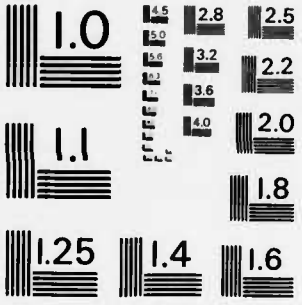
AD-A132 299 THE DESIGN OF BACKTRACK ALGORITHMS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA G LOBERG JUN 83

1/1

UNCLASSIFIED F/G 9/2 NL

END
DATE FILMED
9-83
DTIC



NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN OF BACKTRACK ALGORITHMS

by

Gary Loberg

June 1983

Thesis Advisor: Douglas R. Smith

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Design of Backtrack Algorithms		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June, 1983
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Gary Loberg		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June, 1983
		13. NUMBER OF PAGES 94
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

backtrack algorithms, program synthesis, control structure abstraction, problem reduction, state space search, graph search

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The backtrack control structure is a well known combinatorial problem solving approach in computer science. The strategy can be abstracted into a program schema with slots for lower level functions which is suitable for the automated synthesis of backtrack programs. Employing a known model of program synthesis based on a problem reduction problem representation, two reduction rules are developed for transforming a (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract (Continued) Block # 20

problem specification into a backtrack control structure with specification into a backtrack control structure with specifications for lower level functions. We illustrate these rules with sample problems.

Approved for Public Release, Distribution Unlimited

The Design of Backtrack Algorithms

by

Gary Loberg
Captain, United States Army
B.S., United States Military Academy, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1983

Author: Gary Loberg

Approved by: Douglas R. Smith
Thesis Advisor

Second Reader

David K. Hsiao
Chairman, Department of Computer Science

Dean of Information and Policy Sciences

ABSTRACT

The backtrack control structure is a well known combinatorial problem solving approach in computer science. The strategy can be abstracted into a program schema with slots for lower level functions which is suitable for the automated synthesis of backtrack programs. Employing a known model of program syntnesis based on a problem reduction problem representation, two reduction rules are developed for transforming a problem specification into a backtrack control structure with specifications for lower level functions. We illustrate these rules with sample problems.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

The backtrack control strategy has developed into one of the major classes of algorithms since its first appearance in the literature of computation. This has been recognized by many authors and most current textbooks on algorithms, including those by Aho, Hopcroft and Ullman [Ref. 1] and Horowitz and Sahni [Ref. 2], include substantial sections on the strategy. Skill in the development of backtrack algorithms can be as useful to programmers as their skill with other general algorithm classes, such as the divide and conquer, greedy and dynamic programming control strategies. A minor goal of this paper is to further refine knowledge of the structural relationships within a backtrack algorithm.

This expert knowledge of backtrack programming techniques can also be used in the program synthesis process. The problem reduction approach to program synthesis detailed in Smith [Ref. 3, 4] employs reduction rules in the form of algorithmic schemas and supporting heuristic knowledge concerning subschema specification to decompose a problem specification to a series of simpler specifications. Program solutions to these subspecifications are ultimately composed via the schema structure into a program satisfying the original specifications. The major goal of this paper is to produce

two such schemas for the backtrack control strategy and two corresponding design methods for employing these schemas.

The first discussion of backtrack by Walker [Ref. 5] was a fairly general description of a technique then in use for deciding combinatorial problems. Further descriptions of the technique, such as Golomb and Baumert [Ref. 6] and Bitner [Ref. 7] were oriented towards the efficiency aspects of the strategy. This approach to the study of backtrack algorithms was reflected in texts on combinatorial algorithms, such as that by Reingold, Nievergelt and Deo [Ref. 8]. With this emphasis on the development of specialized techniques for improving efficiency the study of the general properties of the backtrack class was overlooked. The paper by Gerhart and Yelowitz [Ref. 9] reversed this trend. They developed a series of backtrack schemas differentiated by the type of control (recursive or iterative) and the type of solution (first, all or optimal) desired. The emphasis was on the development of schemas proven to be correct along with general specifications for the subschemas which would aid in proving the correctness of the algorithms developed to complete the program.

This paper attempts to address two perceived gaps in the understanding of backtrack algorithms. The first gap lies in the development of schemas in a notation suitable for automated program synthesis. This notation should allow for simpler program verification techniques than those used by

Gerhart and Yelowitz. The schemas should also be accompanied by heuristics for instantiation of the schema to satisfy a given problem specification. Chapters II, III and IV will address these concerns by describing the program syntnesis system (Chapter II), the characteristics of a backtrack algorithm (Chapter III) and a backtrack program schema and associated design method (Chapter IV). The second gap lies in the extension of the backtrack strategy to solve a class of problems which have not generally been solved by a backtrack control structure in the past. Chapter V will develop a schema and associated design method for searching a solution graph of a problem with a hierarchical structure. Chapter VI will conclude this paper and point to further areas of research.

II. THE PROGRAM SYNTHESIS SYSTEM

The program synthesis model for this research is the problem reduction approach as developed by Smith [Ref. 10, 11]. This approach is an attempt to formalize the programming discipline of top down design as a hierarchical, problem reduction structure. A brief examination of this model will help identify the type of knowledge required to synthesize a backtrack program.

A. THE PROBLEM REDUCTION MODEL

The key concept in this model is that program development by top down design is a problem reduction approach to the programming problem. Top down design is accomplished through successive refinement of a problem specification into a series of simpler subspecifications. These subspecifications are related through control structures which direct control through the subprograms. At each step of the refinement process the subspecifications from the previous step are further refined. This continues until all are replaced by the primitive constructs of the programming language. The entire program is then composed from the primitive language constructs and control structures produced during the refinement stages.

A problem reduction problem solving approach attempts a solution by applying reduction operators to a problem goal

statement. These reduction operators decompose the goal into a number of simpler subgoals and additionally provide a framework for composing the solutions to the subgoals into a solution to the original problem goal. Also required is a set of primitive operators which allow direct solving of a subgoal. By successively decomposing a problem until a primitive operator can be applied to each subgoal and then composing these solutions with the structure provided by the reduction operator, a solution to the original problem is found.

The analogy between problem red ction problem solution and top down design is obvious. The goal statement in a program syntnesis system is a formal specification of a problem. A primitive operator of a program syntnesis system is a programming language construct. The reduction operators include a procedure for developing subspecifications (design strategy in Smith [Ref. 12], design method above) and a structure for composition of the subspecification solutions. The structures chosen for the reduction operators are program schemas which reflect the different control strategies. The program synthesis problem is to develop a program schema/design method pair which allows synthesis of correct programs.

A simple example should help illustrate this process. Suppose our specification requires the selection of the maximum of two natural numbers given as input. The goal

specification may look like:

```
MAX(A,B) = C such that
  [A>=B <=> C=A]  &
  [B>A <=> C=B]

where MAX: (NxN) -> N
```

This specification for a function named MAX states that MAX takes two natural numbers as input and returns a single natural number. The logic specification consists of a conjunction of two clauses. Each clause must therefore be true for the output to be correct. Both conjuncts are logical equivalences, which requires both sides of the equivalence to be true or both sides false for the equivalence to be true. Thus we have a specification in which if A>=B, C must equal A, and if B>A, C must equal B. Thus C must be the maximum of A and B. If our programming language had a suitably defined function MAX(X,Y), then a primitive solution to this goal could be applied. If not, the goal must be further reduced to allow for solution. One reduction rule which could be applied is a simple conditional. With this rule a control schema would be imposed and subspecifications would be developed for the schema slots. The schema may look like:

```
if P
  then F
  else G
```

Where P, F, G are functions the rule will specify. The specifications produced by the rule may be:

```
P:(A,B) = b  such that
   [A>=B <=> b]
where P:(NxN) -> B

F:A = C  such that
   [A = C]
where F:N -> N

G:B = C  such that
   [B = C]
where F:N -> N
```

With these specifications P can be directly solved by a simple relational operator and F and G can be solved by an assignment operator, and the final program produced will be:

```
MAX(A,B) =
   if A>=B
      then C <- A
      else C <- B;
   return C
```

B. PROBLEM SPECIFICATION

The program synthesis system requires a formal specification of a problem. This formal specification is a logical description of the input/output relationships for the program. The following format will be used to specify problems in this paper:

```
F:x = z such that I:x => O:<x,z>
where F:D -> R
```

In this instance, F is the name of the specification and the : operator indicates function application.

There are four components to a formal specification. The input condition I details all known properties of objects input to the program. If the input condition applied to some object x is true, then the program must

produce the specified output. In many cases the input condition will be vacuously true. The output condition O specifies the relations that are expected to hold between the input objects and the output objects. The domain D specifies the data type of input objects and the range R specifies the data type of output objects. The program synthesis system will attempt to derive a program F which takes as input an object of type D and produces as output an object of type R. If this input object satisfies the input condition then the output condition applied to the input and output objects will be true.

C. THE PROGRAMMING LANGUAGE

The target programming language for this system is a functional language similar to Backus' FP notation [Ref. 13]. A functional language provides several advantages to the program synthesis process. The most significant is the relative ease of program verification. Although not a trivial task, the proof techniques are more manageable than those for procedural languages. The principal reason for this lies in the nature of expressions. A functional program constitutes a single expression. Within this expression all occurences of a name or subexpression have the same value. Thus the statement by statement state changes within a procedural language which create most of the difficulty in program verification do not exist with functional programs. This permits an algebra of functional

programming, as Backus further discusses [Ref. 14] which permits use of the language as a proof tool. A second advantage lies in the hierarchic nature of functional languages. Higher level functions are constructed from lower level functions and appropriate combining functional forms. The reduction rules in the synthesis system are actually methods for producing specifications for lower level functions and schemas which connect the specifications with the appropriate combining forms.

A functional language contains a set of five components [Ref. 15], which are:

1. a set of objects
2. a set of functions
3. the application operation
4. a set of functional forms
5. a function definition mechanism

The functional language used is fully described in Appendix A. The following paragraphs highlight the major differences between Backus' notation and the language notation used.

1. Set of Objects

The set of objects in this language include specific data types. The particular data types which will be necessary in this paper are N, the natural numbers, LIST(N), lists of natural numbers, I, the integers and B, the boolean values true and false. Also included is the data structure <>, sequences of objects.

2. Set of Primitive Functions

The set of primitive functions are tied to the various data types and structures. A complete set of functions is given in Appendix A.

3. The Application Operation

Function application is enhanced by allowing the use of named parameters in both the application and definition of functions. This deviates greatly from Backus' intentions, but obviates much of the use of selector functions in data manipulation. At the least it increases the clarity of function definitions. A further motivation is the knowledge that efficient algorithms [Ref.16] exist to extract named parameters from function definitions. A declaration mechanism is also included to allow for controlling name visibility.

4. The Function Definition Mechanism

An anonymous function definition mechanism, similar to the LISP lambda function, is included. The syntax is:

```
(lambda <parameter list>
        {function definition})
(actual parameter list)
```

This will be most useful for schema expression, as it allows for fully specifying a lower level function within a higher function. In the backtrack schema we shall use this feature to express a lower level function in terms of its component functions, thereby directly expressing all components of the backtrack strategy and their relationships.

III. THE BACKTRACK CONTROL STRATEGY

The backtrack control strategy is essentially a technique applicable to combinatorial problems. A backtrack algorithm will conduct an uninformed search of a state space to select those states which satisfy the problem constraints. The advantage of a backtracking algorithm over other uninformed search techniques is that it can employ the problem constraints to prune the state space tree, thus reducing the amount of search required.

A. STATE SPACE SEARCH

A state space problem representation attempts to define a problem through description of the various states of the problem world and methods in the problem world for transforming a given state into a new state. In the computer solution of state space problems the fundamental concepts are the symbolic representation of the relevant aspects of the problem state and the computation of permissible state transformations. These permissible transformations are problem world related in that they represent transformations the problem world would permit. For example, a permissible transformation may well lead to a problem state which violates a constraint, but is an allowable action in the world being modelled. The solution technique most often used to solve state space problems is

some form of search. The search commences at a given initial state and proceeds through a directed graph, where the graph nodes represent the possible states and the arcs represent the permissible transformations. The search terminates when a goal state is reached.

An illustrative example is the missionaries and cannibals problem. In this problem we are given an equal number of missionaries and cannibals on a river bank and a boat which can hold at most two persons. The goal is to get all missionaries and cannibals to the other bank without leaving more cannibals than missionaries on either bank at any time. To represent this problem with a state space representation we must identify the relevant aspects of state and develop a symbolic representation for them. We must also develop routines to compute allowable transformations between state descriptions. The solution to this problem will be a sequence of transformations which move the missionaries and cannibals from one bank to the other and which do not violate the problem constraints.

A number of techniques exist for searching state space graphs. They differ principally in the technique used for selecting which already visited state to expand, or to transform to a new state. Uninformed techniques such as depth first, breadth first and generate and test search transform known states in an arbitrary and fixed manner. The backtrack strategy, as we shall see, is an example of an

uninformed search. An informed technique, such as best first search, will use some type of knowledge to evaluate the known states and select the most promising of these for expansion. The decision of whether to use an informed or uninformed search is most often a function of the problem and how well search knowledge can be codified.

B. GENERAL DESCRIPTION OF APPLICABLE PROBLEMS

Backtrack is suited for the solution of combinatorial problems which exhibit certain characteristics. These characteristics include the ability to segment the problem into a set of discrete but interrelated decisions, a solution structured as a vector of decisions, and a set of testable solution constraints which relate the decision elements.

1. Problem Characteristics

Representation of a problem as a set of discrete decisions structures the problem into a tree search problem. Each node of the tree represents a decision to be made and each arc from that node represents a different alternative solution. In the missionaries and cannibals problem a node may represent the decision: who gets in the boat to go to the opposite river bank? Each arc represents a different alternative: one or two missionaries, one or two cannibals or one missionary and one cannibal. By forcing this tree structure onto the problem, backtracking algorithms do not have to be concerned with maintenance of

solved node lists or other storage outside the path from the current node to the root of the tree. In fact, the state space tree is implicit in backtrack algorithms and not explicitly stored.

Representation of the solution by a vector of decision solutions corresponds directly to the path in the state space tree explicitly stored at any time by a backtrack algorithm. In our state space model this path is the current state. This direct solution representation precludes a requirement to construct a solution once the search has concluded.

The problem defined constraints on solution element relationships allow backtrack algorithms to test the current sequence of decisions (path from root to current node) and prune the implicit search tree without explicitly examining all nodes of the tree. The time efficiency of a backtrack algorithm, measured by the number of nodes examined, is a function of how well constrained these relationships are. The tighter the constraints, the less nodes will be examined. Without constraints, the algorithm will examine all nodes of the state space tree.

2. K QUEENS Problem Representation

An example representation will illustrate how a simple combinatorial problem can be represented for solution by a backtrack algorithm. The problem, traditionally used to explain backtrack, is the K QUEENS problem. Simply

stated, the K QUEENS problem is to find all possible board positions on a KxK chessboard for K queens such that no queen attacks any other queen. From the rules of chess, we must find all positions such that no two queens are on the same row, on the same column, or on the same diagonal.

To represent this as a series of decisions we note that no two queens may be on the same row. Also, if we are to place K queens on a KxK board, there must be at least one queen on each row. It follows that there must be one and only one queen on each row of the board. Therefore, the decision to make at level i of the tree is where to place the queen on row i.

The solution vector returned will be a path from the root to a leaf of the tree. Position i of the vector will represent the positioning of the queen on row i. Thus the solution will have the form

$X = (x(1), x(2), \ldots, x(K))$

where each x(i) is the position (column number) of the queen on row i.

The constraint relationships can also be determined from the rules of chess. These constraints reflect the facts that no two queens can be on the same column or diagonal. To express the column constraint in a computable form we note that our representation would depict two queens in the same column as two elements of the solution vector having the same value. We can restrict this with the

constraint:

```
column constraint
  FOR ALL x(i),x(j) IN X
  [i≠j => x(i)≠x(j)]
```

The diagonal constraint is a little more difficult. Two queens are on the same diagonal if their row distance is the same as their column distance. For example, queens at row and column positions (1 4) and (3 6) are on the same diagonal as are queens at positions (1 4) and (3 2). We can thus subtract the queens' row numbers and column numbers and then compare their absolute values to determine if they are on the same diagonal. This gives us the diagonal constraint:

```
diagonal constraint
  FOR ALL x(i),x(j) IN X
  [i≠j => abs(i-j)≠abs(x(i)-x(j))
```

One final constraint identifies a path as a solution and thus may be termed a solution constraint. This constraint is identified by the fact that K decisions must be made to place K queens on the board. A computable solution constraint is thus length(X) = K. The complete representation is given in Figure 1.

C. GENERAL DESCRIPTION OF THE STRATEGY

Backtrack is best defined as an uninformed, exhaustive, depth first tree search strategy. The strategy is uninformed, in that it does not employ problem specific knowledge about how to search for a solution state. It is exhaustive in that it will implicitly or explicitly examine

all possible solution states as it executes. It is a tree search strategy because it implicitly structures the problem into a tree which represents solution states by a path from the root to a leaf. It is a depth first strategy because it fully examines a subtree defined by one alternative before it begins examination of the next alternative.

```
DECISION STRUCTURE
  decision(i) = column placement for queen on row i

SOLUTION STRUCTURE
  <X> where each X = (x(1), x(2), ... ,x(K))
      where x(i) = column number for queen on row i

CONSTRAINT STRUCTURE
  element constraints
    FOR ALL x(i),x(j) IN  X
        [i≠j => x(i)≠x(j)]
        [i≠j => abs(i-j)≠abs(x(i)-x(j))]
        [1 <= x(i) <= K]
  solution constraint
      length:X = K
```

FIGURE 1
K QUEENS Problem Representation

A backtrack strategy attempts to construct a solution vector one element at a time. After deciding on one element, the strategy will expand this solution one element further. If the strategy determines no expansion is possible and a complete solution has not been achieved then it will backtrack, change its most recently made decision, and try to expand the new partial solution.

To implement this strategy, a backtrack algorithm takes as an input parameter a description of the path from the

root of the state space tree to the node being expanded. The algorithm will expand this node by creating descriptions of all possible paths from the root through the expanded node with length equal to one greater than the parameter path. The algorithm will then examine these new paths in an arbitrary order. This examination first tests the path for a solution and returns the path if it is found to be a solution. If not a solution, it tests for any violation of a predefined subset of the problem constraints. If a violation is found, the algorithm determines no solution can be found with further exploration and terminates search on this path and all possible extensions. If there are no constraint violations, the path is recursively expanded to search for a solution deeper in the tree.

Recursion is the natural form of expression for backtrack algorithms. Using standard program transformations Horowitz and Sahni [Ref. 17] and Gernart and Yelowitz [Ref. 18] have developed iterative backtracking procedures from their recursive algorithms. This paper, since it is not concerned with efficiency issues, will develop algorithms and schemas in recursive notation and leave for later program transformation work the translation into iterative notation. With this in mind, Figure 2 gives a simple backtrack function in a procedural notation.

The efficiency of a backtrack algorithm principally depends on how the path element constraints contained in the

predicate FEASIBLE are defined. The pruning efficiency of the predicate is directly related to the degree of constraint being tested. The more constraining the relationships, the more pruning will be accomplished. As discussed above, the pruning constraints will often be a subset of the total problem constraints. For these reasons, a good heuristic is required for selecting the appropriate constraints if a good backtracking algorithm is to be developed by a programmer or an automated synthesis system. A synthesis design method based on such a heuristic is thus desirable.

The computation of the predicate FEASIBLE highlights one further characteristic of the strategy. The relationships expressed in the predicate often involve data about the path elements. This data must be visible to the predicate, which normally implies extensive parameter passing at each call of the function. The data relevant to each element of the path is very often static, however. The data can be seen as properties of the separate elements, and the constraining relationships as relationships between the elements' properties. For this reason, many backtracking algorithms establish these properties as global data, which can be accessed from any level of the recursion.

```
PROBLEM (PARM_LIST)  <-  BACKTRACK (NIL)

where
FUNCTION BACKTRACK (PATH)   is defined as

  ALTERNATIVE_SET <- GENERATE (PATH,PARM_LIST)
    /* generate is a function which will return all
       extensions to PATH */

  SOLUTION_SET <- {};

  FOR P IN ALTERNATIVE_SET DO

      IF SOLUTION (P)
         THEN SOLUTION_SET <- SOLUTION_SET U {P}
        /* solution is a predicate which returns
           true if the parameter is a solution
           to the problem */
      ELSE
          IF FEASIBLE (P)
             THEN SOLUTION_SET <-
                  SOLUTION_SET U BACKTRACK (P);
        /* feasible is a predicate which returns
           true if the parameter can be expanded */


  END FOR;

  RETURN SOLUTION_SET;

END BACKTRACK
```

FIGURE 2
General Backtrack Function

The algorithm described above is a simple description of a backtrack strategy which returns all solutions in the problem defined state space. Two other variants of backtrack often arise. The first variant is a strategy which returns only the first solution discovered. The second variant returns only the best solution encountered, where the solutions have been ordered by some scoring

function. Both of these variants require additional control features which complicate the basic backtrack strategy and will not be further discussed in this paper. For those interested, Gernart and Yelowitz [Ref. 19] provide further discussion of this topic.

IV. A BACKTRACK REDUCTION RULE

A reduction rule for implementing a backtrack algorithm has two components, the program schema and the design method for subschema specification. This chapter develops a schema for a simple backtrack algorithm with slots for three subalgorithms. A design method is then presented for reducing the problem specification into subalgorithm specifications. The method is based on an examination of the required relationships of the three subalgorithms. Two problems are then examined to illustrate the application of the reduction rule.

A. SCHEMA DEVELOPMENT

In developing a program schema one approach is to describe completely the expected input to the schema, the desired output from the schema and the series of transformations on the input the schema is required to perform to produce the output. These transformations can then be translated into lower level functions connected by the language combining forms. The following paragraphs derive a schema in the desired functional language using this procedure.

1. The Expected Input

From the general discussion of the backtrack strategy (see page 23) we can describe the expected input

and its salient characteristics. When a backtrack function is invoked it is passed one parameter, a vector representation of a partial solution to the problem. We will call this vector PATH, since it is a path from the root of the state space tree to the last node (last element of the vector) examined. PATH is of unknown length, since the function is called at every level of the state space tree. A null PATH can also exist, which indicates no decisions have yet been made. This is the problem state when the initial invocation occurs.

Although the length of PATH is unknown, there are characteristics which can be inferred. The most significant is that PATH has been determined not to be a solution. If the previous invocation of the function had determined that PATH was a solution then the function would have terminated prior to the recursive invocation we are concerned with. A second characteristic is that PATH meets the test of the predicate feasible. A major assumption of this design method is the conclusion that although PATH may not satisfy all the output conditions required by the problem specification, it satisfies a major subset of the conditions. Furthermore, there is reason to expect that an expansion of PATH will eventually satisfy all the output conditions. The current backtrack invocation must therefore search for all such expansions.

Another input issue concerns the problem data which will be required by the lower level functions. The assumption made in the development of this paper is that this data will be made global. Figure 2 (see page 27) demonstrates how this is accomplished. All program specifications developed will declare this data as a parameter to the program, then declare the BACKTRACK function and lower level functions at the same scope level, providing the required visibility. The alternative is to declare the data as input to BACKTRACK and pass it as a parameter to every recursive call of the function. In the K QUEENS example the only data is the value of K. The cost of passing this parameter will be minimal. In other examples, such as the Processor Sequencing Problem we discuss later, the data is much more extensive and the parameter passing costs are higher. In any case, it is simpler to consider this data as global and not be concerned with the mechanics of creating parameter lists.

2. The Desired Output

The output from a backtrack algorithm is also a path or list of paths. These paths, in vector form, represent all possible solutions to the problem. Each invocation of the backtrack function examines a subtree of the state space tree to search for an extension to PATH which terminates in a solution. The shorter PATH is the deeper the subtree examined will be. In any subtree there

is a possibility of zero, one or more solutions which will be returned to the invocation examining that subtree. The backtrack function must compose these separate path solutions into a list of subtree solutions.

3. Input Transformations

The input transformations are also apparent from the strategy description (see page 23). There are three transformations to perform. The first of these is an expansion of the current partial solution by one additional decision. At the simplest level this transformation must produce a set of all paths which are possible expansions of PATH. Each path in this set represents expansion of the partial solution by one additional decision element. Each possible decision is represented by a corresponding element in the set. The result of this transformation is a set of paths to be examined.

The second transformation is to execute a series of conditional tests. These tests perform the examination of each path produced by the first transformation. The significant characteristic of the strategy is that the tests and resulting action are completed for each path before any processing begins on any other path. We will call the path under consideration TEST_PATH. The tests and actions can be subdivided into two sets. The first set tests for a solution. If a solution is discovered, the action is to return TEST_PATH. If the first test fails, the second set

tests for feasibility of expansion. If this test decides expansion is feasible, the backtrack function is recursively called with TEST_PATH as the parameter. If the test fails no further expansion is feasible and the nil path is returned to signify no solution is found.

The final transformation is required to eliminate the nil paths in the solution once all expansions have been examined. After this transformation is complete, the value returned will consist of a list of solutions.

4. Schema Translation

Translation into a program schema requires grouping desired transformations into lower level functions and specifying the appropriate functional forms for relating the inputs to and outputs from the functions. The ability to separate the backtrack strategy into three transformations of the input implies that we can define three lower level functions to perform the transforms. The following paragraphs develop these three functions and the proper combining forms.

The first transformation operates on the input to the schema, the parameter PATH. This allows specification as a direct function application to the parameter. The output of this application is to be a list of all paths which are expansions of PATH. Since the operation is to generate all possible expansions, we will name this function GENERATE. In our language notation this is: GENERATE:PATH.

The second transformation operates on the output of the function GENERATE. Its method is to operate on an element, return a value, operate on the next element, return a value and continue until the list provided by GENERATE is exhausted. This operation is clearly an example of the APPLY-TO-ALL functional form available in our language. Since the operation is a test of each element of the list we will name the function TEST. In our language notation this is:

```
αTEST(GENERATE:PATH)
```

We know more about the behavior of the function TEST, however. TEST is a conditional function with two predicates. We can further specify TEST within the schema by employing this knowledge. The first predicate is a solution test. The resulting action is to return the path if the predicate holds. In our language this is:

```
SOLUTION:TEST_PATH -> ID:TEST_PATH;
```

The second test is a check for feasibility. The action is to recursively call the backtrack function with the path as parameter. This can be expressed in our language as:

```
FEASIBLE:TEST_PATH -> BACKTRACK:TEST_PATH;
```

The final action of the function is to return nil. The use of an anonymous function definition will allow definition of TEST within the schema as follows:

```
α(lamoda<TEST_PATH>
        {(SOLUTION:TEST_PATH -> ID:TEST_PATH;
          FEASIBLE:TEST_PATH -> BACKTRACK:TEST_PATH;
          NIL)})
(GENERATE:PATH)
```

The final transformation eliminates all null lists in the list returned by the partial schema above. The appropriate lower level function and combining form already exist in our language. The functional form INSERT will move the function APPEND through the list and eliminate all null list occurences. All that is required is to compose this function and combining form onto the partial schema to produce the backtrack schema of Figure 3.

```
BACKTRACK:PATH
   /APPEND
   (α(lambda<TEST_PATH>
         {(SOLUTION:TEST_PATH -> ID:TEST_PATH;
           FEASIBLE:TEST_PATH -> BACKTRACK:TEST_PATH;
           NIL)})
   (GENERATE:PATH) )
```

FIGURE 3
Backtrack Program Schema

B. DESIGN METHOD FOR SUBSCHEMA SPECIFICATION

The schema developed above is only one component of the required reduction rule. Also necessary is a design method for specifying the lower level functions GENERATE, SOLUTION and FEASIBLE. A rule for derivation of these subspecifications must be based on the expected input and output of the functions and the relationships between the functions which the schema exploits to solve the problem. The reduction rule developed in the following paragraphs builds from these relationships. The rule provides a specification schema for each for each lower level function

and a method for instantiating the schemas for a given problem instance. The method is a pattern matching process which replaces references to the problem specification in the schemas with the referenced components of the problem specification. In developing the schemas the notation used below is the same as the problem specification notation, with two additions: Capital letters refer to the components of the specification and lower case letters refer to the function or problem specification. Thus Op refers to the output condition of the problem specification, while Os, Of and Og refer to the output conditions of the functions SOLUTION, FEASIBLE and GENERATE respectively.

1. GENERATE Specification Schema

To derive a general heuristic for specifying the GENERATE function we need to closely examine the output requirements for the function. Backtrack requires GENERATE to produce all single decision extensions to PATH. It is significant that GENERATE is the only function in the schema which produces output. Each element of this output is a potential solution. The implication is that GENERATE must perform all computation required to construct a decision element and append it to PATH. This computation may require incorporation of constraints from the problem output condition. The K QUEENS problem provides a simple example. In this problem there is a direct constraint on the value of the decision alternatives, this being that the column number

for each decision must be between one and K. Failure to include this constraint in GENERATE may result in the production of an infinite sequence of path extensions.

A heuristic to support this reasoning can be designed. If a constraint exists which places direct restrictions on the computed value of a decision element then this constraint should be included in the specification for GENERATE. What constitutes a "direct restriction" is not well formulated, but two general principles are offered. If a constraint restricts a decision element by a specified relation to constant values, then this is a direct restriction. The K QUEENS constraint above falls in this category. Secondly, if a constraint is formulated as an equality between a decision element and a computable value, then the constraint directly restricts the decision. We will show an example of this later. On a more general note, the issue of which function to include constraints in is a major point of concern to algorithm designers and is further addressed in the section on schema limitations.

There are other output conditions for the GENERATE function. If GENERATE is to produce single decision extensions to PATH then the length of each element of the output must be one greater than the length of PATH. Also, each element of the output minus its last decision is equal to PATH. A clean symmetry exists between these constraints. We have restricted the size of each element of the output,

the value of the last decision of each element, and the values of the rest of the decisions. This suggests a completeness in the specification. The complete output condition Og can be expressed as:

```
Og = FOR ALL TEST_PATH  IN  PATH_LIST
     [length(TEST_PATH) = 1+length:PATH &
      tlr(TEST_PATH) = PATH &
      Opg(last(TEST_PATH))]

where Opg = subset of Op which directly
            restricts a decision
```

There are certain conditions known to be true of the input. As discussed in the paragraph on schema development PATH is known to be feasible. This fact may be used by the synthesis system and needs to be represented as an input condition. The specification input condition is thus:

FEASIBLE(PATH)

To derive the domain and range of GENERATE we need to examine the relationships between the input and output of the function and those of the problem. Generate accepts as input a path representation for which it is to generate allowable expansions. Although not a solution, PATH is the proper type of a solution. We can discover the solution type by examining the range of the problem. The problem is to produce a sequence of solutions. The range of the problem is thus a sequence of the desired type. Given a problem range of <Y>, which signifies a sequence of objects of type Y, where Y is a language type we can extract Y as the domain of GENERATE. The function must output a sequence

of objects, each of which is a potential solution. This is the same output type as the problem and the problem-range can be substituted for the range of GENERATE. This produces a domain and range specification of:

Dg = Y where Rp = <Y>
Rg = Rp

The complete specification schema is given in Figure 4.

2. SOLUTION Specification Schema

The function SOLUTION is the simplest of the lower level functions to specify since it relates directly to the entire problem specification. SOLUTION is a function which accepts a path representation as input and returns a boolean value. The representation SOLUTION tests is the same type as the elements of the problem domain. In the K QUEENS problem, for example, the problem range is <LIST(N)>. We want the program to produce a sequence of lists, where each list is a solution. The corresponding domain for SOLUTION is simply LIST(N). Since the function is a predicate, it must return a boolean value. The domain and range can thus be specified as:

Ds = Y where Rp = <Y>
Rs = B

To derive the input and output specifications we note that SOLUTION must return true when the problem output conditions applied to the parameter TEST_PATH are true and must return false when the problem output conditions applied to the parameter are false. This can be expressed as a logical

equivalence between the boolean value returned and the problem constraints applied to the parameter TEST_PATH. Since some of the constraints may be included in GENERATE, we need only include the subset not in GENERATE. The input condition follows from the input condition to GENERATE. Since the input to GENERATE is known to be feasible, the input to SOLUTION minus the last element must be feasible. The input and output conditions may be expressed as:

```
Is = FEASIBLE(tlr:PATH)
Os = Ops(TEST_PATH) <=> b

where Ops = subset of Op not included
            in GENERATE specification
```

The complete specification schema is given in Figure 4.

3. FEASIBLE Specification Schema

The specification of FEASIBLE is more difficult than that for SOLUTION because the feasibility test is the less constraining of the two. An assumption of this design method is that FEASIBLE is a relaxation of the constraints represented by SOLUTION. One rule for relaxing restrictions is to eliminate one or more expressions within a conjunctive statement of constraints. We attempt to develop a heuristic for identifying which conjunct or conjuncts of the constraints stated in the problem output conditions to include in the feasibility test.

The backtrack schema expects certain characteristics of the path being investigated. A path which is feasible yet not a solution fails to meet one or more of the output

conditions. However, it is feasible that an expansion of the path may meet all the output conditions. A path determined to be unfeasible also fails to meet one or more of the output conditions. The difference is that an unfeasible path will never meet all the output conditions, no matter what sequence of decisions is appended. If we can specify the type of condition which, when failed by a partial solution will also be failed by any extension to that partial solution, then this knowledge can be added to our reduction rule.

A heuristic can be formulated to express this knowledge. A constraint which addresses the solution as a whole is not of this type. If the path as a single entity fails a condition, then any expansion to the path produces a different entity, and may pass the condition. A constraint which limits the relations between the parts of the solution is of this type, however. If a partial solution exhibits a conflict between two elements the same conflict will exist no matter what subsequent elements are appended to the path. The conclusion is that the appropriate constraints are a subset of the problem output conditions and can be selected by an heuristic process which retains only those constraints which relate elements of the proposed solution. The input and output conditions can be expressed as:

If = FEASIBLE(tlr:PATH)
Of = Opf(TEST_PATH) <=> b

where Opf = all conjuncts of Op which
relate elements of TEST_PATH
and are not in GENERATE

Since FEASIBLE is a component of the same conditional expression as SOLUTION, the domain remains the same. Since it is also a predicate, the range remains the same.

Df = Y where Rp = <Y>
Rf = B

The complete specification schema is given in Figure 4.

C. THE K QUEENS PROBLEM

Our first example to illustrate the use of this reduction rule will be the K QUEENS problem discussed earlier. The format to be followed in presenting this and later problems will be to represent the problem with the structure in Figure 1, develop a formal specification of the problem, and then apply the reduction rule of the two previous paragraphs. The output will be a program satisfying the problem in the form of the backtrack program schema with formal specifications for the lower level functions.

```
GENERATE SPECIFICATION SCHEMA
  GENERATE:PATH = PATH_LIST  such that
  FEASIBLE:PATH =>
      FOR ALL  TEST_PATH  ELEMENT OF  PATH_LIST
            [length(TEST_PATH) = 1+length(PATH) &
             tlr(TEST_PATH) = PATH              &
             Opg(last(TEST_PATH))]

  where  GENERATE:Y -> Rp  such that Rp = <Y>
    and  Opg = subset of Op such that all conjuncts
               of Op which directly restrict decision
               elements are in Opg

  Heuristic: To identify Op elements for Opg select
             those in which either
         1) a single decision element is restricted
            by constant values  OR
         2) a single decision element is restricted
            by an equality

SOLUTION SPECIFICATION SCHEMA
  SOLUTION:TEST_PATH = b  such that
  b <=> [FEASIBLE(tlr:PATH) => Ops(TEST_PATH)]

  where  SOLUTION:Y -> B  such that  Rp = <y>
    and  Ops = subset of Op such that all conjuncts
               not in Opg are in Ops

FEASIBLE SPECIFICATION SCHEMA
  FEASIBLE:TEST_PATH = b  such that
  b <=> [FEASIBLE(tlr:PATH) => Opf(TEST_PATH)]

  where  SOLUTION:Y -> B  such that  Rp = <Y>
    and  Opf = subset of Op such that all conjuncts
               which relate decision elements and
               not in Opg are in Opf
```

FIGURE 4
Reduction Rule Specification Schemas

1. Problem Representation

The required problem representation was developed in Figure 1 (see page 24).

2. Problem Specification

The components of the formal problem specification may be extracted directly from the problem representation. The domain of the problem is the type of the variable input parameter. For the K QUEENS problem the variable parameter is K, the natural number denoting the size of the chessboard. The domain is thus N, the natural numbers. The range of the problem is the type of the solution structure. For the K QUEENS problem the solution is expressed as a sequence of lists of natural numbers. Each list represents one solution and the sequence lists all solutions. The domain and range specification can thus be specified as:

K_QUEENS:N -> <LIST(N)>

The output condition is derived from the problem constraint structure. It is simply the conjunction of all constraints in the problem representation, formulated in an appropriate logical specification. Using X to represent a solution and PATH_LIST to represent the sequence of all solutions the output condition is:

```
FOR ALL x(i),x(j) IN X, X IN PATH_LIST
        [i≠j => x(i)≠x(j)  &
         i≠j => abs(i-j)≠abs(x(i)-x(j))  &
         1 =< x(i) =< K]
      & length:X = K
```

The input condition is derived from the observation that the program should produce valid output regardless of the value of the input, as long as the input is of the proper type. This type restriction is already provided by the domain

designation. The input condition is thus vacuously true and reduces the truth of the input/output implication to the truth of the output condition. The complete specification is given at Figure 5.

```
K_QUEENS:K = PATH_LIST such that
true => FOR ALL x(i),x(j) IN  X,
                          X IN  PATH_LIST
            [i≠j => x(i)≠x(j)  &
             i≠j => abs(i-j)≠abs(x(i)-x(j))  &
             1 <= x(i) <= K ]
        &  length(X) = K
where  K_QUEENS:N -> <LIST(N)>
```

FIGURE 5
K QUEENS Problem Specification

3. Function Specification

We will now apply our reduction rule to the formal K QUEENS problem specification. The application of the rule will instantiate the specification schemas for the lower level functions and produce a backtrack schema with formal specifications for the lower level functions. Our discussion of the rule application will illustrate the pattern matching process. Any reference to the problem specification within the function specification schemas will cause a search of the problem specification for the desired components. These components will then be inserted into the instantiated function specification. For example, the GENERATE schema specifies the range of GENERATE to be the range of the problem specification. In instantiating the GENERATE specification the range in the problem

specification is extracted and inserted into the function specification. In this manner, all lower level function specifications are produced.

For the K QUEENS problem the specification is listed in Figure 5 and the reduction rule schemas are listed in Figure 4. We begin the rule application by developing the specification of GENERATE. The schema lists the domain as:

Dg = Y where Rp = <Y>

Since the problem specification lists Rp as <LIST(N)>, Y matches LIST(N) and we have Dg = LIST(N). Similarly, the match for Rg produces <LIST(N)> as the range for GENERATE. The schema input condition is listed as true, which requires no match since there is no reference to the problem specification. The output condion references the problem specification only in the conjunct:

Opg(last(x(i)))

where Opg = subset of Op which directly restricts decision

Employing our heuristic for identifying constraints which directly restrict decisions, the constraint:

FOR ALL x(i) IN X
[1 =< x(i) =< K]

meets the first case and is inserted into the GENERATE specification. All components of the specification have now been produced and are included in Figure 6.

Schema instantiation for the SOLUTION function is accomplished with the same procedure. The specification

schema lists the domain of SOLUTION as:

Ds = Y where Rp = Y

The problem specification lists Rp as <LIST(N)>, which allows a match between Y and LIST(N). List(N) is thus identified as the domain. The schema specification lists B as the range, which requires no match with the problem specification. The input condition also remains true, since no problem reference is required. The output condition does reference the problem specification in:

```
Os = Ops(TEST_PATH) <=> b
where Ops = subset of Op  not included
            in Opg
```

We thus extract all conjuncts of the problem output condition not listed in the output condition for GENERATE and place them in the output condition for SOLUTION. These conjuncts are:

```
FOR ALL x(i),x(j) IN X
        [i≠j => x(i)≠x(j)  &
         i≠j => abs(i-j)≠abs(x(i)-x(j))]
    & length:X = K
```

The complete specification for SOLUTION is listed in Figure 5.

The procedure for FEASIBLE is the same. The domain and range schema specifications are the same as for SOLUTION and produce a domain of LIST(N) and a range of B. The input condition remains true. The output condition specification of:

```
Of = Opf(TEST_PATH) <=> b
where Opf = subset of Op which includes all
            conjuncts which relate elements
            of decision and are not in Opg
```

references Op and forces identification of those constraints not in GENERATE which address the decision elements. From the problem specification these are easily identified as:

```
FOR ALL x(i),x(j) IN X
        [i≠j => x(i)≠x(j)  &
         abs(i-j)≠abs(x(i)-x(j))]
```

The complete specification for FEASIBLE is given in Figure 6.

4. Program Generation

To further illustrate the program synthesis process we will develop programs to satisfy the specifications for the lower level functions. The development process will not be detailed but will be only generally described.

Development of the function GENERATE will be discussed first. Satisfaction of this specification can be accomplished by a program which constructs a sequence of lists. Each list is constructed by appending one natural number to the input path. The natural numbers must fall between one and K. A simple program which accomplishes this is:

```
GENERATE:PATH =
    AUX_GENERATE:<PATH, 1>
where
AUX_GENERATE:<PATH, COUNTER> =
    (EQUAL:<COUNTER, K> -> APPENDR:<PATH, COUNTER>;
     APPENDL:[APPENDR:<PATH, COUNTER>,
              AUX_GENERATE:[PATH, ADD:<1, COUNTER>]])
```

```
K_QUEENS:K =
    BACKTRACK:nil

where
BACKTRACK:PATH =
    /APPEND
    (α(lambda<TEST_PATH>
          {(SOLUTION:TEST_PATH -> ID:TEST_PATH;
            FEASIBLE:TEST_PATH -> BACKTRACK:TEST_PATH;
            NIL)})
    (GENERATE:PATH) )

GENERATE:PATH = PATH_LIST  such that
 FEASIBLE:PATH => FOR ALL  TEST_PATH  IN  PATH_LIST
            [length(TEST_PATH) = 1+length(PATH)  &
             tlr(TEST_PATH) = PATH    &
             1 <= last(TEST_PATH) <= K ]

where  GENERATE:LIST(N) -> <LIST(N)>

SOLUTION:TEST_PATH = b  such that
b <=> {FEASIBLE(tlr:TEST_PATH) =>
        FOR ALL x(i),x(j) IN  TEST_PATH
            [i≠j => x(i)≠x(j)  &
             i≠j => abs(i-j)≠abs(x(i)-x(j))]
        &  length(TEST_PATH) = K}

where  SOLUTION:LIST(N) -> B

FEASIBLE:TEST_PATH = b  such that
b <=> {FEASIBLE(tlr:TEST_PATH) =>
        FOR ALL x(i),x(j) IN  TEST_PATH
            [i≠j => x(i)≠x(j)   &
             i≠j => abs(i-j)≠abs(x(i)-x(j))]
where  FEASIBLE:LIST(N) -> B
```

FIGURE 6
K QUEENS Program Specification

The next function to be developed is FEASIBLE. We wish to develop SOLUTION after FEASIBLE since SOLUTION properly includes all the constraints in FEASIBLE. This will allow inclusion of FEASIBLE within SOLUTION. FEASIBLE is expressed as a conjunction of two constraints. This

translates into the AND of two computable boolean expressions. The first expression compares the values of all elements in the parameter. The input condition tells us that all elements in tlr:PATH meet this condition. Therefore we only need compare the last element with the rest of the elements. The second expression compares the absolute values of the difference of the row positions and the difference of the column positions. Since we know that this condition holds for all elements of PATH except for the last, we only need test the last element. This gives us the program:

```
FEASIBLE:PATH =
     AND:[ROW_MATCH:PATH,
          DIAG_MATCH:PATH]
where
ROW_MATCH:PATH =
     (NULL:PATH -> true;
      AND:[NEQUALS:[LAST:PATH, 1:PATH],
           ROW_MATCH(TL:PATH)])
DIAG_MATCH:PATH =
     (NULL:PATH -> true;
     AND:[NEQUALS[ABS(-:[TL:PATH, 1:PATH]),
                  ABS(-:[LENGTH:PATH, 1])],
          DIAG_MATCH(TL:PATH)])
```

The derivation of the function SOLUTION is now simple. SOLUTION contains the conjunction of three constraints. Two of these are included in FEASIBLE. We can include FEASIBLE and the final constraint in an AND function to complete this program. This gives us:

```
SOLUTION:PATH =
    AND:[FEASIBLE:PATH,
         EQUALS:[K, LENGTH:PATH]]
```

After derivation of these programs the synthesis system

would replace the specifications of Figure 6 with these programs and the process would be complete.

D. THE PROCESSOR SEQUENCING PROBLEM

The Processor Sequencing Problem is a known NP complete problem (Ref. 20). It differs from the K QUEENS problem in that the path elements under examination at any stage of the process have a number of associated properties and the constraint relationships are expressed predominantly in terms of these properties. The solution to this problem will illustrate the use of global data in backtracking algorithms and the incorporation of constraints into the function GENERATE.

1. Problem Representation

The Processor Sequencing Problem (PSP) may be simply stated: Given a set of tasks to be run on a single processor, with each task having an associated release time, processing time and deadline, does there exist a scheduling sequence which will complete all tasks prior to their deadline? The associated properties place a series of constraints on the tasks. The release time is an earliest possible availability constraint. No task is available to run before its release time. Once selected for execution, each task will consume exactly the amount of time specified by its processing time. The deadline places a latest completion constraint on each task.

The first task in representing this problem is to decide on a decision structure. One obvious component of a decision is which task to run next. But this is not complete in that more information is required about scheduling this task than mere selection provides. The time the task is scheduled to run is also a crucial part of the decision. This time is not fixed based on the previous decisions in the partial solution vector, but depends on additional information. For this reason, the decisions made for this problem can be represented by a pair, the first element being the task selected and the second element being the start time of the task.

The second representation task is to transform the decision structure into a solution structure. The solution structure will consist of a sequence of decisions, each decision being of the form specified by the decision structure. Thus a solution will have the form:

X = (x(1), x(2), ... , x(N))

where each x(i) is of form
(task(i), time(i))

The final representation task concerns the problem constraints. A number of constraints relate the elements of the possible solutions. The first we will consider is the deadline restriction imposed on each task. Whether a task meets its deadline depends on two factors: the task start time and the task processing time. The start time is an element of the decision being tested. The processing time

and deadline time are constant values associated with each instance of the problem. A task meets its deadline if the sum of the start time and the processing time is less than the deadline. This can be expressed in computable form as:

```
FOR ALL x(i) IN X
        [deadline(task(i)) >= start(i) + time(task(i))]
```

where deadline, time are problem constants

Another solution element constraint is identified by the fact that no task may be scheduled twice. Thus each task in the sequence must be distinct. We can represent this by noting that if the position of two tasks in the sequence are distinct, then the tasks must also be distinct. This can be expressed as:

```
FOR ALL x(i),x(j) IN X
        [ i≠j => task(i)≠task(j) ]
```

There are also constraints on the start time of each task. These limit the start time to a point after both the completion time of the previous task and the release time of the task under consideration. It follows that the start time may be expressed as the maximum of the two constraints. Assuming a language function to select the maximum of two natural numbers, this constraint may be expressed as:

```
FOR ALL x(i) IN X
        [start(i)=max(release(task(i)),
                      start(i-1)+process(task(i-1)) )]
```

where release, process are problem constants

The final constraint identifies a solution from potential

solutions which meet all other constraints. If all other constraints are met and the number of elements of the proposed solution equals the number of tasks, then we know that all tasks are included in the sequence. This final constraint can be identified as our solution constraint and is expressed as:

LENGTH:X = K

The complete problem representation is given in Figure 7.

```
DECISION STRUCTURE
  decision(i) = ith task to run
                start time of task

SOLUTION STRUCTURE
  <X>  where each X = (x(1), x(2), ... ,x(K))
       where x(i) = (task(i), start(i))

CONSTRAINT STRUCTURE
  element constraints
    FOR ALL x(i),x(j) IN X
        [ i≠j => task(i)≠task(j)]
    FOR ALL x(i) IN X
        [ deadline(task(i)) >=
                  start(i) + process(task(i)) ]
    FOR ALL x(i) IN X
        [ start(i) = max(release(task(i)),
                         start(i-1)+process(task(i-1))) ]
  solution constraint
    length(X) = K

where release, process, deadline are problem constants
```

FIGURE 7
PSP Problem Representation

2. Problem Specification

As with the K QUEENS problem, the four components of the PSP formal problem specification can be easily derived from the problem representation. The domain for the PSP

problem is the type of the variable input. In this case, the variable input is the number of tasks, K, a natural number. The solution structure should provide us the range. In this case, a single solution is structured as a list of pairs of natural numbers. The first element of the pair is a task identifier and the second element is a start time for that task. Since the problem requires a sequence of all solutions, the proper range is a sequence of lists. We can express this as:

PSP:N -> <LIST(NxN)>

The problem output condition is immediately derived from the constraint structure. It is merely the conjunction of all constraints we have identified. The expression of this output condition is more complex than for the K QUEENS problem because the constraints rely on constant values defined by the problem instance. Our notation for declaring these constant values will be the where declaration of our programming language. This declaration in effect defines a scope of visibility for the constants, making them known to the constraints. The problem output condition is:

```
FOR ALL x(i),x(j) IN X
        [i≠j => task(i)≠task(j)  &
         deadline(task(i)) >=
                  start(i)+process(task(i))  &
         start(i)=max(release(task(i)),
                      start(i-1)+process(task(i-1)) )]
    & length:X=K
```

where release, process, deadline are program constants

For reasons the same as with the K QUEENS problem the input

condition is vacuously true. The complete specification is given in Figure 8.

```
PSP:K = TASK_LIST such that
true  => FOR ALL x(i),x(j) IN X, X IN TASK_LIST,
                and x(i) = (task(i),start(i))
       [i≠j => task(i)≠task(j)  &
        deadline(task(i) >=
                start(i)+process(task(i))  &
        start(i) = max(release(task(i)),
                       start(i-1)+process(task(i-1)))  ]
     & length(X) = K

where  PSP:N -> <LIST(NxN)>
  and  release, process, deadline are problem inputs
```

FIGURE 8
PSP Problem Secification

3. Function Specification

We now apply our reduction rule to produce a backtrack program with formal specifications for the lower level functions which will solve the PSP problem. To do so we use the schemas of Figure 4 and the formal problem specification of Figure 8. We begin with the specification of the function GENERATE. The specification schema lists the domain as Y, where Rp = <Y>. Matching this against the problem specification provides Rp as <LIST(NxN)> which gives Y as LIST(NxN). This is placed as the domain of GENERATE. The schema lists the range as Rp so we have:

GENERATE:LIST(NxN) -> <LIST(NxN)>

The schema input condition does not reference the problem specification so it is copied into the GENERATE specification. In a like manner the first two conjuncts of

the schema output condition are copied into the GENERATE specification. The final conjunct references Opg, the subset of the problem output conditions which directly restricts a decision element. Examining the problem specification under the guidance of our heuristic produces a match with the conjunct:

```
FOR ALL x(i) IN X, X IN TASK_LIST
            and x(i) = (task(i), start(i))
      [start(i) = max(release(task(i),
                      start(i-1)+process(task(i-1))) ]
```

and case two of the rule. Case two prescribes the inclusion of problem constraints which in the GENERATE function if a constraint restricts a single decision element by an equality. In this case start(i) is the decision element and it is restricted by an equality. Case one produces no match since no constraint bounds a decision element by constant values. The schema entry for Opg is replaced by the conjunct above producing the full specification of Figure 9.

The same procedure is used to develop the formal specification for SOLUTION. The schema specifies the domain as Y where the problem range is <Y>. The problem range is <LIST(NxN)>, which produces a Y match of LIST(NxN), which we take as the domain of SOLUTION. The schema range does not reference the problem specification, so it is copied into the function specification. The same is done for the function input condition. The schema output condition references Ops, the subset of the problem output condition which is not included in Opg. From the discussion in the

last paragraph this reduces to the first, second and fourth conjuncts in the problem output condition. Replacing Ops with these conjuncts produces the specification of Figure 9.

The specification for SOLUTION is identical to FEASIBLE, as shown in the schemas, with the exception of the output condition. In this case, the reference to Opr in the schema must be replaced by all conjuncts of the problem output condition which relate decision elements and are not in Opg. With this problem the last conjunct does not relate decision elements since it addresses the solution as a whole. The third construct is included in Opg. This leaves the first two constraints to be substituted for Opr. Placing these constraints into the schema produces the specification of Figure 9.

E. SCHEMA LIMITATIONS

The reduction rule developed in this chapter has a number of limitations. The principal deficiency is that it is heuristic in nature and not an algorithm. The underlying reason for this is the failure of the rule to incorporate any proof mechanism in its actions. It is believed that a proof mechanism may be constructed based on the design method developed above. Reduction rules for the simple divide and conquer control strategy have been developed by Smith [Ref. 21] which employ a proven theorem as the basis for specification development.

```
PSP:K =
  BACKTRACK:nil

where
BACKTRACK:PATH =
    /APPEND
    (α(lambda<TEST_PATH>
          {(SOLUTION:TEST_PATH -> ID:TEST_PATH;
            FEASIBLE:TEST_PATH -> BACKTRACK:TEST_PATH;
            NIL)})
    (GENERATE:PATH) )

GENERATE:PATH = PATH_LIST  such that
 FEASIBLE:PATH => FOR ALL x(i),x(j) IN X,
                          X  IN  PATH_LIST
                          and x(i) = (task(i),start(i))
          [length(X) = 1+length(PATH)  &
           tlr(X) = PATH   &
           start(i) = max(release(task(i)),
                          start(i-1)+process(task(i-1)))]

where  GENERATE:LIST(NxN) -> <LIST(NxN)>

SOLUTION:TEST_PATH = b  such that
b <=> {FEASIBLE(tlr:TEST_PATH) =>
            FOR ALL x(i),x(j) IN  TEST_PATH
                    where x(i) = (task(i), start(i))
            [i≠j => task(i)≠task(j)  &
             deadline(task(i)) >=
                    start(i)+process(task(i))]
            &  length(TEST_PATH) = K}

where SOLUTION:LIST(NxN) -> B

FEASIBLE:TEST_PATH = b  such that
b <=> {FEASIBLE(tlr:TEST_PATH =>
            FOR ALL x(i),x(j) IN  TEST_PATH
                      where x(i) = (task(i), start(i))
            [i≠j => task(i)≠task(j)  &
             deadline(task(i)) >=
                    start(i)+process(task(i))]}

where FEASIBLE:LIST(NxN) -> B
where release, deadline, process are program constants
```

FIGURE 9
PSP Program Specification

A second limitation of the rule is the inefficiency inherent in the backtrack schema. As evidenced by our examples there is much duplicate computation between the SOLUTION and FEASIBLE predicates. This could indicate that efficiency is better served by evaluating the FEASIBLE predicate first and then nesting a diminished form of the SOLUTION predicate within the action clause of FEASIBLE. Although our design method would allow this, it restricts the schema to problems where the FEASIBLE constraint includes only restrictions within SOLUTION as well. It is not known whether this is a general condition with problems suitable for the backtrack solution technique and the more general schema of Figure 3 was developed instead.

A general efficiency concern in the development of any backtrack algorithm is the proper subdivision of constraints between GENERATE and the other functions. Obviously, any constraint within GENERATE filters nonfeasible partial solutions from SOLUTION and FEASIBLE. How much total computation is saved is not clear, however. The total number of nodes examined by the predicates is less when more of the constraints are included within GENERATE, but the computation required by GENERATE is greater. A general conclusion that seems valid is that some work is saved if there is also duplicate computation, as discussed above, between SOLUTION and FEASIBLE, but if there is no duplicate computation, then each extension at each level visited is

tested once against each constraint. A more favorable area for related investigation is in program transformation. This work may identify when backtrack programs produce duplicate computation, and transform such programs to eliminate the duplication.

V. AN EXTENSION TO BACKTRACK

The backtrack algorithm has traditionally been employed to solve problems of the type described in Chapter III. Research on the strategy has been oriented towards efficiency improving techniques, [Ref. 22, 23] program proving [Ref. 24], problem representation formalisms [Ref. 25] and control structure abstraction [Ref. 26, 27]. The problem of extending the strategy for solution of a different class of problems has not been significantly addressed. The second reduction rule proposed by this paper extends the backtrack strategy by adapting it for solution of the problem reduction problem type. The result is a general purpose schema with a heuristic design method for lower level function specification. As this result is less rooted in existing knowledge, the design method presented will be described in general terms.

A. PROBLEM REDUCTION PROBLEM REPRESENTATION

A problem reduction problem representation is another formalism for symbolic problem description. As with the state space representation discussed in Chapter III, representation of a problem with a problem reduction format will impose a particular graphic structure onto the problem. With this structure we can employ a graph search procedure to search for a solution. The goal in this

chapter is to adapt the backtrack strategy to search the problem reduction graph. In this paragraph we will first develop the representation, then depict the graph structure produced by the representation, and then illustrate a sample problem representation.

1. Problem Representation

There are three key components of a problem reduction representation. The first component is the problem state. This is a symbolic description of the state of the problem at any point in the search process. The initial problem state is a description of a goal which is to be satisfied. As the search process executes, the initial goal state will be decomposed into one or more subgoal states, which, when both are satisfied, will cause the original goal to be satisfied. An example of this is the symbolic integration process. Given a goal state of the form:

$\int(f(x) + g(x))\,dx$
where f,g are known functions

A solution to this problem is a symbolic representation of the integral. An initial decomposition may produce the two subgoals:

$\int f(x)\,dx$
$\int g(x)\,dx$
where f,g are known functions

Solving both of these two subgoals will lead to the solution of the original problem. In this case, the two subsolutions must be added.

In order to decompose states and compose solutions some means must be provided for these actions. The second component of a problem reduction representation is a set of reduction rules. Each rule will act on a goal description and provide one or more decomposed subgoals. The rule also provides a method for combining solutions to subgoals into solutions to the original goal. The most significant aspect of rule application is that all subgoals must be solved for the original goal to be solved. In our symbolic integration example the reduction rule applied may be of the form:

```
If Integrand is form f(x) + g(x)
   where x is variable of integration
then
   solve f(x) and g(x)
   compose solutions with +
```

It is important to note that there is an applicability condition (If) and a conjunctive solution.

The representation we have described thus far allows only goal decomposition. The third component of the representation allows for a solution of a subset of goals we will call primitive. This component is a set of rules, also called primitive, which, when applied to a primitive goal will return a solution. In our symbolic integration example, one primitive rule may be:

```
If Integrand is of form cos x
   where x is variable of integration
then return sin x
```

The primitive operators provide the only means of finding a solution in a problem reduction representation. They are a

means to represent those goals which we know how to directly solve.

2. And/Or Tree Representation

The graph structure imposed by this representation is similar to the structure of a state space tree, but contains an additional node type. We will represent goal descriptions by nodes and rule applications by arcs. The path from the root of a tree to a subgoal description describes the sequence of rule applications which produced the goal description. Given a node (goal description) there is a range of reduction rules which may be applied. This range is represented by the set of arcs leaving the node. The complicating factor of the problem reduction representation lies in reduction rules which decompose a goal description into two or more subgoals. The relationship between these subgoals is tightly constrained, representing the fact that both of these subgoals must be solved to solve the goal. This logical AND relationship contrasts with the subgoals produced by the other reduction rules. Satisfaction of the subgoals produced by any reduction rule will satisfy the goal. The graphic solution is to tie together the arcs representing application of one rule with a hyperarc. This creates an AND node, which signifies that all descendants of the AND node must be satisfied. The application of distinct rules are represented by OR nodes, or arcs not connected by hyperarcs.

which represent the logical OR nature of their relationship. Figure 10 depicts a sample search space. Given an initial goal represented as node A, three reduction rules can be applied. Rule 1 produces subgoals B and C, rule 2 produces subgoal D and rule 3 produces subgoals E and F. A can be solved by solving either set of goals, B and C, or D, or E and F. Ultimately, if B and C are to be solved then G or H, and I must be solved. If D is to be solved, then J and K must be solved. If E and F are to be solved, then L and M and N must be solved. To solve this problem the search process must search for subgoals which can be solved by primitive operators and tie together the separate paths represented by and nodes. Unlike the state space search, the result of this search process will be a solution tree. From any node the separate branches represent the different subgoals produced by a single rule application. As an example, Figure 11 depicts the four solution trees present in Figure 10 if all leaf nodes can be solved.

3. An Example Representation

The example we present here and develop for the remainder of the chapter is a simple arithmetic theorem prover. Given a goal statement in terms of an arithmetic assertion in any number of variables, and a number of propositions about those variables we know to be true, can we prove the statement is true. In this paragraph we will develop a problem reduction representation for the problem,

and in later paragraphs we will adapt the backtrack control strategy to search the representation defined AND/OR tree and return a proof of the assertion.

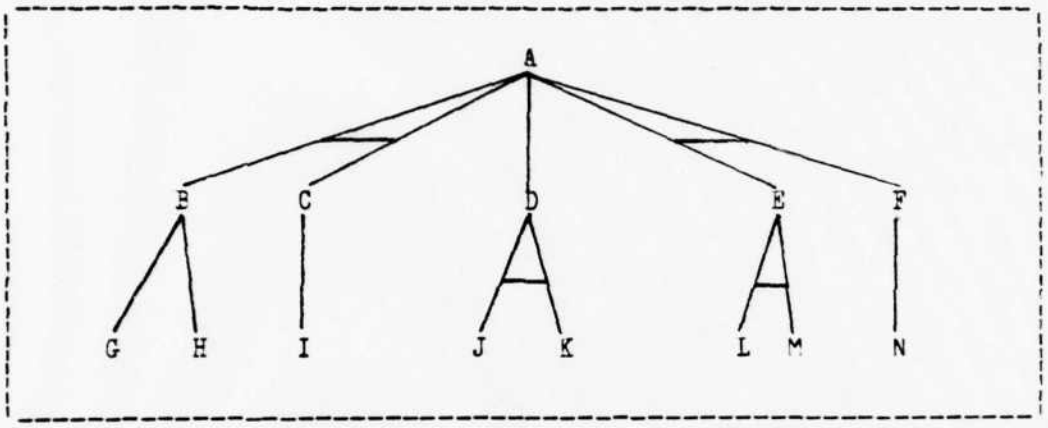


FIGURE 10
AND/OR Graph

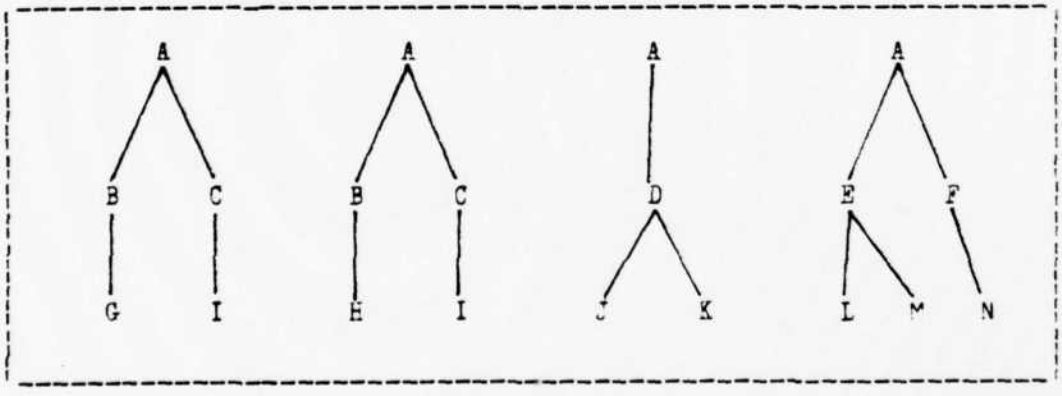


FIGURE 11
Solution Graphs

To represent our problem with a problem reduction formalism we need to define the three components of the representation. The first component is the goal description. The initial goal is an arithmetic assertion.

A suitable goal description is the assertion itself. The result of applying a reduction rule will be one or more subgoals, each of which should be a simpler arithmetic assertion. For our problem representation we can express this as:

GOAL DESCRIPTION
form: arithmetic assertion
initial: [B * (A + C)]/E > B

The initial goal description represents the particular problem instance to solve.

The next component we describe is the set of primitive rules. These rules need to be described before the reduction rules because they provide the basis towards which the reduction rules should simplify the goal. Primitive rules represent the knowledge possessed about the problem. They specifically apply to goal descriptions that can be directly solved. In the theorem prover, these rules are expressions of the propositions which are known to be true. For the problem instance we are concerned with these are:

PRIMITIVE RULES
A > 0
B > 0
C > 0
E > 0
C > E

To complete our problem representation we need only specify the reduction rules. The purpose of a reduction rule is to simplify a goal state which cannot be directly solved by a primitive rule. It follows that reduction rules

embody general knowledge about problem area relationships which allow transformation of goal descriptions into one or more simpler descriptions. In simple theorem proving these relationships can be described with logical implications which represent general known theorems. They can be stated in the form:

P1 & P2 & ... & PK => P0

where P0 represents a goal and P1, ... ,PK represent subgoals. If P0 can be matched against a goal description, then subgoal P1 ... PK will be produced. We will use four reduction operators for the theorem prover:

```
REDUCTION OPERATORS
  x>0 & y>0 => x*y > 0
  x>0 & y>z => x+y > z
  x>0 & y>z => x*y > x*z
  x>z*y & y>0 => x/y > z
```

The complete problem representation is given in Figure 12.

```
GOAL DESCRIPTION
  form: arithmetic expression
  initial: [3 * (A + C)] / E > B

PRIMITIVE RULES
  A > 0          B > 0
  C > 0          E > 0
  C > E

REDUCTION RULES
  x>0 & y>0 => x*y > 0
  x>0 & y>z => x+y > z
  x>0 & y>z => x*y > x*z
  x > z*y & y>0 => x/y > z
```

FIGURE 12
Theorem Prover Problem Representation

B. SCHEMA DEVELOPMENT

In developing the backtrack schema for a problem reduction representation the procedure described in Chapter IV will again be followed. This procedure requires description of the expected input, description of the desired output, identification of the operations required to transform the input to the output and then translation of the operations into lower level functions and appropriate functional forms.

1. The Expected Input

As in the state space backtrack schema a representation of a path is expected as input. This path is a symbolic description of the sequence of rule applications which have reduced the initial goal description to the current goal description. Since the path does not include the current goal description, this must also be included in the expected input. The resulting input is a sequence consisting of a path and a symbolic representation of the current goal.

The relevant characteristics of the input are two. The first is that all rules in the path have been successfully applied. The second is that this current goal may be primitive. This situation is a result of the backtrack strategy applying the SOLUTION predicate before the FEASIBLE predicate. This will be further discussed in the section on input transformations.

2. Desired Output

The output desired from a problem reduction representation is often dependent on the problem. For example, the desired output for the symbolic integration problem is a symbolic description of the integral. With the simple theorem prover we desire proof of the input assertion. A commonality between these and all problem reduction representations is the sequence of operations performed to arrive at a solution. For this reason the general output desired will be a solution graph consisting of the reduction rules and primitive rules applied to solve the problem. The return from this most general case can be transformed into the desired output form.

3. Input Transformations

In describing the input transformations required we will stay as close as possible to the simple backtrack schema developed in Chapter IV. The goal is to produce a schema which can be applied to either the state space representation or the problem reduction representation. The design method will differ based on the problem representation. To do so we will identify those aspects of the simple backtrack schema which require enhancement to search an AND/OR graph, and develop those enhancements in either the schema or the design method.

The initial transformation required is to extend the path parameter. In this case, the extension consists of

appending one more reduction rule to the path of rules previously applied. This extension does not apply the rule, but lists it as one possible alternative. The result of this transformation will be a new sequence of path, state pairs. Each pair represents a different alternative extension to the path of applied rules.

The second transformation is the conditional test. The SOLUTION predicate will again be executed first. In a problem reduction representation a solution is not recognized by examining the sequence of decisions (rule applications), but by examining the current goal description. Upon recognition of a solution, the action is to return the sequence of rules, and not the goal description. If the SOLUTION predicate fails then the FEASIBLE predicate will be executed. This predicate is a test of the path to determine if a solution can feasibly be discovered through expansion of the path. The clearest way to test this in a problem reduction representation is to test the reduction rule appended by the path expansion transformation. If this rule can be applied to the goal, then further subgoals can be produced which may lead to solutions. If the rule can be applied then the appropriate actions are more complex than those in the state space schema. The obvious first action is to apply the rule and produce new subgoals. If only one subgoal is produced we have created an OR node. In this case the appropriate

action is to recursively call the backtrack function with the new subgoal and path. If more than one subgoal is produced an AND node has been created and more complex action is required. If an AND node is created then a separate path is created for each descendant of the node. To solve the AND node each path must return a solution. To solve this problem by a backtrack search we must search each path and compose the solutions. If any path returns nil, the result of the node will be nil. The order of transformations on AND node is thus to apply the rule, create separate <PATH, GOAL> pairs for each subgoal, backtrack on each pair and finally compose solutions.

The final transformation is to filter the nil solutions returned by the examinations of the expansions. The value returned will consist of a list of solutions.

4. Schema Translation

To derive a schema from the required transformations we will again group the transformations into lower level functions combined with the appropriate functional forms. The first transformation is the generation of expanded paths. This transformation can again be accomplished by a single function GENERATE. In our language notation this is:

GENERATE:<PATH, GOAL>

Where the parameter PATH is a representation of the sequence of rules applied, and the parameter GOAL is a description of the current goal.

The second transformation is the conditional testing function. As with the state space representation, this function is applied to one element of the output of GENERATE and the results are returned before it is applied to the next element. This APPLY-TO-ALL operation is:

```
∝TEST (GENERATE:<PATH, GOAL>)
```

We can expand the function TEST since we know the actions required of it. The first predicate tests the goal for being primitive. If it is, the action is to return the path. This can be expressed as:

```
SOLUTION:GOAL -> ID:PATH;
```

The second predicate is a test for feasibility of expansion. The corresponding action is to apply the rule, decompose the path, subgoals pair into separate path, subgoal pairs, apply backtrack to each pair and finally compose the results. This can be expressed as:

```
FEASIBLE:<PATH, GOAL> ->
        COMPOSE( BACKTRACK(DECOMPOSE:<PATH, GOAL>));
```

Using the lambda definition of our language we now have:

```
(lambda <PATH>
      {(SOLUTION:GOAL -> ID:PATH;
        FEASIBLE: <PATH, GOAL> ->
            COMPOSE( BACKTRACK(DECOMPOSE:<PATH, GOAL>));
        NIL)})
(GENERATE:<PATH, GOAL>)
```

The final transformation filters the nil values returned by the process. This can be expressed as inserting the APPEND function through the list of solutions returned. The complete schema is given in Figure 13.

```
BACKTRACK:<PATH, GOAL> =
  /APPEND
  (α(lambda<PATH>
       {(SOLUTION:GOAL -> ID:PATH;
         FEASIBLE:<PATH,GOAL> ->
            COMPOSE( BACKTRACK(DECOMPOSE:<PATH,GOAL>));
         NIL)})
      (GENERATE:<PATH,GOAL>) )
```

FIGURE 13
Backtrack Program Schema

C. SUBSCHEMA SPECIFICATION

The schema developed above provides a structure for composing the solutions to the subproblems GENERATE, SOLUTION, FEASIBLE, DECOMPOSE and COMPOSE. A design method for specifying these subproblems is also required. This paragraph discusses the relationships between the functions the schema requires to solve a problem. A detailed design method similar to the method presented in Chapter IV is not developed, but left for further research.

1. GENERATE Specification

The function GENERATE must accept an input pair which represents the path of reduction rules applied and the current goal specification. The output must be a sequence of pairs. Each pair contains a new path representation and the goal specification. The new path is the input path to which one reduction rule of those available to apply has been appended. The sequence contains a separate pair for each available rule. As an example, if GENERATE is given

the input:

<(R1, R3, R2), GOAL>

and there are four reduction rules R1...R4 available to apply then GENERATE must output:

<<(R1, R3, R2, R1), GOAL>,
<(R1, R3, R2, R2), GOAL>,
<(R1, R3, R2, R3), GOAL>,
<(R1, R3, R2, R4), GOAL>>

This output represents all possible expansions of the rule application path. It does not represent all expansions with applicable rules. A different function of the schema will delete nonapplicable rules.

2. SOLUTION Specification

The function SOLUTION will again be the easiest to describe. The intent of SOLUTION is to test a goal description to see if it is a solution. In the problem reduction representation there is one operation to test for a solution. If the goal description can be solved by a primitive rule then a solution has been found. The specification of SOLUTION must express this relationship between the goal and the primitive rules. The form of the relationship may differ from problem to problem. For example, in the symbolic integration problem the relationship is an application. If a primitive rule can be applied to the goal, then it can be solved. In the theorem proving problem the relationship is membership. If the goal is a member of the primitive rules then the goal is solved.

3. FEASIBLE Specification

The function of the predicate FEASIBLE is to test a path for the possibility it may lead to a solution. The path under consideration exhibits two characteristics. The last element of the path is a reduction rule which has not been applied to the goal description. The remainder of the path is a sequence of reduction rules which have been applied to the initial goal description to produce the current description. If the path under consideration is to be considered feasible, then the last rule of the path must be applicable to the goal description. In more concrete terms, a reduction rule applies to a goal if the rule produces one or more subgoals from the goal. The FEASIBLE predicate must test this relationship between the goal and the reduction rule. It is significant that the FEASIBLE function does not actually apply the rule to produce subgoals. It need only ensure the rule can be applied.

4. DECOMPOSE Specification

If the path is feasible (the rule can be applied), the next step is to produce a <PATH, GOAL> pair. This pair will be the input to the recursive call to BACKTRACK. In many instances the application of a reduction rule will produce more than one subgoal. For this reason the function DECOMPOSE must do more than apply the rule to prepare input for BACKTRACK. For every subgoal produced by the rule application, DECOMPOSE must construct a <PATH, GOAL> pair.

The path in all pairs is identical: the sequence of rule applications which produced the associated goal description. The goal description in each pair is unique: it describes one of the subgoals produced by the rule application. There are two important characteristics of the output of this function. If the rule application produces only one subgoal, then there is only one <PATH, GOAL> pair produced, and the APPLY-TO-ALL functional form of the schema reduces to a straight application of BACKTRACK to the pair. This operation is similar to the state space backtrack schema. Secondly, DECOMPOSE has transformed a <PATH, GOAL> pair where the path description contains a nonapplied rule (the final rule) to a pair with all rules applied and a new goal description. This is the type of input expected by the function BACKTRACK.

5. COMPOSE Specification

BACKTRACK is applied to each <PATH, GOAL> pair produced by DECOMPOSE. The result returned by the application is a sequence of paths. Each path represents a sequence of rules applied to the goal description which terminated in a solution. For goal descriptions which could not be reduced to a primitive goal the algorithm returns the nil path. If the nil path is found in the sequence of paths returned it signifies that one of the subgoals produced by the reduction rule is not solvable. From our discussion of the problem reduction formalism, this means that the goal is

not solvable with this reduction since all subgoals produced must be solved to solve the goal. The inference is that the sequence of reduction rules which produced the subgoals does not lead to a solution and the nil path must be returned to indicate such. If no nil path is returned in the sequence then all subgoals were solved and the sequence is returned.

Figure 14 depicts the requirements of the lower level functions of the problem reduction backtrack schema.

D. A SIMPLE ARITHMETIC THEOREM PROVER

Our example to illustrate this reduction rule is the simple theorem prover developed throughout this chapter. A formal problem specification will not be developed as the informal description of the reduction is not detailed enough to exploit the formalism of a specification. Instead, this paragraph will develop informal specifications based on the problem representation of Figure 12 and the function requirements of Figure 14.

1. GENERATE Specification

Our problem representation lists a set of four reduction rules, R1...R4. GENERATE must produce a sequence of four <PATH, GOAL> pairs. Each PATH will terminate with a different reduction rule. We can express this in our informal notation as:

```
GENERATE:<PATH, GOAL> = <<PATH(1), GOAL>,<PATH(2), GOAL>,
                        <PATH(3), GOAL>,<PATH(4), GOAL>>
such that
     TLR:PATH(i) = PATH &
     TL:PATH(i) = RULE(i)
```

This will provide the desired input to the conditional test.

```
GENERATE REQUIREMENTS
  GENERATE:<PATH, GOAL> = NEW_STATE  such that
       NEW_STATE = {<NEW_PATH, GOAL>|
         NEW_PATH = APPENDR:<PATH, RULE> for each RULE}

SOLUTION REQUIREMENTS
  SOLUTION:GOAL = boolean   such that
       boolean <=> R1:<PRIMITIVES, GOAL>
       where  R1 is a problem dependent relation

FEASIBLE REQUIREMENTS
  FEASIBLE:<PATH, GOAL> = boolean   such that
       boolean <=> R2:[tlr:PATH, GOAL]
       where R2 is a problem dependent relation

DECOMPOSE REQUIREMENTS
  DECOMPOSE:<PATH, GOAL> =
       <<PATH, NEW_GOAL(1)>,...,<PATH, NEW_GOAL(N)>>
  such that
       [N = number conjuncts in precondition TL:PATH] &
        R2:<CONJUNCT(i), NEW_GOAL(i)> =
           R2:[tlr:PATH, GOAL]  ]

COMPOSE REQUIREMENTS
  COMPOSE:SOLUTION_SEQUENCE = SOLUTIONS  such that
      [MEMBER:<NIL, SOLUTION_SEQUENCE> =>
                        SOLUTIONS = NIL] &
      [NOT(MEMBER<NIL, SOLUTION_SEQUENCE>) =>
                        SOLUTIONS = SOLUTIONS_SEQUENCE]
```

FIGURE 14
Reduction Rule Requirements

2. SOLUTION Specification

The major difficulty in specifying the SOLUTION function is in determining the appropriate relation between the primitive rules and the goal description. In the theorem prover problem we attempt to reduce the initial goal

descriptions to descriptions which are known to be true. The descriptions known to be true are the problem propositions, which the problem representation designates as primitive rules. The problem, therefore, is to find goal descriptions which are in the set of primitive rules. The appropriate relation is a membership test and we can express this as:

```
SOLUTION:GOAL = boolean such that
    boolean <=> [MEMBER:GOAL, PRIMITIVES>]
```

3. FEASIBLE Specification

The FEASIBLE predicate is a test of the applicability of the final rule of the path to the current goal description. The specification question is to determine what relation tests this applicability. In this problem a goal description is given in terms of constants, literals and arithmetic operators. The rules are expressed in terms of variables, constants and operators. An appropriate applicability test is a pattern match between the conclusion of the rule and the goal description. This test can match any subexpression or literal of the goal against the rule variables, but the constants and operators must be exact matches. If a match is found the rule can be applied to the goal. This can be expressed as:

```
FEASIBLE:<PATH, GOAL> = boolean such that
    boolean <=> MATCH:[TL:PATH,GOAL]
```

4. DECOMPOSE Specification

After the predicate FEASIBLE has determined that the rule applies to the goal description, DECOMPOSE must apply the rule and construct a sequence of <PATH, GOAL> pairs for input to BACKTRACK. To determine subgoals we note that the precondition of the reduction rule lists the form of the subgoals to be produced. The difficulty in creating subgoals is in replacing the rule variables with the problem literals and subexpressions. We can identify the appropriate literals with a matching process identical to that conducted by the feasibility test. For each subgoal produced DECOMPOSE must then create a <PATH, GOAL> pair. We can express these requirements as:

```
DECOMPOSE:<PATH, GOAL> =
  <<PATH, NEW_GOAL(1)>,...,<PATH, NEW_GOAL(N)>> such that
  N = number conjuncts in precondition of TL:PATH &
  FOR EACH  CONJUNCT(i)  IN   tlr:PATH
  MATCH:<CONJUNCT(i), NEW_GOAL(i)> =
       MATCH:[tlr:PATH, GOAL
```

5. COMPOSE Specification

The final function to specify is COMPOSE. This is also the simplest to specify. COMPOSE must return nil if nil is a member of the parameter sequence. If nil is not a member of the sequence then the sequence is to be returned. This can be expressed as:

```
COMPOSE:SOLUTION_SEQUENCE = RETURN_SEQUENCE  such that
  [MEMBER:<NIL, SOLUTION_SEQUENCE> =>
                RETURN_SEQUENCE = NIL] &
  [NOT(MEMBER:<NIL, SOLUTION_SEQUENCE>) =>
                RETURN_SEQUENCE = SOLUTION_SEQUENCE]
```

The complete informal specification for the functions is given at FIGURE 15.

```
PROOF:<GOAL, PRIMITIVES, RULES> =
      BACKTRACK:<NIL, GOAL>
where

BACKTRACK:<PATH, GOAL> =
  /APPEND
  (α(lambda<PATH>
       {(SOLUTION:GOAL -> ID:PATH;
         FEASIBLE:<PATH,GOAL> ->
            COMPOSE( BACKTRACK(DECOMPOSE:<PATH,GOAL>));
         NIL)})
    (GENERATE:<PATH, GOAL>))

GENERATE:<PATH, GOAL> =
              <<PATH(1), GOAL>, ... ,<PATH(4), GOAL>>
such that  FOR ALL PATH(i)
           [TLR:PATH(i) = PATH    &
            TL:PATH(i)  = RULE(i)]

SOLUTION:GOAL = boolean     such that
    boolean <=> MEMBER:<GOAL, PRIMITIVES>

FEASIBLE:<PATH, GOAL> = boolean     such that
    boolean <=> MATCH:[TL:PATH, GOAL]

DECOMPOSE:<PATH, GOAL> =
      <<PATH, NEW_GOAL(1)>,...,<PATH, NEW_GOAL(N)>>
such that
     [N = number conjuncts in precondition tl:PATH    &
      FOR EACH  CONJUNCT(i)  IN  tlr:PATH
      MATCH:<CONJUNCT(i), NEW_GOAL(i)>  =
           MATCH:[tlr:PATH, GOAL]

COMPOSE:SOLUTION_SEQUENCE = RETURN     such that
     [MEMBER:<NIL, SOLUTION_SEQUENCE> =>
                   RETURN = NIL     &
      NOT(MEMBER:<NIL, SOLUTION_SEQUENCE>) =>
                   RETURN = SOLUTION_SEQUENCE]
```

FIGURE 15
Theorem Prover Program Specification

VI. CONCLUSION

The success of future efforts in program synthesis will depend in large part on the ability of system developers to codify expert knowledge about the programming process. As synthesis systems become more complex and attempt to solve more difficult problems the search space created in the solution process suffers the effects of combinatorial explosion. As the search space grows the search strategy must become more efficient. The larger the space the closer the search process must approximate a straight line search. The ability to execute a straight line search is a function of the knowledge the search strategy employs to solve the problem. The better the knowledge, the fewer incorrect alternatives will be explored.

The principal goal of this paper is the development of a reduction rule for a synthesis system based on the problem reduction representation formalism. This rule encapsulates specific knowledge about how to solve a class of combinatorial problems. It includes a control strategy based on the backtrack class of algorithms and a design method for developing subproblem specifications which, when solved, can be incorporated into the control strategy to solve the original problem. It is believed that the design method is sufficiently specific to guide the synthesis process through the first level specification of any problem in its class.

The secondary goal of this paper is the refinement of general programming knowledge concerning the backtrack control structure. It is believed that current knowledge concerning the strategy is deficient in two areas. While the backtrack procedure has been schematized, general principles concerning the relationships between the components of the procedure have not been clearly articulated. The design alternatives for the lower level functions have not been specified and the relationship of the functions to the problem has not been defined. It is believed that the reduction rule of Chapter IV can provide a design basis for programmers as well as a synthesis system.

The second area of knowledge refinement concerns the extension of the basic strategy to a problem domain to which it has not been previously applied. Chapter V adapts the basic strategy to search the AND/OR graphs produced by a problem reduction representation formalism. The informal reduction rule developed in this chapter can again be applied by programmers as a basis for design.

The backtrack strategy is clearly not the most efficient technique for searching state space or AND/OR trees. Whenever problem area knowledge can be codified for use by a control strategy, a search process which selects a best path for expansion will be more efficient than a backtrack search. In many cases it is either not possible to codify such knowledge in an efficiently computable format or the

search efficiency is not worth the added effort of including the knowledge. The backtrack strategy offers an attractive option. With readily available program schemas and design methods the control strategy is easily developed. Once developed, the strategy can significantly prune a search tree provided problem constraints are sufficiently restrictive. This places emphasis on rigorous identification and specification of the problem constraints, an activity beneficial to programming.

There are significant research areas remaining to be investigated. These include a formal proof of the reduction rule proposed in Chapter IV, formalizing the rule proposed in Chapter V with a formal proof, investigation of efficiency improving constraint allocation techniques for the lower level functions FEASIBLE, SOLUTION and GENERATE, and other design methods for the backtrack strategy based on different assumptions than those discussed.

APPENDIX A - THE PROGRAMMING LANGUAGE

The following is a description of the programming language used in the definition of the program schemas and developed examples. The descriptive format and most definitions are derived from Backus [Ref. xx].

A. THE SET OF OBJECTS AND TYPES

type	description	example values
B	boolean values	true false
N	natural numbers	0, 1, 2, ...
I	integers	..., -1, 0, 1, 2,...
LIST(N)	lists of natural numbers	nil (1) (1,2,3)
<>	sequence of objects	<nil> <1, 3, true, (2,3)>

B. THE SET OF FUNCTIONS

function	domain	range	definition
s:x	any	any	if x=<x1,...,xn> and n >= s then xs else undefined
t1:x	any	any	if x=<x1> then <nil> if x=<x1,...,xn> and n>=2 then <x2,...,xn> else undefined
id:x	any	any	x

function	domain	range	definition
atom:x	any	B	if x B or x N then true else false
equal:x	NxN	B	if x=<y, z> and y=z then true else false
nequal:x	NxN	B	if x=<y,z> and y=z then false else true
null:x	LIST(N)	B	if x=nil then true else false
length:x	LIST(N)	N	if x=nil then 0 if x=<x1,...,xn> then n
+:x	NxN	N	if x=<y,z> then y+z
-:x	NxN	N	if x=<y,z> then y-z
and:x	BxB	B	if x=<true, true> then true else false
or:x	BxB	B	if x=<false, false> then false else true
appendr:x	any	any	if x=<nil, z> then <z> if x=<(x1,...,xn), z> then <x1,...,xn,z> else undefined
appendl:x	any	any	if x=<z, nil> then <z> if x=<z, (x1,...,xn)> then <z,x1,...,xn> else undefined

function	domain	range	definition
append:x	any	any	if x=<z, nil> or x=<nil,z> then <z> if x=<z,(x1,...,xn)> then <z,x1,...,xn> else undefined
tlr:x	any	any	if x=<x1> then <nil> if x=<x1,...,xn> and n>=2 then <x1,...,xm> where m=n-1 else undefined
abs:x	I	N	|x|

note: the result of functions applied to invalid types is undefined

C. THE APPLICATION OPERATION

The application operation allows the use of named parameters. Function definitions include formal parameter names. The scope of these names is restricted to the function application. The actual parameter/formal parameter correspondence is positional. If a single actual parameter is required the syntax is as follows:

function_name:parameter

If multiple parameters are required, they are listed as a sequence as follows:

function_name:<parameter_1, ... ,parameter_n>

D. THE SET OF COMBINING FORMS

form	name	definition
f(g:x)	composition	f:<g:x>

form	name	definition
[f1, ... ,fn]:x	construction	<f1:x, ... ,fn:x>
(p:x f:y;g:z)	condition	if p:x then f:y else g:z where x,y,z are named parameters not necessarily distinct
/f:x	insert	if x=<x1> then x1 if x=<x1, ... ,xn> then f:<x1, /f:<x1,...,xn>> else undefined
f:x	apply to all	if x=nil then nil if x=<x1,...xn> then <f:x1,...,f:xn>

E. THE FUNCTION DEFINITION MECHANISM

The operator binds a function name to a function definition. The syntax is as follows:

```
function name:<parameter list>
              function definition
```

The language also permits the use of anonymous function definitions. The lambda operator is used to define the function as follows:

```
(lambda <parameter list>
        {function definition})
        (actual parameter list)
```

LIST OF REFERENCES

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., Data Structures and Algorithms, Addison-Wesley, 1983.

2. Horowitz, E. and Sahni, S., Fundamentals of Computer Algorithms, pp. 323-359, Computer Science Press, 1978.

3. Naval Postgraduate School Report NPS52-82-011, Top-Down Synthesis of Simple Divide and Conquer Algorithms, by D. R. Smith, November 1982.

4. Smith, D. R., A Problem Reduction Approach to Program Synthesis, to be published in Proceedings of the Eighth International Joint Conference on Artificial Intelligence, August 1983.

5. Walker, R. L., "An Enumerative Technique for a Class of Combinatorial Problems," Proceedings of the Symposium on Applied Mathematics, vol. X, 1960.

6. Golomb, S. and Baumert, P., "Backtrack Programming," Journal of the Association for Computing Machinery, vol. 12, no. 4, pp. 516-524, 1965.

7. Bitner, J. R. and Reingold, E. M., "Backtrack Programming Techniques," Communications of the Association for Computing Machinery, vol. 18, no. 11, November 1975.

8. Reingold, E. M., Nievergelt, J., and Deo, N., Combinatorial Algorithms: Theory and Practice, pp. 126-130, Prentice-Hall, 1977.

9. Gernart, S. L. and Yelowitz, L., "Control Structure Abstractions of the Backtrack Programming Technique," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 285-292, December 1976.

10. Naval Postgraduate School Report NPS52-82-011, Top-Down Synthesis of Simple Divide and Conquer Algorithms, by D. R. Smith, November 1982.

11. Smith, D. R., A Problem Reduction Approach to Program Synthesis, to be published in Proceedings of the Eighth International Joint Conference on Artificial Intelligence, August 1983.

12. Ibid., p. 1.

13. Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style of Programming and Its Algebra of Programs," Communications of the Association for Computing Machinery, vol. 21, no. 8, pp. 613-641, August 1978.

14. Ibid., pp. 618, 624.

15. Ibid., p. 620.

16. Turner, D. A., "Another Algorithm for Bracket Abstraction", The Journal of Symbolic Logic, vol. 44, no. 2, pp. 267-270, June 1979.

17. Horowitz, E. and Sanni, S., Fundamentals of Computer Algorithms, p. 332, Computer Science Press, 1978.

18. Gernart, S. L. and Yelowitz, L., "Control Structure Abstractions of the Backtrack Programming Technique," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, p. 288, December 1976.

19. Ibid., pp. 285-289.

20. Garey, M. R. and Johnson, D. S., Computers and Intractibility: A Guide to the Theory of NP-Completeness, p. 236, W. H. Freeman and Company, 1979.

21. Naval Posteradute School Report NPS52-82-011, Top-Down Synthesis of Simple Divide and Conquer Algorithms, by D. R. Smith, November 1982.

22. Bitner, J. R. and Reingold, E. M., "Backtrack Programming Techniques," Communications of the Association for Computing Machinery, vol. 18, no. 11, November 1975.

23. Golomb, S. and Baumert, P., "Backtrack Programming," Journal of the Association for Computing Machinery, vol. 12, no. 4, pp. 516-524, 1965.

24. Gernart, S. L. and Yelowitz, L., "Control Structure Abstractions of the Backtrack Programming Technique," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, p. 288, December 1976.

25. Walker, R. L., "An Enumerative Technique for a Class of Combinatorial Problems," Proceedings of the Symposium on Applied Mathematics, vol. X, 1960.

26. Lindstrom, G., "Backtracking in a Generalized Control

Setting," Association of Computing Machinery Transactions on Programming Languages and Systems, vol. 1, no. 1, pp. 8-26, July 1979.

27. Gerhart, S. L. and Yelowitz, L., "Control Structure Abstractions of the Backtrack Programming Technique," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, p. 288, December 1976.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	Curricular Officer, Code 37 Computer Technology Curricular Office Naval Postgraduate School Monterey, California 93940	1
5.	Associate Professor Douglas R. Smith, Code 52sc Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
6.	Associate Professor Bruce R. MacLennan, Code 52ml Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
7.	CPT Gary Loberg HQ, US Army CECOM ATTN: DRSEL-TCS-CR Fort Monmouth, New Jersey 07703	2


DATE
ILME