CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

NL 339 (r. 86/09)

IMPLEMENTATION STRATEGIES FOR A PLAN-BASED DEDUCTION SYSTEM

by

Kenneth Wesley Forsythe

Thesis

submitted to the School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Masters

in

Computer Science

University of Ottawa

May 1984

ACKNOWLEDGEMENTS

I would like to thank Dr. S. Matwin for his encouragement, support and much needed advice provided during my work on this thesis. I would also like to thank my fiancee Lam Nguyen for her patience and assistance during this time as well.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ABSTRACT

A plan-based deduction system is a mechanical theorem prover which utilizes an intelligent backtracking strategy. This type of system eliminates much of the futile repetitive deductions associated with standard linear backtracking techniques. During implementation of such a system several important questions concerning the control of the deduction process arose with special emphasis on maintaining a complete solution space. This thesis examines these issues and presents solutions to the problems raised. It also takes a brief look at related work and shows that the issues discussed here, have been realized but largely ignored. Finally, the results of applying our anti-redundancy strategy to a well known computer vision problem are presented.

INTRODUCTION

With the advent of digital computers the concept of building a "thinking machine" has gained wide popularity. The research directed towards this idea has been categorized by computer scientists as the field of "Artificial Intelligence". Incorporated into this field is the study of mathematical logic in context with its application to computer reasoning processes. This category of logic is more specifically known as mechanical theorem proving or - more recently - as automated deduction.

Given a theorem expressed in mathematical logic, mechanical theorem proving is an algorithmic method of ascertaining if that theorem is correct, based on previously known facts and axioms. Interest in finding such an algorithm was first shown by mathematicians such as Leibniz (c 1700), and

then later by Peano (c 1900) and Hilbert (c 1920) (see [Chang & Lee 75]). Unfortunately, in 1936 both Church and Turing showed that proving a theorem mechanically is an undecidable problem. In other words, there is no general decision procedure that will prove the validity of formulas of first order logic, because for invalid formulas, the procedure may never terminate. Around 1930, Herbrand took a different approach and derived a mechanical method which can prove a given formula is false. For valid formulas, however, no such proof exists and the process will eventually terminate. Herbrand's method renewed interest in finding a theorem prover but it wasn't until the advent of digital computers that practical applications became feasible as his algorithm is very time consuming.

Gilmore (1960) first attempted to implement Herbrand's method on a digital computer. Since this strategy would only prove false formulas as being false, he negated those formulas he wished to prove as true. This approach for proving theorems is called a refutation procedure and is now the most common strategy. However, the implementation of Herbrand's original algorithm overwhelmed the computer's resources and the need for a more practical solution was recognized. Davis and Putnam (1960) made a few improvements to the algorithm but still could not reduce its complexity, sufficiently. A major breakthrough came in 1965, when Robin-

son introduced his so-called resolution principle. He developed a refutation procedure which eliminated much of the ineffiency inherent to Herbrand's method. Since then, there has been much research to further refine Robinson's resolution procedures. These refinements include semantic resolution, lock resolution, linear resolution, unit resolution and derivations of these.

All these types of resolution procedures are based on negating the formula representing original theorem, and then proving that this new formula is false, which allows the conclusion that the original formula, O, is true. This involves showing that for all valuations, V, $V(\neg O)$ = False, i.e. $V(O)$ = True, hence $\vdash O$, where O is in Disjunctive Normal Form after skolemizing. To show $V(\neg O)$ = False, each literal in $\neg O$ is resolved with a complementary literal from some given set of therorems; this is called a deduction step. Whenever a deduction step fails, that is we cannot show $V(\neg O)$ = False, then we backtrack to a literal which can be resolved with a different complementary literal from the known set of theorems. This process continues searching for valid deductions until a refutation is found or until all the theorems in the set have been tried and rejected. To control which literals have been resolved with which theorems from the set have been tested, there must be a search strategy and a backtracking algorithm built into the

deduction process. Most of the more well known resolution procedures use a depth-first search with linear backtracking techniques.

Linear backtracking, also called "chronological" or "blind" backtracking, means that subproblems created during the deduction process are linearly ordered and solved in that order. If a failure of a subproblem occurs the system will backtrack to the last such subproblem for which an alternative solution exists, discarding all deductions made between these two points. The system then tries to solve this subproblem using the alternative. The reason for this is that the algorithm does not analyze where the source of failure is, but tries to find a solution through trial and error according to the preset search strategy, until all alternatives have been exhausted. This means that when an alternative match is selected, the deduction process continues as though it was attempting refutation for the first time. Any previously computed subproblem, whose solution was discarded via backtracking, has to be solved again. If the source of failure occurred near the beginning of the process then many of the subproblems may have to be continually recalculated. It is clear that in the worst case this type of behavior will generate an exponential number of deduction steps (in searching for a solution) compared to the actual number of facts and axioms that are used.

In order to eliminate this repetitive processing in a deduction system, two criteria have to be met. The first one is that all the sources of failure of the subproblem must be determined. The second criterion is that the information must be stored in such a way that backtracking from any source of failure is possible. If these two requirements are met then the artificial ordering imposed on the solving of subproblems can be ignored and intelligent backtracking can be implemented.

In 1982, Pietrzykowski and Matwin [Pietrzykowski & Matwin 82] devised a refutation procedure based on linear resolution which incorporates an intelligent backtracking strategy. This strategy determines exactly which deductions lead to the failure of the refutation process, and which of these are appropriate backtracking points. Backtracking starts at a given point and traces back through the chain of successive deductions to the formula O looking for alternative deductions, where O represents the original negated theorem and is in Disjunctive Normal Form after skolemization. Each backtracking point is tried in turn until a refutation has been found or the set of backtracking points is exhausted. If backtracking terminates unsuccessfully from every point then a refutation is said not to exist. In many cases this strategy will limit the number of deduction steps to be

linearly proportional to the number of facts and axioms used.

To illustrate this concept consider the following example set of clauses with the first one being the goal statement (or theorem) to be refuted.

P(x)Q(x,y)R(z)S(y,z)

-P(x)N(x,y)

-Q(x,x)

-R(d)

-R(e)

-R(f)

-S(c,d)

-N(a,c)

-N(b,c)

-N(c,c)

Example 1.

In this example the capital letters N, P, Q, R and S represent predicate names, the letters x, y and z represent variables and the letters a, b, c, d, e and f represent constants. (The convention used throughout the paper will be the use of the capital letters M through T for predicate names, the small letters a through h for functions and constants and the small letters u through z for variables.) If we examine the deduction process on this set of clauses using standard (blind) backtracking techniques we would find that the number of deduction steps needed would be 21. This is because the order in which the literals are resolved is P - N - Q - R - S, which yields a backtracking order of S - R

- Q - N - P. The system must backtrack through all the alternatives for these predicates in this order. That is, all the alternatives for a predicate are retried each time the system backtracks past it. In this example the three alternatives for R are attempted each time a choice for N is tried. This progress of the deduction process is shown in figure 1. Each line shows the lists of literals resolved before a failure occurs. The literal at which backtracking succeeds, is underlined and the number of deductions made is in parenthesis at the end of the line.

-P(x) -N(a,c) -Q(x,x) <u>-R(d)</u> -S(c,d)	(5)
-P(x) -N(a,c) -Q(x,x) <u>-R(e)</u> -S(c,d)	(2)
-P(x) <u>-N(a,c)</u> -Q(x,x) -R(e) -S(c,d)	(2)
-P(x) -N(b,c) -Q(x,x) <u>-R(d)</u> -S(c,d)	(4)
-P(x) -N(b,c) -Q(x,x) <u>-R(e)</u> -S(c,d)	(2)
-P(x) <u>-N(b,c)</u> -Q(x,x) -R(e) -S(c,d)	(2)
-P(x) -N(c,c) -Q(x,x) -R(d) -S(c,d)	(4)

Figure 1. A trace of example 1 using blind backtracking.

However, if this artificial order imposed on the selection of backtracking points was removed then this repetition could be eliminated. The intelligent system proposed

by Pietrzykowski and Matwin first establishes that the sources of failure after the first attempt at deduction, are with the predicates N, Q and S. Since backtracking terminates at the goal statement, the backtracking path for N starts at N and terminates at P; and the paths for Q and S consists only of Q and S, respectively. Backtracking would then be invoked from one of the three backtracking points. Since predicates Q and S have no other alternatives, backtracking from these points fails immediately. Backtracking from N succeeds and the second alternative for N is tried. This too, fails to yield a refutation and the same three predicates are determined as appropriate backtracking points. Q and S still have no alternatives, but backtracking from N succeeds. Trying the last alternative for N produces the desired refutation after executing a total of 7 deduction steps. This reduction in the number of steps has two sources: the elimination of trying the alternatives associated with predicate R when it is clear that R has no relationship to the source of the problem, and the termination of the backtracking process when the goal statement is encountered. The trace of the deduction process using intelligent backtracking techniques is shown in figure 2. Each line is the same as in the figure above except that a "|" divides some of the literals to show that backtracking is restricted to these partitions.

-P(x) -N(a,c) | -Q(x,x) | -R(d) | -S(c,d) (5)

-P(x) -N(b,c) | -Q(x,x) | -R(d) | -S(c,d) (1)

-P(x) -N(c,c) | -Q(x,x) | -R(d) | -S(c,d) (1)

Figure 2. A trace of example 1 using intelligent backtracking.

As mentioned above, this intelligent backtracking technique has been utilized by Pietrzykowski and Matwin in a plan-based deduction system. The author's association with this research was initially with the development of a computer implementation of their results. However, in the testing of this plan-based deduction system it was found that certain inefficiencies, referred to as redundancies, are inherent to this type of backtracking algorithm. During the deduction process, a deduction step fails because the fact or axiom being tried as a solution is incompatible with an existing fact or axiom. This means there exists at least two points from which to backtrack from every time a failure occurs.

As the system tries to obtain a refutation from processing each of these backtracking points it is possible that some overlapping of effort occurs. This duplication of effort arises from the system designating two or more back-

tracking points every time a deduction step fails, but it only resolves one of them in a single deduction step. Now, suppose the next step fails too, and the same backtracking points are determined. Likewise, suppose that when all of the other original set of backtracking points get resolved they, too, fail with the same identical backtracking points. We now have the situation where there are more backtracking points then there are literals to be resolved; consequently, duplicate deductions are generated. Another cause of redundancy arises from attempting to obtain multiple refutations for the same theorem. When a refutation is found there are no sources of incompatibility from which to derive a backtracking point. Consequently, in order for the refutation process to continue, artificial failures must be induced into the solution space so that backtracking will be initiated and another refutation found. It was discovered that in order to generate a complete solution set without redundant refutations a sophisticated algorithm was needed to induce these artificial failures. It was also found that the performance of the system was dependent on when the backtracking procedure was initiated, as the solutions to some subproblems could be delayed.

The derivation of algorithms which eliminate these inefficiencies from the plan-based deduction system of Pietrzy-

kowski and Matwin is the topic of this thesis <1>. It is organized into the following sections. The first section presents an introduction to the work of Pietrzykowski and Matwin and defines their terminology which will be used throughout this paper. The second section discusses current related work in this field and shows that these problems have not been addressed. The next three sections present the author's solutions to these three deficiencies of the original system mentioned above. Section six shows how our deduction process applies to a real world situation and the last section contains our concluding remarks.

<1> Part of this research has been previously published in [Forsythe & Matwin 84], and also as a University of Ottawa Technical Report.

PLAN-BASED DEDUCTION

This section presents the research of Pietrzykowski and Matwin and defines the terminology used in their papers: [Pietrzykowski & Matwin 82], [Matwin & Pietrzykowski 82] and [Matwin & Pietrzykowski 84]. The reader is assumed to have a sufficient understanding of first order logic and mechanical theorem proving and is directed to works such as [Chang & Lee 75] for a more complete study of this field.

As mentioned in the introduction, the main intent of Pietrzykowski and Matwin was to propose a deduction system which eliminates the redundancy inherent to linear backtracking systems. To achieve this they needed a method of pinpointing the sources which cause a deduction step to fail and storing the deductions in such a way that backtracking could begin from any one of these sources.

In order to meet these criteria they divided the deduction process into two phases. The first of these is the preprocessing phase which determines which pairs of literals are potential resolvents. That is, all the clauses in the base are scanned for pairs of complementary literals and the most general unifier for each, if it exists, is recorded. The *base* refers to all the database of known clauses plus the theorem to be refuted. The second phase is the dynamic processing phase. In this phase the deduction algorithm is applied and a refutation sought. This consists of refuting all the literals in the goal statement (the original theorem negated). As a resolvent for each literal is found, the most general unifier for them is merged with the most general unifier computed for the deduction process thus far. Any new literals resulting from the logical consequence of the two resolved literals are also refuted. When all the literals have been refuted the most general unifier for the plan is checked to see if any illegal substitutions have been made. If none have then a refutation has been found, otherwise the backtracking procedures are invoked and an alternative resolvent tried.

The description of this deduction system will be divided into two subsections. The first will describe the data structures and the second the main processes. These are presented next.

THE DATA STRUCTURES

This subsection presents the four main data structures used in Pietrzykowski and Matwin's plan-based deduction systems. These four structures are the Basic Structure (BS), the Graph of Basic Constraints (GBC), the Graph of Dynamic Constraints (GDC) and the Plan. These data structures are the key to intelligent backtracking as they provide the means to finding where the sources of a failure in a deduction step are located, plus the ability to backtrack from them. These structures are basically graph structures with special links to facilitate traversal within a structure and between structures. The functions of these four structures are discussed below.

The *Basic Structure* is a representation of the base. Special emphasis is given to the handling of variables, which although unique between clauses, become bound to other variables or terms through unification. It is through these bindings that the sources of failure are established. In addition, each literal in the BS provides a link to all the other complementary literals in the base with which it is unifiable.

The *Graph of Basic Constraints* records the most general unifier for every pair of unifiable complementary literals. This structure may actually consist of a number of separate graphs where each graph records the substitutions for variables which are bound together. Every distinct set of bindings is represented by a different graph. This structure also maintains a link to the two basic structures corresponding to the two literals from which it was derived.

The *Graph of Dynamic Constraints* contains the most general unifier for the deduction process. In other words, it keeps track of which substitutions have been made and what bindings have occurred. As with the GBC this structure may be composed of several distinct graphs, where each of the graphs represents a partitioned set of substitutions. If two variables in two substitutions become bound to each other then their corresponding graphs are merged together. After each deduction step, the GDC is updated with the substitutions recorded in the GBC for the two resolvent literals. Also contained in this structure are the appropriate links to the BS and the Plan.

The *Plan* represents the deduction process. It can be viewed as a graph with the *top node* of the graph representing the formula to be refuted. Nodes in the graph correspond to clauses in the base. When a clause is selected in a

deduction step it gets inserted as a node in the plan. Each literal in a node becomes a key or a goal. A *key* is the literal which was used as the resolvent, i.e, the complementary literal chosen to resolve a goal. A *goal* is any other literal in the node and needs to be resolved. It contains a pointer to a key. Thus the top node consists of all goals while every other node consists of one key and zero or more goals. Associated with every goal is a list of *potentials*. This corresponds to all the goal's unifiable literals which have not yet been tried as a resolvent for it. This list is derived from the list of unifiable literals associated with the basic structure.

In general, the graph of a plan will represent a tree. If an ancestor node of a goal contains a literal which is unifiable with this goal then these two may be resolved instead of introducing a new node. Thus, the graph would contain an upwards link and can no longer be considered as a tree. The Plan also maintains links to the BS and GDC.

There are a few other concepts which bear mentioning at this time. A *Storage Unit* refers to both a Plan and its corresponding GDC. It is an instance of the deduction process which can be preserved and later retrieved. Because many backtracking points many be determined within one plan, copies of the plan and GDC need to be made so each of the

backtracking points can be processed individually without having to reconstruct the storage unit. Backtracking points correspond to goals in the plan which introduce complementary literals which are directly responsible for the plan's nonunifiability. These goals are grouped into sets called clashes, such that removal of all the bindings introduced by resolving these goals will restore unifiablitity to the plan. Each goal in a clash is referred to as a conflict. In this paper, we sometimes may refer to a node in conflict but we actually mean that the goal pointing to this node is the conflict.

At the end of this section there is an example which shows the organization of a Plan and its corresponding Graph of Dynamic Constraints.

THE DEDUCTION PROCESS

In this subsection a description of the algorithms for the plan-based deduction system of Pietrzykowski and Matwin is given. The algorithms can be grouped into three partitions: the preprocessing phase, the dynamic processing phase and the backtracking phase. These three phases are discussed individually in the next three subsections. In the last section we present the basic algorithm which describes the deduction process at the top most level and serves as a base for further refinements.

THE PREPROCESSING PHASE

The preprocessing phase is, as the name implies, a preliminary routine which performs all the static processing. It reads in the base of clauses and builds the corresponding Basic Structure for each literal. It then searches all the clauses for pairs of complementary literals. For every pair it finds, it traverses the two Basic Structures for them to determine if they are unifiable; and if they are, it records the most general unifier for them in the Graph of Basic Constraints. Proper links to the GBC with the two Basic Structures are maintained. It should be mentioned that the preprocessing phase is not necessary in order for the algorithm to work. However, it greatly improves efficiency by avoiding repetitive and identical unifications during the dynamic processing phase.

THE DYNAMIC PROCESSING PHASE

This phase is responsible for performing the deductions and managing the search strategy. To begin the deduction process the Basic Structure pertaining to the goal statement (usually the first clause in the base) is converted into a node and made the root of the Plan. Every literal in this clause becomes a goal to be refuted. Associated with each goal is its list of potentials. Initially, this list will contain all the complementary literals it is unifiable with. The insertion of a node into the plan causes each goal in the node to be denoted as *open*.

The deduction process consists of selecting each open goal in turn and choosing one of its potentials to be a resolvent for it. (We have adopted a depth first search strategy for choosing open goals while potentials are ordered according to their position in the base.) The clause in which this potential is located is transformed into a node and inserted into the graph. The potential becomes the key in the new node and all other literals are converted into goals and marked as open. The potential is then deleted from the resolved goal and the goal itself is now considered *closed*. The GBC for the goal and the potential is then merged with the GDC for the plan. It is possi-

ble that the potential is contained in a clause that already exists as a node in the plan. This implies that a solution for this goal exists and a pointer to this potential is created. This situation is the only exception where a goal will point to another goal rather than to a key. The process then resumes selecting the next open goal.

The dynamic processing phase continues until the plan is completely closed or until an open goal is found which has no potentials. In the former case if the plan is unifiable (i.e. the GDC has no illegal substitutions associated with it) then a refutation has been found. If a most general unifier for the plan does not exist then the backtracking phase is initiated. In the latter case if the goal belongs to the top node then the problem has no solution otherwise the backtracking procedure is invoked on that goal.

If a refutation has been found then the process terminates and displays the solution. If multiple solutions are requested then a special procedure is used to induce artificial conflicts so that the backtracking phase can be invoked. This procedure is described in the section titled Artificial Conflicts.

THE BACKTRACKING PHASE

This phase is responsible for controlling the backtracking procedures which allow alternative potentials to be chosen. As mentioned above there are two cases which invoke this routine. The first case is when the plan is closed and a most general unifier for the plan does not exist. Here the backtracking phase must determine the sources of the failure as well as backtrack from them. The other case is when an open goal has no potentials; then it is labelled as a conflict and backtracking is immediately invoked. Hence, we can partition this phase into two routines, one which determines the source of failure and the other which actually does the backtracking. Both of these are described below.

To every plan there corresponds a Graph of Dynamic Constraints. Since a most general unifier for a nonunifiable plan does not exist, there must have been at least one illegal substitution made. This fact will be recorded within the GDC. An illegal substitution is reflected by a variable being bound to two or more functions (a constant is considered a function with zero arguments). These two functions will be denoted as being in conflict. The GDC was designed so that this condition is easily recognized and links, to both the node and the Basic Structure, which these functions

reside in, are maintained. Both these structures are then traversed together to determine where the sources of failure occur.

A source of failure is considered to be such a set of nodes in the plan that when these nodes are removed from the plan, the resulting plan is unifiable. These nodes do not neccessarily have to contain one of the literals which failed to be unified. They may also be nodes which unifies two variables together such that if the binding is broken between these variables then unification will be possible. The goals which introduce these nodes are collectively called a *clash* and individually a *conflict*. That is, a minimal set of goals which, if removed from the plan will restore its unifiability, is called a clash. A clash will consist of one or more conflicts. However, there must be two or more clashes for a nonunifiable plan as this condition arises from two different terms being bound to the set of variables. To determine the clashes for a nonunifiable plan, the plan must be traversed systematically to locate every goal between each pair of terms in conflict. The sequences of goals one must traverse in the plan to get from one of the nodes containing a function in conflict to anoth-

er is called a *path* <2>. To determine all the paths between all the functions in conflict is not a trivial problem, whose solution is described in the above references.

Once we have found these paths, we can eliminate all the nonleaf nodes from them, as only leaf nodes are required as starting points for the backtracking algorithm. Since each of these paths represents a way to reach pairs of terms in conflict and there may be more then one path between the same pair, to make them unreachable we have to remove a goal from every path. Again, this is a nontrivial problem and is also outside the scope of this paper. Once the two nodes are unreachable, removing the corresponding links from the GDC reinstates unifiability to the plan. Again, note that each of the collections of goals which make two nodes unreachable is called a clash of which the individual goals are conflicts.

Once the clashes for a plan have been determined, the deduction process can start backtracking from every goal in the clash looking for an alternative potential. Since more than one clash may exist and the one we choose may not lead to a refutation, we need a method of restoring the plan and

---

<2> An example of a path is given in a later section (see the clauses of example 2b and figure 4).

GDC to its original state so we can backtrack from an alternative clash. The natural way to do this is, after a clash has been selected copy both the plan and GDC with the remaining clashes (called a storage unit) to some storage area where it may be later retrieved for reinstatement (see [Forsythe & Matwin 83]).

The backtracking phase commences by checking every conflict in a clash for a nonempty list of potentials. For every conflict which has an empty list, backtracking is invoked on it. This consists of repeatedly checking ancestor goals from the conflict to the goal statement for a goal with potentials. If the top node is reached unsuccessfully then backtracking for that conflict fails. The backtracking phase for a clash succeeds only if the backtracking of every goal in the clash succeeds. The failure of a clash causes a storage unit to be reinstated so that another clash can be selected. If the backtracking phase succeeds in finding a goal with potentials for every conflict in the clash then the plan is pruned to remove these clashes. This is accomplished by designating every goal which was found to contain a potential, as open. All the nodes for which these goals are ancestors for, have to be removed from the plan and any bindings caused by these nodes have to be eliminated from the GDC. Once the plan and GDC are updated appropriately, the dynamic processing phase is reinvoked.

These two phases of backtracking and goal processing are mutually recursive: as clashes are processed they may lead to further plans which contain clashes which may lead to still more plans which contain clashes. Since other clashes from previous plans are still in storage when new ones are added a selection process retrieving them in some order is necessary. We have chosen to "stack" them in storage using a last-in-first-out selection strategy. The deduction process terminates when a refutation is found or the stack of storage units is exhausted.

THE BASIC ALGORITHM

The above phases represent the fundamental concepts of a plan-based deduction system. In this paper we are concerned not with the development of these routines but with the refinement of them. That part of the system which we will be refining, is listed below as an algorithm and will serve as a basis for modification.

```
procedure Main
   clashes := empty
   gdc     := empty
   topnod  := first clause from base of clauses
   plan    := topnod
   opens   := topnod.goals
   su      := plan + gdc + clashes + opens
   stop    := false
   result  := false
   store(su)
```

(continued)

```
while (su not empty) and (not result) do
-- while there is no solution but more plans
--   get a plan and remove any clashes to
--   make it unifiable and then try a new
--   alternative clause
      retrieve(su)
      if su.clashes not empty then
           clash := first(su.clashes)
           removeconflicts(su, clash, stop)
      if not stop then
           develop(su)
-- if a deduction is found then display it
-- or find more solutions otherwise store
-- the plan for further processing
      if (su.clashes empty) and
                    (su.opens empty) then
           if allrefutations then
                display refutation
                su.clashes := artificialcon(su)
           else
                result := true
      else
           store(su)
end while
```

(continued)

```
    -- display the final refutation
      if result then
           display refutation
      else
           no refutation
end main

procedure Develop(su: storage unit)

     while (su.opens not empty) do
          y  := first(su.opens)
          n  := nodeof(y)
          z  := first(y.potentials)
          nn := newnode(n, z)
          add(nn, z, su.plan)
          merge(graph(y, z), su.gdc)
       end while
     su.clashes := conflictcheck(su)
  end develop
```

Algorithm 1. The original algorithm for plan-based deduction

In this algorithm there are a number of notations, conventions and procedures which need to be expounded upon:

1) The "." indicates a subfield of a record. Thus "su.clashes" should be read as " the clashes of su".
2) The "--" indicates a comment
3) su refers to a "storage unit" and consists of a plan, a GDC, clashes and opens, which are defined above.
4) Topnod is initialized to be the preselected, negated clause from the base of clauses and becomes the formula to be refuted. It consists of all goals which initially make up the list of opens.
5) Stop is a global flag which, if false, means that the current plan has open goals with potentials to be developed.
6) Result is a flag to indicate that one refutation has been found.
7) Store is a procedure which places a storage unit on disk for later retrieval.
8) Retrieve is a procedure which fetches a storage unit from disk and makes it the current one.
9) First is a procedure which returns the first element of the specified set. The element returned is deleted from the set.
10) Removeconflicts is a procedure which removes all the conflicts of a clash from the plan of the current storage unit. The parameter "stop" is returned with a value

of true if the backtracking strategy fails in locating a goal with potentials.

11) Allrefutations is a user defined flag which indicates that all refutations are desired as opposed to only one.

12) Artificialcon is a procedure which generates a clash on a nonunifiable plan so that the deduction process can be reactivated to generate another solution. This procedure is defined in more detail later on.

13) Nodeof is a function which returns the node in which a specified goal resides.

14) Newnode is a procedure which creates a new node from the clause in which the specified potential is a literal.

15) Add is a procedure which inserts the given node, nn, in the specified plan, as a child of z.

16) graph is a function which returns the Graph of Basic Constraints (GBC), i.e. the unifier, of the two arguments (the literal corresponding to the goal and the potential).

17) merge is a routine which updates the current Graph of Dynamic Constraints (su.gdc) with the substitutions recorded in the given GBC.

18) Conflictcheck is a function which determines if a given plan is unifiable. If it is, it returns empty otherwise it returns the appropriate set of clashes.

This algorithm outlines the deductive strategy as described in the preceding subsections. Procedure Main controls the deductive process at the top most level. It selects which plan will be developed and ensures that every possible plan is tried until a refutation is found. Storage units containing nonunifiable plans are stacked in a last-in-first-out strategy to promote a depth first search strategy for developing these plans. If all refutations are desired then artificial conflicts are determined on closed unifiable plans. Procedure Develop is responsible for constructing each individual plan. After a plan is closed, it is checked to see if it is unifiable.

AN EXAMPLE OF PLAN-BASED DEDUCTION

To clarify the above concepts, the deduction process is examined on the following clauses in example 2.

```
 P(x) Q(x,y) R(z) S(y,z,x)
-P(x) N(x,y)
-Q(x,x)
-R(d)
-S(x,d,y) M(x,c) T(y)
-N(a,c)
-M(b,c)
-T(a)
```

Example 2a.

In this example the goal statement is the first clause in the base. Since the same names occur more than once we will rewrite these clauses appending a number to each function and variable name as in example 2b. This will avoid confusion later as to which instance of the name is referenced. As a reminder to the reader, all variables of the

same name are bound to each other within the same clause but are distinct between clauses.

```
 P(x1) Q(x2,y1) R(z1) S(y2,z2,x3)
-P(x4) N(x5,y3)
-Q(x6,x7)
-R(d1).
-S(x8,d2,y4) M(x9,c1) T(y5)
-N(a1,c2)
-M(b1,c3)
-T(a2)
```

Example 2b.

Figure 3 shows a representation of the seven GBC's found during the preprocessing phase. It indicates which variables are bound together by horizontal lines and the substitutions determined via the unification algorithm are depicted by vertical lines. The names of the two literals on which the unification was performed label the graphs.

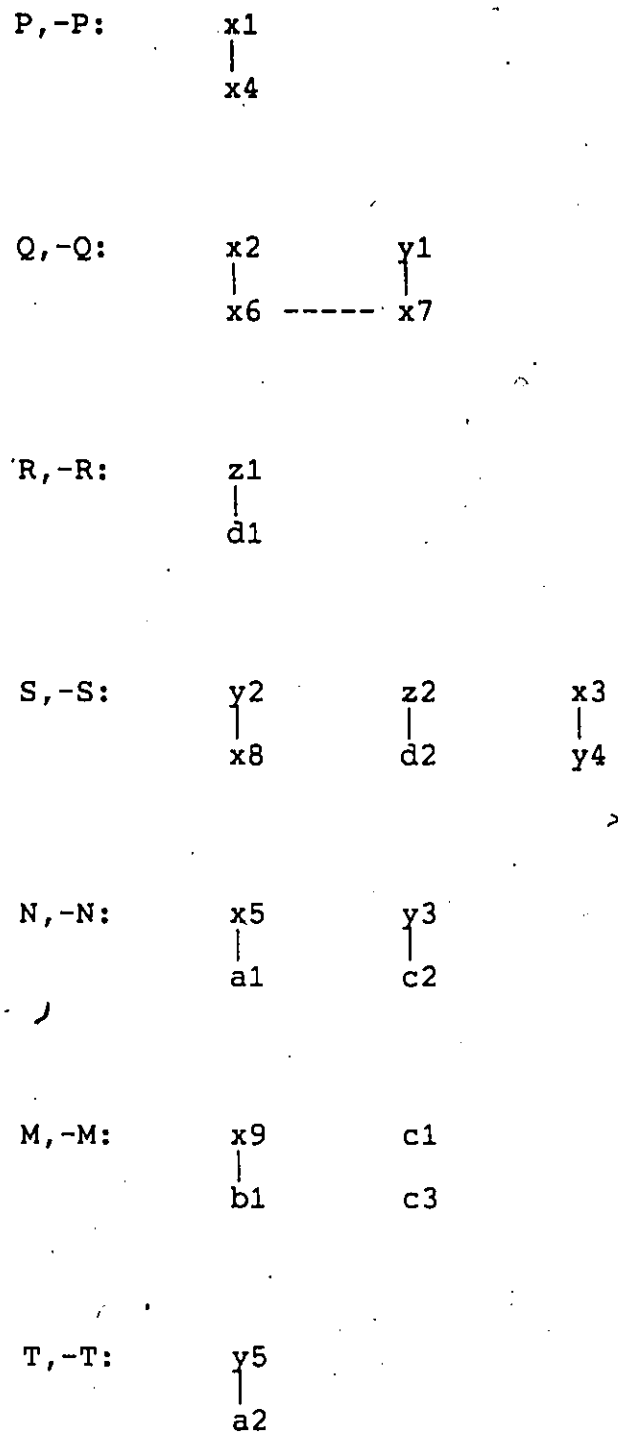

Figure 3. The GBC's for example 2b.

In figure 4, a Plan for this set of clauses is shown. The arcs indicate that the two literals at each end are a pair of resolvent literals. Since one of these literals is called a goal and the other a key we shall denote the two together as a goal-key pair. The number labelling the arcs on goal-key pairs will be used later to define a path through the plan. Figure 5 presents the corresponding GDC for this plan.

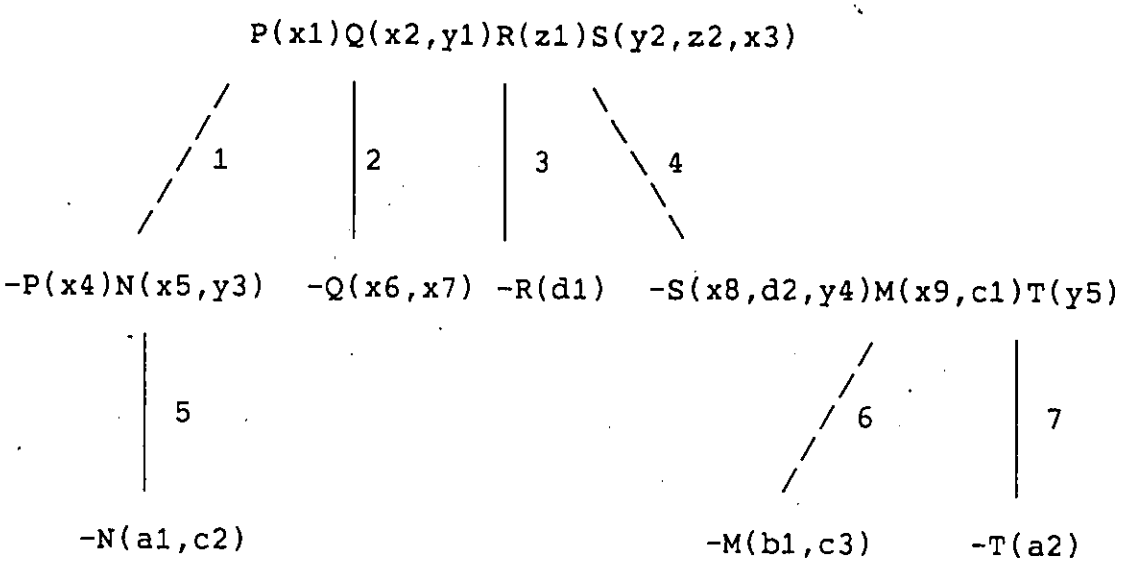

Figure 4. A Plan for example 2b.

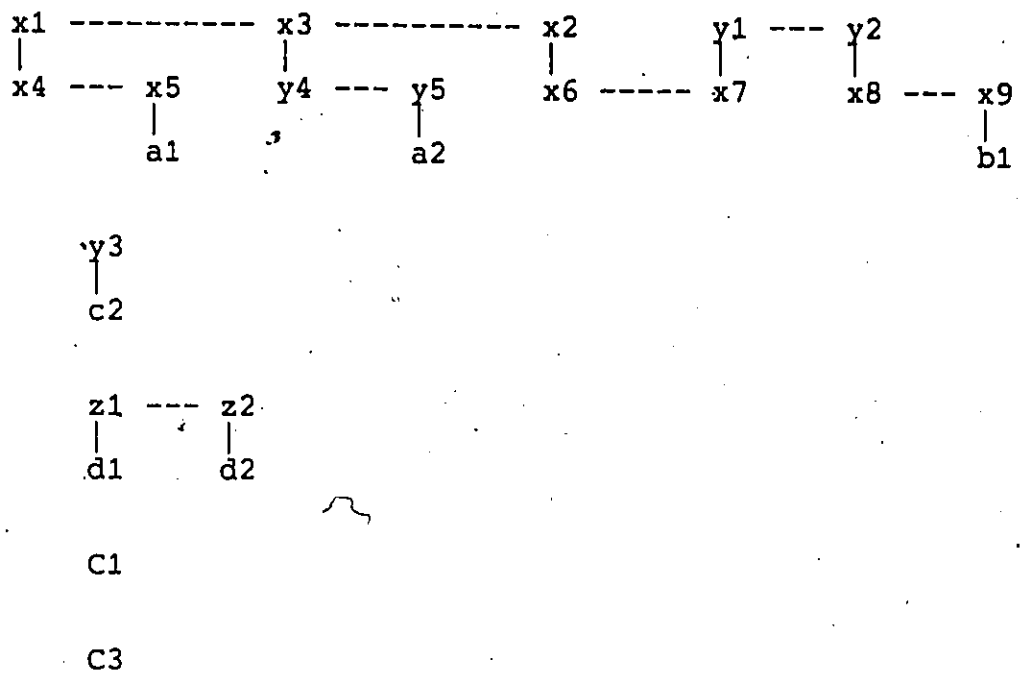

Figure 5. A GDC for example 2b.

From figure 5, one can easily verify that this set of clauses is not unifiable as the constants a1, a2 and b1 all belong in the same graph. Once the plan is closed the backtracking algorithm must determine the paths in the plan between these nonunifiable constants. In this example, this means the algorithm must find the paths between a1 and b1 and between a2 and b1. These paths will be represented by the labels attached to the goal-key pairs. For the clauses of example 2b they are: {5, 1, 2, 4, 6}

and {7, 4, 2, 4, 6}. These paths are derived through the following links: a1 - x5 - x4 - x1 - x2 - x6 - x7 - y1 - y2 - x8 - x9 - b1 and a2 - y5 - y4 - x3 - x2 - x6 - x7 - y1 - y2 - x8 - x9 - b1. Next the nonleaf nodes are deleted to obtain the minimal paths which are {5, 2, 6} and {7, 2, 6}.

The next step is to convert these paths into clashes. We now have a set of paths where we have to remove specific elements from each path to restore unifiability to the plan. We wish to transform these paths into clashes, where the removal of all the elements from only one clash restores unifiability to the plan. This process results in the following set of three clashes: {5, 7}, {2}, and {6}, which corresponds to: {N(x5, y3), T(y5)}, {Q(x2, y1)} and {M(x9, c1)}. One of these three clashes would then be chosen for backtracking and a copy of the plan, DCG and the remaining two clashes would then be placed in storage for later processing.

RELATED RESEARCH

This sections presents a brief discussion of current research in graph based deduction algorithms. The papers discussed here present interesting approaches to theorem proving systems using graphs but differ from the method of Pietrzykowski and Matwin in terms of what a plan is and how it is operated on to obtain a refutation. The main objective is to show that most of the current research tends to all but ignore the problems associated with backtracking and redundant processing. The papers relevant for discussion are [Bibel 83], [Bruynooghe & Pereira 83], [Chang & Slagle 79], [Kowalski 75] and [Sickle 76].

In [Bibel 83] the deduction process is performed on first order logic axioms arranged in the form of a

matrix. This matrix is a nonclausal representation of the complete search space. The deduction process consists of finding a path through this matrix by establishing connections between complementary literals. If the matrix can be spanned (i.e. each path through the matrix contains a connection) then a proof is obtained. Unification is performed at each step in the spanning process and when a literal chosen for connection cannot be unified then selective backtracking is invoked. What selective backtracking is or how it affects the performance of the algorithm is not discussed. Bibel's approach is unique compared to the others as his method is the only one which is not based on resolution.

In [Kowalski 75] a complete representation of the search space is also used as the basis for deduction. This search space is called a connection graph and is based on a clausal representation of the axioms. The resolution process differs from [Bibel 83] in that links are maintained between every pair of unifiable complementary literals as opposed to selected pairs. When a literal is resolved, the link is removed and new links to the remaining literals in the resolvent's clause are added to the graph. The substitutions required for the new links are directly computed from the substitutions associated with the old link. If a node in the graph

contains no links outside the clause in which the node resides or if the clause is a tautology, then all the links associated with that clause are removed from the graph. A refutation is realized if the graph contains the empty clause.

In Kowalski's paper, the problem of redundancy is addressed using an example based on propositional calculus axioms. How his solution can be extended to predicate calculus is not discussed. However, the paper does suggest that the problem of redundancy in general can be solved through the concept of ordering which is also the fundamental philosophy upon which our solution is based.

[Chang & Slagle 79] also use the notion of connection graphs as the basis for finding a refutation on a set of clauses. After building this graph and modifying it to be a directed graph, they use a set of rewriting rules which determine sequences of substitutions which may lead to a refutation. Consistency of these substitutions is only verified after the complete sequence of substitutions has been generated. If the verification fails then another sequence is tried, which is equivalent to backtracking. The effects of backtracking on this system are not discussed, neither is the problem of redundant solutions as understood here.

The approach to refutation on a set of clauses presented in [Sickle 76] is via clause interconnectivity graphs. The underlying concept of these graphs is to construct a representation of the original search space and to modify it incrementally as clauses are selected for resolution. The paper stresses intelligent selection of these clauses to avoid the generation of redundant and/or useless search spaces. However, if the selection algorithm leads to an inconsistent set of substitutions there is no mention of how backtracking is invoked or what affect it has on the efficiency of that method.

The work of [Bruynooghe & Pereira 83] more closely resembles the research of Pietrzykowski and Matwin than any of the other papers. The aim of their research is also to eliminate the repetitive solving of subproblems which are not related to the source of failure caused by blind backtracking techniques. They define their approach as different, but complementary to the approach used by Pietrzykowski & Matwin. According to them, their method finds the minimal subtrees of a plan such that unification is impossible, whereas the approach of Peitrzykowski and Matwin finds the maximal subtree such that unification is possible. Because their approach is similar in some respects to Pietrzykowski and Matwin's, the technique used by them to eliminate redundancy also has simi-

larities to the method presented in this paper; they both are founded on the concept of restricting the search space of potential solutions. However, Bruynooghe and Pereira control this restriction according to the order in which the input clauses are selected. This technique prevents the use of heuristics in situations where redundancy has to be sacrificed in order to preserve completeness of the search space. It is also unclear how the algorithm handles situations where two or more input clauses need to be processed simultaneously. They also claim that their approach to refutation permits parallel processing but they do not elucidate as to how it can be implemented with their selection criteria.

In the next three sections the algorithms that have been developed by the author for minimizing the redundancy inherent to a plan based deduction system are presented. The first section deals with the processing strategy of a single plan. The next section is concerned with removing redundancy associated with the backtracking strategy and the third discusses how artificial conflicts can be introduced into a closed unifiable plan so multiple refutations may be found.

7

SINGLE PLAN PROCESSING

A refutation for a set of clauses in plan-based deduction exists when the plan is closed and a most general unifier for the plan exists. In the original design of the plan-based deduction system developed by Pietrzykowski and Matwin the deduction process constructed a closed plan (if possible) and then verified that all the substitutions performed were valid. The intent was that all the conflicts in the plan could be detected during one phase. These conflicts would then be arranged into appropriate clashes which the backtracking routine would process. During the testing of the implementation of this system it was made evident that this strategy of verifying the plan after it was closed led to redundant processing. The reason for this redundancy can

be made clear through the use of an example such as the base of clauses in example 3.

```
 P(x) Q(x,e)
-P(a)
-Q(c,z) M(z)
-Q(b,z) M(z)
-Q(a,z) M(z)
-M(z) R(z,x) S(y,z) T(x,y)
-R(c,e)
-S(d,e)
-T(c,d)
```

Example 3.

In order to see the limitations of the original design an inspection of the results using the original system should be made. Figure 6a shows the closed plan as developed by the original strategy. In this figure, alternative (potential) clauses which could replace the selected one are shown within pointed brackets beneath the chosen clause.

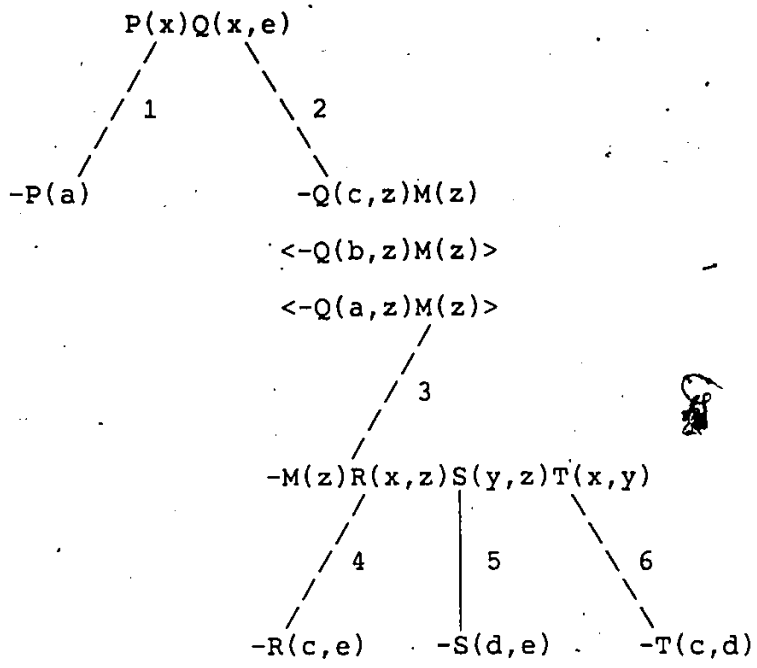

Figure 6a. A Plan for example 3.

It is obvious from figure 6a that the sources of failure for this plan are the literals -P(a) and -Q(c,z). The deduction system operating on this plan would then determine that the path between these literals lay through the goals P(x) and Q(x,e) or arcs 1 and 2 in the diagram. Dividing these goals into clashes results in the two clashes {1} and {2}. Choosing the clash {1} for backtracking, results in a failure as there are no potentials for this conflict. As this conflict resides in the

topnode, there are no other goals which can be checked for potentials. Backtracking from {2} succeeds with the alternative clause -Q(b,z)M(z). The arc to the node in conflict is removed and replaced by the alternative clause, resulting in the new plan shown in figure 6b. What needs to be made clear at this point is that the transition from the plan in figure 6a to the plan in figure 6b results in the arcs 3, 4 and 5 being resolved twice. When arc 2 is cut to remove the node -Q(c,z)M(z) the four descendant nodes of this node are removed from the plan as well. Because the replacement clause also has the literal (goal) M(z) to resolve then these four other nodes necessarily get resolved again. This same process is repeated on the plan of figure 6b to obtain the plan of figure 6c which shows a refutation for this set of clauses.

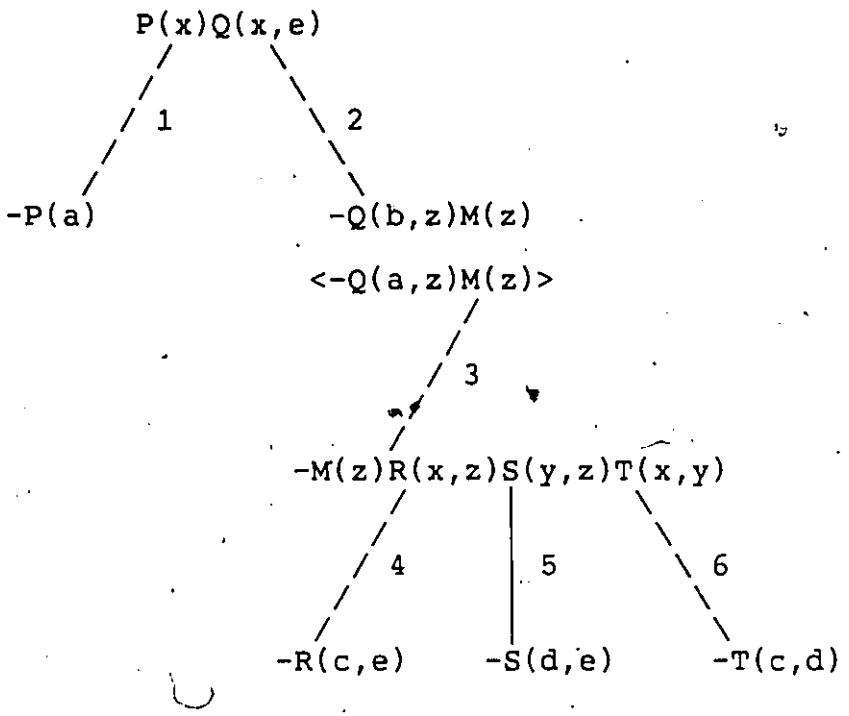

Figure 6b. A second-level plan for example 3.

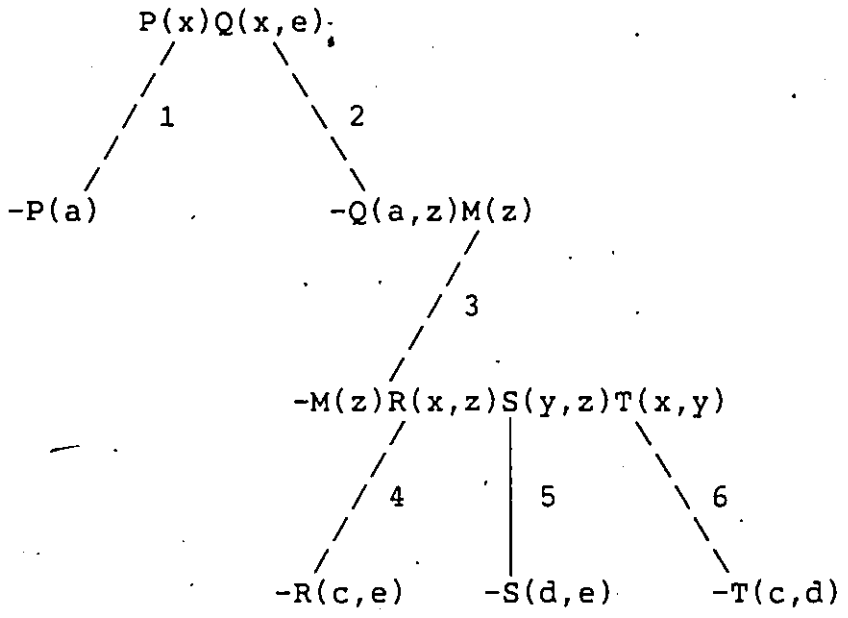

Figure 6c. A third-level plan for example 3.

In order to measure the amount of work done by this strategy we introduce the notions of arc insertion and arc deletion which refer to every instance of inserting or deleting an arc. Thus after the plan of figure 6a has been constructed a total of 5 insertions and 0 deletions have been made. Construction of the plans in 6b and 6c each make 2 deletions and 4 more insertions. Thus after refutation the total number of insertions is 13 and the total number of deletions is 4.

To reduce the amount of work, the resolving of the goal M(z) and thus R(x,z), S(y,z) and T(x,y) as well, should be delayed until the final plan is constructed. To do this we need to verify the substitutions made at every resolution step. If an illegal substitution was made then the dynamic processing phase should be interrupted immediately and the backtracking phase invoked. This new strategy would prevent the goal M(z) from being processed until the correct substitution is performed. The modifications to the algorithm are restricted to the procedure Develop which is shown below. New or changed lines are highlighted using capital letters.

```
procedure Develop(su: storage unit)

    while (su.opens not empty) AND
          (SU.CLASHES EMPTY) DO
        y  := first(su.opens)
        n  := nodeof(y)
        z  := first(y.potentials)
        nn := newnode(n, z)
        add(nn, z, su.plan)
        merge(graph(y, z), su.gdc)
        SU.CLASHES := CONFLICTCHECK(SU)
    end while
end develop
```

Algorithm 2. Version 2 of the original algorithm

The effect of moving the conflictcheck routine inside the loop gives a dramatic increase to the performance of the system. The results of applying this new strategy to the clauses of example 3 are as follows.

This new strategy results in two new plans, as shown in figures 6d and 6e, with the final plan being the same

as that in figure 6c. The work done by this approach leads to only 2 insertions for the first plan, 1 insertion and 1 deletion for the second plan and 1 deletion and 4 insertions for the second plan, for a total of 7 insertions and 2 deletions.

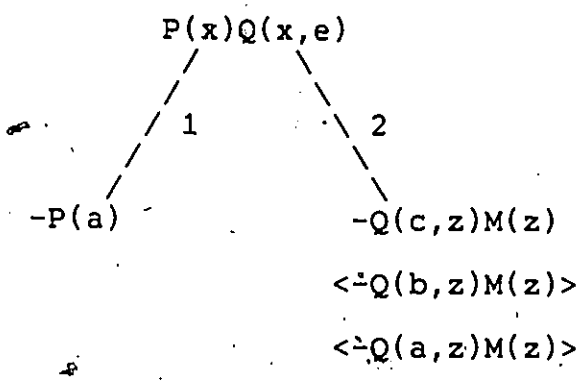

Figure 6d. An improved plan for example 3.

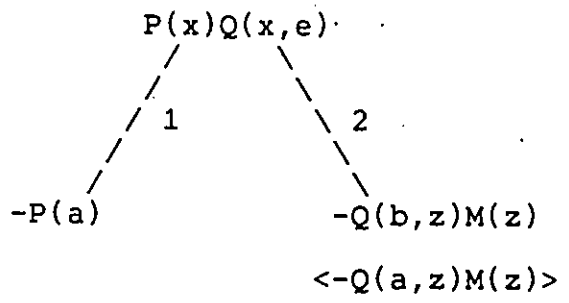

Figure 6e. An improved second-level plan for example 3.

It is interesting to note that if the clause -Q(a,z)M(z) was replaced with the clause -Q(d,z)M(z) then no refutation would exist for this new set of clauses. Using the original strategy the same number of insertions and deletions would be made but with the new strategy, since the literal M(z) would never be resolved, only 4 insertions and 2 deletions would be made.

REDUNDANCY REMOVAL

This section describes how redundant plans are generated by the original plan based deduction system designed by Pietrzykowski and Matwin. It then presents an algorithm which minimizes this redundancy yet maintains a complete search space. A complete search space is one that guarantees that all potential solutions will be attempted before a "no refutation" conclusion is reached. It is a well known trade-off that many heuristic algorithms have a tendency to trade completeness for efficient execution, but the approach taken here is to sacrifice redundancy for completeness in situations where a choice has to be made.

To show that redundant solutions are inherent to plan-based deduction systems the following example is

discussed. Since the emphasis here is with the presentation of an improved algorithm the examples will be kept simple. It should be noted that the amount of redundancy will increase with the complexity of the problem and that the algorithm presented is general enough to handle all cases. Consider the following set of clauses in example 4.

P(x) Q(x)

-P(a)

-P(b)

-P(e)

-Q(c)

-Q(d)

-Q(e)

Example 4.

The deduction process which leads to a refutation for this base of clauses can be best shown via a graph as displayed in figure 7a. Each node in this graph represents a plan or a failure. A failure occurs when a clash cannot resolve at least one of its conflicts because none

of the goals between it and the topnode have any potentials. Leaf nodes depict nonunifiable plans whose children correspond to its resolved clashes. Leaf nodes are either refutation nodes, which indicate a closed unifiable plan, or failure nodes. Arcs in the graph extend from a clash to a new plan which is derived from resolving that clash.

A condensed notation is used to display each plan as we are concerned with how a plan is developed rather than how a refutation is derived. Each plan then, is represented by the goal statement with the terms of each resolved complementary literal shown underneath each goal. In braces alongside each of these terms are the terms of the potential literals which have not yet been resolved. It should be emphasized here that this section refers to the processing of a plan between solutions. Once a solution is found than the previous clashes no longer correspond to the artificial conflicts and new priorities need to be determined.

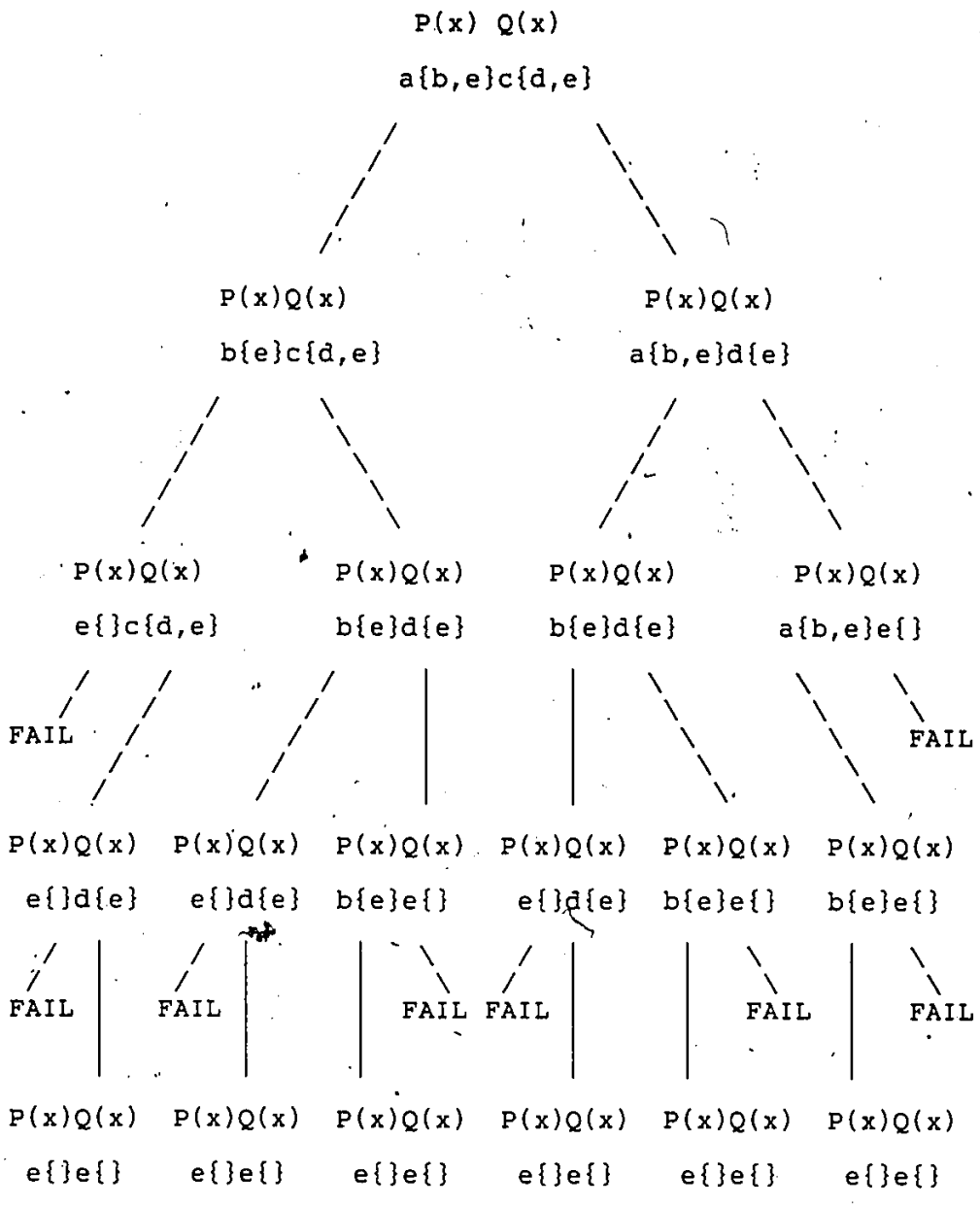

Figure 7a. A deduction trace for the clauses of example 4.

Consider the topmost node of the graph in figure 7a. The goal statement consists of the clause: P(x) Q(x). From the base of clauses of example 4, we see that the first literal which could be unified with P(x) is -P(a). The "a" from this term is shown below P(x) in the figure. Likewise, the "c" from -Q(c) is shown beneath the goal Q(x). From the base we also note that two other literals could have been unified with P(x) and two more with Q(X). The terms b, e, d and e from these two sets of potentials are shown within braces to the right of the term of the selected literal. This type of notation is sufficient because we are interested only in how the terms within the braces are selected to replace those terms contained within the clash.

Looking again at the first node in figure 7a, we see that this node represents a nonunifiable plan. There are two clashes for this plan, one consisting of the goal P(x) and the other consisting of the goal Q(x). From these two clashes we can derive two more plans each of which replaces the literal removed during the backtracking phase with one of the conflicts' potentials. These two new plans are shown as the descendant nodes of the root node. The left descendant replaces the literal -P(a) with -P(b) while the right descendant replaces the literal -Q(c) with -Q(d). These two new plans each have

the same two clashes: {P(x)} and {Q(x)}, corresponding to the nonunfiable nodes: -P(b) & -Q(c) and -P(a) & -Q(d). Resolving each of these clashes leads to four more plans which then yield 6 more plans and two failure nodes. These 6 plans then lead to 6 more failure nodes and 6 more plans. These last 6 plans are all closed and unifiable and therefore, refutations. Although, each of these refutations were derived through separate paths, they all represent the same solution!

Consider again the development of these refutations as shown in figure 7a. Graphically, we can see that the right subtree of every left descendant is identical to the left subtree of every corresponding right descendant. This is the redundancy we wish to eliminate, yet the algorithm must also preserve the completeness of the search space, i.e., if a refutation exists, the algorithm must be capable of deriving it. The algorithm of Pietrzykowski and Matwin is proven to be complete in [Pietrzykowski & Matwin 84]. The completeness of the algorithm presented here relies on this fact. If we can show that our new algorithm only restricts the deductive process from generating the redundant subtrees' then we can assume that it, too, is complete.

To see why redundant plans are generated consider again the deductive trace of the clauses of example 4 as shown in figure 7a. A property of this example is that the clashes produced necessarily involve the same goals in each nonunifiable plan. In other words, the same two goals, P(x) and Q(x), are selected as the only conflicts of two clashes in each subsequent level. Looking at the second level of the deductive trace, we see that both plans have two clashes which cause four more plans to be generated though only three are unique. This is because the same two goals, P and Q, are involved in both sets of clashes but only one of the goals is being replaced in deriving the new plan: P for the left plan and Q for the right one. Thus, when clashes are determined for the second level plans, all but one of the potentials from the first level are still available in each of the second level plans. Because the clashes for both second level plans involve the same conflicts, when developing the third level plans one clash from each of the two plans develops the same potential which the other second level plan developed. Thus, we get the combination of -P(b) with -Q(d) occurring in two plans at the third level. In the next level, this redundancy is even more pronounced as the combinations of -P(e) with -Q(d) and -P(b) with -Q(e) each occur three times.

In order to eliminate this redundancy we must provide communication between subsequent levels of plans. This is accomplished by assigning priorities to the individual clashes to prevent the additional generation of identical plans. That is, we determine which clashes will generate identical plans and eliminate all but one of those clashes. Once the priorities for the clashes have been set then the next generation of plans can eliminate those clashes with a higher priority than the priority of the clash used to generate that particular plan. The effect of this strategy is to control which plans can develop the common set of potentials available to all the plans at a particular level. Implementing this control requires the following modifications to the original algorithm (highlited by the use of capital letters).

```
procedure Main
    clashes    := empty
    gdc        := empty
    topnod     := first clause from base of clauses
    plan       := topnod
    opens      := topnod.goals
    su         := plan + gdc + clashes + opens
    stop       := false
    result     := false
    SU.ALLCLASHES    := EMPTY
    CURRENTPRIORITY := 0
    store(su)
    while (su not empty) and (not result) do
         retrieve(su)
         if su.clashes not empty then
              clash := first(su.clashes)
              -- RETAIN THE PRIORITY OF THE CLASH
              CURRENTPRIORITY := CLASH.PRIORITY
              removeconflicts(su, clash, stop)
         if not stop then
              develop(su)
```

(continued)

```
        if (su.clashes empty) and
                    (su.opens empty) then
            if allrefutations then
               "display refutation"
                su.clashes := artificialcon(su)
            else
                result := true
        else
            store(su)
    end while
    if result then
        display refutation
    else
        no refutation
end main
```

(continued)

```
procedure Develop(su: storage unit)

    SU.ALLOLDCLASHES := SU.ALLCLASHES
    while (su.opens not empty) AND
          (su.clashes empty) DO
        y  := first(su.opens)
        n  := nodeof(y)
        z  := first(y.potentials)
        nn := newnode(n, z)
        add(nn, z, su.plan)
        merge(graph(y, z), su.gdc)
        su.clashes := conflictcheck(su)
    end while
    SU.ALLCLASHES := SU.CLASHES
    SETPRIORITY(SU.CLASHES)
end develop
```

(continued)

```
PROCEDURE SETPRIORITY(CLASHES: SET OF CLASH)

   WHILE CLASHES NOT EMPTY DO
     -- CHECK IF THE CLASHES EXISTED IN THE
     -- PREVIOUS PLAN. IF IT DID NOT ASSIGN
     -- IT A PRIORITY. THEN ELIMINATE  ANY
     -- WHICH HAVE A HIGHER PRIORITY THEN THE
     -- CURRENT ONE
      CLASH := INITIAL(CLASHES)
      IF INOLDCLASHES(CLASH,
                      SU.ALLOLDCLASHES,
                      OLDCLASH)
      THEN
         IF OLDCLASH.PRIORITY -> CURRENTPRIORITY
         THEN REMOVE(CLASH)
         ELSE CLASH.PRIORITY := OLDCLASH.PRIORITY
      ELSE
         CLASH.PRIORITY := NEWPRIORITY(CLASHES)
      CLASH := NEXT(CLASHES)
   END WHILE
END SETPRIORITY
```

Algorithm 3. Version 3 of the original algorithm

With these refinements to the original algorithm there are some additional procedures which need to be commented on. ~

1) Su.allclashes is used to remember the original set of clashes as the actual set of clashes may not contain all of them.
2) Su.alloldclashes is used to save su.allclashes for use by the next plan generated..
3) Initial returns the first element of the specified list without deleting it.
4) Inoldclashes is a boolean function which determines if the specified clash was a member of the previous set of clashes by checking for an identical clash. If it is then the parameter oldclash is set appropriately, and the function returns true, otherwise the function returns false.
5) Remove is a routine to delete a clash from the set of clashes.
4) Newpriority is a function which returns a new priority value. How this routine derives it's value is discussed below.
7) Next is a function which fetches the next element from the specified set without deleting it.

This algorithm places an ordering (priority) on each clash which dictates which clashes (of sequel plans) can

lead to new plans. The ordering value placed on a clash of a plan P1 limits the number of new clashes in the plan P2 generated from P1 after the removal of the clash. To acheive this, the set of previous clashes needs to be remembered so that the set of new clashes can be matched with the previous set. This ensures consistent assignment of priority values as each clash in the new plan is assigned the same value that it had in the old set of clashes. Those clashes in the new plan which have a priority greater than the clash from which the plan was derived are discarded. This strategy ensures that only one of the many clashes which can generate the same plan will actually be used to generate that plan. The exceptions to this occur when 1) a goal which was previously open becomes a conflict of the current plan and a new ordering scheme is essential and 2) when the conflicts in the new set of clashes get rearranged into different grouping so that they no longer correspond to the previous set of clashes. When exception (1) occurs the new priority numbers are assigned values from one to the number of clashes. In the second case, where clashes contain the same conflicts but in different patterns, then the value assigned is a heuristic one which is high enough to ensure that no combinations will be missed. Since this situation causes a reevaluation of the priori-

ty number, it is possible that redundant solutions may be generated.

The effect of this algorithm can be seen in figure 7b. In this figure the priority number associated with each clash is shown to the right of the arc which joins the clash to the plan it generates. As indicated by the figure no redundant plans are generated. In the second level the plan on the right is limited to only one clash which prevents the redundant plan from the initial trace from being derived. This effectively removes the redundant left subtree phenomena described above. In the third level since the two right most plans were generated by clashes with a priority of one, only one plan is generated by each. This again removes the redundant left subtree of the first trace. As a result this trace only consists of twelve nodes where the the original one consisted of twenty-seven.

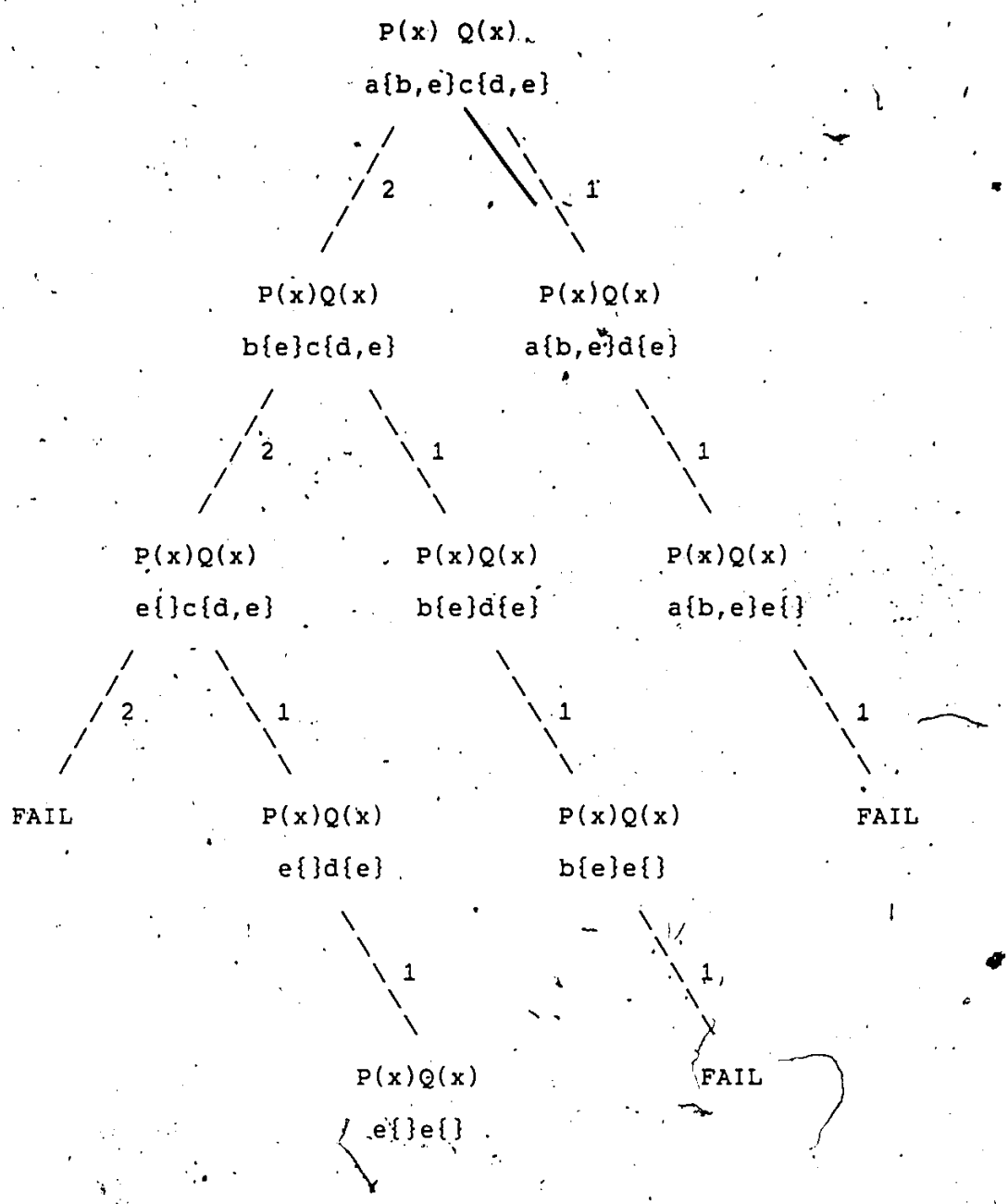

Figure 7b. A second deduction trace for the clauses of example 4.

Now, let us consider the case where the clashes of the resulting plan are not identical to the clashes of the previous plan, as illustrated by the set of clauses in example 5. A trace for the deduction process is shown in figure 8. In this figure, if a clash contains more than one conflict, a line is drawn underneath the node to connect those conflicts.

$-P(x)-Q(x)-R(x)$

$P(a)$

$P(e)$

$Q(a)$

$Q(b)$

$Q(e)$

$R(b)$

$R(e)$

Example 5.

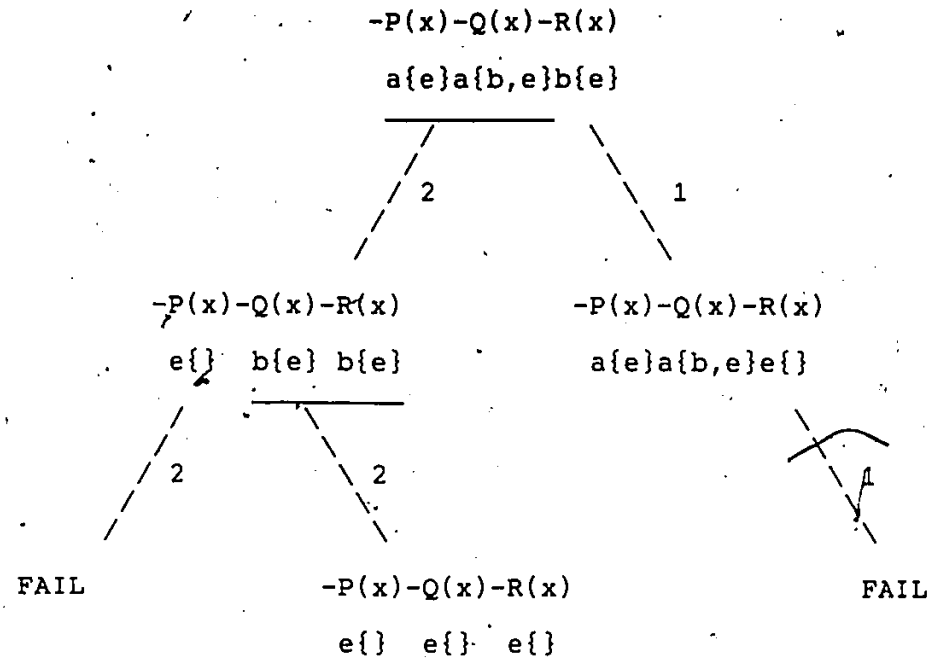

Figure 8. A deduction trace for the clauses of example 5.

In the first plan of the trace we see that the clashes are: {-P(x), -Q(x)} and {-R(x)}. However, in the left plan of the second level the clashes are {-P(x)} and {-Q(x), -R(x)}. Neither of these two clashes occurred in the first plan, therefore a new ordering scheme must be used to set the priority value for these two clashes. In this case, we set both clashes to have the same value as the clash from which it was derived. (An algorithm which

decides this value so that the minimal number of plans will be generated, yet which preserves completeness, is a topic for further research. However, if the value of the deriving clash is used, completeness will be preserved but in some cases redundant plans will be generated.) In this example since the next plan is a solution, the actual priority value assigned is irrelevant. As the figure shows, only 6 nodes were generated using this algorithm. If the orginal strategy was used then three more nodes would have been generated from the clash which we eliminated from the right hand node of level two.

From these two examples it should be evident that the refinements to the original algorithm prevent much of the inherent redundancy yet maintain completeness. To prove that completeness is preserved is a difficult problem. Instead, let us assume that the original algorithm is complete and show that we have not removed any clashes except those which lead to redundant plans.

In the original algorithm, the relationship of clashes with potentials can be considered as elements of several lists which interact with each other, such that one and only one element of each list is changed with each new plan generated. However, redundant plans occur because no information is recorded as to which combinations of

potentials within a plan have already been selected. By assigning a priority to each clash, we are effectively placing an ordering on how the potentials are to interact with each other. Suppose we have three clashes and we want to obtain every possible combination of associated potentials only once, we could assign priorities to each clash such that the potentials in the third clash change after every combination, the potentials in the second clash change after every pass of the third clash and the potentials of the first clash change after every pass of the second clash.

For example, consider an algorithm for the generation of all possible combinations of triples for three lists of {1,2,3} with length three. A complete set of triples is as follows: {(1,1,1), (1,1,2), (1,1,3), (1,2,1), (1,2,2), (1,2,3), (1,3,1), (1,3,2), (1,3,3), (2,1,1), ..., (3,3,3)}. To remember which combinations have been obtained, all we have done is placed a priority on each element's position which dictates which element gets changed to generate the next triple. That is, we have selected the third position to be changed before the second and the second position is changed before the first. In other words, the third position has the lowest priority (1) the second position the next (2) and the

first position the highest priority (3). This priority only controls the order of generating the triples.

The lack of a strategy which restricts the combination of clashes generated for a plan was the problem with the original algorithm. The orginal algorithm was designed to ensure that every possible plan could be generated but without having to keep track of those plans previously generated. This strategy was possible because the ordering of the potentials is well defined and constant, so that each list of potentials was traversed sequentially. However, the number of plans which generated a new plan became greater than the number of lists to choose from and so duplicate plans were necessarily, developed.

The refinements made to the algorithm also rely on the fact that the lists of potentials are well defined and ordered. However, these refinements try to restrict the number of plans generated so that this number seldom exceeds the number of lists of potentials. Since clashes control which potentials are selected, limiting which clashes can develop plans allows us to restrict the number of plans produced. We achieve this control by placing a priority value on each clash which is similar to the ordering placed on each position during the generation of triples in the above example. Each priority

defines an order in which the clashes should be resolved; if this ordering is respected then a complete, nonredundant set of plans will be generated.

The important thing to note in the above example, is that when the second position changed, the list in the third position was reset to its first element. Likewise, when the first was changed the second and third lists were reset. That is, changing an element in a position with a given priority resets all the lists with lower priorities; thus allowing a systematic nonredundant method of controlling the selection of triples. This concept has to be applied to the generation of clashes in order to eliminate the redundancy.

Unfortunately, this concept is a not easily applied to the deduction algorithm because a list of potentials cannot actually be reset during the resolution process. When a list of potentials is exhausted a failure condition arises. However, a single plan may change more than one list by generating a number of plans. The priority strategy we have derived, maps the above strategy to these conditions. That is, instead of resetting a list of potentials, we determine which lists of potentials need to be changed to generate the next level of plans. The priority value placed on a clash corresponds to the

priority we placed on a postion. A priority of one means only one clash can be generated, a priority of two means that two clashes can be generated and so on (in the above strategy a change in position one changed one list, a change in position two changed two lists, etcetera). Since the priority associated with each clash is passed on to successive plans, the algorithm can retain control over which lists of potentials need to be changed for the next level of plans, thus avoiding the redundancy produced by the original strategy.

It should be clear from the above example, where triples for three lists were generated, that this strategy is complete. Since our algorithm maps this strategy onto the deduction process a complete search space is also generated. Alternatively, by comparing figure 7a to figure 7b, one can see that only redundant plans were eliminated, thus the completeness already proved for the base algorithm is preserved.

In the cases where the conflicts of a subsequent clash are not the same, we cannot assume that our imposed ordering values are still valid and some recovery action must be taken. As mentioned earlier, lack of redundancy is to be sacrificed for completeness where a choice has to be made. This situation occurs when the elements in

the list are changed, either by the rearrangement of the conflicts within a set of clashes or by the introduction of a new conflict. In the first case, we can assume that the change only affects the ordering of those clashes involved. That is, if we tamper with the second and third lists, the first remains unaffected. Thus, we need only give a priority to the affected clashes a value which is no higher than any of those concerned. Giving all the affected clashes the highest of these clashes' priorities will ensure that the next generation of plans changes all the potentials which needed to be changed so that a complete search space is produced.

In the second case, if a new conflict is added then we have to start afresh and assign each clash a new priority. We cannot just give the clash containing the new conflict the next highest priority value because the rest of the clashes will not remain the same and we cannot know that all combinations will still be generated. We know that a new conflict affects the way the old clashes were ordered because a new open goal is only developed if the plan is unifiable. Assigning new priorities ensures that all potentials will be changed so that completeness is maintained.

Although this is not a formal proof, it should be clear that this strategy will minimize the redundancy inherent to the original algorithm, yet maintain its completeness.

ARTIFICIAL CONFLICTS

This section describes how artificial conflicts are generated by the original algorithm and how this strategy also leads to redundant processing. It then presents some enhancements to the algorithm which reduces the number of duplications.

The concept of artificial conflicts is derived from the fact that the backtracking strategy is dependent upon specific points from which to backtrack. Deduction algorithms are designed to continually select potential clauses which are to be used in the refutation of a goal statement. Clauses which are determined to prevent unification are discarded and replaced with new clauses by the backtracking phase of the deduction process. When a closed unifiable plan has been derived then a refutation

has been found and the process terminates. In order to determine all the refutations which exist for a given base of clauses, the deduction process has to be reactivated. This is done by selecting certain points, called artificial conflicts, from which to reinvoke the backtracking routine.

With standard linear backtracking procedures the selection of this backtracking point is trivial. Simply designate the last literal unified as an artificial conflict and allow the deduction process to continue. This guarantees that every possible refutation will be found without duplication as the deduction process's search strategy still controls the selection of untried clauses. However, for a plan-based deduction system choosing the backtracking points is not so easy, as backtracking can originate from any node in a plan and must always terminate at the topnode.

To obtain a set of backtracking points with these restrictions, yet ensure that a complete solution set would still be found, Matwin and Pietrzykowski proposed the following strategy. Traverse a closed unifiable plan and designate every leaf node as a single conflict in a clash. Since every branch in a plan must terminate with a leaf node and because the backtracking system looks at

every node between a leaf node and the topnode for potentials, this strategy will find every possible refutation. However, many redundant solutions may be derived by this approach as illustrated by the following example.

P(x)Q(x)R(y)

-P(a)

-P(b)

-Q(a)

-Q(b)

-R(c)

-R(d)

Example 6.

To see how redundant solutions are obtained on the set of clauses of example 6 consider the deduction trace as shown in figure 9a. In this figure, all failure nodes and all nonunifiable plans are omitted so that the trace of refutations is easier to observe. We note that this figure contains eight refutations but four of them are redundant. An analysis of the trace of the deduction process is given in following the figure.

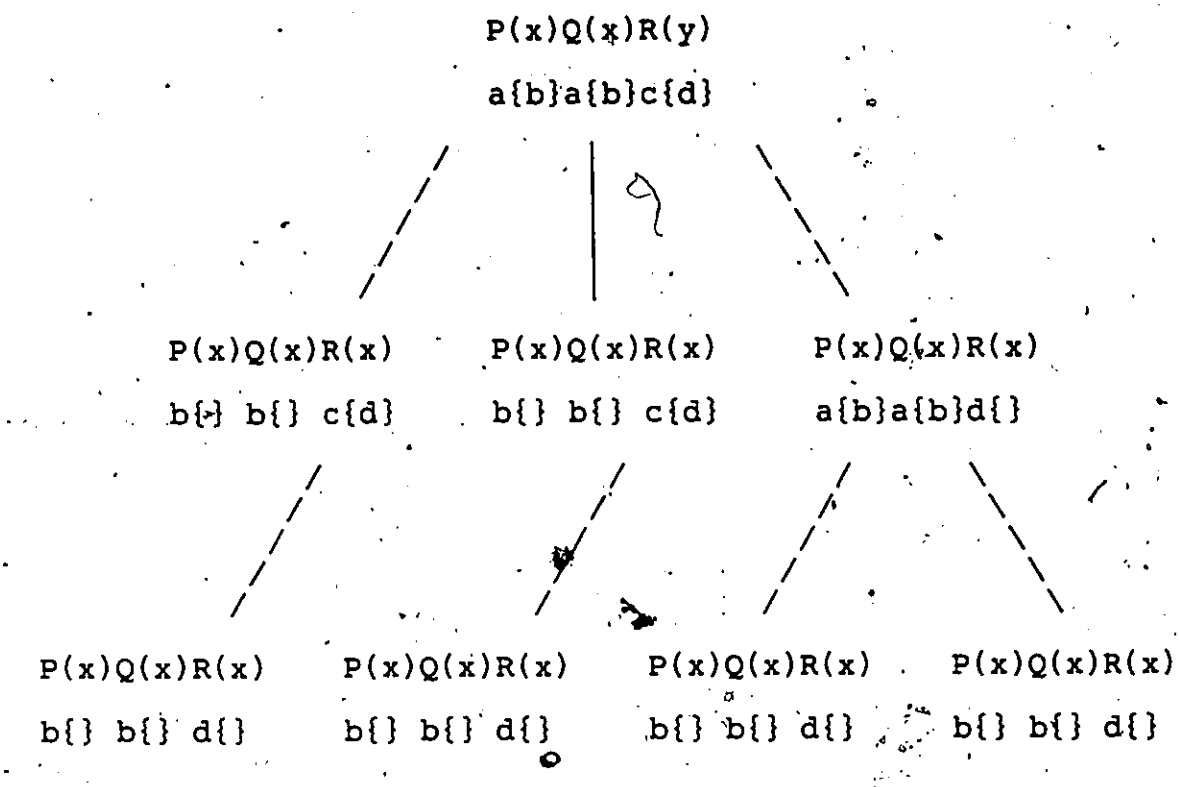

Figure 9a. A deduction trace for the clauses of example 6.

The initial choice of clauses leads immediately to a closed unifiable plan. Each goal in this plan is then chosen as an artificial conflict which yields three clashes, each containing a single conflict. Developing the clash containing P(x) produces the leftmost plan at the second level. Notice, that in order to obtain a unifiable plan the goal Q(x) had to be resolved as well. Similarly, when the clash containing Q(x) is processed,

it leads to the same plan as that derived by P(x). The third clash leads to the rightmost plan which is unique.

Looking at the third level of the this trace, we see that all four plans are identical. Obviously, two of these plans are the same because of the duplication at the second level. The other two plans are duplicates of each other for the same reason we had duplicates at the second level. The reason all four plans are identical is due to the same redundant method of processing clashes described in the previous section. That is, duplication occurs because individual plans process clashes without considering if that particular clash has been previously developed on a similar plan.

We now propose solutions to these sources of redundancy. The first source of redundancy we consider is that which caused the clashes {P(x)} and {Q(x)} to yield the same solution. This duplication stemmed from the fact that P(x) and Q(x) both had to be resolved simultaneously due to the binding of x to x. Consequently, when choosing the artificial conflicts all the goals which introduce constants that are bound together through variables must belong to the same clash.

The second source of redundancy is due to the lack of consideration for clashes which may already been developed by a similar plan. This suggests we must apply the algorithm derived in the previous section to the development of artificial conflicts. Consequently, the selection strategy which determines the clashes to be used as backtracking points for the reactivation of the deduction process is described by the following algorithm.

```
procedure Main
    clashes    := empty
    GDC        := empty
    topnod     := first clause from base of clauses
    plan       := topnod
    opens      := topnod.goals
    su         := plan + dcg + clashes + opens
    stop       := false
    result     := false
    su.allclashes          := empty
    SU.ALLACCLASHES        := EMPTY
    SU.ACPRIORITY          := 0
    SU.ALLOLDACCLASHES     := EMPTY
    ARTIFICIALPROCESSING := FALSE
    store(su)
    while (su not empty) and (not result) do
        retrieve(su)
        if su.clashes not empty then
            clash := first(su.clashes)
            currentpriority := clash.priority
            removeconflicts(su, clash, stop)
```

(continued)

```
            IF ARTIFICIALPROCESSING then
                SU.ACPRIORITY := CURRENTPRIORITY
            ARTIFICIALPROCESSING := FALSE
        if not stop then
            develop(su)
        if (su.clashes empty) and
                    (su.opens empty) then
            if allrefutations then
                display refutation
                ARTIFICIALPROCESSING := TRUE
                su.clashes := artificialcon(su)
            else
                result := true
        else
            store(su)
    end while
    if result then
        display refutation
    else
        no refutation
end main
```

(continued)

```
PROCEDURE ARTIFICIALCON(SU: STORAGE.UNIT)

    -- RETAIN THE PRIORITY OF THE PREVIOUS
    -- ARTIFICIAL CONFLICTS, THEN DETERMINE
    -- THE NEW SET OF CLASHES AND ASSIGN
    -- THEM NEW PRIORITIES
    CURRENTPRIORITY := SU.ACPRIORITY
    SU.ALLOLDACCLASHES := SU.ALLACCLASHES
    SU.CLASHES := FROMGDC
    SU.ALLACCLASHES := SU.CLASHES
    SETPRIORITY(SU.CLASHES)
END ARTIFICIALCON
```

(continued)

```
PROCEDURE FROMGDC

    --  DETERMINE WHICH GOALS ARE TO BECOME
    --  ARTIFICIAL CONFLICTS
    CLASH := EMPTY
    CLASHES := EMPTY
    DCG := INITIAL(SU.GDC)
    WHILE DCG NOT EMPTY DO
        TERM := INITIAL(DCG.SUBSTITUTIONS)
        WHILE TERM NOT EMPTY DO
            CONFLICT :=  TERM.PLANNODE.KEY.FATHERGOAL
            IF TERMINALNODE(TERM.PLANNODE) THEN
            APPEND(CLASH, CONFLICT)
            TERM := NEXT(DCG.SUBSTITUTIONS)
        END WHILE
        UNIQUECONFLICTS(CLASH)
        APPEND(CLASHES, CLASH)
        CLASH := EMPTY
        DCG := NEXT(SU.GDC)
    END WHILE
  FROMGDC := CLASHES
END FROMGDC
```

Algorithm 4. Version 4 of the original algorithm.

Again, with more refinements to the original algorithm there are some additional identifiers which need to be commented upon.

1) Su.allacclashes is used to remember the original set of artificial conflict as the actual set of clashes may not contain all of them.
2) Su.alloldacclashes is used to save su.allacclashes for use by the next closed plan generated.
3) Artificialprocessing is a flag which tells the deduction process that the last set of clashes generated were artificial conflicts as special actions have to be taken.
4) Term is a temporary which holds the current term in the list of substitutions of the Graph of Dynamic Constraints.
5) Conflict is a temporary which holds the goal that introduced node the term resides in.
4) Terminalnode is a function which returns true if the node the the term resides in is a leaf node.
7) Append is a routine which adds either a conflict to a clash or a clash to a set of clashes.
8) Uniqueconflicts is a procedure which determines that the clash does not contain any duplicate occurrences of any of its conflicts.

The modifications to procedure Main include a flag which indicates that the last set of conflicts generated are artificial conflicts. We also need to save the priority value of the clash which reactivated the backtracking routines, as this is used to control which clashes of the next set of artificial conflicts have to be discarded.

The procedure Artificialcon first retrieves the priority of the last artificial conflict and then saves the set of previous artificial conflicts. The new set of conflicts is determined via the routine Fromgdc and its elements are given the proper priority values.

The procedure Fromgdc chooses which goals are to be designated as conflicts and how they are to be arranged into clashes. This is accomplished by traversing the Graph of Dynamic Constraints. As mentioned earlier this graph is composed of several distinct graphs, each of which represent a set of substitutions. Each of these graphs are then traversed to select those terms in the graph which constitute a leaf node. All these goals, corresponding to one set of substitutions, are grouped together in the same clash. Every individual graph of substitutions will determine a new clash. It is conceivable that many terms in the same graph of substitutions may reside in the same node, which results in the same

goal being named as a conflict many times in the same clash. A post processing routine then has to be invoke to ensure uniqueness of conflicts within a clash.

To clarify how these refinements reduce the redundancy consider the application of this new algorithm to the clashes of example 6. A trace showing the deduction process is shown in figure 9b. In this figure, we again omit intermediate plans and failure nodes. The number to the right of each arc indicates the priority value placed on the clash which originates the arc. Clashes which include more than one conflict are indicated by a line under the node which connects those conflicts.

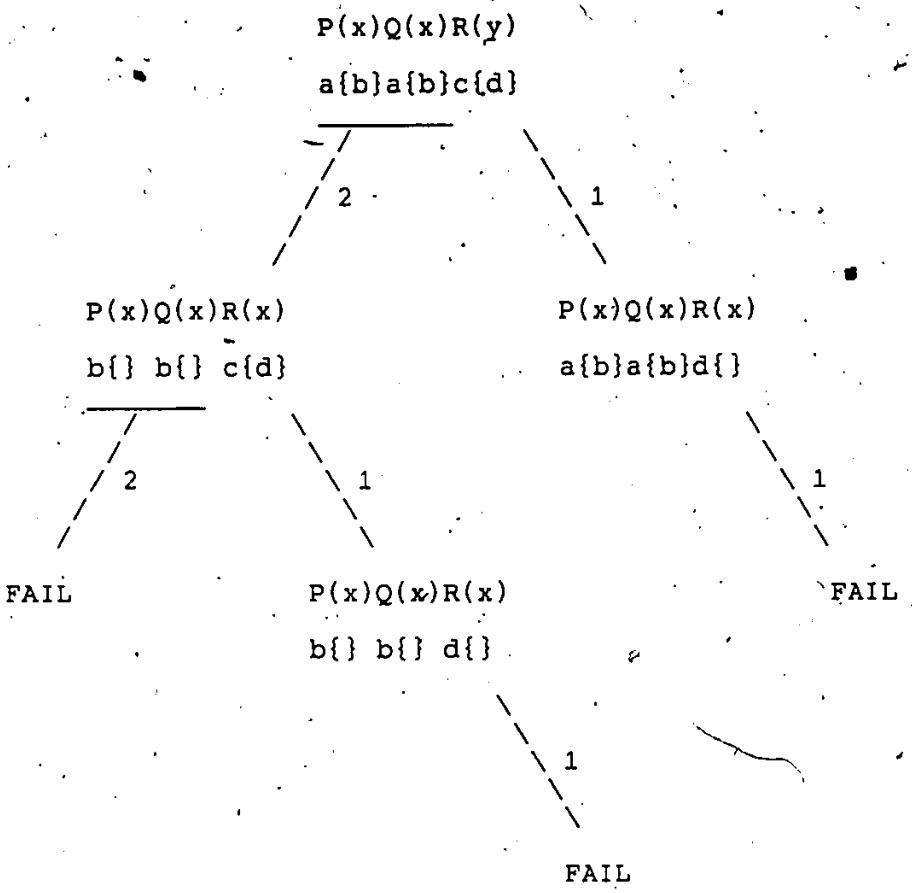

Figure 9b. A second deduction trace for the clauses of example 6.

From this figure it is clear that the refinements to the original algorithm do minimize the redundancy. In fact, for this example there are no redundant refutations generated. As one can see in figure 9b, the duplicate refutations produced at the second level in figure 9a are eliminated by combining all the conflicts whose terms are

bound through variables. So rather than having three nodes at this level, there now resides only two; both of which are unique solutions. At the third level, only one node exists. This is because one of the redundant refutations was removed by eliminating the redundancy at the second level. Another redundant refutation was not generated for the same reason of grouping bound conflicts together. The third redundant solution is eliminated by applying the ordering strategy to clashes. Placing priorities on the clashes keeps the leftmost node at the second level from developing both clashes. So rather than having eight solutions with four duplicates, we now generate only the four unique refutations.

Of course this strategy does not guarantee that all redundant solutions will be eliminated. The same causes of redundancy for this algorithm, as described in the previous section, are still present. Also, if the first plan is not a solution then the ordering strategy for the artificial conflicts does not have a single starting point. In other words, this strategy can only be invoked after a solution is found and if more than one branch of the initial plan leads to a refutation then this strategy will have several starting points. Nonetheless, this strategy has a significant improvement on the number of redundant solutions generated.

THE HUFFMAN-CLOWES POLYHEDRAL WORLD EXAMPLE

This section describes a practical application for the research described in this paper. The example presented here is based on the Polyhedral World of Huffman and Clowes and was suggested to us by M. van Emden. Huffman and Clowes have defined an important problem in computer vision, where certain basic assumptions and theorems are made relating geometric drawings of scenes to real world interpretations. Their theorems are expressed as first-order logic formulas so a deductive algorithm can then answer predefined queries. The example we wish to present here shows how our deductive system can be used to determine the different Huffman-Clowes labelling schemes of a tetrahedron.

To introduce the Huffman-Clowes theory, we need to make the following three assumptions.

1) At every line no more than two planes meet.
2) At every point exactly three planes meet.
3) The point of view is "general". That is, lines of one edge do not coincide with the lines of another edge so that a small change in the viewpoint does not cause a change in the number of lines in the scene.

There are four types of junctions of lines at a point. These are called V, W, Y and T which correspond visually to how the lines meet. There are a total of 18 different possible interpretations of these junctions. To describe these interpretations we use the following notations:

1) A line labelled by "+" indicates that two planes meet and both are visible which gives rise to a convex edge.
2) A line labelled by "-" indicates that two planes meet and both are visible but which yields a concave edge.
3) A line marked with an arrow, ">", indicates that two planes meet and only one is visible. We use the convention that the plane which is visible is to the right when one faces in the direction of the arrow.

We now present the different ways in which the four junctions can be labelled. We note that there are six

possible ways to label a V junction, three for a W junction, five for a Y and four for a T. These are all shown in figure 10.

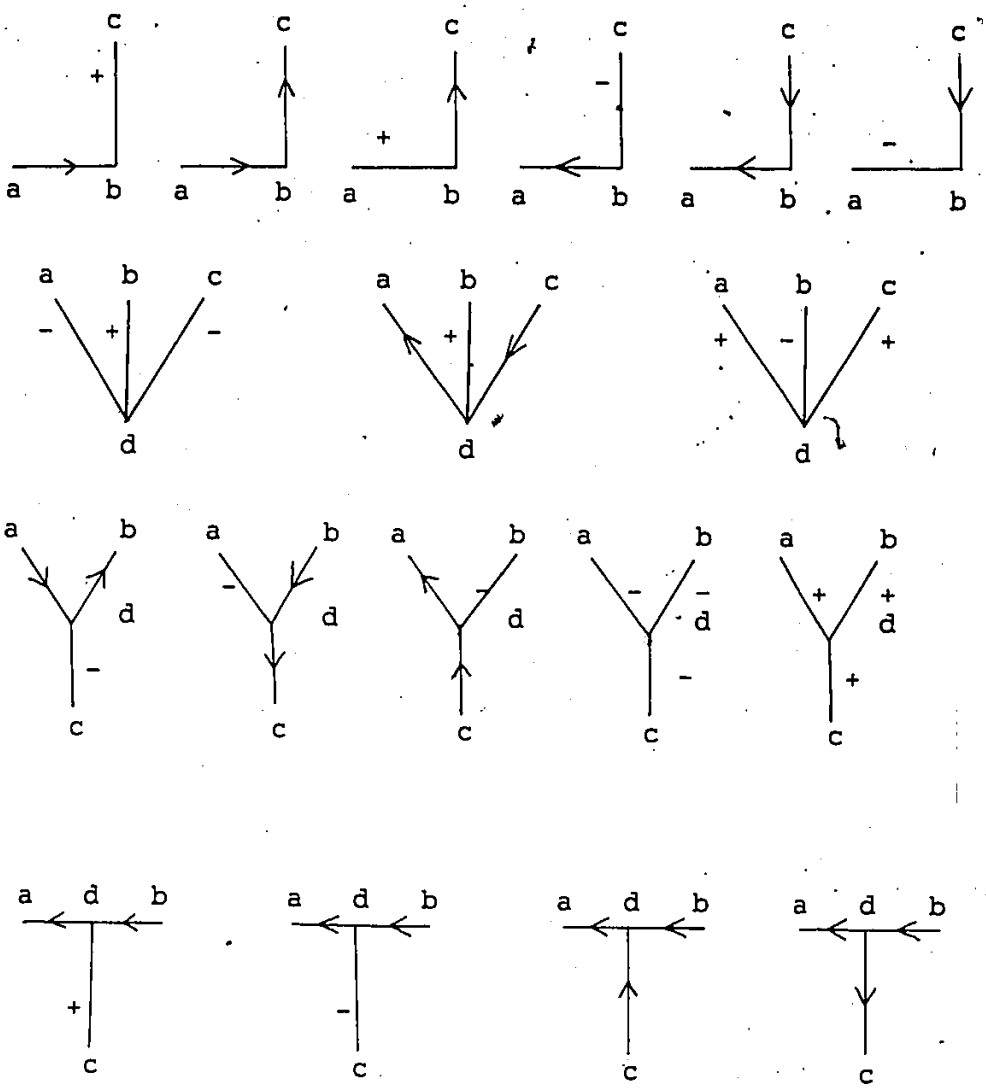

figure 10. The Huffman-Clowes junctions

The next step is to code the knowledge represented by the diagrams in figure 10, into some form which can be used by a deduction system. Therefore, we translate all of the eighteen junctions into first order logic formulas. In each of these clauses the predicate name is used to represent the type of junction. Each clause represents one particular junction, based on a general form using the labels shown in figure 10.

For V-junctions we have the general form:

V(ab,bc,ba,cb),

where each of the arguments represents a label for the specified line segment. Since the direction of the line is relevant we need for example the segment ab and the segment ba. The names of the segments correspond to the names given in figure 10. Thus, for the six possible V-junctions we obtain the following six clauses.

V(>,+,<,+)

V(>,>,<,<)

V(+,>,+,<)

V(<,-,>,-)

V(<,<,>,>)

V(-,<,-,>)

For W-junctions we have the following general form and the three specific junctions:

W(ad,bd,cd,da,db,dc)

W(<,+,>,>,+,<)

W(-,+,-,-,+,-)

W(+,-,+,+,-,+)

For Y-junctions we have the following general form and the five individual junctions:

Y(ad,bd,cd,da,db,dc)

Y(>,<,-,<,>,-)

Y(-,>,<,-,<,>)

Y(<,-,>,>,-,<)

Y(-,-,-,-,-,-)

Y(+,+,+,+,+,+)

Finally, for T junctions we have the following general form and the four particular junctions.

T(ad,bd,cd,da,db,dc)

T(<,>,+,>,<,+)

T(<,>,-,>,<,-)

T(<,>,>,>,<,<)

T(<,>,<,>,<,>)

Interpreting a specific scene now becomes a problem of correctly labelling each junction. For example, consider the tetrahedron in figure 11. We would like to know all the possible ways in which this figure can be labelled to give a valid, real world interpretation.

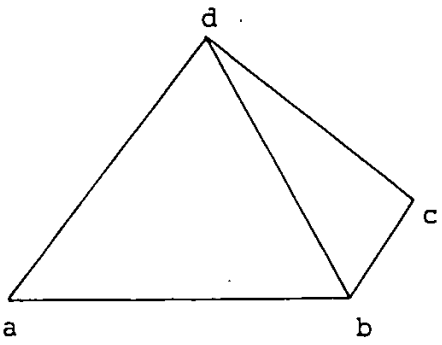

Figure 11. An unlabelled tetrahedron scene.

To determine the possible ways in which the tetradral can be labelled we must formulate a query in first order logic, which uses the clausal form of the junctions we have defined above. The query must consider all the possible junctions. The four junctions in figure 11 are formed by the following sets of sides:

ad, ab --> V(da, db, ad, bd)

ab, bd, bc --> W(ab, db, cd, ba, bd, bc)

bc, cd --> V(bc, cd, cb, dc)

cd, bd, ad --> W(cd, bd, ad, dc, db, da)

The conjunction of these four clauses form the query we wish to solve. To obtain a base of clauses to perform the required deductions we must negate our query and add it to the set of known facts listed above. This results in the following base of clauses on which we can determine all the possible refutations.

-V(da, db, ad, bd) -W(ab, db, cd, ba, bd, bc)
-V(bc, cd, cb, dc) -W(cd, bd, ad, dc, db, da)

V(>,+,<,+)

V(>,>,<,<)

V(+,>,+,<)

V(<,-,>,-)

V(<,<,>,>)

V(-,<,-,>)

W(<,+,>,>,+,<)

W(-,+,-,-,+,-)

W(+,-,+,+,-,+)

Y(>,<,-,<,>,-)

Y(-,>,<,-,<,>)

Y(<,-,>,>,-,<)

Y(-,-,-,-,-,-)

Y(+,+,+,+,+,+)

T(<,>,+,>,<,+)

T(<,>,-,>,<,-)

T(<,>,>,>,<,<)

T(<,>,<,>,<,>)

Using the original plan based deduction system on this base of clauses we obtained the following measurements.

The number of goal insertions is	271
The number of goal deletions is	310
The number of refutations is	6

Obviously, from the magnitude of the numbers, a trace of the deduction process would be impractical to include here. However, we can present the six refutations which were derived. These are shown in figure 12. Notice that of the six refutations, three of them are duplications (iii & iv the same as i, and v the same as ii).

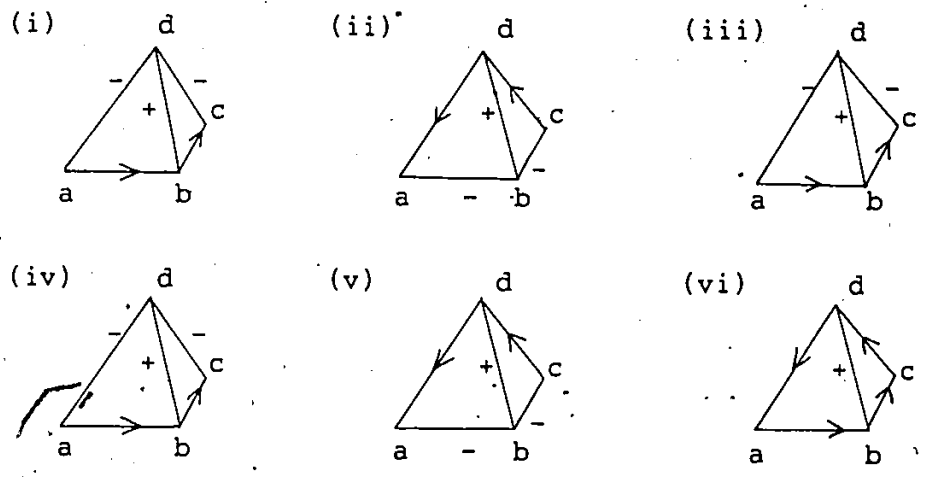

Figure 12. The six tetrahedrons from the original strategy.

Using the modified plan based deduction system on this base of clauses we obtained the following measurements.

The number of goal insertions is	106
The number of goal deletions is	126
The number of refutations is	3

The three unique refutations which were derived are shown in the following figure 13.

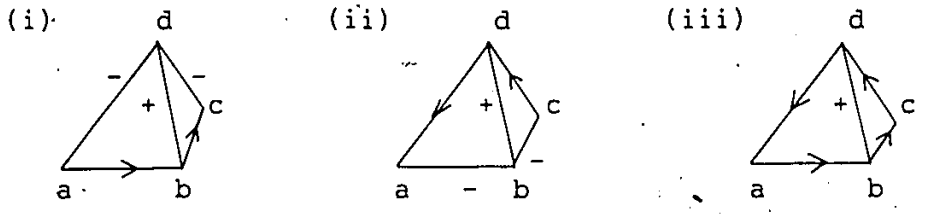

Figure 13. The three tetrahedrons from the enhanced strategy.

From these measurements it should be clear that a significant amount of work was saved through the enhancements outlined in this paper. The three solutions found are unique and are the only three for this base of clauses, so completeness was perserved.

CONCLUSION

An implementation of an enhanced plan based deduction system has now been completed. It involves some 7000 lines of PASCAL code and runs under CMS on an AMDAHL 470/V5.

Reviewing the results of experimentation and suggested implementation strategies, we feel that our research will lead to a practical and efficient deduction system. Our emphasis on attempting to control the system so that it avoids generating redundant solutions is particularly significant. Without some kind of constraint, the system tends to generate an unacceptably large number of identical solutions which makes it impractically slow and inflates its memory requirements. The problem lies with imposing a constraint which does

not restrict the system from generating a complete solution set. Completeness is an important consideration when applying an automated deduction system to cope with the intensional clauses of a large data base. We believe that the algorithms outlined in this paper provide such a constraint, where completeness (of [Matwin & Pietrzykowski 84]) is preserved and reduncancy is significantly decreased.

However, an open and interesting question, still remains. During deduction the system may proceed either depth-first or breadth-first when developing open goals. Is the choice of either strategy relevant for efficiency (if so, in what way?) or is this dependent upon some topological properties of the plan being asserted?

REFERENCES

[Bibel 83]

Bibel, W. "Matings in Matrices", Communications of ACM, Vol 26, No 26, pp. 844-852, 1983.

[Bruynooghe & Periera 83]

Bruynooghe, M. and Periera, L., "Deduction Revision by Intelligent Backtracking", Universidade Nova de Lisboa, Research Report, July 1983.

[Chang & Slagle 79]

Chang, C.L. and Slagle, J.R., "Using Rewriting Rules for Connection Graphs to Prove Theroms",

Artificial Intelligence, Vol. 12, pp. 159-180, 1979.

[van Emden 83]

van Emden, M, personal correspondence.

[Forsythe & Matwin 83]

Forsythe, K. and Matwin, S., "Copying of Multi-level Structures in a PASCAL Environment", submitted to Software - Practice and Experience, 1983.

[Forsythe & Matwin 84]

Forsythe, K. and Matwin, S., "Implementation Strategies for Plan-based Deduction", accepted for publication in proceedings of CADE-7, San Fransisco, USA, May 1984.

[Kowalski 75]

Kowalski, R., "A Proof Procedure Using Connection Graphs", Journal of ACM, Vol 22, No 4, pp. 572-595, 1975.

[Matwin & Pietrzykowski 82]

Matwin, S. and Pietrzykowski, T., "Exponential Improvement of Exhaustive Backtracking: Data

Structure and Implementation", Procs. of CADE-6, pp.240-259.

[Matwin & Pietrzykowski 84]

Matwin, S. and Pietrzykowski, T., "Intelligent Backtracking in Plan-Based Deduction", submitted to IEEE Trans. on Pattern Analysis and Machine Intelligence.

[Pereira & Porto 80]

Pereira, L.M., and Porto, A., "Selective Backtracking for Logic Programs", Procs. of CADE-5, pp. 306-317.

[Pietrzykowski & Matwin 82]

Pietrzykowski, T. and Matwin, S., "Exponential Improvement of Exhaustive Backtracking: A Strategy for Plan-Based Deduction", Procs. of CADE-6, pp.223-239.

[Sickle 76]

Sickle, S. "A Search Technique for Clause Interconnectivity Graphs", IEEE Trans. on Computers, Vol 25, No 8, pp. 823-835, 1976.