consider the following SC A-model for the equality predicate with the translation:
2.2

\((x, y) \Rightarrow \) the clause set:
\@x, y;
\@y, x;

\@=(x, y) \Rightarrow \) the clause set:
x, y;
\@x, \@y;

\(a \Rightarrow \) the sentential letter: \(a\)
\(b \Rightarrow \) the sentential letter: \(b\)

and assume no ground case information.

If we evaluate the unit clause \(=(a, b);\), the clause set in LM is:

\(a, b;\)
\@a, \@b;

which is consistent, so that the clause \(=(a, b);\) is false.

Then, evaluating the clause \(@=(a, b);\), which translates to

\@a, b;
\@b, a;

we also have a consistent clause set, thus yielding false for
\@=(a, b);.

Clearly this \(A\)-model is not evaluating clauses correctly according to any HI. What is actually happening is that the evaluation is according to some class, \(M\), of HI's. In some of these \(=(a, b);\) is false, so that it is evaluated false. In the rest of \(M\), \@=(a, b); is false, so that it too is evaluated to false.
One thing that can be done in cases like this (i.e. when the A-model is using a class of HI's) is to ensure that every clause that is evaluated is uniformly true or uniformly false over the class, M, of HI's being used. When encountering a clause not uniform in truth in M, then M must be replaced by an $M' \subseteq M$, such that the clause is uniform in $M'$. $M'$ would then be used until it became necessary to choose an $M'' \subseteq M'$. Thus successive clause evaluations would cause modifications of the ground case information, or the algorithmic parts of the model, or both. Notice that once a clause is evaluated, its truth evaluation is not affected by future changes in the model, since it was uniformly true or false when evaluated, and the only model changes permitted are those that replace the model class with a subclass of itself.

For simplicity, and definiteness, we assume that the only way to modify a model, i.e. restrict M, is to add ground case information so as to make a clause that is not uniform in M become uniformly true in $M' \subseteq M$. The way to do this is:

1. If the clause evaluates to true in M, no modification is performed since it must be uniformly true in M.
2. If the clause evaluates to false in M, but is not uniform in M (see below), then we try to find various items of ground information, $G_1, G_2, \ldots$, such that:
   a) The current ground information is consistent with the expression $G_1$ and $G_2$ and $\ldots$;
   b) The clause evaluates to true when evaluated by the model using the combined old ground case information plus $G_1, G_2, \ldots$. 
If $a$ and $b$ above are both satisfied, then the new model is the same as the old one but with the ground case information augmented by $G_1, G_2, \ldots$. We do not elaborate here how to find a suitable set of ground facts, but mention that sometimes there will exist no such set of ground facts. In this case the model is left unmodified, and the clause is evaluated as false in $M$. Such a default assignment of false to a clause causes no problems (beyond a decrease in search efficiency) for the types of refinement strategies discussed in this report (i.e. TMS, SR and HLR).

A simple example of this model updating process is the equality example, above, where now, when the $=(a,b)$; evaluation comes out false, we augment the ground (in this instance empty) information set with the information in the clause, namely

$$
@a,b; \\
@b,a;
$$

The ground information set is still consistent, so this is acceptable. The clause $=(a,b)$; would now evaluate true.

Now the clause $@=(a,b)$; translates to

$$
@a,b; \text{ ground information} \\
@b,a; \text{ ""} \\
@a,b; \text{ from the clause } @=(a,b) \\
@b,a; \text{ "" "" } \\
$$

and this is a consistent set, so that $@=(a,b)$; evaluates to false.
We are left with the problem of determining, when a clause evaluates to false in $M$, if it is uniformly false. A clause is uniformly false in $M$ if the set of statements in $LM$ representing the negation of the clause are not only consistent, but are in fact consistent in all interpretations which satisfy the ground case information and the information implicit in the algorithm for testing consistency. As an example consider the evaluation of clause GAX.4 as done in section 2.1. The SC clause set obtained there was consistent, and thus GAX.4 was evaluated to false. However, that SC clause set is a true set of statements only when the sentential letter "x" is assumed false. If "x" is assumed true, then the SC clause set is untrue. Thus GAX.4 is not uniformly false in this model. Furthermore there is no ground case information that can be added to this model which will make GAX.4 uniformly true. We note here that this does not seem to be an unresolvable difficulty with A-models. We do not develop potential solutions to this difficulty in this report.

The above explanation is in no way yet sufficiently formalized so that it is possible to prove that it is a consistent way of viewing A-models. Such a formalization has not yet been attempted. However, this type of orientation toward A-models does seem to have some utility, since there are other types of A-models which use different internal languages and different processing algorithms, but are still quite analogous (with respect to the topics of this section) to the SC A-models. One of these models is the simultaneous linear equation model (SLE). This model translates literals into equations involving the arguments of the literal.
Thus a clause would be translated into a set of simultaneous linear equations, and the processing algorithm is a decision procedure testing the set for consistency. To see that the SLE model acts as a theorem proving system analogous to the SC model, one need merely consider a set of equations to be a set of unit clauses, each using the two place equality predicate (negated for inequalities), with terms built out of constant symbols, existentially quantified variables, and function symbols corresponding to the usual arithmetic operations. In addition there would be other clauses expressing the rules of algebraic manipulations. In an actual implementation it would probably not be efficient to do consistency checking in the SLE model as a theorem proving search, but it is reasonable to consider it that way conceptually. Section 3.2 of this report is an example of an HLR search using a SLE model.

Other possible A-models are the decidable parts of Euclidean geometry or of real analysis. Our viewpoint of models is not meant to be restricted to what are typically thought of as mathematical models. The discussion of models will however be limited to the SC and SLE models in this report, since these two models will be familiar to the reader, and are quite sufficient for illustrative purposes.

We leave now the discussion of A-models. In the remainder of this report the word "model" may be thought of as signifying a single HI.
2.3 A Defect in The Model Strategy

TMS evaluates the truth values of clauses when they are generated, giving them a single truth value for all of the remainder of the search. But often a clause that has been determined to be false and gets resolved with a true clause, actually doesn't qualify as a false clause because the unification operation on the false clause eliminates all of its false instances. An example of this is the false clause

\[ P(x, 1, y), <(x, y); \]

and the true clause

\[ \neg P(a, 1, b); \]

where the model is taken to be

\[ P(x, y, z) \Rightarrow x \text{ times } y \text{ is } z \]
\[ <(x, y) \Rightarrow x \text{ is less than } y \]
\[ a \Rightarrow 10 \]
\[ b \Rightarrow 20 \]
\[ 1 \Rightarrow \text{identity element for multiplication} \]

and the domain of individuals is that of the positive integers.
The resolvent of these two clauses is

\((a,b):\)

which is true in the above model.

The problem is not that the resolvent is true. There is no way in TMS to maintain completeness if true resolvents are forbidden. Rather, the problem is that the resolvent has a literal in it which came from the false parent, but which now has no false instances. It is clear (refer to section 1.5 for the proof of TMS using LR) by looking at the ground case proof that must exist, that such an occurrence of loss of false instances can be forbidden in a TMS search without a loss of completeness. Further reflection on the form of the clauses and the resolutions which appear in the ground proof trees involved in the proof of TMS leads to a complete refinement strategy for unrestricted resolution which combines LR and semantic considerations stronger than TMS. The next chapter is a presentation of such a strategy.
3.0 **Hereditary-Lock Resolution**

This chapter presents the main result of this report. This result is the development of a sound and complete refinement strategy for resolution, called Hereditary-Lock Resolution (HL-Resolution, or HLR). This strategy combines a semantic refinement (TMS) and a syntactic refinement (LR) in such a manner that the refinement restricts the way that a clause may be used based upon the way that clause was generated in the search.

Sections 3.1, 3.2 and 3.3 are an informal introduction to the strategy, including a hand worked example. Section 3.4 is a formal definition of the HLR refinement. Section 3.5 is the statement and proof of the soundness and completeness theorem for HL-Resolution. Sections 3.6 and 3.7 give some perspectives on the utility of the strategy and indicate some areas that require further investigation.

3.1 **HL-Resolution: An Informal Description**

This section states what HL-Resolution is in an informal manner. The purpose here is neither to give an exact formal definition nor to consider implementation details but rather to give the reader an idea of the overall structure of the strategy.

Let \( M \) be a Herbrand interpretation and \( L \) a set of literals. A grounding substitution for \( L \) is a substitution which when applied to every literal in \( L \) converts the literal to a ground instance. \( M \)
is said to satisfy a set of literals, \( L \), iff for all grounding substitutions, \( \Sigma \), there exists a \( k \) such that \( k \in L(\Sigma) \) and \( k \) is true in \( M \). If \( L \) is a clause and \( M \) satisfies \( L \) (i.e. \( L \) viewed as a set of literals), then we say that \( L \) is true in \( M \).

An HL-clause, \( C \), is an ordered pair, \( C = \langle SD(C), FSL(C) \rangle \), with each element being a set of HL-literals. \( SD(C) \) is called the set of standard literals of \( C \), and \( FSL(C) \) is called the (set of) false substitution literals of \( C \).

The standard literals correspond to the usual literals in a clause in normal resolution. The FSL literals constitute a restriction on the ground instances which a clause represents. An HL-clause represents all ground instances of its standard literals in which the grounding substitution, \( \Theta \), is a grounding substitution for both the standard and FSL literal sets, and such that for every literal, \( L \), in the FSL set, \( L(\Theta) \) is not true in \( M \).

Let \( C \) be an HL-clause. Let \( \rho \) be the set union over all variables appearing anywhere in \( C \) and all literals appearing anywhere in \( C \). Let \( \mu \) be the set of variables appearing in the standard literals of \( C \). We execute iteratively the following two steps until no further change occurs to \( \mu \):

1. Move all literals in \( \rho \) containing a variable in \( \mu \) into \( \mu \).
2. Move all variables in \( \rho \) which are contained in a literal in \( \mu \) into \( \mu \).
We define a variable or literal to be influential in $C$ iff it is in $MU$. If $L$ is a literal in the FSL set of the clause $C$, and if $L$ is not influential in $C$, then $L$ may be deleted from the FSL of $C$.

Let $L$ be a literal, then by $\@L$ is meant the same literal but with the opposite sign, i.e. $\@L$ contains a negation sign iff $L$ does not contain a negation sign.

Every standard literal has associated with it two numbers. One is called the true lock number, and the other is called the false lock number. It is through the use of these lock numbers that HL-Resolution incorporates LR as part of its refinement. To understand why each literal must have more than one lock number refer back to section 1.5, where TMS was proved complete by an argument based on LR. There it is seen that in the ground case it is necessary to have all the lock numbers of true literals smaller than all the lock numbers of false literals, in order to make the LR search automatically satisfy TMS restriction of having a false parent in every resolution step. When searching at the general level, then, it is necessary to have a mechanism which can accomplish the same effect as assigning small lock numbers to true literals and large lock numbers to false literals. The problem at the general level is that a given clause may really need to be used to represent several distinct ground instances, and the truth value of any particular general level literal might be different in the different ground instances. HL-Resolution searches are concerned with keeping track of the use of clauses with respect to which ground instances are the ones intended to be represented by the
clause. This information is implicitly held, in the FSL of the clause, at the level of specifying a set of literals that must be false. Completeness of the HL-search is maintained through the use of the double lock numbers on literals, the true (false) lock number being used if a literal is supposed to stand for true (false) instances of itself. Exactly how this is accomplished should be apparent later when some examples are given. Most of the complications arise, in explaining the HL-Resolution method, in treating standard literals for which the FSL does not contain enough information to restrict them to necessarily true or necessarily false instances.

The input clauses for HL-Resolution are similar to input clauses in normal resolution, except that:

1. each clause has an FSL part, and it is empty;
2. true and false lock numbers, all distinct, have been assigned to all the literals (and all of the true lock numbers are smaller than all of the false lock numbers).

Each input clause, and during the search every clause generated, is evaluated according to M, and marked "T" if it only has true ground instances, "F" if it only has false ground instances, or "T/F" if it has both true and false ground instances. Note that this evaluation is relative to the FSL restriction, i.e. we only consider grounding substitutions that make every FSL literal false. Section 3.2 contains an example of how a HL-clause can be evaluated relative to its FSL restrictions. We may view the search process as a breadth first binary resolution search
(factoring will be discussed later) with several refinements and one extension.

The extension is that there is an asymmetry in the role played by the two parents in a resolution step. Thus given two clauses, C1 and C2, we consider two distinct resolution possibilities, and express this as resolving C1 against C2, and C2 against C1. This leads to the first refinement condition which is that for C1 to resolve against C2, it must be the case that C1 is false in M. Thus C1 may be either "F" or "T/F" in its evaluation according to M. Furthermore, after choosing the two literals on which to unify in the resolution of C1 against C2, C1 must still be false after the most general unifier (mgu) is applied to C1. Note that the mgu is also applied to the FSL literals. The purpose of this is simply to make sure that a false clause in a resolution actually contributes to the resolvent produced at least one of its false instances.

It is not the case however, that the search would be complete if false clauses were allowed to contribute none of their true instances to the search space. Thus if C1 and C2 are both "T/F" in M, then we would try to resolve C1 against C2, producing the resolvent R, making sure C1 can pass on only false literal instances to R. In this case C2 acts as a true clause since at the very least, the literal of C2 resolved on is representing true ground instances. We then try the other possibility, namely resolving C2 against C1, producing R', in which we make sure the literals in R' received from C2 have false ground instances, and
here C1 acts as the true clause.

Thus our first refinement condition states that if C1 and C2 are both "T" we do not resolve them together at all. If one has true instances and the other has false instances (but not both are "T/F"), then we try to resolve the false one against the true one. If both C1 and C2 are "T/F", then we resolve C1 against C2 and also C2 against C1. In each case the resolution of C against D is blocked if, after unification, C has no false instances left. This first refinement is clearly just a strengthening of TMS, and is sensitive to the structure of the model which is used for the truth evaluations of the clauses.

Now we consider the second refinement, which is syntactic in nature and is an elaboration of the Lock Resolution strategy.

When we resolve C1 against C2, we know that C1 must stand only for those ground instances in which every one of its literals is false. What we then do is say that the literal in C1 on which we are allowed to resolve is constrained to be a literal of lowest false lock number in C1. We also see that after unification, the corresponding literal on which C2 is being resolved must be true for all the ground instances that C2 represents in this resolution step. From this we can require that every literal in C2 with a smaller true lock number than the literal in C2 on which we resolve, must represent ground instances which are false. Thus we see rather stringent conditions being statable about what must hold before a resolvent can be produced. The single most important feature of HL-Resolution search organization is the movement of
these conditions in a simple form into the FSL portion of the resolvent being produced. This is how we organize the individual resolution steps so that we can keep track of what is going on with respect to the allowed substitution instances for the literals in the resolvent produced.

As an example of how this is done in a typical resolution step, consider the two HL-clauses, C1 and C2, where C1 has at least some false instances, and C2 has at least some true instances, and we wish to resolve C1 against C2. We write the HL-clauses as a set of standard literals, terminated by a semicolon, and followed by the set of FSL literals. Each standard literal is written as an ordered triple, in which the first component is the true lock number, the third is the false lock number, and the second component is a normal literal.

Thus we have

C1 = <1,A(x),50>, <9,B(f(y)),61>, <2,G(c),57>; FSL = (L1,L2)
C2 = <10,θA(u),46>, <15,E(u,f(u)),41>, <17,θA(g(v)),38>;
FSL = (L3,L4)

where L1, L2, L3 and L4 represent some literals that have been put into the FSL's due to previous resolution steps in the derivation trees for C1 and C2. We assume that these literals are not the same as any of the standard literals of C1 and C2 for the purposes of this example.
Now to resolve $C_1$ against $C_2$, $C_1$ must stand for ground instances in which every one of its literals is false, and it will be allowed to resolve only on its literal of lowest false lock number, which in this case is its first literal. We first try to resolve on $C_2$ on the literal of lowest true lock number, again in this case its first literal. There is a mgu for these two literals, $\nu = (x/u)$, and we form the resolvent $R_1$:

$$R_1 = \langle 15, E(x, f(x)), 41 \rangle, \langle 17, A(g(v)), 38 \rangle, \langle 9, B(f(y)), 61 \rangle, \langle 2, G(c), 57 \rangle;$$

$$FSL = (L_1(\nu), L_2(\nu), L_3(\nu), L_4(\nu), A(x), B(f(y)), G(c))$$

Notice that the FSL set consists of everything we know must be false, namely those things already in the FSL's of $C_1$ and $C_2$, plus all of the literals from the false parent, $C_1$.

There is another resolvent that can be produced from $C_1$ against $C_2$, by using the third literal in $C_2$. Thus we produce $R_2$, using the mgu $\mu = (g(v)/x)$:

$$R_2 = \langle 10, A(u), 46 \rangle, \langle 15, E(u, f(u)), 41 \rangle, \langle 9, B(f(y)), 61 \rangle, \langle 2, G(c), 57 \rangle;$$

$$FSL = (L_1(\mu), L_2(\mu), L_3(\mu), L_4(\mu), A(g(v)), B(f(y)), G(c), A(u), E(u, f(u)))$$

Here we notice that since the first and second literals of the true clause, $C_2$, had lower true lock numbers than the true literal selected for resolution from $C_2$, namely the third literal of $C_2$, those first two literals of $C_2$ must represent false ground instances. This is why they appear in the FSL of $R_2$. 
Thus we see a strong interaction between the Lock Resolution literal ordering and the allowed substitution instances that a resolvent can have. This interaction is mediated through the action of the model. This brings us to the last refinement condition for HL-Resolution steps, namely that the set of ground instances that a clause stands for must be non-empty. In the case of R1 and R2 above, we would actually have to check within the model, M, to see if the FSL's had any substitution instances that simultaneously makes all of the FSL literals false ground literals. This is simply done by considering the FSL literals as if they were the literals of an ordinary clause (i.e. "OR-ed" together), and submitting them to the model for evaluation just like a normal clause would be evaluated. If the model returns the answer "false", then the FSL set has a falsifying substitution, and the corresponding clause exists. If the answer is "true", then the clause has no allowed instances and can be deleted.

We make the following comments in general on the HL-Resolution search, as it has been presented up to this point:

1. When the parent clauses have non-empty FSL's then the mgu is applied to both FSL's and all their literals go into the FSL of the resolvent, along with any other new literals from:
   a) all the standard literals of the false parent, including the literal resolved upon;
   b) any literals among the standard literals of the true parent which have true lock numbers lower than the true lock number of the literal resolved upon in the true parent.
2. There is usually only one possible literal for resolution in the false parent.
3. Often there will be several literals to resolve upon in the true parent.
4. The input clauses start out with no literals in their FSL lists.
5. As clauses deeper in the search are formed, the FSL's tend to grow and become more restrictive.
6. Some resolvents are blocked due to having empty sets of ground instances.
7. Literals which contain only ground terms can be eliminated from an FSL if they are false; if they are true then the whole clause can be eliminated.
8. If a literal is no longer influential in a clause, then it may be deleted from the FSL if it has false instances; if it has only true instances, then the clause is deleted.

Factoring in HLR is somewhat different than in unrestricted resolution. For simplicity in explanation we will treat factoring as explicit factoring. We approach the factoring issue by first stating that any factoring can be thought of as a sequence of elementary factoring steps, where an elementary factoring step is defined to be a factoring of a clause where the set of literals on which we factor contains just two literals. We also note that in HLR, as in other binary resolution refinement strategies, it is only necessary to produce factors which unify literals with the literal which will be next resolved away in a resolution step. Thus in HLR we identify the next literal to be resolved on, and
perform elementary factoring steps, yielding a set of factors. These factors, and any further factors produced from them by elementary factoring steps on the same literal, constitute the set of factors of a clause that must be produced in HLR. We do not here give a full description of the factoring process, but offer the following example to illustrate the nature of the considerations that go into the factoring step. Suppose we wish to factor the clause C:

\[ C = \langle 1, A(x), 50 \rangle, \langle 2, A(f(y)), 49 \rangle, \langle 11, B(a), 41 \rangle; \text{FSL} = \emptyset \]

which can be syntactically factored on the second components of its first and second literals by using the mgu \( f(y)/x \). As in LR, we delete the literal with the higher lock number, i.e. we "factor low". The problem is that the first two literals of C have no unique low number, since we do not know whether to use the true or false lock numbers of these literals. Therefore we tentatively produce the following two elementary factors of C:

\[
\begin{align*}
F1 &= \langle 1, A(f(y)), 50 \rangle, \langle 11, B(a), 41 \rangle; \text{FSL} = (\emptyset A(f(y))) \\
F2 &= \langle 2, A(f(y)), 49 \rangle, \langle 11, B(a), 41 \rangle; \text{FSL} = (A(f(y)))
\end{align*}
\]

We see that \( F1 \) stands for ground instances in which \( A(f(y)) \) is true, and \( F2 \) for ground instances in which \( A(f(y)) \) is false. Now, having done this, we can see that \( F2 \) is not a legitimate elementary factor. The reason is that since \( A(f(y)) \) is false in \( F2 \), the first literal in \( F2 \) must use its false lock number. But then the second literal of \( F2 \) will be the one resolved away next. This means we have factored \( C \) on a literal other than the one to be next resolved.
away, and this is unnecessary in HLR. Thus the only factor of $C$ is $F_1$. Notice that it is necessary to check the FSL of $F_1$ according to the model to see that it is indeed possible to have $\exists \mathcal{A}(f(y))$ evaluate false. If the model says that $\exists \mathcal{A}(f(y))$ is true, then $F_1$ will be deleted also.

Our explanation of HL-Resolution thus far has been informal in nature, but has covered enough of the relevant features of the strategy so that we are now in a position to see and understand an actual example of an HL-search space. This is done in the next section.
This section works through an example theorem taken from Chang and Lee, p. 302, (Chang and Lee, 1973) and is stated as:

"If S is a nonempty subset of a group such that if x, y belong to S then x * INVERSE(y) belongs to S, then S contains INVERSE(x) whenever it contains x."

In order to have a search space of manageable size, for a hand worked example, we leave out the associativity axioms for the group. The true and false lock numbers are shown by writing literals as ordered triples, with the first component being the true lock number. We also write after each clause, "T", "F" or "T/F" to indicate the following.

1. "T" if the clause has only true instances.
2. "F" if the clause has only false instances.
3. "T/F" if the clause has both true and false instances.

The model scheme, M, used for these truth evaluations for this example will be that of simultaneous linear equations (SLE), including inequalities. The following correspondence holds between the language of the clause set, LC, and the language of the model, LSLE:

\[
\begin{align*}
P(x,y,z) & \implies x+y-z=0 \\
I(x) & \implies -x \\
1 & \implies 0 \\
B & \implies B \\
S(x) & \implies x \geq B
\end{align*}
\]
where the constant $B$ in LC is a Skolem function of no arguments introduced by negating the theorem and transforming to CNF, and the unary function $I(x)$ is an abbreviation for INVERSE$(x)$.

The domain of individuals for $M$ will be taken as real numbers, and the symbols in LSLE have their usual meaning in real analysis. There is one piece of ground case information in $M$, which is that $B > 0$. We will not discuss how this model was obtained here.

The clauses for this theorem are:

1. $\langle 10, P(1,x,x), 1000 \rangle$; $FSL = ( ) T$
2. $\langle 9, P(x,1,x), 900 \rangle$; $FSL = ( ) T$
3. $\langle 8, P(x,I(x),1), 800 \rangle$; $FSL = ( ) T$
4. $\langle 7, P(I(x),x,1), 700 \rangle$; $FSL = ( ) T$
5. $\langle 6, S(B), 600 \rangle$; $FSL = ( ) T$
6. $\langle 5, @S(I(B)), 500 \rangle$; $FSL = ( ) T$
7. $\langle 4, @S(x), 1000 \rangle, \langle 3, @S(y), 200 \rangle, \langle 1, @P(x,I(y),z), 300 \rangle, \langle 2, S(z), 400 \rangle$; $FSL = ( ) T/F$

We will perform a breadth first search using HL-Resolution with the SLE model given above, and the indicated lock numbering. No factoring will be performed in order to keep this example as simple as possible. The notation for following the search is that

\[ n Tn x m Tm = r \]

means that clause number $n$, used for its $Tn$ (i.e. true or false) instances resolves with clause number $m$ using its $Tm$ instances, resulting in the resolvent numbered $r$. 


There is only one clause that can be produced at level 1. This is because there exists only one clause with false instances (clause 7), and we must resolve it on its lowest false lock numbered literal, and there is only one other literal in the clause set at this time that can possibly unify with it (the literal in clause 5). Thus we produce the only clause at level 1, namely:

$$5Tx7F = 8. \ <3, \theta S(y), 200>, \ <1, \theta P(B, I(y), z), 300>, \ <2, S(z), 400>,$$

FSL = ($$\theta S(y), \theta P(B, I(y), z), S(z)$$) F

Notice that $$\theta S(B)$$ is not included in the FSL of clause 8 since it is a ground literal and is false in M. Also notice that clause 8 has only false instances in M, as will always be the case in HL-Resolution when one of the parents is a true unit clause.

At level 2 of the search there are two resolvents produced.

$$5Tx8F = 9. \ <1, \theta P(B, I(B), z), 300>, \ <2, S(z), 400>,$$

FSL = ($$\theta P(B, I(B), z), S(z)$$) F

$$7Tx8F = 10. \ <4, \theta S(x), 100>, \ <3, \theta S(y'), 200>, \ <1, \theta P(x, I(y'), y), 300>,$$

$$<1, \theta P(B, I(y), z), 300>, \ <2, S(z), 400>,$$

FSL = ($$\theta S(y), \theta P(B, I(y), z), S(z), \theta P(x, I(y'), y)$$) T/F

Now we start producing the clauses at level 3.

$$3Tx9F = 11. \ <2, S(1), 400>; \ FSL = ( ) F$$

We notice that $$5Tx10F$$ produces no resolvents since the only allowed unification according to the lock numbering, namely letting x in clause 10 unify to B, results in a resolvent whose FSL set
becomes

\[
\text{FSL} = (\text{S}(B), \text{S}(y'), \text{P}(B, I(y'), y), \text{P}(B, I(y), z), \text{S}(z), \text{S}(y))
\]

and the model \( M \) cannot make all of these literals false at the same time, i.e. the resolvent would not represent any ground instances.

It turns out that we have completed level 3 already, since no more pairs of clauses, both at level 2 or below, can resolve. Thus level 3 consists of clause 11. We now start producing level 4.

\[7Tx11F = 12. \quad <4, \text{S}(x), 100>, <1, \text{P}(x, I(1), z), 300>, <2, \text{S}(z), 400>;
\]

\[
\text{FSL} = (\text{P}(x, I(1), z), \text{S}(z)) \quad T
\]

\[7Tx11F = 13. \quad <3, \text{S}(y), 200>, <1, \text{P}(1, I(y), z), 300>, <2, \text{S}(z), 400>;
\]

\[
\text{FSL} = (\text{P}(1, I(y), z), \text{S}(z), \text{S}(y)) \quad F
\]

\[10Tx11F = 14. \quad <4, \text{S}(x), 100>, <1, \text{P}(x, I(1), y), 300>,
\]

\[
<1, \text{P}(B, I(y), z), 300>, <2, \text{S}(z), 400>;
\]

\[
\text{FSL} = (\text{S}(y), \text{P}(B, I(y), z), \text{S}(z), \text{P}(x, I(1), y)) \quad F
\]

\[10Tx11F = 15. \quad <3, \text{S}(y'), 200>, <1, \text{P}(1, I(y'), y), 300>,
\]

\[
<1, \text{P}(B, I(y), z), 300>, <2, \text{S}(z), 400>;
\]

\[
\text{FSL} = (\text{S}(y), \text{P}(B, I(y), z), \text{S}(z), \text{P}(1, I(y'), y)) \quad T/F
\]

This concludes level 4 of the search.

Notice that both clauses 14 and 15 contain the literal \( \text{S}(y) \) in their FSL sets, and this literal came from the FSL set of clause 10. This literal is one of the reasons why clause 14 only has false instances. Without that literal clause 14 would be classed "T/F". It is worthwhile here to interrupt the search briefly to
show what the model evaluation is like for clause 14, for our chosen model M.

We write the FSL and standard literals, each negated, in the language of the model, for clause 14. This gives 8 conditions, one for each literal.

1. \( x \geq B \)
2. \( x-0=y \)
3. \( B-y=z \)
4. \( z < B \)
5. \( y \geq B \)
6. \( B-y=z \)
7. \( z < B \)
8. \( x-0=y \)

This is a consistent set of conditions, so M returns false as its evaluation. In order to show that there are no true instances requires more work however. We see that the 2nd, 3rd and 4th standard literals of clause 14 can never be true, since they are represented in the FSL set. But we are not sure about the first literal. To make a decision we translate the negations of the FSL literals and the first literal (directly) into the language of the model. This gives

1. \( x < B \) from \( @S(x) \)
2. \( y \geq B \) from \( @S(y) \)
3. \( B-y=z \) from \( @P(B,I(y),z) \)
4. \( z < B \) from \( S(z) \)
5. \( x - 0 = y \) from \( @P(x, I(1), y) \)

where the first condition is the direct translation of the first literal in clause 14, and the rest are translations of the negations of the clause 14 FSL literals, as shown on the right. We see that the 5th condition says \( x = y \), and this causes the first two conditions to be contradictory. From this we conclude that the set of substitutions that make all of the FSL literals simultaneously false are disjoint from the set of substitutions that make \( @S(x) \) true. Therefore the set of ground instances which clause 14 stands for all have their first literal (i.e. \( @S(x) \), grounded) false. We already knew that the other literals must be false since they were exactly represented in the FSL of clause 14. Thus clause 14 represents only false ground instances.

The above explanation of how the SLE model works is on an intuitive level. For automatic theorem proving this model must be implemented as a decision procedure for the class of expressions we are interested in. As an example of such a decision procedure see Cooper (Cooper, 1972). A somewhat different procedure for the SLE model is currently being implemented for experimental purposes, but it is not yet certain if it is actually a decision procedure.

We now pick up the search process again at the point we left off, starting level 5.

\[ 11F \times 12T = 16. \quad <1, @P(1, I(1), z), 300>, <2, S(z), 400>; \]
\[ \text{FSL} = (@P(1, I(1), z), S(z)) \]
5Tx13F = 17. \( <1, \theta \Pi(1, I(B), z), 300>, <2, S(z), 400> \);
\[ \text{FSL} = (\theta \Pi(1, I(B), z), S(z)) \]

7Tx13F = 18.

In order to keep this example as clean as possible we will not write out the forms of the clauses, unless they are involved in the proof, or are of other interest, from this point onward.

5Tx14F = 19.
7Tx14F = 20.
11Fx15T = 21.
5Tx15F = 22.
7Tx15F = 23.

This concludes level 5, and we start level 6.

1Tx16F = 24.

At this point, clauses 3T and 18F resolve to give

\( <2, S(1), 400>; \text{FSL} = (\ ) \)

which is a duplicate of clause 10. In general it could be difficult to make decisions about subsumption possibilities or to determine if two clauses which differ only in their FSL sets are equivalent, without incurring a high computational load. We would expect however, that a reasonable implementation of HL-Resolution would detect the identity of this 3Tx18F resolvent and clause 11. Thus we do the same in this example here, and delete this clause. The next clause to be produced is
Clause 25 is a unit, and we assume that there is a one step unit look-ahead in the search process. Thus, since clause 25 resolves with clause 6

we have found a proof. The FSL is an important part of the HL-null clause. If a null clause is formed during an HL-Resolution search but the FSL requirement has no ground instances for which the clause stands, then a proof has not been found of the required form for HL-Resolution.

It should be emphasized that the 26 clauses (7 input and 19 generated) constitute the complete breadth first search space (without factoring) of HL-Resolution for this problem. The null clause appeared at level 7, and this was detected when the second clause at level 6 was generated (i.e. clause 25).

This same clause set was also submitted to a breadth first normal resolution theorem prover, and a proof was obtained after reaching a search space size of 165 clauses. The theorem prover used the one step look ahead for generated units, and a special type of factoring which produces fewer factored clauses than the usual factoring. The null clause was obtained at level 4, and this was seen after generating one clause at level 3, by the look ahead for units. The total number of clauses at the end of level 2 was 163 clauses. Thus the search was growing rapidly in the number of
clauses per level already at level 2. An estimate of the number of clauses contained in level 3 is 1300 clauses.

The search was also tried by the same theorem prover but without any factoring at all. This time the proof was obtained after generating 224 clauses. Again, a unit look ahead found the proof at level 4 before level 3 was finished. This time 148 clauses were produced at level 3 before the look ahead discovered the proof.

It is seen that the HL-Resolution search generates far fewer clauses at any given level. However the null clause was 3 levels deeper in HL-Resolution than in unrestricted resolution. To make a more valid comparison between HL-Resolution and unrestricted breadth first, with factoring (BF-FAC) and without (BF), we compare the number of clauses generated at the time a proof was detected, and the number of clauses contained in the search space at the level two levels before the proof level.

<table>
<thead>
<tr>
<th>type of resolution</th>
<th>number of clauses at proof time</th>
<th>level of proof</th>
<th>number of clauses 2 levels before the proof level</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLR</td>
<td>26</td>
<td>7</td>
<td>8(level 5)</td>
</tr>
<tr>
<td>BF-FAC</td>
<td>165</td>
<td>4</td>
<td>147(level 2)</td>
</tr>
<tr>
<td>BF</td>
<td>224</td>
<td>4</td>
<td>63(level 2)</td>
</tr>
</tbody>
</table>

We see that HL-Resolution decreased the number of clauses by about a factor of eight, for this particular problem.
3.3 Some Comments Concerning HL-Resolution Searches
and Other Resolution Searches

Resolution developed rapidly as a theorem proving method through the development of a large number of strategies. Most of these strategies were syntactically oriented, as was the basic resolution procedure. The outstanding feature of these strategies was their lack of uniformity in application effect. Any given strategy might help the search tremendously on one specific problem, and help very little, or become even detrimental when used on a slightly different problem. Clearly these strategies were not sufficiently sensitive to what was happening in the search space.

The two fundamental problems with resolution methods in general are:

1. the lack of any effective techniques addressing themselves to the Term Substitution Problem (TSP);
2. the restriction to local interaction in the search space (LIP, for Local Interaction Problem).

The term substitution problem refers to the fact that in first order logic (and therefore in first order resolution) the semidecidability (as opposed to decidability) has its origin in the inability to detect the non-existence of a grounding substitution which makes a set of clauses an unsatisfiable set of ground clauses. This means that the real procedural search, i.e., the part of the search that cannot both be bounded in the amount of work performed and still maintain completeness of the search, is
the search over the Herbrand universe.

Most resolution strategies have no adequate method for directly handling the TSP. In fact, they hide the problem by prescribing the use of the most general unifier. This mgu is automatically called forth, as a result of which clauses, and which literals within these clauses, are chosen by the search strategies. These strategies are typically insensitive to the TSP, and when they are sensitive to it, they seem to be sensitive to the wrong aspect of it. An example of a strategy which is insensitive to the TSP is LR. Strategies which prevent the formation of clauses which exceed a given bound on the depth of function nesting are an example of strategies that are sensitive to the wrong aspect of the TSP. A fundamental shortcoming of the use of the mgu in resolution steps is that unification is essentially a two literal process, while the resolution step involves all of the literals in the parent clauses.

Even when the proper terms appear in a resolution search, the sentential connections necessary are only found after, usually, many additional clauses with new irrelevant, or redundant, terms are generated. Thus resolution tends to be term intensive (and indiscriminately so) rather than truth table intensive.

We identify, then, two deficiencies in the way resolution handles the TSP. The first is the promiscuous generation of substitution instances, and the second is the lack of address in following the sentential consequences of terms shortly after their formation. These two problems are very closely connected, but not
inseparably so. We will return to these two aspects of the TSP after a brief discussion of the local interaction problem.

The local interaction problem (LIP) is, for large theorem proving problems, the single most important factor causing resolution search spaces to grow as fast as they do. There are two recognizable aspects to this problem. The first is the pure syntactic redundancy of a sentential nature which occurs whenever the search procedure fails to have the property called singly connectedness (Wos et al., 1967). Singly connectedness is almost achieved by Lock Resolution and this aspect of the LIP can be considered reasonably well handled by resolution strategies that include LR within themselves.

The second aspect of the LIP is the lack of adequate communication of partial search results from one part of the search space to other parts. Typically, when a resolvent is produced, the only influence it has is through subsumption of or by other clauses, or through direct resolution with other clauses. Both of these modes of interaction, but particularly resolutions, become impractical in a search which is combinatorially growing. What is needed is some form of interaction that uses information in a clause, immediately upon the generation of the clause, to simplify the rest of the search space. This interaction could be semantic, syntactic, or both.

HL-Resolution does something about both the TSP and the LIP which most other resolution strategies do not. It avoids producing new term substitutions at the high rate that other strategies do.
Looking back at the hand worked example of section 3.2, we see that HL-Resolution produced no new terms at all of higher function nesting depth than that which existed in the input set. This is in distinction to the unrestricted breadth first resolution searches with which it was compared, which by level 2 had already introduced a term with 2 additional levels of function nesting. To whatever degree HLR retards the introduction of new term classes in the search it must be instead concentrating effort in the direction of exploring the sentential connections of term classes already in the search space. This is true in a weak sense only, however. The problem here is that we really need to achieve a much stronger characterization (and concomitant ability to control) the ground instances that a general level clause represents. At present HLR controls the allowed ground instances according to a truth characterization evaluated by a model. There is reason to believe that HLR can be extended (semantically and syntactically) so as to exert even more control over the form of the search, and it is expected that such extensions will specifically help with the TSP. The whole issue of proper term substitution is a delicate one since semidecidability of the first order logic limits what can be done, but yet it is clear that current methods are not yet optimal within this limit.

The LIP is handled in a partial manner by HL-Resolution as formulated. An HL-clause carries with itself, implicitly in the information in the FSL set, a notion of where it came from. Remember that the model, M, is used in a manner similar to the way it would be used in TMS. However, TMS is quite sloppy with respect
to just what instances of a clause are the relevant ones in a resolution step. This gives rise in TMS to a search space which at the general level may locally make sense, but globally is inconsistent with the central idea behind TMS. In particular it is possible, using TMS and a model, M, to find a refutation for a set of clauses such that

1. The general level proof tree satisfies The Model Strategy as formulated in Luckham (Luckham, 1968), using M.
2. There does not exist any grounding substitution of the proof tree such that the ground level tree formed satisfies The Model Strategy using M.

In HL-Resolution it is the FSL set, in conjunction with the model that prevents such a situation from occurring (in fact, the lock numbers are also involved in this issue, but only accidentally, i.e. TMS could have this defect corrected without the utilization of lock numbers). Thus HL-Resolution introduces a strong global constraint on the nature of the underlying ground proof trees that it is searching over at the general level. This is accomplished by including in the FSL set the information which is relevant concerning how that clause was derived. An additional effect of doing this is that it helps reduce the number of term instances that can be formed.

Unfortunately, the real issue in the LIP, that of relating information in one part of the search rapidly and effectively to the rest of the search has not been handled very well at all. There is some weak immediate information transfer in the sense that
3.3

A resolvent, when generated, may result in a change in the model, which then affects all future model evaluations. As in the case of the TSP, there are indications here also that HL-Resolution can be extended so as to handle the LIP more effectively.

This section has forced a somewhat unnatural conceptual distinction between the TSP and the LIP. They are both really reflections of a single underlying difficulty in resolution theorem proving, which is that there is strong context dependency in the structure of a resolution search space which is not being adequately handled. HL-Resolution is an initial attempt, through the use of FSL's which are evaluated by a model, to increase the context size, or scope, of the basis on which refinement strategy decisions are made.
3.4 Definition of the Basic HL-Resolution Refinement Strategy

This section defines the basic HL-Resolution refinement strategy in a precise fashion, and is not oriented toward an implementation viewpoint and description. Notice that in this section in order to make things precise, we modify the notion of HLR clauses and clause sets. Input clause sets are transformed into primitive representation HL-clause sets. Doing this clarifies both the formal definition of the HLR refinement and the proof of its completeness. The refinement strategy that results is referred to as the basic HLR strategy. This strategy is defined in this section in terms of primitive representation clauses. It is important to distinguish between the phrases "basic HLR strategy", and "primitive representation clauses". The former is a refinement strategy, while the latter refers to a notational scheme in which this section presents the basic HLR strategy. Later in this report (near the end of section 3.5) we indicate how the notions of HLR stated here relate to the description of HLR given in sections 3.1 and 3.2.

An example of many of the definitions used in this and the next section are to be found in the Appendix.

The usual notions of resolution are assumed familiar (e.g. treating clauses as sets (and as lists) of literals, most general unifier (mgu), and the notions of Herbrand universe and base). We extend these notions in an obvious way implicitly in this section (e.g. the mgu of two HL-literals is the mgu of their second components). We define a Herbrand interpretation to be a set of
ground literals such that every literal is either an element of the Herbrand base or the negation of an element of the Herbrand base, and every element of the Herbrand base appears in the interpretation exactly once, either directly or with a negation sign. If a literal is an element of a given Herbrand interpretation then it is said to be true in that interpretation. Otherwise it is said to be false in that interpretation.

**HL-Literal**

An HL-literal is a 3-tuple whose first and third components are integers. The second component is a normal literal. We refer to the assignment of integer values to the first and third components of HL-literals as a lock numbering. Any lock numbering of a set of HL-literals in which all the first component values are smaller than all of the third component values is called a HL-proper lock numbering. If all of the integers used in a HL-proper lock numbering are distinct, the numbering is referred to as unambiguous. We refer to an HL-literal as just a literal, and specifically use the word "normal" when referring to normal literals. The first component of a literal is called the true lock number, and the third component is called the false lock number. The selector functions for true and false lock numbers are t and f, respectively. For all definitions which define new clauses in terms of old clauses (e.g. resolving two clauses, or factoring a clause) we assume that the old clauses have been replaced by copies of themselves in which new unique variable names have been substituted.
A literal is a ground literal if no variable appears within it. We define the truth value of a ground literal with respect to a Herbrand interpretation, $M$, to be the same as the truth value of its second component.

**HL-Clause, Standard Literals, False Substitution Literals**

An HL-clause, $C$, is an ordered pair of sets of literals. The first set is denoted by $SD(C)$ and elements of $SD(C)$ are called standard literals of $C$. The second set is denoted $FSL(C)$ and elements of $FSL(C)$ are called False Substitution Literals of $C$. We write

$$C = <SD(C),FSL(C)>$$

as an obvious identity. Both standard literals and FSL literals are HL-literals. This report makes no specific use of the lock numbers of the FSL literals.

A grounding substitution, $G$, for a literal $L$, is any substitution such that $L(G)$ is a ground literal. A grounding substitution, $G$, for a set of literals, $S$, is any substitution that qualifies as a grounding substitution for each element of $S$. A grounding substitution, $G$, for an HL-clause, $C$, is any substitution that qualifies as a grounding substitution for $SD(C)$ and for $FSL(C)$. If $G$ is a grounding substitution for the HL-clause, $C$, then $C(G)$ is a ground HL-clause.
In what follows M will designate an arbitrary but fixed Herbrand interpretation. We define an evaluation function, also denoted M, for the Herbrand interpretation M, such that if L is a set of literals

\[ M(L) = \begin{cases} 
T & \text{if for all grounding substitutions, THETA, of } L, \\
& \text{at least one literal of } L(\text{THETA}) \text{ is true in } M. \\
F & \text{otherwise} 
\end{cases} \]

If L happens to be the set of normal literals of a normal clause, then the above definition of M is usually what is meant by a truth evaluation of a clause in resolution strategies (e.g. in TMS).

**Feasible Clauses**

A HL-clause, C, is said to be feasible (with respect to M), iff \( M(\text{FSL}(C)) = F \). A HL-clause is infeasible iff it is not feasible.

**HL-Null Clause**

An HL-null clause is any feasible HL-clause, C, such that SD(C) is empty.
Exact Primitive Representation

For a given HL-clause, C, we say that the HL-clause k is an exact primitive representation (with respect to M) of C iff:

1. SD(k) = SD(C)
2. .FA. 1: 1 ∈ FSL(C) --- > 1 ∈ FSL(k)
3. .FA. 1: 1 ∈ SD(k) --- > [ 1 ∈ FSL(k) OR θ1 ∈ FSL(k) ]
4. .FA. 1: 1 ∈ FSL(k) --- > [ θ1 ∈ SD(C) OR ( 1 ∈ SD(C) U FSL(C) ) ]
5. k is feasible

An exact primitive representation is an HL-clause. An HL-clause, k, that fails to satisfy condition 4, but satisfies the other conditions is called an inexact primitive representation of C. When it is immaterial as to whether a primitive representation is exact or inexact, it will be called just a primitive representation. Notice that if k is a primitive representation of C, then k is an exact primitive representation of k.

Exact Primitive Representation Set

The exact primitive representation set (with respect to M) of C, is defined to be the set of all exact primitive representations of C.
M-Evaluations of HL-Clauses

We now extend the domain of the model evaluation function, M, previously defined on sets of literals, to include also feasible HL-clauses, as follows.

Let C be a feasible HL-clause, and let G(x,y) be the relation which is true when x is a grounding substitution for the HL-clause y. Then

\[
M(C) = \begin{cases} 
T & \text{if } .FA. \Theta: [G(\Theta,C) \\
& \text{and } M(FSL(C)(\Theta)) = F] \\
& \implies [M(SD(C)(\Theta)) = T] \\
F & \text{if } .FA. \Theta: [G(\Theta,C) \\
& \text{and } M(FSL(C)(\Theta)) = F] \\
& \implies [M(SD(C)(\Theta)) = F] \\
T/F & \text{otherwise}
\end{cases}
\]

The above definition is for HL-clauses in general. Notice that if the clause C is a ground clause (i.e. SD(C) and FSL(C) are ground sets of literals), or if C is a primitive representation clause, then M(C) will never be "T/F".

Selected True Literal

Let k be a primitive representation of C, then for each literal l such that

1. \( l \in SD(k) \)
2. \( \Theta l \in FSL(k) \)
3.4

3. .FA. m: [m ∈ SD(k) AND

\[ \emptyset m \in FSL(k) \] --- > t(l) ≤ t(m) \]

we say that l is a selected true literal of k.

Selected False Literal

If k is a primitive representation of C, and

.FA. m: m ∈ SD(k) --- > m ∈ FSL(k)

then for each literal l such that

1. l ∈ SD(k)
2. .FA. m: m ∈ SD(k) --- > f(l) ≤ f(m)

we say that l is a selected false literal of k.

Selected Literal

Let k be a primitive representation HL-clause. If M(k) = T then a selected literal of k is any selected true literal of k. If M(k) = F then a selected literal of k is any selected false literal of k.

Selected Factor

Let k be a primitive representation HL-clause with a selected literal L whose second component is L'. If there exists a set of one or more literals L1, L2, ..., each distinct from L and each in SD(k), and with second components L1', L2', ..., respectively, such that OMEGA is the mgu of the set of literals
Let \( L', L_1', L_2', \ldots \), then the clause

\[
F = \langle (SD(k) - L_1 - L_2 \ldots), FSL(k) \rangle(\Omega)
\]

is a selected factor of \( k \) on selected literal \( L \) iff \( F \) is feasible. \( \Omega \) is called the factoring unifier. A reduced selected factor, \( RF \), of \( k \) on selected literal \( L \) is any selected factor of \( k \) on selected literal \( L \) such that there is no literal in \( SD(RF) \) whose second component is identical to the second component of \( L(\Omega) \), except \( L(\Omega) \) itself, where \( \Omega \) is the factoring unifier.

All selected factoring is assumed to be reduced, and will be referred to simply as selected factoring. Notice that, aside from the literals factored out, the factoring unifier may also cause two or more standard literals to have identical second components, but with none of them equal to the second component of the selected literal on which factoring is being performed. These other literals are all retained in the factored clause.

Let \( C \) be a primitive representation clause, and \( L \) a selected literal of \( C \). If there is no other literal in \( SD(C) \) with a second component identical to the second component of \( L \), then \( C \) is said to be a (reduced) selected factor of itself on selected literal \( L \).

**Primitive Representation Binary Resolvent**

Let \( C \) and \( D \) be two primitive representation clauses. Let \( LC \) be a selected false literal of \( C \), and \( LD \) a selected true literal of \( D \), such that \( LC \) and \( LD \) are unifiable with \( mgu \) \( \Theta \). Then the clause \( R \) is a primitive representation binary resolvent (on
literals LC and LD, and with respect to model M) of C against D iff:

1. SD(R) = ((SD(C)-LC) U (SD(D)-LD))(THETA)
2. FSL(R) = (FSL(C) U FSL(D))(THETA)
3. R is feasible in M.

C is called the false parent of R, and D is called the true parent of R.

**Basic Primitive Representation Deduction**

We define a basic primitive representation deduction (with respect to M) of R from S to be a finite binary rooted tree with:

1. all of its leaves being primitive representation clauses in S;
2. each internal node a primitive representation binary resolvent of selected factors of the two nodes above it, with the condition that the two literals resolved upon in the resolution are the selected literals actually used as the literals on which the selected factoring occurred;
3. R at the root.

A basic primitive representation deduction from S of an HL-null clause is called a basic primitive representation refutation of S.
HL-Associated Input Set

Let $S'$ be a set of HL-clauses, and $S$ a set of normal clauses. Then $S'$ is called the HL-associated input set (with lock numbering LN) for $S$ iff:

1. LN is a HL-proper lock numbering of $S'$;
2. there is a 1-1 mapping, $C$, from $S$ to $S'$, and a 1-1 mapping $L$, from normal literals of $S$ to literals of $S'$ such that for all $x \in S$ and for all $k$ we have:
   a) $L(k) \in SD(C(x))$ iff $k \in x$
   b) $k \in x \implies k$ equals the second component of $L(k)$;
3. for all $u, u \in S' \implies FSL(u)$ is empty.

Exact Primitive Representation HL-Associated Input Set

The exact primitive representation HL-associated input set of $S$ is a set, HLS, of exact primitive representation clauses formed by replacing each clause, $C$, in the HL-associated input set for $S$ by the exact primitive representation set of $C$. In HLS there will in general be many pairs of clauses that share the same lock numbers. If the literals in clauses in HLS have their lock numbers re-assigned so that HLS has an unambiguous HL-proper lock numbering, then we say that HLS has been disambiguated.

This completes the set of definitions needed for the basic HLR strategy expressed in primitive representation form. The next section states and proves the soundness and completeness theorem for this strategy.
3.5 Soundness and Completeness of HL-Resolution

This section first states a soundness and completeness theorem for basic primitive representation HL-Resolution, and then gives a proof of this theorem. Finally it is indicated how the basic primitive representation results are related to the basic non-primitive representation HL-Resolution. Refer to the Appendix for an example of some of the notions used in this section.

THEOREM
(Soundness and Completeness of Primitive Representation Deduction)

Let $S$ be a set of normal clauses, and let $M$ be an arbitrarily chosen Herbrand interpretation in the language of $S$. Let $HLS$ be the exact primitive representation HL-associated input set (with HL-proper lock numbering LN) of $S$, with respect to $M$. Then $S$ is unsatisfiable iff there exists a basic primitive representation refutation (with respect to $M$) of $HLS$. 
By normal resolution will be meant unrestricted binary resolution with implicit factoring. Normal resolution is known to be sound and complete.

**soundness**

It is to be shown that if $S$ is satisfiable, then there exists no basic primitive representation refutation of HLS. The proof is by contradiction. We assume both the satisfiability of $S$ and the existence of a basic primitive representation refutation of HLS, which we denote by $R$-HLS. We form the tree $R'$ from the tree $R$-HLS by replacing each clause in $R$-HLS by the set (actually list) of second components of its standard literals. By looking at the definitions of basic HLR at the primitive representation level, it should be clear that the new tree, $R'$, is in fact a normal resolution refutation of $S$. But normal resolution is sound, so that we have the contradiction. Thus basic primitive representation resolution is sound.

**completeness**

For ease of reference the characteristics of the clause sets and other items involved in the completeness proof are listed here.

$S$ is a set of normal general level clauses which is unsatisfiable.
SG is a set of normal ground clauses which are ground instances of clauses in S, and is minimally unsatisfiable.

GR(S,SG) is a list of sets of substitutions which, when applied to S yields SG.

HLS is a set of general level exact primitive representation clauses, which is the exact primitive representation HL-associated input set of S (with respect to model M, and with HL-proper lock numbering LN). We assume for now that HLS is not ambiguously numbered. Later this requirement is dropped.

HLSG is a set of ground exact primitive representation clauses formed by applying GR(S,SG) to HLS.

SGL is a set of ground lock normal clauses identical to SG except that lock numbers have been added to the literals in SGL in a specific way relative to the structure of HLSG.

MLR is a Modified form of Lock Resolution.

R-SGL is a refutation of SGL according to MLR.

R-HLSG is a ground basic primitive representation refutation of HLSG.

R-HLS is a general level basic primitive representation refutation of HLS.
The completeness proof requires that we establish the existence of a basic primitive representation refutation of HLS based upon the unsatisfiability of $S$, and that this be done for an arbitrary Herbrand interpretation, $M$, as the model, and for an arbitrary HL-proper lock numbering, $LN$, of HLS. The notion of satisfiability of HLS is neither defined nor used in this proof.

Let $S$, HLS, LN and M be as stated in the hypothesis of the theorem. For the completeness proof we further assume that $S$ is unsatisfiable. We will consider $S$ to be a non-ground set of clauses and state the proof for that situation. If $S$ is a ground set of clauses then a large part of the proof given here is superfluous, and the reader should be able to identify which parts are relevant to treating the ground case. The following is an outline of the proof steps.

1. Preliminary definitions and lemma statements.

2. Herbrand's theorem applied to $S$ yields a ground set $SG$, which is minimally unsatisfiable.

3. A set of ground clauses, HLSG, each of which is a strict grounding of a clause in HLS, is defined and related to clauses in $SG$. A lock numbered set of clauses, SGL, is formed from $SG$.

4. A Modified form of Lock Resolution (MLR) is defined and applied to SGL to yield a refutation of SGL, called R-SGL. R-SGL is then transformed into a basic primitive representation refutation of HLSG, called R-HLSG.
5. The Lifting Lemma is used to transform R-HLSG into a general level refutation, R-HLS, of HLS which is a basic primitive representation refutation of HLS.

6. The Lifting Lemma is proved.

In the completeness proof it is best to think of clauses as "bags" or "heaps" of literals, as opposed to ordinary sets of literals.

Step 1.

The following lemma will be used in step 6 in the proof of Lemma IV.

**Lemma I**

Let \( L \) be a set of literals

\[
A_1, A_2, \ldots, A_m, B_1, B_2, \ldots, B_n
\]

and \( \Sigma \) a grounding substitution for \( L \) such that

\[
A_1(\Sigma) = A_2(\Sigma) = \ldots = A_m(\Sigma).
\]

If \( \Lambda \) is the mgu of the set of literals \( A_1, A_2, \ldots, A_m \), then there exists a substitution, \( \rho \), such that \( \Sigma = (\Lambda)(\rho) \).

We do not prove this lemma.
Strict Grounding

Let $U$ be a primitive representation clause and $V$ a ground primitive representation clause. Then $V$ is a strict grounding of $U$ under substitution $\rho$, relative to the Herbrand interpretation $M$, iff all of the following conditions are met:

1. $V$ is feasible with respect to $M$.
2. There exists a total 1-1 mapping, $g$, from $\text{SD}(U)$ onto $\text{SD}(V)$ such that for all $k \in \text{SD}(U)$ it is the case that $g(k)$ is identical to $k(\rho)$ in all three components.
3. There exists a total 1-1 mapping, $h$, from $\text{FSL}(U)$ onto $\text{FSL}(V)$ such that for all $k \in \text{FSL}(U)$ it is the case that $h(k)$ is identical to $k(\rho)$ in all three components.

If a literal of $U$ and a literal of $V$ map to each other under $f$ or $g$, then those two literals are said to be naturally associated (by $g$ or by $h$).

**LEMMA II**

Let $x$ be a strict grounding of the primitive representation clause $X$ under substitution $\tau$. If $x'$ is a selected true (false) literal of $x$, then for every literal $X'$ in $X$ such that $X'(\tau) = x'$ (i.e. equality for all three components), it is the case that $X'$ is a selected true (false) literal of $X$. Furthermore there will exist at least one $X'$ in $X$ such that $X'(\tau) = x'$. 
PROOF OF LEMMA II

Case 1: $x'$ is a selected true literal in $x$.

The proof is by contradiction, where we assume $x'$ is a selected true literal of $x$ and that there exists some $X'$ in $X$ such that $X'(\tau) = x'$, and $X'$ is not a true selected literal of $X$. If $x'$ is a selected true literal in $x$, then $\theta x'$ is an element of the FSL of $x$, and by the definition of a strict grounding there exists at least one literal $\theta X''$ in the FSL of $X$ such that $\theta X''(\tau) = \theta x'$, and at least one standard literal $X''$ such that $X''(\tau) = x'$, where equality between literals here means that they are identical in all three components. Let $X'$ be any specific literal in $X$ such that $X'(\tau) = x'$. By the definition of a strict grounding, $x$ must be feasible. This means that $\theta X' \in \text{FSL}(X)$. Assume now that $X'$ is not a selected true literal of $X$. By the definition of a selected true literal the only way that $X'$ can fail to be a selected true literal of $X$ is if there exists a standard literal $L$ in $X$ such that $\theta L$ is in the FSL of $X$, and the true lock number of $L$ is less than the true lock number of $X'$. But if this were the case, then there would exist a standard literal, $L(\tau)$, and an FSL literal, $\theta L(\tau)$, both in $x$, and the true lock number of $L(\tau)$ would be less than the true lock number of $x'$. But then $x'$ could not be a selected literal of $x$. Therefore we have the contradiction, and $X'$ must be a selected literal of $X$.

Case 2: $x'$ is a selected false literal in $x$.

Similar to case 1.
Immediate Primitive Representation Deduction

Let \( w, u \) and \( v \) be primitive representation clauses. An immediate primitive representation deduction of \( w \) from \( u \) and \( v \) is a basic primitive representation deduction tree with \( u \) and \( v \) as leaves, and \( w \) at the root, and containing no other clauses. If there is an immediate primitive representation deduction of \( w \) from \( u \) and \( v \), then \( w \) is said to be immediately deducible from \( u \) and \( v \).

Ground Normal Lock Image

Let \( x \) be a ground clause that consists of a set of ground normal literals with lock numbers, i.e. each literal is an ordered pair in which the first component is a normal ground literal, and the second component is a number. Let \( X \) be a ground primitive representation clause. Then \( x \) is a ground normal lock image of \( X \) iff there exists a function, \( g \) (not necessarily unique), which is a 1-1 total mapping from standard literals of \( X \) onto literals of \( x \) such that the following holds:

1. \( L \in \text{SD}(X) \) and \( L \in \text{FSL}(X) \)
   \[
   \quad \Rightarrow \quad g(L) = \langle \text{second component of } L, f(L) \rangle
   \]
2. \( L \in \text{SD}(X) \) and \( \theta L \in \text{FSL}(X) \)
   \[
   \quad \Rightarrow \quad g(L) = \langle \text{second component of } L, t(L) \rangle
   \]

where \( t \) and \( f \) are the selector functions for the true and false lock numbers of HL-literals. The function \( g \) is called the ground normal lock function.
The next lemma uses the MLR, which is defined in step 4 of the completeness proof.

**LEMMA III**

Let X and Y be ground primitive representation clauses, and x, y be ground normal lock images of X and Y respectively. If there exists a clause z, which is a resolvent of x and y according to MLR, then there exists an immediate primitive representation deduction of a clause, Z, from X and Y, such that z is a ground normal lock image of Z.

**PROOF OF LEMMA III**

We do not give a detailed proof here, but just indicate how to proceed.

First we realize that some specific model, M, must exist, in order for X and Y to be well defined. This is because a primitive representation clause is by definition a feasible clause, and so is defined relative to some model. Also we can assume a ground normal lock function, G1, exists (not necessarily unique) which associates literals from X to x, according to the definition of a ground normal lock image. Similarly for a G2 associating literals from Y to y. Then, for the z which exists by the lemma hypothesis, we can construct a set of HL-literals (using G1 and G2) to represent the standard literals of clause Z. An FSL set for the clause Z is simply the union of the FSL sets of X and Y. Now it is necessary to show that this constructed Z actually can be at the root of an
immediate primitive representation deduction with X and Y as leaves. This requires checking the definitions of MLR and HLR and seeing that everything done to construct z from ground normal locked literals in x and y according to MLR, is also allowed in HLR to produce Z from the corresponding ground HL-literals in X and Y. Specifically the factoring of MLR corresponds to the reduced selected factoring of HLR on ground clauses. Also, every selected literal in x under MLR is mapped to a selected literal of X in HLR by the function G1. Similarly for G2 (if we were doing this in detail this would require a lemma statement similar to Lemma II). Finally the constructed Z must be feasible since it has an FSL which is just the union of FSL sets which consist of only ground literals, and which came from clauses that were feasible.

**LEMMA IV**

*(ground to general level lifting lemma)*

Let X, Y, x, y, z be primitive representation clauses, in which X and Y have their variables standardized apart, and with x a strict grounding of X under substitution TAU, and y a strict grounding of Y under substitution NU. If it is the case that there exists an immediate primitive representation deduction of z from x and y, then there exists an immediate primitive representation deduction of a clause Z, from X and Y, such that z is a strict grounding of Z.

PROOF OF LEMMA IV (see step 6)
Step 2.

S is an unsatisfiable set of normal clauses, and we assume that the clauses in S have their variables standardized apart. Consider S as ordered so that we may speak of the k-th clause in S.

By Herbrand's theorem (Chang and Lee, 1973) there exists a set of ground clauses, SG, each of which is an instance of a clause in S, and SG is minimally unsatisfiable (in particular SG does not contain two clauses which are identical). The substitutions which convert S to SG are denoted by GR(S,SG). GR(S,SG) is a list of sets of substitutions, such that the k-th element of GR(S,SG) is a set of j(k) distinct substitutions to be applied to the k-th clause in S, generating j(k) distinct ground instances of that clause in SG. We do not delete duplicate literals in any clause in SG. A clause x in S is said to be naturally associated with a clause y in SG iff x is the k-th clause in S, for some k, and x(TAU) = y for some TAU in the k-th element of GR(S,SG). The symmetric relation "naturally associated" defines a function which is many-one from SG into S, and is total on SG. If x in S and x' in SG are naturally associated, we define an extension of the naturally associated relation to include a natural association of literals L in x and L' in x' in the obvious way. The natural association of literals in x and x' defines a total 1-1 mapping from x onto x'.

Notice that Herbrand's theorem does not imply a unique SG. We choose any minimally unsatisfiable one for the completeness proof. Having S and a specific SG still does not necessarily give a unique GR(S,SG). We are free to choose any specific GR(S,SG) for the
completeness proof. Having SG and GR(S,SG) specific allows a unique relation of naturally associated to be defined on pairs of clauses, one in S and one in SG. However the naturally associated relation on pairs of literals is still not unique, and we are free to choose any specific one.

Step 3.

We are now going to define a set, HLSG, which is a set of ground exact primitive representation clauses. Each clause in HLSG will be a strict grounding of some clause in HLS.

Assume that the k-th element of GR(S,SG) is non-empty. Let c be the k-th clause in S, and EPRS(c) the set of exact primitive representations in HLS for the normal clause c in S. Notice that this is actually a two stage connection. First there is a unique clause, c', in the HL-associated input set of S, which is the non-primitive representation HL-clause corresponding to c. Then EPRS(c) is the exact primitive representation set of c'. Any element of EPRS(c) is said to correspond to c. Assume each exact primitive representation clause in EPRS(c) uses the same variable names as are used in c. Let RHO be an element of the k-th list element of GR(S,SG). Then applying RHO to each member of EPRS(c) will result in precisely one feasible ground clause, which we denote by apply(RHO, EPRS(c)). The clause apply(RHO, EPRS(c)) will be considered as a strict grounding, i.e. no deletion of identical literals will be done.
The set HLSG is the set of ground exact primitive representation clauses which are generated by applying all substitutions contained in all of the elements of GR(S,SG) to the appropriate clauses in HLS. Each clause in HLSG is ground, is an exact primitive representation clause, and is a strict grounding of a clause in HLS.

Again we extend the naturally associated relation, this time to relate normal clauses (and normal literals) in SG to clauses (and standard literals) in HLSG. We relate the clause \( x \) in SG to the clause \( y \) in HLSG iff there exists clauses \( u, v \) and substitution \( \Omega \) with the following properties:

1. there exists a \( k \) such that \( u \) is the \( k \)-th clause of \( S \), \( \Omega \) is an element of the \( k \)-th element of \( \text{GR}(S,SG) \), and \( u(\Omega) = x \).

2. \( v \) is an exact primitive representation clause in HLS which corresponds to \( u \), and \( v(\Omega) = y \).

The pairing of clauses between SG and HLSG is unique (if a specific choice has been made for \( \text{GR}(S,SG) \)). This pairing of clauses by the naturally associated relation constitutes a 1-1 total function between SG and HLSG.

The naturally associated relation between literals in clauses which are naturally associated is defined in the obvious way by noting that a clause in SG is identical to the set (actually "bag") of second components of the standard literals of a clause with which it is naturally associated in HLSG. Thus for \( x \) in SG, and \( y \)
in HLSG, if $x$ and $y$ are naturally associated then there will exist a total 1-1 mapping of literals from $x$ to literals of $y$. The pairing of literals between SG and HLSG is not unique. We arbitrarily choose any one of them for the completeness proof. We note here that this lack of a unique extension of the naturally associated relation on literals is not essential, and that by keeping careful track of the identity (or source) of literals in $S$, SG, HLS, and HLSG, we could have arrived at this point in the proof with a unique pairing of literals. This was deemed an unnecessary complication since the completeness proof does not require it.

Now we define $SGL$ to be a copy of $SG$ in which each literal, $L$, of $SGL$, is assigned a lock number in the following way.

If $L'$ is the literal in HLSG which is naturally associated with $L$, we assign to $L$ the true lock number of $L'$ if $L'$ is true in $M$, else we assign the false lock number of $L'$ to $L$.

In the above the relation naturally associated on clauses and literals of $SGL$ to clauses and literals of HLSG has been assumed to be defined in the obvious way.

Each clause in $SGL$ is a ground normal lock image of the clause in HLSG with which it is naturally associated.

**Step 4.**

We define $MLR$ to be a Modified Lock Resolution refinement strategy for ground sets of arbitrarily lock numbered normal clauses. $MLR$ is similar to ordinary Lock Resolution except with
respect to factoring. No explicit factoring is allowed, which means at the ground level that the input set of clauses does not have duplicate literals removed, and when resolvents are formed there will be no merging of duplicate literals in the resolvent. There is, however, mandatory implicit factoring of the following form. When resolving clause C1 on literal L1 with clause C2 on literal L2, all literals, L1', in C1 which are identical (except possibly for lock number) to L1 are implicitly factored away. Similarly for all literals L2' in C2 identical to L2. MLR is a complete ground refinement strategy. This can be proved easily in several ways, one of which is the same as is used to prove LR complete (Chang and Lee, 1973). Thus there will exist a MLR refutation, R-SGL, of SGL.

Now we transform the refutation tree R-SGL by replacing the ground normal lock clause at each node by a ground primitive representation clause in the following way. If the node is a leaf, and \( x \in \text{SGL} \) is the clause there, then replace it by the naturally associated exact primitive representation clause of \( x \) which is in HLSG. For each internal node, \( k \), of R-SGL which has not had its clause replaced, but whose parents have had their clauses replaced, we apply Lemma III, which asserts the existence of a primitive representation clause which is immediately deducible from the new clauses at the parents of \( k \). Furthermore this deduced clause has the original clause at node \( k \) as a ground normal lock image, and thus it becomes the new clause at node \( k \). We continue this process of clause replacement, by virtue of Lemma III, in a breadth first manner, i.e. first all level 1 nodes are replaced, then all level
2 nodes, etc., until the root is finally replaced. At this point we recognize that at the root of the transformed tree must be a ground HL-null clause, since the new clause at the root has a ground normal lock image which has no literals.

Thus we have shown the existence of a basic primitive representation refutation of the ground set HLSG, and we call this refutation R-HLSG.

Step 5.

Now we transform R-HLSG into a general level refutation of HLS, called R-HLS. This is done one level at a time by first replacing each leaf clause, x, of R-HLSG by the clause y in HLS such that x is a strict grounding of y. We remember that HLSG was constructed as a set of strict groundings of HLS, so such a y will exist, and will in fact be unique. Now, by a process completely analogous to the transformation, in the previous step, of R-SGL into R-HLSG, we apply Lemma IV repeatedly to transform R-HLSG into R-HLS. The new clause, x, that replaces the old root clause, y, must be an HL-null clause, since y had no standard literals and y is a strict grounding of x. The resulting new tree is a general level basic primitive representation refutation of HLS.

Thus we have shown the existence of a basic primitive representation refutation of HLS based on the unsatisfiability of the set S. The proof of completeness now just requires the proof of Lemma IV, the Lifting Lemma.
Step 6.

We repeat the statement of Lemma IV here.

LEMMA IV: Let $X, Y, x, y, z$ be primitive representation clauses, in which $X$ and $Y$ have their variables standardized apart, and with $x$ a strict grounding of $X$ under substitution $\text{TAU}$, and $y$ a strict grounding of $Y$ under substitution $\text{NU}$. If it is the case that there exists an immediate primitive representation deduction of $z$ from $x$ and $y$, then there exists an immediate primitive representation deduction of a clause $Z$, from $X$ and $Y$, such that $z$ is a strict grounding of $Z$.

PROOF OF LEMMA IV

We will need mainly to concentrate on the standard literals of clauses.

Let

$$x = x_1, x_2, \ldots, x_n; \text{ FSL} = ---$$

$$y = y_1, y_2, \ldots, y_m; \text{ FSL} = ---$$

$$z = z_1, z_2, \ldots, z_k; \text{ FSL} = ---$$

By the hypotheses of the lemma, $x$ and $y$ resolve to produce $z$. Without loss of generality we assume that $x$ is the false clause and $y$ the true clause, and that $x_1$ and $y_1$ are the selected literals used in the resolution of $x$ against $y$ to yield $z$. Thus $x_1$, where the negation sign is taken to be an operator that syntactically "inverts" the negation status of a literal. Let $Fx$
be the set of literals in $x$ that have the same second components as $x_1$, including $x_1$ itself. Let $F_y$ be the similar set relative to literal $y_1$ for clause $y$. Then the immediate primitive representation deduction of $z$ from $x$ and $y$ must produce a $z$ with the following structure:

$$SD(z) = (SD(x) - F_x) \cup (SD(y) - F_y)$$

$$FSL(z) = FSL(x) \cup FSL(y)$$

The clause $z$ is of necessity feasible since $x$ and $y$ were feasible ground clauses. By the lemma hypothesis $x$ is a strict grounding of $X$ (under TAU) and $y$ is a strict grounding of $Y$ (under NU). This allows us to assume the existence of a (not necessarily unique) naturally associated relation between literals of $X$ and $x$, and also between literals of $Y$ and $y$. This relation is defined by the functions $g$ and $h$ which must exist for the definition of a strict grounding to apply. The functions $g$ and $h$, and therefore this definition of the naturally associated relation are not necessarily unique. Let $X_1$ be the literal in $X$ naturally associated with $x_1$ in $x$, and $F_X$ the set of literals of $X$ whose naturally associated literals in $x$ are in $F_x$. Similarly for $Y_1$ and $F_Y$. Let

$$X = X_1, X_2, \ldots, X_n; \quad FSL = \cdots$$

$$Y = Y_1, Y_1, \ldots, Y_m; \quad FSL = \cdots$$

have their literals written in the same order as the naturally associated literals in their strict groundings, $X(TAU)$ and $Y(NU)$, respectively. Now, knowing that $x_1 = \theta y_1$ we can assert that there exists a unifier of the set $F_X \cup F_Y$ (ignoring negation signs). By
the Lemma hypothesis X and Y have their variables standardized apart. Thus we can assert by Lemma I for any mgu, OMEGA, of the set FX U FY (ignoring negation signs), there exists a substitution SIGMA such that for 1 ≤ i ≤ n and 1 ≤ j ≤ m

\[ X_i(OMEGA)(SIGMA) = x_i \]
\[ Y_j(OMEGA)(SIGMA) = y_j \]

By Lemma II we can assert that X1 is a selected false literal of X, and Y1 is a selected true literal of Y. This means that the clause

\[ X' = ((SD(X) - FX) U X1)(OMEGA); \quad FSL = --- (OMEGA) \]

is a reduced selected factor of X, and is feasible. Similarly for a Y'. Therefore there will exist a primitive representation binary resolvent, Z, of X' and Y' on selected literals X1(OMEGA) and Y1(OMEGA), in X' and Y' respectively. This constitutes an immediate primitive representation deduction of Z, from X and Y, on selected literals X1 and Y1, respectively. Specifically Z is defined by

\[ SD(Z) = (SD(X) - FX)(OMEGA) U (SD(Y) - FY)(OMEGA) \]
\[ FSL(Z) = FSL(X)(OMEGA) U FSL(Y)(OMEGA) \]
From the construction of $Z$ it should be realized that

$$SD(Z)(\Sigma) = SD(z)$$

and

$$FSL(Z)(\Sigma) = FSL(z)$$

Furthermore the clause $Z$ is feasible since $z$ is feasible. Thus there is an immediate primitive representation deduction of $Z$ from $X$ and $Y$, such that $z$ is a strict grounding of $Z$. This proves the Lifting Lemma.

The above completeness proof shows the existence of a basic primitive representation refutation of HLS based on the unsatisfiability of $S$. It was assumed in the statement of the soundness and completeness theorem that HLS had been disambiguated. The proof of completeness itself is a valid argument independently of whether HLS is disambiguated or not. A stronger refinement results if HLS is disambiguated. However, in order to relate the basic primitive representation HLR to the basic non-primitive representation HLS, HLS should be left ambiguous. This relationship involves the notion of a non-primitive representation clause "standing for", or containing within itself several distinct primitive representation clauses. Such a relationship is analogous to the relationship of a general level clause to the ground instances it represents or contains. In order to support the completeness of HLR at the non-primitive representation level, we would need to use something of the form:
LEMMA V: (primitive representation to non-primitive representation "lifting" lemma)

Let u, v be primitive representations, respectively, of HL-clauses U and V. If there is an immediate primitive representation deduction of w from u and v, then there is an immediate HL-deduction of a clause W from U and V such that w is a primitive representation of W.

Such a lemma statement requires that "immediate HL-deduction" be defined. This requires producing, at the non-primitive representation level, a complete set of definitions of HL-Resolution, much as has been done in section 3.4 for primitive representations.

We will not actually carry out this program of development here for the following reasons:

1. We are not dwelling on implementation issues in this report, and even if we were it is not clear that the prescription in sections 3.1 and 3.2 are the best "lifted" versions of the basic primitive representation strategy stated here.

2. HL-Resolution is being stressed as a theoretical refinement strategy in this report, and in this vein the primitive representation viewpoint lays bare more of the structure of the strategy, and is thus the level at which the strategy should be studied.
3. Further refinements and extensions of HLR will undoubtably best be initially phrased at the primitive representation level, and only later, if at all, "lifted" above the primitive representation level. There is also a strong possibility that future extensions of HLR might be best expressed in primitive representations even in an implementation.

The basic issue with respect to implementation of HLR is that, on a per clause basis, primitive representations require more storage. In addition, in the early part of the search, there will be many more clauses than if a non-primitive representation form is used. On the other hand, because of the ability to disambiguate the lock numbering at the primitive representation level, we see that, as formulated in this report, the primitive representation level is essentially a stronger statement of the basic HLR strategy. What one would wish to have is some form of dynamic assignment of lock numbers at the non-primitive representation level so as to achieve the same (or perhaps even stronger) degree of singly connectedness as is available at the primitive representation level.
3.6 Evaluation of the HLR-Resolution Strategy

The HLR strategy as stated in this report is a semantically oriented resolution strategy which attempts to achieve search efficiency by extending the notion of a clause to include information about the derivation history of the clause. This information then restricts the way in which the clause may further be used, so as to constrain the search at the general level to correspond only to ground refutations of a very restricted form. This ground refutation is exactly a lock refutation on the ground set, with a lock numbering meeting some additional conditions relative to a chosen Herbrand interpretation. HLR is more faithful to its ground form in the sense that an HL-refutation found at the general level will always correspond (in a strong sense) to some HL-refutation of some unsatisfiable ground set. This is not the case in SR (or in LI-clash resolution, (Slagle, 1972)), nor is it the case in TMS.

HL-Resolution was seen to help alleviate both the term substitution problem and the local interaction problem, but still was not an adequate treatment of these problems. There are indications that much more can be done in HLR to neutralize these two sources of search combinatorics.

Judging the relative merits of various resolution strategies is often very difficult to do with any certainty. This is particularly the case when the basic HLR-Resolution strategy is involved because of two factors:
1. There is no full implementation of the basic HL-refinement yet available.

2. There are two additional degrees of freedom in an HL-search, namely the choice of model, and the particular assignment of lock numbers.

Within the limitations imposed by these two factors the following items, at least, can be safely stated.

1. For a sentential clause set HLR automatically becomes normal Lock Resolution, which is among the more efficient of the known resolution refinement strategies, particularly for (near or exactly) minimally unsatisfiable sets.

2. When the model is chosen to be all negative literals, HLR specializes to a refinement of P1 deduction. Thus, in the worst case of a completely trivial model, HLR can be assumed at least superior to P1 deduction (or to P1 deduction under some renaming).

3. On simple problems the search seems to grow at about the same rate as for SL-resolution (Kowalski and Kuehner, 1971). There are reasons to believe that HLR will gain a relative advantage over SL-resolution both as the clause sets become more complex (but near minimally unsatisfiable), and as clause sets become cluttered with irrelevant clauses. These reasons have to do with the (intuitively) expected characteristics of FSL's of clauses generated at the deeper levels of search. As with most questions of resolution search efficiency, these characteristics seem quite resistant to theoretical analysis.
We mention here that there are some immediate surface analogies between SL-resolution (similar also to Model Elimination (Loveland, 1968, 1972)) and HLR. These become apparent if connections are made between FSL literals and framed literals, and ordered clauses and lock numbered ordered literals in an HL-clause. There are some similarities, but the differences are basic enough to raise the question as to whether HLR could be advantageously extended by incorporating some analogue of the A- and B-literal concepts of SL-resolution.

4. The HLR refinement, while performing about as well as the best general purpose complete resolution strategies available on simple problems and expected to increase in relative advantage on harder problems, is still quite inadequate when compared with what it would seem possible to do on theorem proving tasks.

5. The real worth of the HL-Resolution refinement lies in what seem to be realizable extensions of the method. Some of these are given in the next section.
3.7 Extensions of the HL-Resolution Strategy

This section enumerates some of the specific directions in which it is thought that the HL-Resolution strategy should be extended.

1. HL-Resolution could produce in its search a clause in which two standard literals have the same lock numbers. It would seem that completeness can be maintained if separate new lock numbers are then assigned (either true or false lock numbers, whichever is involved in the collision). It remains to be seen if such a collision signals any identifiable situation which would add new restrictions to the search process. It would be interesting to try to develop LR into an even more restrictive strategy, and then combine that strategy with TMS to obtain a stronger form of HLR. Peterson (Peterson, 1976) has shown how LR can be used to extend strategies which are complete for Horn sets to strategies which are complete in general. The main result, called LNL-T resolution, is a lock resolution strategy which is not directly compatible with HLR, but it would seem that there would be similar strategies that would be compatible.

2. In SL-resolution framed literals act as a derivation tree history marker in the ordered clause in which they occur. This information signals the specific points in the search where the search deviates from input resolution constraints. If a similar (i.e. input) type of refinement could be grafted into the HLR structure the expected result would be a further
increase in search efficiency.

3. Subsumption has not been considered yet in HLR. The obvious thing to try to do is to develop some criteria for semantic subsumption through the use of the model. This seems difficult to do if it is required that completeness be maintained.

4. The presentation of the basic HLR strategy was independent of what Herbrand interpretation is used as the model, M. Thus HLR is sound and complete independent of the choice of M. However, it is assumed that part of the TSP will depend upon the choice of the model. There are numerous questions concerning the relationship in HLR of a specific M to a specific set of clauses. If we assume M is a model scheme, and thus has parameters which must be specified before the search begins, then we want to know what parameter choices are best. At present very little is known about how to compare the worth of two model schemes before the search begins.

5. The ultimate sensitivity of HLR to the exact choice of model is not yet known. If the nature of the model proves to be crucial in search efficiency for difficult problems, as it would seem reasonable to assume will be the case, then it becomes important to develop methods that will facilitate the process of bringing the proper model to bear upon a search effort. Such methods might include a library of established model schemes and procedures for deciding which model to use on a given problem. These procedures might even be re-invoked after some partial search has been done and a new model
selection made on the basis of the partial search results.

6. It would be highly effective if a way were found to combine several models into a new model having more desirable features than any of its constituent pieces. This seems rather difficult to do at the present time. Henschen discusses how to combine several Herbrand interpretations into a new interpretation for the case of Horn sets and has developed a semantic refinement strategy for Horn sets (Henschen, 1975). It remains to be seen if A-models can be combined in an analogous manner and the results extended to non-Horn cases.

7. HLR is sound and complete with an arbitrary Herbrand interpretation as the model, M. There are procedures that are simple extensions to the basic HLR strategy which may or may not affect soundness and completeness, depending upon the particular clause sets they are applied to. It is important to characterize in which situations this occurs. One of these procedures we call evaluating out a predicate letter (a simple extension of "elimination by evaluating predicates", (Nilsson, 1971, p.218)). Evaluating out a given literal in a clause involves simply removing the literal from the standard literals of a clause and adding it to the FSL set, and then keeping the transformed clause iff it is feasible in M (the original clause is deleted). Evaluating out a predicate letter involves evaluating out every standard literal in a clause set using that predicate letter. Evaluating out a predicate letter is not a soundness preserving operation, in general. What happens is that the meaning of unsatisfiability is modified from
"unsatisfiable means false in every interpretation" to "unsatisfiable means false according to M". If M is a model scheme then it contains many individual interpretations, and unsatisfiable will mean false over all of the individual interpretations in M. In particular we notice that the modified notion of unsatisfiability pertains only to the predicate letter(s) actually evaluated out. We see here what is the strongest single extension available in the HL-Resolution structure, namely the ability to set a context, through the use of the model, for theorem proving. The process of evaluating out predicate letters brings model based information into the declarative clause set structure in a complexity reducing manner (by removing standard literals and turning them into FSL literals), rather than by complexity increasing mechanisms (such as adding new declarative information in the form of additional clauses). The process of evaluating out literals needs to be better understood.

8. The LIP needs much further work. The obvious choice for a carrier of global information in the HLR search, the model, is not yet adequately transmitting information of sufficient strength. There are several possible ways to attack the LIP. All seem to require large amounts of additional computer resource on a per clause generated basis, and thus would only be applicable for problems with large search spaces. As an example of such an attack on one aspect of the LIP, consider the following. Let S be an unsatisfiable set of general level clauses, and C a clause in S. We say that the clause x
properly subsumes the clause \( y \) iff \( x \) subsumes \( y \) and \( y \) does not subsume \( x \). Then in searching for a proof one need never consider \( C \) to represent ground instances which are properly subsumed by any other clause in \( S \). In HLR all of the literals of \( C \) survive (as either standard or FSL literals) in clauses derived from \( C \). As clauses derived from \( C \) which are deeper in the search are produced, it may be possible to detect at some point that a resolvent is in fact using only ground instances of \( C \) which can be subsumed by some other input clause (or clauses). If such a situation occurs, then the resolvent may be deleted. Such a deletion strategy is a complete refinement of HLR. There are also some variants of this strategy which are not known to be complete (nor known to be incomplete) refinements of HLR. An important point to be emphasized here is that the above example is totally syntactic in its orientation (assuming that subsumption is defined syntactically). One is led to wonder if there are analogous semantic refinements.
4.0 Summary

A new resolution strategy, called HL-Resolution, has been presented as a sound and complete refinement strategy combining Lock Resolution and The Model Strategy. HL-Resolution combines both semantic and syntactic refinement criteria, and in particular tries to achieve an increase in the globality (or context) of the individual resolution steps (through the use of FSL sets). A simple example was given using HLR indicating that it was an efficient strategy. The basic HLR strategy was judged comparable to such strategies as SL-resolution for simple problems, but would be expected to have a relative advantage on more complex problems. The primary utility of HLR is that it offers a suggestive framework for the development of further strategies, particularly through the use of models.

The models useful in HL-Resolution were described as model schemes which initially are specified at a certain level of generality, and become more specified as the resolution search proceeds. This is done in a way which allows a deferment of the decision as to exactly what Herbrand interpretation the model represents until more information is available from the search process.

Considerably more work remains to be done both in formalizing the results already obtained concerning models, and in extending the basic HL-Resolution strategy.
REFERENCES


To help the reader check on his understanding of the definitions used in sections 3.4 and 3.5, an example is given here. The problem statement describes the following world, and is taken with minor changes from Moore (Moore, 1975).

Three blocks, B1, B2, and B3 are stacked with B1 on the top and B3 on the bottom and B2 in the middle. B1 is blue in color, and B3 is green. It is not known if B2 is blue or green, but it is one or the other.

The problem task is to show that in this world there are two blocks, one on the other, and the upper one is blue and the lower one is green. The set of clauses this translates into will be called "Colored Blocks". Colored Blocks is the initial clause set, S.

S:  Colored Blocks

S1. ON(B1,B2);
S2. ON(B2,B3);
S3. COLOR(B1,blue);
S4. COLOR(B3,green);
S5. COLOR(B2,blue), COLOR(B2,green);
S6. @ON(x,y),@COLOR(x,blue),@COLOR(y,green);
The predicate and constant names are self explanatory. Clause S6 is the denial of the existence of the situation that we are going to show actually does hold, while the first 5 clauses are just the facts we do know to be true.

We pick some arbitrary HL-proper lock numbering for the HL-associated input set of Colored Blocks.

**HL-associated Input Set of Colored Blocks**

1. \(<1,\text{ON}(B1,B2),100>\); \(\text{FSL} = []\)
2. \(<2,\text{ON}(B2,B3),200>\); \(\text{FSL} = []\)
3. \(<3,\text{COLOR}(B1,\text{blue}),300>\); \(\text{FSL} = []\)
4. \(<4,\text{COLOR}(B3,\text{green}),400>\); \(\text{FSL} = []\)
5. \(<5,\text{COLOR}(B2,\text{blue}),500>, <6,\text{COLOR}(B2,\text{green}),600>\); \(\text{FSL} = []\)
6. \(<7,\text{@ON}(x,y),700>, <8,\text{@COLOR}(x,\text{blue}),800>, <9,\text{@COLOR}(y,\text{green}),900>\); \(\text{FSL} = []\)

Now, in order to form the exact primitive representation HL-associated input set for Colored Blocks, it is necessary to have a particular model specified. We pick as the domain of the model the set of real numbers, and use only the usual equality relation within this model. We pick some specific individual real numbers to represent the constants in LC. Specifically we have the following correspondence from LC to LM.
This gives us the following exact primitive representation HL-associated input set for Colored Blocks, with respect to M.

**Exact Primitive Representation**

**HL-Associated Input Set for Colored Blocks**

Here we underline the active lock number for each literal, and also underline the second component of any selected literals in each clause. The particular lock numbering we have chosen gives just one selected literal per clause. For ease in reading, the lock numbers are omitted for the FSL literals.

1. \(<1, \text{ON}(B1,B2),100>\); \(\text{FSL} = \{\text{ON}(B1,B2)\}\) \(F\)
2. \(<2, \text{ON}(B2,B3),200>\); \(\text{FSL} = \{\text{ON}(B2,B3)\}\) \(F\)
3. \(<3, \text{COLOR}(B1,\text{blue}),300>\); \(\text{FSL} = \{\text{COLOR}(B1,\text{blue})\}\) \(T\)
4. \(<4, \text{COLOR}(B3,\text{green}),400>\); \(\text{FSL} = \{\text{COLOR}(B3,\text{green})\}\) \(T\)
5. \(<5, \text{COLOR}(B2,\text{blue}),500>, <6, \text{COLOR}(B2,\text{green}),600>\); \(\text{FSL} = \{\text{COLOR}(B2,\text{blue}),\text{COLOR}(B2,\text{green})\}\) \(F\)
6.1. \( \langle 7, \text{ON}(x,y), 700 \rangle, \langle 8, \text{COLOR}(x,\text{blue}), 800 \rangle, \langle 9, \text{COLOR}(y,\text{green}), 900 \rangle; \)
FSL = [\text{ON}(x,y), \text{COLOR}(x,\text{blue}), \text{COLOR}(y,\text{green})] T

6.2. \( \langle 7, \text{ON}(x,y), 700 \rangle, \langle 8, \text{COLOR}(x,\text{blue}), 800 \rangle, \langle 9, \text{COLOR}(y,\text{green}), 900 \rangle; \)
FSL = [\text{ON}(x,y), \text{COLOR}(x,\text{blue}), \text{COLOR}(y,\text{green})] T

6.3. \( \langle 7, \text{ON}(x,y), 700 \rangle, \langle 8, \text{COLOR}(x,\text{blue}), 800 \rangle, \langle 9, \text{COLOR}(y,\text{green}), 900 \rangle; \)
FSL = [\text{ON}(x,y), \text{COLOR}(x,\text{blue}), \text{COLOR}(y,\text{green})] T

6.4. \( \langle 7, \text{ON}(x,y), 700 \rangle, \langle 8, \text{COLOR}(x,\text{blue}), 800 \rangle, \langle 9, \text{COLOR}(y,\text{green}), 900 \rangle; \)
FSL = [\text{ON}(x,y), \text{COLOR}(x,\text{blue}), \text{COLOR}(y,\text{green})] T

6.5. \( \langle 7, \text{ON}(x,y), 700 \rangle, \langle 8, \text{COLOR}(x,\text{blue}), 800 \rangle, \langle 9, \text{COLOR}(y,\text{green}), 900 \rangle; \)
FSL = [\text{ON}(x,y), \text{COLOR}(x,\text{blue}), \text{COLOR}(y,\text{green})] T

6.6. \( \langle 7, \text{ON}(x,y), 700 \rangle, \langle 8, \text{COLOR}(x,\text{blue}), 800 \rangle, \langle 9, \text{COLOR}(y,\text{green}), 900 \rangle; \)
FSL = [\text{ON}(x,y), \text{COLOR}(x,\text{blue}), \text{COLOR}(y,\text{green})] T

6.7. \( \langle 7, \text{ON}(x,y), 700 \rangle, \langle 8, \text{COLOR}(x,\text{blue}), 800 \rangle, \langle 9, \text{COLOR}(y,\text{green}), 900 \rangle; \)
FSL = [\text{ON}(x,y), \text{COLOR}(x,\text{blue}), \text{COLOR}(y,\text{green})] T
We disambiguate the numbering to give us the unambiguously HL-proper lock numbered exact primitive representation
HL-associated input set for Colored Blocks. This corresponds to HLS of the completeness theorem in section 3.5.

**HLS: Exact Primitive Representation**

**HL-Associated Input Set of Colored Blocks (disambiguated)**

- HLS1. \(<1, ON(B_1,B_2), 100>\); \(FSL = [ON(B_1,B_2)]\)  
  \(F\)

- HLS2. \(<2, ON(B_2,B_3), 200>\); \(FSL = [ON(B_2,B_3)]\)  
  \(F\)

- HLS3. \(<3, COLOR(B_1,blue), 300>\); \(FSL = [@COLOR(B_1,blue)]\)  
  \(T\)

- HLS4. \(<4, COLOR(B_3,green), 400>\); \(FSL = [@COLOR(B_3,green)]\)  
  \(T\)

- HLS5. \(<5, COLOR(B_2,blue), 500>, <6, COLOR(B_2,green), 600>\);  
  \(FSL = [COLOR(B_2,blue),COLOR(B_2,green)]\)  
  \(F\)

- HLS6.1. \(<7, @ON(x,y), 700>, <8, @COLOR(x,blue), 800>\),  
  \(<9, @COLOR(y,green), 900>\);  
  \(FSL = [ON(x,y),COLOR(x,blue),COLOR(y,green)]\)  
  \(T\)

- HLS6.2. \(<10, @ON(x,y), 701>, <11, @COLOR(x,blue), 702>\),  
  \(<12, @COLOR(y,green), 703>\);  
  \(FSL = [ON(x,y),COLOR(x,blue),COLOR(y,green)]\)  
  \(T\)

- HLS6.3. \(<13, @ON(x,y), 704>, <14, @COLOR(x,blue), 705>\),  
  \(<15, @COLOR(y,green), 706>\);  
  \(FSL = [@ON(x,y),COLOR(x,blue),COLOR(y,green)]\)  
  \(T\)

- HLS6.4. \(<16, @ON(x,y), 707>, <17, @COLOR(x,blue), 708>\),  
  \(<18, @COLOR(y,green), 709>\);  
  \(FSL = [ON(x,y),COLOR(x,blue),COLOR(y,green)]\)  
  \(T\)
HLS6.5. \(\langle 19, \text{ON}(x,y), 710 \rangle, \langle 20, \text{COLOR}(x, \text{blue}), 711 \rangle,\)  
\[\langle 21, \text{COLOR}(y, \text{green}), 712 \rangle;\]
FSL = \([\text{ON}(x,y), \text{COLOR}(x, \text{blue}), \text{COLOR}(y, \text{green})]\) T

HLS6.6. \(\langle 22, \text{ON}(x,y), 713 \rangle, \langle 23, \text{COLOR}(x, \text{blue}), 714 \rangle,\)  
\[\langle 24, \text{COLOR}(y, \text{green}), 715 \rangle;\]
FSL = \([\text{ON}(x,y), \text{COLOR}(x, \text{blue}), \text{COLOR}(y, \text{green})]\) T

HLS6.7. \(\langle 25, \text{ON}(x,y), 716 \rangle, \langle 26, \text{COLOR}(x, \text{blue}), 717 \rangle,\)  
\[\langle 27, \text{COLOR}(y, \text{green}), 718 \rangle;\]
FSL = \([\text{ON}(x,y), \text{COLOR}(x, \text{blue}), \text{COLOR}(y, \text{green})]\) T

SG for Colored Blocks

Using Colored Blocks as \(S\), we have an SG in which clauses 1 through 5 are just as they are in Colored Blocks, and clause 6 of Colored Blocks yields two ground instances.

SG1. \(\text{ON}(B1, B2)\);
SG2. \(\text{ON}(B2, B3)\);
SG3. \(\text{COLOR}(B1, \text{blue})\);
SG4. \(\text{COLOR}(B3, \text{green})\);
SG5. \(\text{COLOR}(B2, \text{blue}), \text{COLOR}(B2, \text{green})\);
SG6. \(\text{ON}(B1, B2), \text{COLOR}(B1, \text{blue}), \text{COLOR}(B2, \text{green})\);
SG7. \(\text{ON}(B2, B3), \text{COLOR}(B2, \text{blue}), \text{COLOR}(B3, \text{green})\);
GR(S,SG) for Colored Blocks

A GR(S,SG) that connects this S and SG is:

\[ GR(S,SG) = [(null),(null),(null),(null),(null),
(B1/x,B2/y),(B2/x,B3/y)] \]

which is a list of 6 elements. The first 5 elements are identical, and consist each of a single substitution which we call null. Null causes no substitutions to occur on the corresponding clause but does cause one copy of that clause to be in SG. The sixth element of GR(S,SG) consists of two distinct substitutions, generating the two ground instances (clauses SG6 and SG7) in SG that correspond to clause S6 of Colored Blocks.

HLSG for Colored Blocks

The set HLSG for Colored Blocks has its first 5 clauses the same as the first 5 clauses in HLS, which is the exact primitive representation HL-associated input set for Colored Blocks (disambiguated). The reason is that the first 5 elements of GR(S,SG) each consists of just one substitution, namely the null substitution. Clauses 6 and 7 of HLSG result from applying each of the elements of the sixth component of GR(S,SG) to each clause among HLS6.1 through HLS6.7. This gives one feasible ground clause from HLS6.6, which becomes clause 6 of HLSG, and one feasible ground clause from clause HLS6.4, which becomes clause 7 of HLSG.
1. \(<1, ON(B1,B2), 100>\); \(FSL = \{ON(B1,B2)\}\) \(F\)
2. \(<2, ON(B2,B3), 200>\); \(FSL = \{ON(B2,B3)\}\) \(F\)
3. \(<3, \text{COLOR}(B1,\text{blue}), 300>\); \(FSL = \{\text{COLOR}(B1,\text{blue})\}\) \(T\)
4. \(<4, \text{COLOR}(B3,\text{green}), 400>\); \(FSL = \{\text{COLOR}(B3,\text{green})\}\) \(T\)
5. \(<5, \text{COLOR}(B2,\text{blue}), 500>, <6, \text{COLOR}(B2,\text{green}), 600>\);

\(FSL = \{\text{COLOR}(B2,\text{blue}), \text{COLOR}(B2,\text{green})\}\) \(F\)

6. \(<22, @ON(B1,B2), 713>, <23, @\text{COLOR}(B1,\text{blue}), 714>,\)

\(<24, @\text{COLOR}(B2,\text{green}), 715>\);

\(FSL = \{ON(B1,B2), @\text{COLOR}(B1,\text{blue}), \text{COLOR}(B2,\text{green})\}\)

7. \(<16, @ON(B2,B3), 707>, <17, @\text{COLOR}(B2,\text{blue}), 708>,\)

\(<18, @\text{COLOR}(B3,\text{green}), 709>\);

\(FSL = \{ON(B2,B3), \text{COLOR}(B2,\text{blue}), @\text{COLOR}(B3,\text{green})\}\) \(T\)

\textbf{SGL for Colored Blocks}

Finally this yields an SGL for Colored Blocks as follows:

1. \(<\text{ON}(B1,B2), 100>\);
2. \(<\text{ON}(B2,B3), 200>\);
3. \(<\text{COLOR}(B1,\text{blue}), 3>\);
4. \(<\text{COLOR}(B3,\text{green}), 4>\);
5. \(<\text{COLOR}(B2,\text{blue}), 500>, <\text{COLOR}(B2,\text{green}), 600>\);
6. \(<@\text{ON}(B1,B2), 22>, <@\text{COLOR}(B1,\text{blue}), 714>, <@\text{COLOR}(B2,\text{green}), 24>\);
7. \(<@\text{ON}(B2,B3), 16>, <@\text{COLOR}(B2,\text{blue}), 17>, <@\text{COLOR}(B3,\text{green}), 709>\);
R-SGL for Colored Blocks

6x1=8.  <@COLOR(B1,blue),714>, <@COLOR(B2,green),24>;  
7x2=9.  <@COLOR(B2,blue),17>, <@COLOR(B3,green),709>;  
9x5=10. <COLOR(B2,green),600>, <@COLOR(B3,green),709>;  
10x8=11. <@COLOR(B3,green),709>, <@COLOR(B1,blue),714>;  
11x4=12. <@COLOR(B1,blue),714>;  
12x3=13.  *BOX*;

R-HLSG for Colored Blocks

Here the FSL literals will be written with all three components to make it easier to see where they came from.

6x1=8.  <23, @COLOR(B1, blue), 714>, <24, @COLOR(B2, green), 715>;  
FSL = [<22, ON(B1, B2), 713>, <23, @COLOR(B1, blue), 714>,  
<24, COLOR(B2, green), 715>, <1, ON(B1, B2), 100>]  T

7x2=9.  <17, @COLOR(B2, blue), 708>, <18, @COLOR(B3, green), 709>;  
FSL = [<16, ON(B2, B3), 707>, <17, COLOR(B2, blue), 708>,  
<18, @COLOR(B3, green), 709>, <2, ON(B2, B3), 200>]  T

9x5=10. <6, COLOR(B2, green), 600>, <18, @COLOR(B3, green), 709>;  
FSL = [<16, ON(B2, B3), 707>, <17, COLOR(B2, blue), 708>,  
<18, @COLOR(B3, green), 709>, <2, ON(B2, B3), 200>,  
<5, COLOR(B2, blue), 500>, <6, COLOR(B2, green), 600>]  F
10x8=11.  \[18, \text{COLOR}(B3, \text{green}), 709], \langle 23, \text{COLOR}(B1, \text{blue}), 714 \rangle; \]

\[\text{FSL} = \{16, \text{ON}(B2, B3), 707], \langle 17, \text{COLOR}(B2, \text{blue}), 708 \rangle,\]

\[\langle 18, \text{COLOR}(B3, \text{green}), 709], \langle 2, \text{ON}(B2, B3), 200 \rangle,\]

\[\langle 5, \text{COLOR}(B2, \text{blue}), 500 \rangle, \langle 6, \text{COLOR}(B2, \text{green}), 600 \rangle,\]

\[\langle 22, \text{ON}(B1, B2), 713], \langle 23, \text{COLOR}(B1, \text{blue}), 714 \rangle,\]

\[\langle 24, \text{COLOR}(B2, \text{green}), 715], \langle 1, \text{ON}(B1, B2), 100 \rangle\]  F

11x4=12.  \[23, \text{COLOR}(B1, \text{blue}), 714; \]

\[\text{FSL} = \{16, \text{ON}(B2, B3), 707], \langle 17, \text{COLOR}(B2, \text{blue}), 708 \rangle,\]

\[\langle 18, \text{COLOR}(B3, \text{green}), 709], \langle 2, \text{ON}(B2, B3), 200 \rangle,\]

\[\langle 5, \text{COLOR}(B2, \text{blue}), 500 \rangle, \langle 6, \text{COLOR}(B2, \text{green}), 600 \rangle,\]

\[\langle 22, \text{ON}(B1, B2), 713], \langle 23, \text{COLOR}(B1, \text{blue}), 714 \rangle,\]

\[\langle 24, \text{COLOR}(B2, \text{green}), 715], \langle 1, \text{ON}(B1, B2), 100 \rangle\]

\[\langle 4, \text{COLOR}(B3, \text{green}), 400 \rangle\]  F

12x3=13. *BOX*

\[\text{FSL} = \{16, \text{ON}(B2, B3), 707], \langle 17, \text{COLOR}(B2, \text{blue}), 708 \rangle,\]

\[\langle 18, \text{COLOR}(B3, \text{green}), 709], \langle 2, \text{ON}(B2, B3), 200 \rangle,\]

\[\langle 5, \text{COLOR}(B2, \text{blue}), 500 \rangle, \langle 6, \text{COLOR}(B2, \text{green}), 600 \rangle,\]

\[\langle 22, \text{ON}(B1, B2), 713], \langle 23, \text{COLOR}(B1, \text{blue}), 714 \rangle,\]

\[\langle 24, \text{COLOR}(B2, \text{green}), 715], \langle 1, \text{ON}(B1, B2), 100 \rangle,\]

\[\langle 4, \text{COLOR}(B3, \text{green}), 400 \rangle, \langle 3, \text{COLOR}(B1, \text{blue}), 300 \rangle\]

R-HLS for Colored Blocks

The refutation for HLS is identical (in this particular example) to R-HLSG except that clause 6 of HLSG is replaced by clause HLS6.6, and clause 7 of HLSG is replaced by clause HLS6.4.
INDEX

Items in this index are only those which are defined in the text, and the page references are only to the definitions. For abbreviations not listed here see the table following this index.

Against (resolving against) 30
Algorithmic based models (A-models) 13
Ambiguous lock numbering (see disambiguated and unambiguous)

Basic HL-Resolution strategy 53
Basic primitive representation deduction 61
Basic primitive representation refutation 61

Disambiguated 62
Elementary factoring step 35
Evaluating out (a predicate letter) 90
Evaluation function (for a Herbrand interpretation) 56, 58
Exact primitive representation 57
Exact primitive representation HL-associated input set 62
Exact primitive representation set 57

f (selector function) 54
F 29, 38
Factoring unifier 60
False lock number 28, 32, 54
False parent 61
False substitution literals 27, 55
Feasible clause 56
FSL 27, 55

Ground case model 12
Ground HL-clause 55
Ground literal 55
Ground literal truth value 55
Ground normal lock function 70
Ground normal lock image 70
Grounding substitution (for a literal) 55
Grounding substitution (for a set of literals) 26, 55
Grounding substitution (for an HL-clause) 55

Herbrand interpretation 53
HL-associated input set 62
HL-clause 27, 55
HL-literal 54
HL-null clause 56
HL-proper lock numbering 54

Immediate primitive representation deduction 70
Immediately deducible 70
Inexact primitive representation 57
Infeasible 56
index

Influential variable 28
Influential literal 28

Lemma I 67
Lemma II 68
Lemma III 71
Lemma IV 72
Lemma V 83
Lock numbering 54
Lock Resolution 5
Local interaction problem 49
LSLE 38

M (as a function) 56, 58
MLR (Modified Lock Resolution) 76
Minimally unsatisfiable 2
Model evaluation function 56, 58
Modified lock resolution 76

Naturally associated 68, 73-76, 80
Negation (of a literal) 28
Normal resolution 3

Primitive representation 57
Primitive representation binary resolvent 60

Reduced selected factor 60
Resolution step 2

Satisfy (a set of literals) 26
SD 27, 55
Selected factor 59, 60
Selected false literal 59
Selected literal 59
Selected true literal 58
Simultaneous linear equation (SLE) model 22
Singly connected 3
Standard literals 27, 55
Strict grounding 68

t (selector function) 54
T 29, 38
T/F 29, 38
The Model Strategy 7
Theorem (soundness and completeness of HLR) 63
Term substitution problem 47ff.
Trivial model 3, 12
True clause 27
True lock number 28, 32, 54
True parent 61

Unambiguous (lock numbering) 54
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Conjunctive Normal Form</td>
</tr>
<tr>
<td>.FA.</td>
<td>The quantifier &quot;for all&quot;</td>
</tr>
<tr>
<td>HI</td>
<td>Herbrand interpretation</td>
</tr>
<tr>
<td>HLR</td>
<td>Hereditary-Lock Resolution</td>
</tr>
<tr>
<td>LC (LM)</td>
<td>Language of the clauses (of the model)</td>
</tr>
<tr>
<td>LIP</td>
<td>Local Interaction Problem</td>
</tr>
<tr>
<td>LR</td>
<td>Lock Resolution</td>
</tr>
<tr>
<td>M</td>
<td>Model (usually meaning a collection of Herbrand interpretations), or a model evaluation function</td>
</tr>
<tr>
<td>SC</td>
<td>Sentential Calculus</td>
</tr>
<tr>
<td>SR</td>
<td>Semantic Resolution</td>
</tr>
<tr>
<td>.TE.</td>
<td>The quantifier &quot;there exists&quot;</td>
</tr>
<tr>
<td>TMS</td>
<td>The Model Strategy</td>
</tr>
<tr>
<td>TSP</td>
<td>Term Substitution Problem</td>
</tr>
<tr>
<td><em>BOX</em></td>
<td>The empty list (or set) of literals</td>
</tr>
<tr>
<td>-</td>
<td>subtraction or set difference</td>
</tr>
<tr>
<td>@</td>
<td>negation sign</td>
</tr>
</tbody>
</table>
FORMAL SPECIFICATIONS OF MODELS FOR SEMANTIC THEOREM PROVING STRATEGIES

D. M. Sandford

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, New Jersey

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Grant #DAHC15-73-G6 to the Rutgers Project on Secure Systems and Automatic Programming.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U. S. Government.
ACKNOWLEDGMENTS

I would like to thank Professor A. Yasuhara for a careful proofreading and suggestions on improving the clarity of presentation.

This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under grant DAHCIS-73-G6.
CONTENTS

Chapter 1
1.0 Introduction............................................. 1
1.1 Models in General...................................... 1
1.2 MDS.................................................... 5
1.3 Model Schemes Instead of Models..................... 6

Chapter 2
2.0 Introduction............................................. 7
2.1 General Definitions and Nomenclature................. 8
2.2 Basic Model Scheme Nomenclature...................... 14
2.3 Basic Model Scheme Results......................... 22
2.4 Basic Model Scheme Specification..................... 25
2.5 A Sound and Complete Refinement
   Using Incomplete Model Schemes......................... 23
2.6 An Example of a Basic Modeling Structure............ 31

Chapter 3
3.0 Discussion............................................... 35

REFERENCES.................................................. 40

TABLE OF ABBREVIATIONS.................................... 42
1.0 Introduction

This report is a companion report to SOSAP-TR-30 (Sandford, 1977, hereafter TR-30), and is a more formal treatment of material introduced informally in Chapter 2 of TR-30. This report is theoretically oriented, and constitutes the initial presentation of material which will be explored in greater depth in a thesis by the author (forthcoming in 1978).

1.1 Models in General

There is considerable interest in, and belief in the utility of, using models to incorporate semantic information into the processing repertoire of sophisticated problem solving systems. Models seem to be important in controlling search space size when the search space is syntactically defined. One of the most important situations of this type is in theorem proving in a specific domain of discourse using a general purpose theorem prover. In such a situation it is hoped that the model will supply the needed search guidance to make the general purpose inference rules efficient in the specific domain of application. An example of such a specific domain is that of program verification. In this specific case the theorem proving problems which are generated as subproblems are in general well beyond the ability of currently available syntactically oriented theorem provers. One approach to solving this difficulty is to employ semantic information to guide the theorem proving search by using models. In order to do this one must be able both to specify and manipulate models, and be able
to relate the model effectively to the syntactic search. TP-3U was principally concerned with presenting a new resolution refinement strategy which uses model based information in its search. This report is concerned primarily with a paradigm for firstly, specifying models, and secondly, (at an abstract level) utilizing models so that their semantic information is available to a model based guidance mechanism. This material is presented in terms of a first order resolution refutation situation (Robinson, 1965), but the framework seems to be general enough so as to have application in search space situations which are phrased in other notations.

The principal result obtained in this research and presented here is that sound and complete syntactic resolution theorem proving search procedures can be guided by models which are themselves incomplete, just so long as the models are internally consistent.

A model is not a well defined notion unless we also consider what it is a model of. The notion of model we wish to emphasize is that a model is a structure that abstracts the relevant structure of what it models, and organizes that abstracted relevant structure in a manner that allows efficient problem solving on the abstraction (i.e. efficiency in the model, not necessarily efficiency in the original structure).

When considering problem solving activity as an automated process there is a constant source of difficulty which appears due to the present state of development of the field of artificial intelligence. This difficulty is that, while we wish to have
information processing performed which is of a given level of complexity when understood semantically, we are forced to find fully declarative syntactic representations of these tasks in order to express them to a computer. This mapping to syntactic representations is currently unavoidable, but what one hopes is avoidable is the concurrent loss of semantic information which accompanies this translation as accomplished by current representation techniques.

Model use (for the purposes of this report) is an approach which tries to obtain heuristic efficiency in a search process by making the declarative syntactic search process be guided by semantic information. Thus the efficiency of a search process is a function of the model, but the legitimacy of the resulting search is totally a function of the syntax of the search space language and the syntactic rules of manipulation, and is thus not affected by the semantic content of the model.

There are several different ways in which models may be related to the declarative information processing task. One of these is in top down or hypothesis oriented use of the model to govern how the search space is explored. In this approach the task to be accomplished is performed first in the model, and then the model solution is projected onto the syntactic search space to see if the model solution is feasible syntactically. Thus the syntactic search space is expanded only under the guidance of a relatively complete solution scheme proposed by the model. A converse way to use models is in a bottom up syntactic data driven
manner, such as HL-Resolution (Sandford, 1977). In this approach the syntactic search space is the initiator of processing and individual syntactic search possibilities are locally evaluated as to their worth by the model. While ultimately sophisticated systems will probably use both top down and bottom up approaches, as well as other modes of model utilization, the top down manner seems to be the most important for complex information tasks in which there is a strong model available.

In this report material is presented which comes closest to adequately establishing an abstract paradigm of model use in bottom up resolution refutation procedures. The top down use of models does not require abandoning the structures contained herein, but does require a considerably more extensive development than has been accomplished at this time.
1.2 MDS

The Meta Description System (MDS) (Srinivasan, 1976) is a system which was developed ab initio with a very general and strong capacity to accept and use information in a manner which is semantically oriented, and in fact constitutes a modeling facility of considerable power. In MDS there is also the general capacity for theorem proving (and there is no specific commitment to the exact choice of inference system employed). MDS is in a reverse position to most other general purpose artificial intelligence systems in the sense that its modeling capability is much stronger with respect to its formal theorem proving capability than is usual. In particular resolution theorem proving programs are virtually without exception extremely weak in modeling capability.

This report presents a bottom up view of the relationship between a formal theorem proving domain and a model for that domain, which is applicable to both MDS and to semantically oriented resolution refutation procedures. In particular if the theorem proving component of MDS is taken to be a resolution refutation theorem prover, then this report can be viewed as specifying the formal characteristics the modeling components of MDS must have to ensure refutation completeness of the theorem proving component.
1.3 Model Schemes Instead of Models

One of the difficulties in utilizing models for efficient information processing is the difficulty in finding a good model, and then specifying the model in a manner in which it can actually be used by an automated reasoning system. This report finesses these, and other problems, by taking a theoretical view of models which, while not dealing with the problem of model finding or with pragmatic computation of model use directly, does frame things in a manner which makes these problems pragmatically approachable.

The fundamental notion is that a model is actually a model scheme. A modal scheme is a class of interpretations which is represented by a set of logical statements. By doing this the following happens:

1. By using a class, we obtain partial freedom from introducing a semantic bias in the model when the available semantic information is insufficient to choose a unique interpretation;

2. By representing the class by a set of logical statements we have a natural way of expressing models which are to be utilized to guide a theorem proving search (since much of the attendant techniques, and not unimportantly viewpoints, are already available).

Chapter 2 presents these notions is greater detail.
2.0 Introduction

This chapter develops the formal paradigm of the use of model schemes in resolution refutation procedures. This chapter will also appear in a forthcoming (1978) thesis by the author.

The basic structure as presented in the remainder of this chapter is centered around the situation where $L$ is a first order language in which a set, $S$, of clauses is expressed, and it is necessary to find a refutation of $S$. The language $L'$ would then be a language for a model scheme, and a set of first order formula, $M$, would be a scheme defining set of statements written in the language $L'$. The syntactic search space is generated in the language $L$ according to a semantic resolution refinement (e.g. ML-Resolution), and during the expansion of the search space the model scheme is called upon to perform truth evaluations of various sets of literals. The actual content of the search space is a function of what the truth evaluations actually are, which in turn are a function of the actual model scheme used. The method used for making truth evaluations, and the properties of the entire theorem proving system are presented at an abstract level in the following sections.
2.1 General Definitions and Nomenclature

The notion of a first order language will be taken as primitive here, and is to be the standard notion of a first order language found in logic texts (e.g. (Kleene, 1967)). This report is concerned more with the semantics of languages than with the exact syntactical structure of languages. In later sections some attention is paid to syntax in the context of the relationship between languages. Most of the time, when it is necessary to consider syntax, we will assume that (sets of) well formed formulas (wff's) have been put into conjunctive normal form, as is usual in resolution. The definitions of terms, atoms and literals are all standard (Nilsson, 1971) (Chang and Lee, 1973). Unless otherwise stated all languages considered are first order languages.

If $L$ is a first order language, then we denote the Herbrand universe of $L$, which is the set of all terms of $L$, by $\mathbb{U}(L)$, and we denote the atom set of $L$, also called the Herbrand base of $L$, by $\mathcal{A}(L)$. $\mathcal{A}(L)$ is the (usually infinite) set of all possible ground atoms.

A Herbrand interpretation (HI) for the language $L$ is a set of ground literals such that every element of $\mathcal{A}(L)$ appears, either negated or unnegated in the interpretation, and no element of $\mathcal{A}(L)$ appears both negated and unnegated in the interpretation. The set of all possible Herbrand interpretations of the language $L$ is written as $\mathbb{H}(L)$.
If \( k \) is a literal of \( L \), then by \( \overline{k} \) is meant the same literal but with its negation status inverted, i.e. \( k \) has a negation sign on it iff \( \overline{k} \) does not have a negation sign on it.

A literal (wff, clause, etc.) is said to be ground if it contains no variable symbols within it. If \( x \) is a literal (wff, clause, etc.) and \( \text{Tau} \) is a substitution, then \( \text{Tau} \) is said to be a grounding substitution for \( x \) iff \( x(\text{Tau}) \) is ground.

**Evaluation Function**

A clause is often considered as a set of literals for the purpose of deciding the truth value of the clause in some given interpretation. Let \( h \) be some \( \text{HI} \) for the language \( L \). If \( K \) is a set of ground literals of \( L \), then \( K \) is said to be false in \( h \) iff \( K \cap h \) is empty. Otherwise \( K \) is true in \( h \). We write \( h(K) \) to designate the truth value of the set of ground literals, \( K \), i.e. we assume that there is a function, \( h \), which maps sets of ground literals to their truth value in the interpretation \( h \).

We extend the function \( h \) to apply to general level literals (i.e. literals in which free variables occur) as follows. Let \( K \) be a set of literals in the language \( L \), and \( h \) some \( \text{HI} \) for \( L \), then

\[
h(K) = \begin{cases} 
  \text{false} & \text{iff there exists a grounding substitution,} \\
  & \text{TAU, for K such that } (K(TAU)) \cap h \text{ is empty.} \\
  \text{true} & \text{otherwise.}
\end{cases}
\]
2.1

We call \( h \) the evaluation function for the interpretation \( h \).

This definition of the evaluation function is consistent with the view of a set of literals being a formula which is a disjunction of the literals in the set and with the variable symbols being universally quantified, with the scope of quantification being the entire set of literals. Therefore we see that the set of literals to be evaluated is being treated as if it was a clause. Thus this definition of the evaluation function is the usual one used in resolution strategies. The reason that evaluation functions are not specifically defined to have clauses as the domain of definition is that there are resolution strategies which utilize evaluations of sets of literals, as we have defined them, but for which the sets of literals do not correspond to any clauses of the search space.

The Model Strategy

The Model Strategy (TMS) (Luckham, 1968) is a sound and complete refinement of unrestricted binary resolution. In TMS clauses are considered as sets of literals and some arbitrarily chosen but fixed HI, \( h \), is used for truth evaluations. Two clauses are allowed to resolve together iff they could be resolved together in unrestricted resolution and at least one of the clauses has the truth evaluation "false" in \( h \).
Hereditary-Lock Resolution

Hereditary-Lock Resolution (HL-Resolution, or HLR) (Sandford, 1977) is a sound and complete refinement of unrestricted resolution which combines both the Lock Resolution (Boyer, 1971) and TMS refinements. HLR is a refinement whose semantic component, like that of TMS, is confined to being a function solely of the truth evaluations of sets of literals with respect to some arbitrarily chosen but fixed HI. Thus TMS and HLR are quite similar with respect to their interfacing with the process of truth evaluations by some interpretation.

Semantic Resolution

Semantic Resolution (SR) (Slagle, 1967) is a sound and complete refinement of resolution which is clash oriented, and includes some literal ordering. The SR refinement, like TMS and HLR, has a semantic component which is solely a function of the truth evaluations of sets of literals.

Semantic Strategies and Functions

We call TMS, HLR, and SR, literal set semantic refinements. A literal set semantic refinement is a refinement in which the semantic (or model, or interpretation) based component of the refinement is only a function of the truth evaluations of sets of literals according to some function whose values are "true" or "false". This function is called the semantic function of the strategy. We define a semantic function for the language L to be
any total function on sets of literals from \( L \) into \{true, false\}. TMS, HLR and SR are all sound refinements, independently of what semantic function is used. These three strategies are each complete for the class of clause sets in the language \( L \) if the semantic function is taken to be the evaluation function for some \( \mathcal{H}, \mathcal{h} \), such that \( \mathcal{h} \in \mathcal{H}(L) \).

**False Permissive Semantic Functions**

Let \( s \) and \( s' \) be two semantic functions for the language \( L \), and let \( K \) range over sets of literals from \( L \). Then \( s' \) is said to be false permissive with respect to \( s \) iff

\[
\forall K: (s(K) = \text{false}) \implies s'(K) = \text{false}
\]

and \( s' \) is said to be properly false permissive with respect to \( s \) iff

\[
\exists K: s(K) = \text{true} \text{ and } s'(K) = \text{false} \text{ and } s' \text{ is false permissive with respect to } s.
\]

Clearly the relation of false permissive is transitive, as is the relation of properly false permissive.

**Sound Semantic Functions**

A semantic function, \( s \), for the language \( L \) is said to be sound iff there exists a \( \mathcal{h} \in \mathcal{H}(L) \) such that \( s \) is false permissive with respect to \( \mathcal{h} \). It is the case that if \( s' \) is false permissive with respect to \( s \), and \( s \) is sound, then \( s' \) is also sound.
False Permissive Complete Refinements

A literal set semantic refinement is said to be false permissive complete iff it is refutation complete over the class of clause sets of the language L for every sound semantic function for L.

TMS, HLR and SR are all false permissive complete. We do not prove this in this report. The proof of this is trivial provided that deletion strategies (Chang and Lee, 1973) (Nilsson, 1971) are not used. The completeness of HLR using both deletion strategies and false permissive semantic functions will be dealt with in future work. The completeness of TMS and SR with deletion strategies and false permissive semantic functions is an issue that has not been explored.
2.2 Basic Model Scheme Nomenclature

First Order Languages

A first order language will be specified by an ordered pair, the first element of which is the set of predicate symbols, and the second element of which is the set of function symbols of the language. Thus the language $L$ has a description which we write as

$$\text{DESC}(L) = \langle \text{PRED}(L), \text{FUN}(L) \rangle$$

where $\text{PRED}$ and $\text{FUN}$ are the functions which map languages to their sets of predicate and function symbols, respectively. We only consider languages, $L$, where $\text{PRED}(L)$ is non-empty. Each predicate and function symbol has associated with it a finite non-negative integer, called its arity. We only consider languages, $L$, such that $\text{FUN}(L)$ contains at least one function symbol of arity zero, and for which $\text{PRED}(L)$ and $\text{FUN}(L)$ are both finite sets. Function symbols of arity zero are called constants. A function (predicate) symbol of arity $k$ is also called a $k$-place function (predicate).

We assume the existence of an infinite set of variable symbols, and all languages share this same set of variable symbols implicitly. Also implicitly contained in every language description are the usual logical symbols for negation, quantification, "implies", "and", "or", and any others that are desired and can be defined in terms of those just explicitly mentioned. The definitions of terms, atoms, literals and well formed formulas are all standard. All unbound variable symbols are to be understood as universally quantified, except where explicitly...
noted differently. When writing formulas the letters 
\[x,y,z,u,v,w,q,r,s,t,x_1,y_1, \ldots, x_2,y_2, \ldots\]
will be variable symbols.

Most of the time we will consider expressions of theorem proving problems to have been already converted to conjunctive normal form as is usual in resolution.

Language Amicability

Let \( L \) and \( L' \) be two first order languages. Then \( L' \) is said to be amicable to \( L \) iff, for all \( k \), if there exists a \( k \)-place predicate symbol in \( \text{PRED}(L) \) then there exists at least one \( k \)-place predicate symbol in \( \text{PRED}(L') \), and if there exists a \( k \)-place function symbol in \( \text{FUN}(L) \) then there exists at least one \( k \)-place function symbol in \( \text{FUN}(L') \).

Translation Functions

A symbol translation function from \( L \) to \( L' \), usually denoted by "SYMT", is any function which for all \( k \), is a total function on the set of predicate (function) symbols of \( L \) of arity \( k \) into the set of predicate (function) symbols of \( L' \) of arity \( k \). There can be a symbol translation function from \( L \) to \( L' \) iff \( L' \) is amicable to \( L \). Notice that SYMT need not be a 1-1 mapping and that it need not be onto. We define \( \text{SYMT}(\text{ALPHA}) = \text{ALPHA} \) when \( \text{ALPHA} \) is any variable symbol, or the negation sign.
In the obvious way, SYMT induces a term translation function (TTF), and a literal translation function (LTF). A TTF applied to a list of terms has as its value the list of terms obtained by applying TTF to each individual term in the list. Similarly for LTF applied to a set of literals.

As an example, if the SYMT from L to L' is given by

- \( \text{SYMT}(f) = g \)
- \( \text{SYMT}(h) = g \)
- \( \text{SYMT}(p) = q \)
- \( \text{SYMT}(a) = b \)

then the set of literals

\[ \{P(a,f(x,y)), Q(h(y,a),a)\} \]

translates under the induced LTF to

\[ \{Q(b,g(x,y)), Q(q(y,b),b)\} \]

We further extend LTF to apply to substitutions of L, such that if \( \text{RHO} \) is a substitution of L, then

\[ \text{LTF} (\text{RHO}) = \{ \text{TTF}(t)/v : t/v \text{ is an element of RHO} \} \]

where v is a variable symbol and t is a term from \( \text{HU}(L) \). Thus LTF maps a substitution of L to a substitution of L'.

The Language \( L\backslash L' \)

Assume \( L' \) is amicable to \( L \) and some SYMT has been chosen. Let the subset of \( \text{PRED}(L') \) which is actually in the range of SYMT from \( L \) to \( L' \), be denoted by \( L\backslash\text{PRED}(L') \). Similarly for a \( L\backslash\text{FUN}(L') \). Then there is a first order language whose description is

\[
<L\backslash\text{PRED}(L'), L\backslash\text{FUN}(L')>.
\]

We denote this language by \( L\backslash L' \), where the dependence on a specific SYMT is not explicitly noted, but is assumed to be known from the particular context in which \( L\backslash L' \) is being used.

The Mapping of Interpretations

Let \( L' \) be amicable to \( L \), and let SYMT be some specific symbol translation function. This induces term and literal translation functions \( \text{TTF} \) and \( \text{LTF} \), and a language \( L\backslash L' \).

\( \text{LTF} \) was defined so as to map sets of literals from \( L \) to sets of literals from \( L' \), but clearly every set of literals in the range of \( \text{LTF} \) is also a set of literals in the language \( L\backslash L' \). A similar remark holds for \( \text{LTF} \) applied to substitutions.

Let \( K \) be a set of literals from \( L \). Then \( \text{LTF}(K) \) is called the homomorphic image of \( K \), or just the image of \( K \).

Let \( h \) be a HI of \( L \). If the image of \( h \) is a HI of \( L\backslash L' \), then this image is called a condensate of \( h \). In those situations where SYMT is not a 1-1 mapping there will exist a HI for \( L \) whose image is not a HI of \( L\backslash L' \), and therefore whose image is not a condensate.
Reduct and Expansion Relationships

Let $L'$ be amicable to $L$, and let $\text{SYMT}$ be a symbol translation function from $L$ to $L'$. Then, in general, for each $HI$, $h$, of $L \setminus L'$, there will be more than one $HI$, $h'$, of $L'$, such that $h \subseteq h'$ (remember that the predicate and function symbols of $L \setminus L'$ are a subset of those in $L'$).

Let $h$ be a $HI$ of $L \setminus L'$. The set

$$\{h': h \subseteq h' \text{ and } h' \in H(L')\}$$

is called the expansion cluster of $h$. For each $h'$ in the expansion cluster of $h$, $h'$ is called an expansion of $h$, and $h$ is called a $L \setminus L'$ reduct of $h'$, or just a reduct of $h'$.

Model Schemes

A model scheme for a language $L$ is any non-empty subset of $H(L)$. Notationally, a model scheme for a language $L$ will usually be written as $\text{MSL}$.

Evaluations Over Model Schemes

Let $K$ be a set of literals in the language $L$, and let $\text{"MSL}$ be a model scheme for $L$. Then we define a model scheme evaluation function, also denoted $\text{MSL}$, for the model scheme $\text{"MSL}$:
2.2

The model scheme evaluation of $K$ over $\text{MSL}$ is the value of $\text{MSL}(K)$.

**Associated Semantic Function**

Let $M$ be a model scheme for the language $L$, and let $K$ range over sets of literals of $L$. Then the function $s$ defined by

$$s(K) = \begin{cases} 
\text{false} & \text{iff } K \text{ is "F" over } M \\
\text{true} & \text{iff } K \text{ is "T" over } M 
\end{cases}$$

is called the associated semantic function for $M$.

Let $M$ and $M'$ be two model schemes for the same language. We say that $M$ is (properly) false permissive with respect to $M'$ iff the associated semantic function for $M$ is (properly) false permissive with respect to the associated semantic function for $M'$.

**Induced Model Schemes**

Let $L'$ be amicable to $L$, and SYMT be a symbol translation function from $L$ to $L'$. Then for every model scheme, $\text{MSL}'$, of $L'$, there is an induced (or corresponding) model scheme, $\text{MSL}L'$, of $L\backslash L'$, defined by

$$\text{MSL}L' = \{ m : m \text{ is the } L\backslash L' \text{ reduct of some } m' \text{ in } \text{MSL}' \}$$
Thus every model scheme in $L'$ induces a unique model scheme in $L \setminus L'$.

Let $MSL \setminus L'$ be any model scheme of the language $L \setminus L'$. Then there is an induced model scheme, $MSL$, of the language $L$, defined by

$$MSL = \{ m : LTF(m) \in MSL \setminus L' \}$$

Equivalently we can say that $MSL$ is the subset of $H(L)$ whose elements have condensates which are elements of $MSL \setminus L'$.

Thus every model scheme of $L \setminus L'$ induces a unique model scheme of $L$ (with respect to a given SYMT). As a result of this, every model scheme of $L'$ induces a unique model scheme of $L$. Notice that it is possible, in general, for two distinct model schemes in $L'$ to induce the same model scheme in $L \setminus L'$, and therefore the same model scheme in $L$. We refer to the collection of $L$, $L'$, $L \setminus L'$, SYMT, etc. as a model structure or model system.

**Model Scheme False Permissiveness**

Let $L'$ be a language amicable to the language $L$, and let $SYMT$ be a symbol translation function from $L$ to $L'$ which induces the literal translation function $LTF$. Let $M$ be a model scheme for $L$ and $M'$ a model scheme for $L'$. Then $M'$ is false permissive with respect to $M$ iff for all sets of literals, $K$, of $L$,

$$M(K) = "F" \implies M'(LTF(K)) = "F",$$

and $M'$ is properly false permissive with respect to $M$ iff it is
false permissive with respect to $M$ and there exists a set of literals, $K$, of $L$, such that

$$M(K) = "T" \text{ AND } M'(LTF(K)) = "F".$$ 

If it happens that $M$ is the induced model scheme corresponding to $M'$, then we say that the model structure is properly or improperly false permissive according to whether $M'$ is properly or improperly false permissive with respect to $M$. 
2.3 Basic Model Scheme Results

Lemma I

Let \( L' \) be amicable to \( L \), and \( \text{SYM} \) a symbol translation function from \( L \) to \( L' \), and let \( \text{MSL}' \) be any model scheme for \( L' \). Further let \( \text{MSL} \) be the model scheme in \( L \) induced by \( \text{MSL}' \), and let \( C \) be a set of literals in the language \( L \). Then for all grounding substitutions, \( \text{SIGMA} \), such that \( C(\text{SIGMA}) \) is false in some \( h \in \text{MSL} \), there exists at least one \( h' \in \text{MSL}' \) such that \( \text{LTF}(C(\text{SIGMA})) \) is false in \( h' \).

Proof: By the definition of how \( \text{MSL} \) is induced by a \( \text{MSL}' \), which is in turn induced by the \( \text{MSL}' \) of the lemma hypothesis, it is the case that \( h \in \text{MSL} \) must be such that there exists an \( x \in \text{MSL}' \) such that \( \text{LTF}(h) = x \). By the definition of a literal translation function from \( L \) to \( L' \), it is the case that if \( C(\text{SIGMA}) \cap h \) is empty, so is

\[
\text{LTF}(C(\text{SIGMA})) \cap \text{LTF}(h) = \text{LTF}(C(\text{SIGMA})) \cap x.
\]

But \( x \) must be the reduct of at least one interpretation in \( \text{MSL}' \). Let \( h' \) be any interpretation which is both in \( \text{MSL}' \) and in the expansion cluster of \( x \). This \( h' \) is an expansion of \( x \). Therefore \( \text{LTF}(C(\text{SIGMA})) \) will also be false in \( h' \).
Lemma II

Let L, L', SYMT, MSL, 'MSL' and C be the same as in the hypothesis of Lemma I. Then if the MSL model scheme evaluation of C is "F", then the MSL' model scheme evaluation of LTF(C) is "F".

proof: If C is "F" over MSL, then there must exist a grounding substitution, SIGMA, of C, and a HI, h, in MSL such that C(SIGMA) is false in h. By Lemma I there is then some h' ∈ 'MSL' such that LTF(C(SIGMA)) is false in h'. But by the definition of a literal translation function, it is the case that

\[ LTF(C(SIGMA)) = (LTF(C))(LTF(SIGMA)). \]

Thus there is a substitution, LTF(SIGMA), in the language L', and an interpretation, h' ∈ 'MSL', such that (LTF(C))(LTF(SIGMA)) is false in h'. Therefore LTF(C) is "F" over 'MSL'.

Lemma III

Let L, L', SYMT, 'MSL' and MSL be the same as in the hypothesis of Lemma I. Let C be a ground set of literals in L whose model scheme evaluation is "T" over MSL. Then the model scheme evaluation of LTF(C) over 'MSL' is also "T".

proof: Let h' be an arbitrary HI in 'MSL'. Then there exists a h'' in 'MSL\L' which is the reduct of h', and an h in MSL such that h'' is the condensate of h. By hypothesis C ∩ h is non-empty. Therefore LTF(C) ∩ h'' is non-empty. Therefore LTF(C) ∩ h' is non-empty. Thus LTF(C) is true in every h' in 'MSL' and consequently LTF(C) is "T" over 'MSL'.

At this point in the development one would like to have a result that shows that for any non-ground level set of literals, C, such that C evaluates to "T" over MSL, that LTF(C) evaluates to "T" over MSL'. However, such is not the case for the situations we are talking about in this report. In section 2.6 an example is given that illustrates this point.
2.4 Basic Model Scheme Specification

We now show how to specify model schemes and obtain model scheme evaluations in the language L'.

**Satisfying Interpretation Set**

Let L' be amicable to L, and let SYMT be a symbol translation function from L to L'. Let MSF be a set of first order well formed formula in the language L'. Let MSF be satisfiable. Then there exists a non-empty subset of \( \mathcal{H}(L') \) which consists of exactly those interpretations which satisfy MSF. We call this set the satisfying interpretation set of MSF, and denote it by \( \text{SIS}(\text{MSF}) \). By the definition of a model scheme, \( \text{SIS}(\text{MSF}) \) is a model scheme of L'.

We wish to be able to perform model scheme evaluations of sets of literals over the model scheme \( \text{SIS}(\text{MSF}) \).

**Theorem 1**

If \( K \) is a set of literals, \( K = \{k_1, k_2, \ldots, k_n\} \), in the language L', and MSF is a set of satisfiable formula in L', then the model scheme evaluation of K is "T" over \( \text{SIS}(\text{MSF}) \) iff the set of statements

\[
[\text{MSF}] \cup [K]
\]

is a satisfiable set of statements. (N.B. The meaning of "?" applied to a set of literals is that all variables are changed from universal to existential quantification and the negation status of
each literal in the set is inverted, and the resulting literals are
to be treated as a conjunction.)

proof:

(=>) Assume \(K\) is "F" over SIS(MSF). Then there exists an
interpretation, \(h'\), in SIS('MSF), and a substitution, RHO, such that

\[
\{(k_1(RHO)), (k_2(RHO)), \ldots (k_n(RHO))\}
\]

are all false ground literals in \(h'\). But then the set of literals

\[
\{\neg(k_1(RHO)), \neg(k_2(RHO)), \ldots \neg(k_n(RHO))\}
\]

is a set of ground literals all of which are true in \(h'\). Thus the
set of literals \(\neg(K(RHO))\) is true in \(h'\), and therefore the set of
literals \(\neg K\) is true in \(h'\). Since MSF is true in \(h'\) by the
definition of \(h'\) as an element of SIS(MSF), \(\{MSF\} \cup \{\neg K\}\) is true in
\(h'\), and is therefore satisfiable.

(<=) Assume \(\{MSF\} \cup \{\neg K\}\) is satisfiable. Then there exists a \(h, h'\), which is an element of SIS(MSF), and for which there exists a
substitution, RHO, such that

\[
\neg(k_1(RHO)), \neg(k_2(RHO)), \ldots \neg(k_n(RHO))
\]

is a set of literals all of which are true in \(h\). Thus the set of
literals

\[
K(RHO) = \{k_1(RHO), k_2(RHO), \ldots k_n(RHO)\}
\]

is a set such that every literal in it is false in \(h\). Thus there
exists a substitution, RHO, and a \(h, h'\), such that \(K(RHO)\) is false
in \(h\). Thus the model evaluation of \(K\) over the model scheme
SIS(MSF), which contains \(h\), is "F".
At this point we have connected the notion of model scheme evaluation to the notion of satisfiability of the negation of the set of literals being evaluated and the set of formula specifying the model scheme, all in the language L'. Thus the model scheme evaluations can be computed by any decision procedure for satisfiability testing. For ease of reference we call the model scheme evaluation in L' of a set of literals over the SIS(MSF) scheme as either the MSF evaluation or the L' evaluation.

Depending upon the particular language, L', and the particular set of scheme defining statements MSF, there may or may not exist a decision procedure for satisfiability testing for the class of sets of literals we are interested in evaluating. The class of sets of literals that we will want to evaluate is itself dependent upon the language L, the chosen SYMV, the set of input statements in L, and the particular strategy being employed for theorem proving in L. Thus it would seem in general, to be difficult, if not actually impossible to ensure that some arbitrarily established model scheme would be tractable. While this is true, it causes considerably less difficulty than one would at first think. Why this is the case is the main result of this report, and we now continue directly to the development of that result.
2.5 A Sound and Complete Refinement Using Incomplete Model Schemes

Theorem II

Let \( L \) be a first order language and \( MSL \) be any model scheme for \( L \). Let \( R \) be a false permissive complete resolution strategy, with semantic function \( h \) defined by:

\[
h(K) = \begin{cases} 
\text{false} & \text{iff } K \text{ is "F" over the model scheme } MSL. \\
\text{true} & \text{iff } K \text{ is "T" over the model scheme } MSL.
\end{cases}
\]

(i.e. \( h \) is the associated semantic function for \( MSL \)). Then \( R \) will be a complete refinement strategy when using this \( h \) as its semantic function.

proof: Since \( R \) is false permissive complete, one need merely show that \( h \) as defined above, is a sound semantic function. This can be done by showing that \( h \) is false permissive with respect to some \( h' \in \mathcal{H}(L) \), as follows. Let \( h' \) be any fixed element of \( MSL \). Then for any set of literals, \( K \), if \( h'(K) \) is false, then \( MSL(K) = "F" \), and by the definition of \( h \) given in the theorem, \( h(K) \) is false. Thus \( h \) is false permissive with respect to the \( \mathcal{H} \) \( h' \). Thus \( h \) is a sound semantic function.

We are now ready for the main theorem of this report.
Theorem III

Let $R$ be a false permissive complete resolution refinement for the language $L$. Let $L'$ be any language amicable to $L$, and $LTF$ a literal translation function from $L$ to $L'$. Let $R'$ be a sound refutation procedure for the language $L'$, which is not necessarily complete, and which always terminates in a finite amount of time when evaluating a set of statements (reporting only refutation success or failure). Let $MSF$ be any set of satisfiable statements in the language $L'$. Let $s$ be a semantic function defined for sets of literals of $L$ by:

$$s(K) = \begin{cases} 
\text{false} & \text{iff } R'(\exists (LTF(K)) \cup MSF) = \text{failure} \\
\text{true} & \text{iff } R'(\exists (LTF(K)) \cup MSF) = \text{success} 
\end{cases}$$

Then $R$ will be a complete strategy when using $s$ as its semantic function.

proof: It is only necessary to show that $s$ is a sound semantic function. First we notice that $MSF$ induces a model scheme, $M_{SL'} = \text{SIS}(MSF)$, of $L'$, and a corresponding model scheme, $M_{SL}$, of $L$. Consider the semantic function $h$, as defined in the previous theorem. It was shown to be sound. Consider the semantic function $s^*$, defined on sets of literals, $K$, of $L$, by:
By Lemma II we can assert that \( s^* \) is false permissive with respect to \( h \). But by Theorem I and the soundness of \( R' \), it is the case that \( s \) is false permissive with respect to \( s^* \). Therefore \( s \) is false permissive with respect to \( h \). But \( h \) is a sound semantic function. Thus \( s \) is a sound semantic function.

Theorem III is a crude but interesting result. What we have achieved is the establishment of a structure which allows a complete theorem proving search process to be guided by an interpretation environment which is itself incomplete. The completeness of the theorem proving process rests only on the soundness of the interpretation process. If the semantic strategy used is TMS, HLR, or SR, then the theorem proving process is also sound.
2.6 An Example of a Basic Modeling Structure

Our example uses a language of the clauses, \( L \), and a language of the model, \( L' \), which are both such that they each have only a finite number of distinct HI's.

We consider a language \( L \) which has one two place predicate symbol and two constants:

\[
\text{DESC}(L) = \langle \{=,\rightarrow\}, \{c_1, c_2\} \rangle.
\]

Thus the Herbrand universe of \( L \) (\( \text{HU}(L) \)) is just the set consisting of the two constants. The Herbrand base of \( L \) (\( \text{HB}(L) \)) is:

\[
\text{HB}(L) = \{ (c_1, c_1), (c_1, c_2), (c_2, c_1), (c_2, c_2) \}
\]

Thus there are 16 possible Herbrand interpretations for \( L \).

We will set up a model in the language \( L' \) such that the induced model scheme in \( L \), \( \text{MSL} \), will be the singleton set consisting of the following interpretation:

\[
h = \{ (c_1, c_1), (c_2, c_2), @=(c_1, c_2), @=(c_2, c_1) \}
\]

Thus we will have \( \text{MSL} = \{h\} \).

We choose \( L' \) to be specified by:

\[
\text{DESC}(L') = \langle \{R(-,-)\}, \{k_1, k_2, k_3\} \rangle.
\]

The set \( \text{HU}(L') \) has the three constants as its only members and the set \( \text{HB}(L') \) has 9 ground atoms in it. Therefore there are 512 possible HI's for \( L' \).
We choose the following set of clauses as the defining set, \( MSF \), of clauses for the model scheme, \( MSL' \):

1. \[ \exists R(k_1, k_2); \]
2. \[ \exists R(k_1, k_3); \]
3. \[ \exists R(k_2, k_3); \]
4. \[ R(x, x); \]
5. \[ \exists R(x, y), R(y, x); \]
6. \[ \exists R(x, y), \exists R(y, z), R(x, z); \]
7. \[ R(x, k_1), R(x, k_2), R(x, k_3); \]

What we have done here by using clauses 4, 5 and 6, is to allow only interpretations for \( L' \) which treat \( R \) as an equivalence relation over the elements of \( HU(L') \). Then clauses 1, 2 and 3, allow only interpretations for which \( k_1 \), \( k_2 \), and \( k_3 \) are all inequivalent. Finally clause 7 restricts the allowed interpretations to those in which every \( HU \) element is equivalent to one of \( k_1 \), \( k_2 \) or \( k_3 \). There is exactly one \( HI \) for \( L' \) which satisfies \( MSF \). We call this interpretation \( h' \):

\[
h' = \{ R(k_1, k_1), R(k_2, k_2), R(k_3, k_3), \exists R(k_1, k_2),
\exists R(k_1, k_3), \exists R(k_2, k_1), \exists R(k_2, k_3), \exists R(k_3, k_1), \exists R(k_3, k_2) \}
\]

Thus we have the following:

\[ MSL' = SIS(MSF) = \{ h' \} \]

The function \( SYMT \) is chosen to be:

\[ SYMT = \{ \langle c_1, k_1 \rangle, \langle c_2, k_2 \rangle, \langle =, P \rangle \} \]

The \( HU, H3, \) etc., for the language \( L\backslash L' \) are all identical to the
ones for L except that k1 replaces cl, k2 replaces c2, and R replaces "=". Thus, for example, 'MSL\L' would be the singleton set \{h*\}, where
\[
h* = \{R(k1,k1), R(k2,k2), \exists R(k1,k2), \forall R(k2,k1)\}
\]
The reader should be able to verify that MSF is satisfied only by h' in H(L'), and that h* is the reduct of h', and that h* is the condensate of h. Thus h and h' are corresponding interpretations, and given that MSL' = \{h'\} it is the case that MSL = \{h\}.

We now exhibit some clause evaluations. Where it is otherwise unclear, sets of literals that form a clause are underlined. First consider the clause C, in the language L, where
\[
C = = (x,cl),= (x,c2);
\]
It is the case that h(C) = true, and thus :MSL(C) = "T". Also it is the case that h*(LTF(C)) = h*({R(x,k1),R(x,k2)}) = true. Thus :

\[
:\text{MSL\L'}(LTF(C)) = "T".
\]
On the other hand
\[
h'(LTF(C)) = h'({R(x,k1),R(x,k2)}) = false
\]
since the substitution TAU = {k3/x} is such that
\[
h'((LTF(C))(TAU)) = h'({R(k3,k1),P(k3,k2)}) = false.
\]
Thus MSL'(LTF(C)) = "F". We see that the model scheme, 'MSL', even though it consists of just one HI, is still properly false permissive with respect to MSL.
An example of a clause that is "T" over both MSL and MSL' is the clause D:

\[ D = \{(x, c_1), (x, c_2), (y, c_1), (y, c_2), (x, y)\} \]

It should be clear that MSL(D) = "T" since every grounding substitution for D in the language L must substitute either c_1 or c_2 for x, and thus either the first or second literal of D will be true in h. To show that MSL'(LTF(D)) = "T", i.e. that h'(LTF(D)) = true, we show that Q(LTF(D)) \Upsilon \text{MSF} is unsatisfiable, as follows.

We have as the clause set 7 clauses from \text{MSF}, listed above, and 5 more clauses, numbered 8 through 12 which are from \text{Q(LTF(D))}:

\begin{align*}
3. & \exists R(a, k_1); \\
9. & \exists R(a, k_2); \\
10. & \exists R(b, k_1); \\
11. & \exists R(b, k_2); \\
12. & \exists R(a, b);
\end{align*}

The constants a and b are zero place Skolem functions introduced by the process of negating the set of translated literals. The reader should keep in mind that this set of 12 clauses is not in the language L', since L' does not contain either a or b. However this set of 12 clauses is unsatisfiable iff \text{Q(LTF(D))} \Upsilon \text{MSF} is unsatisfiable.
A refutation of this set of 12 clauses is:

\[
\begin{align*}
7 \times 8 & = 13. \ R(a, k2), R(a, k3); \\
9 \times 13 & = 14. \ R(a, k3); \\
7 \times 10 & = 15. \ R(b, k2), R(b, k3); \\
11 \times 15 & = 16. \ P(b, k3); \\
5 \times 16 & = 17. \ R(k3, b); \\
6 \times 14 & = 18. \ R(k3, z), R(a, z); \\
17 \times 13 & = 19. \ R(a, b); \\
12 \times 19 & = 20. \ \ast \text{BOX}\ast;
\end{align*}
\]

Thus \( \exists (LTF(D)) \cup \text{MSF} \) is unsatisfiable, and it is the case that \( \text{MSL}'(LTF(D)) = "T" \).

An example of a clause which is "F" over both MSL and MSL' is clause E:

\[
E = \exists (x, y), = (x, c1), = (y, c1);
\]

which is false in h (the falsifying substitution is \( \{c2/x, c2/y\} \), and \( \text{MSL}'(LTF(E)) = \text{MSL}'(\{R(x, y), R(x, k1), R(y, k1)\}) = "F" \) since the set of clauses

\[
\{ R(c, d), \exists R(c, k1), \exists R(d, k1) \} \cup \text{MSF}
\]

is satisfiable (where c and d are again new Skolem constants introduced by negating LTF(E)).
Chapter 2 has presented an abstract structured situation in which a sound and complete theorem proving search can be guided in a bottom up fashion by a sound but incomplete model.

The incompleteness of the model has two distinct sources. One is simply that we wish to be unbiased in those situations where the available information is insufficient to specify a single interpretation. This is a pragmatic choice which leads naturally to the notion of a model scheme as a set of interpretations each of which satisfies a common set of conditions (the available model specification as a set of scheme defining statements, MEF, in the model language L'). The second source of model incompleteness is related to the question of efficiency and practicality of performing model evaluations. We are willing to permit an increase in the degree of false permissiveness in the model evaluations to obtain a reduction in the average amount of effort expended in performing model evaluations. In those cases where there exists no decision procedure for the model evaluations this increased false permissiveness is mandatory. In other cases it is merely a tradeoff designed to reduce total theorem proving effort. This tradeoff arises because, for the false permissive complete strategies, as the model evaluation effort is reduced by allowing an increase in the degree of false permissiveness of the semantic function, the syntactic search space contains an increasing number of clauses at each level. A theoretical analysis of this tradeoff seems well beyond current capabilities, and an understanding of the
dynamics of this situation will have to be sought in empirical investigations. For a discussion of model incompleteness and the difficulties it causes in problem solving searches expressed in the PLANNEER formalism see Moore (Moore, 1975). These difficulties do not arise in the context of the theorem proving formalism using false permissive complete semantic refinements.

In the example given in section 2.6 we saw a modeling structure which was demonstratably false permissive for clause C in the sense that MSL(C) = "T" while MSL'(LTP(C)) = "F". This false permissiveness is not a result of indefiniteness as to which interpretation is to be used, since the model schemes MSL and MSL' each consisted of just one interpretation. Neither was it a result of incompleteness of the satisfiability testing inside the model scheme MSL'. Its origin is the fact that L' is a language which has a more extensive Herbrand universe than does L. As a result, when using the particular SYMT of the example, a statement with universally quantified variables may be true in L, but its translated form in L' is not necessarily true in the larger Herbrand universe of L'. This is a specific case of a more general phenomenon of model structures as presented in this report. This phenomenon is that the model scheme in L' can represent and manipulate individuals, functions, and predicates which are inexpressible in the language L. It is this increased capacity of the model to represent information that gives rise to the expansion clusters. We can think of the model as dealing only with an abstraction of the original problem, but dealing with that abstraction in a context which brings in constructs which are
unobservable in the original problem structure. The purpose of including these unobservable constructs is to give the model sufficient structure so that it may be efficiently manipulated in performing evaluations. We avoid speculation here on the ultimate role and utility of unobservables in the type of modeling situations we are concerned with in this report.

We now discuss some issues concerning the practical application of the structures presented in this report. It should already be clear that the efficiency of the model evaluation process is of importance. Model schemes have been abstractly described as sets of formula, MSF, in L'. In the typical situation the structure of these formula will be such that it will be prohibitive to perform model evaluations in L' as a theorem proving type computation using MSF. Thus one criteria in establishing a model scheme is to make sure that there will exist a practical evaluation mechanism. This is a largely uninvestigated area at present, but some of the possibilities are clear. One possibility is that MSF constitutes a decidable theory in L', and there is an efficient decision procedure known. An example of this is the theory of simultaneous linear equations over real variables. This model is illustrated in an example problem in TR-30. This model has also been programmed and empirical results have been obtained using it with HL-Resolution. Cooper has presented a specific decision procedure (Cooper, 1972) for this model.
Another possibility is that no efficient decision procedure is known for the model scheme SIS(MSF). This may be because MSF is an undecidable theory, or is decidable but is inherently complex, or we may just be ignorant of any existing decision procedure. In this case we could try pure theorem proving procedures on MSF with some cutoff on the effort expended. This would introduce some additional incompleteness (and thus increase the degree of false permissiveness in the associated semantic function of SIS(MSF)) but it would be an acceptable way to proceed in some circumstances. Alternatively one might try to model the set of statements "SF itself by another model structure. Notice that such a second level of modeling is not just an iteration or recursion type of relationship of the first level of modeling. The first level model must be sound, but can be incomplete. The second level model need neither be sound nor complete, just so long as its answers are checked by the first level model. At present no work has been done in exploring the use of multiple levels of models.

The overall generality of the paradigm is high; the model language, L', can be any first order language, and the model scheme can be defined by any set of satisfiable formula of L'. The "IDS modeling capability is, however, more than general enough to be well matched to this paradigm for models.
REFERENCES


<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FA</td>
<td>The quantifier &quot;for all&quot;</td>
</tr>
<tr>
<td>H(L)</td>
<td>Set of all HI's for the language L</td>
</tr>
<tr>
<td>HB(L)</td>
<td>Herbrand base for the language L</td>
</tr>
<tr>
<td>HI</td>
<td>Herbrand interpretation</td>
</tr>
<tr>
<td>HLR</td>
<td>Hereditary-Lock Resolution</td>
</tr>
<tr>
<td>HU(L)</td>
<td>Herbrand universe for the language L</td>
</tr>
<tr>
<td>LTF</td>
<td>Literal translation function</td>
</tr>
<tr>
<td>MSL (&quot;L', MSL\L'&quot;)</td>
<td>Model scheme in the language L</td>
</tr>
<tr>
<td></td>
<td>(in the languages L', L\L')</td>
</tr>
<tr>
<td>SR</td>
<td>Semantic Resolution</td>
</tr>
<tr>
<td>SYMT</td>
<td>Symbol translation function</td>
</tr>
<tr>
<td>TE</td>
<td>The quantifier &quot;there exists&quot;</td>
</tr>
<tr>
<td>TMS</td>
<td>The Model Strategy</td>
</tr>
<tr>
<td>TTF</td>
<td>Term translation function</td>
</tr>
<tr>
<td><em>BCX</em></td>
<td>The empty list (or set)</td>
</tr>
<tr>
<td></td>
<td>of literals</td>
</tr>
<tr>
<td>\</td>
<td>negation sign</td>
</tr>
</tbody>
</table>