ED 086 167 IR 000 013

AUTHOR	Biermann, A. W.; And Others
TITLE	Automatic Program Synthesis Reports.
INSTITUTION	Ohio State Univ., Columbus. Computer and Information Science Research Center.
SPONS AGENCY	Department of Defense, Washington, D.C. Advanced Research Projects Agency.; National Science Foundation, Washington, D.C.
REPORT NO	OSU-CISRC-TR-73-6
PUB DATE	Oct 73
NOTE	66p.
EDRS PRICE	MF-$0.65 HC-$3.29
DESCRIPTORS	*Algorithms; *Artificial Intelligence; Computational Linguistics; *Computer Programs; Computer Science; Information Science; Man Machine Systems
IDENTIFIERS	*Automatic Program Synthesis

ABSTRACT

Some of the major results of future goals of an automatic program synthesis project are described in the two papers that comprise this document. The first paper gives a detailed algorithm for synthesizing a computer program from a trace of its behavior. Since the algorithm involves a search, the length of time required to do the synthesis of nontrivial programs can be quite large. Techniques are given for preprocessing the trace information to reduce enumeration, for pruning the search using a failure memory technique, and for utilizing multiple traces to the best advantage. The results of numerous tests are given to demonstrate the value of the techniques. The other paper gives a brief overview of the automatic programming system within which the above algorithm is being used. This system is still under development. (Author/CH)

TECHNICAL REPORT SERIES

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(OSU-CISRC-TR-73-6)

AUTOMATIC PROGRAM SYNTHESIS REPORTS

by

A. W. Biermann, R. Baum
R. Krishnaswamy, and F. E. Petry

Work performed under

Grant No. SD-183, Advanced Research Projects Agency
Grant No. GJ-776, National Science Foundation
Grant No. GN-534.1, National Science Foundation
Grant No. GJ-34739X, National Science Foundation

The Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210
October 1973

PREFACE

The two papers included here describe some of the major results of our automatic program synthesis project and indicate some of our goals for the future. The first paper gives a detailed and very precise description of our algorithm for program synthesis from computation traces. The other paper gives a brief overview of the automatic programming system within which the above algorithm is being used.

The research reported in the first paper was done partly in the Computer Science Department, Stanford University and was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183) and the National Science Foundation Grant No. GJ-776. Part of the work was done in the Computer and Information Science Research Center at the Ohio State University and was supported by Grant Numbers GN-534.1 and GJ-34739X from the National Science Foundation.

The second paper was done in the Computer and Information Science Research Center and was supported by National Science Foundation Grant No. GJ-34739X.

The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report presents research accomplished in cooperation with the Department of Computer and Information Science. The research was administered and monitored by The Ohio State University Research Foundation.

TABLE OF CONTENTS

Preface ii

Speeding Up a Program Synthesizer 1

I. Introduction 2

II. A Synthesis Procedure 6

III. Pruning Techniques 14

IV. A Program for Comparison 24

V. The Experiments 26

VI. Conclusion 38

VII. Acknowledgment 39

Appendix 1 42

Appendix 2 45

A System for Program Synthesis from Examples 47

I. Introduction 48

II. The Language 49

III. The Current Status of the System 55

IV. Some Problems 59

V. Acknowledgment 61

VI. References 62

Speeding up a Program Synthesizer

Alan W. Biermann, Richard Baum, Frederick E. Petry

Department of Computer and Information Science
The Ohio State University

Abstract

An algorithm is given for synthesizing a computer program from a trace of its behavior. Since the algorithm involves a search, the length of time required to do the synthesis of nontrivial programs can be quite large. Techniques are given for preprocessing the trace information to reduce enumeration, for pruning the search using a failure memory technique, and for utilizing multiple traces to the best advantage. The results of numerous tests are given to demonstrate the value of the techniques.

keywords: program synthesis, inference, learning, program inference, finite-state machine synthesis, incompletely specified machines

I. INTRODUCTION

The two most important problems to overcome before a practical automatic program synthesizer can be developed are

(1) how to design the input to the synthesizer so that it is in a form easily used by human beings, and

(2) how to speed up the synthesis through intelligent use of the input information so that large programs can be synthesized in a practical amount of time.

Since it is always possible to simply enumerate the set of all programs until one is found which satisfies the given conditions, the question is not whether the program can be synthesized but whether it can be discovered after a reasonable amount of computation. The designer of an automatic program synthesizer, therefore, needs a technique for transmitting information from the human user to the machine in a form which is easily used by both. This paper will suggest that example computations may be a proper vehicle for such man-machine communications.

We know that when humans communicate ideas to each other, examples play a primary part. It is difficult, for example, to imagine any kind of explanation of how to write a program or do a particular calculation without including some particular cases of how the process is done. Since teaching and learning by example seem to be a natural mode of communication for humans, it does not seem unreasonable to believe that a man and a machine might communicate in the same way.

Recent results in inference theory [9, 10, 11, 12, 14,15,17] show that not only can man learn from examples, but machines can also. With regard to the program synthesis problem, Biermann [9] has given an algorithm for generating programs from computation traces, and Raulefs [26] and Barzdin [6] have also studied the problem. These papers seem to indicate that correct programs can be synthesized on the basis of relatively little input information, often only one sample computation, but that the amount of time required to do the synthesis grows very quickly as a function of program complexity. Typical computation times for constructing the one, three, and five state Turing machine controllers given in [9] were about one, five, and one hundred seconds, respectively. The implications are clearly that programs of practical interest will never be synthesized unless either the old methods are speeded up or new ones are developed. This paper will present a number of techniques for speeding up the synthesis process and will show that most of the searching done by the algorithm in [9] is, in fact, unnecessary. Many of the examples that were done in that research involved long searches with dozens or hundreds of backups whereas often only two or three were necessary. The greatly increased power of the techniques discussed here makes it possible to synthesize programs of significant complexity in just a few seconds of time.

The goal of our research is the development of what we call an autoprogramming system. Such a system gives the user a facility for the easy generation of example computations. We are currently using a computer display system on which appear data structures declared by the user and the commands available for doing a calculation. The user executes an example calculation by referencing the commands and their operands (among the data structures) with a light pen in scratch pad fashion. The contents of the data structures are continuously updated as the calculation proceeds. While the example is being carried out, the machine records the sequence of commands, and later, after one or several examples have been completed, it constructs the shortest computer program which is consistent with these computation traces. As an illustration, the user might spend several minutes sorting by hand with the light pen the list of integers (3, 2, 1, 4) using some algorithm. After a fraction of a second of computing, the machine would type out a general program for the algorithm which sorts N integers. This example is explained in more detail in Section V.

If the synthesized program should exhibit some shortcomings, the individual could input additional computations which would force the system to appropriately revise the generated program. It is well known [9, 18] that any program (or its equivalent) may be generated in this manner.

This paper will discuss the core of the autoprogramming system, the program synthesizer, and another paper [13] will describe the autoprogramming language that we have implemented and its operation. Fortunately, the program synthesizer is quite language independent and so no details of any language need be discussed here.

It is important that our goals in this work not be confused with those of many of the well known researchers on program synthesis methods: Amarel [1, 2], Balzer [5], Waldinger and Lee [29], Manna and Waldinger [23] and others[1]. These individuals are interested in designing systems which actually synthesize the algorithm for doing the problem from very weak input information such as input-output pairs or a formal specification of the desired performance. Such work is extremely important from the artificial intelligence point of view because it addresses the problems of what is thought, how should knowledge be represented, and so forth. Our interest however is in man-machine communication, and we assume that the user already knows how to solve the problem. Our goal is not to automatically synthesize an algorithm but rather to provide an easy means for transmitting the user's concept of the algorithm to the machine. We suspect that the major problem for today's practical programmers is more often a problem of communication with the machine than discovery of an algorithm.

[1]See the Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford, California, August, 1973.

The careful reader will notice that in this paper we are synthesizing minimal incompletely specified machines as was done in [3, 4, 20, 21, 22, 25]. In fact, our preprocessing stage where "difference vectors" are constructed has some resemblance to these earlier methods where "compatibility sets" were assembled in the first stage of the synthesis. However, our search technique is quite different from those of previous approaches and heavily uses the special characteristics of our problem to obtain maximum efficiency. The synthesis methods for completely specified machines such as described in [19] and [28] of course require no searching at all but they are not applicable to the problem discussed here.

II. A SYNTHESIS PROCEDURE

Suppose an individual sitting at a display terminal executes an example computation using some kind of autoprogramming language: read into register 1, add register 1 to register 2, shift register 2 right one digit, and so forth. The contents of all of the registers are visible on the display and the results of each command are immediately updated. Assume that the sequence of commands is $I_1,I_2,I_2,I_2,I_2C_1:I_1,I_2,C_1:I_2,I_2,C_1:I_1,C_2$:halt. I_1 and I_2 represent instructions executed, and C_1 and C_2 represent the existence of some tested condition such as register 1 is positive or register 1 exceeds register 2. Then the task of the synthesizer is to produce the simplest possible program which is capable of executing this trace.

Each instruction I_j may occur several times in the same program so the k^{th} occurrence of I_j will be denoted kI_j. Then the program can be constructed as shown in Figure 1 where the task is to find which occurrence kI_j in the program corresponds to each particular I_j in the trace. Let us assume that there will be a limit of L=4 or fewer instructions in the final program not including tests and branches so that the instruction set will be either $\{1I_1,1I_2,1I_3\}$, $\{1I_1,2I_1,1I_2,1I_3\}$, $\{1I_1,1I_2,2I_2,1I_3\}$, or $\{1I_1,1I_2,1I_3,2I_3\}$. (Note that the halt instruction is denoted I_3.) Thus each instruction I_1,I_2, and I_3 must appear at least once and there can be no more than L=4 instructions.

A detailed description of the example of Figure 1 is given in Figure 2. The pointer IND is advanced down the trace and each instruction execution I_j is associated with a particular kI_j in the program where k is set to the lowest number which has not been found to be contradictory. After the eighth step in Figure 2, the partial program of Figure 3(a) has been constructed. Notice that this partial program implies that the fourth instruction executed must be $1I_2$. This is called a forced move and when it is entered by step 9 of Figure 2 it is parenthesized. An infinite loop would result except that condition C_1 is observed at the sixth entry in the trace followed by an

	condition	instruction	initial synthesis	first backup	second backup	third backup	fourth backup
1		I_1	$1I_1$				
2		I_2	$1I_2$				
3		I_2	$1I_2$				$2I_2$
4		I_2	$(1I_2)$				$1I_2$
5		I_2	$(1I_2)$				$(2I_2)$
6	C_1	I_1	$1I_1$	$2I_1$		$3I_1$	$1I_1$
7		I_2	$(1I_2)$	$1I_2$	$2I_2$		$(1I_2)$
8	C_1	I_2	X	X			$1I_2$
9		I_2					$(2I_2)$
10	C_1	I_1					$(1I_1)$
11	C_2	halt					$(1I_3)$

X indicates a contradiction found.

Figure 1. The example trace and the creation of the flow diagram.

Step	Condition	Execution	IND	Entry made
1		Read L		
2		IND ← 1	1	
3		Give instruction I_j the name $1I_j$		$1I_1$
4		IND ← IND + 1	2	
5		Give instruction I_j the name $1I_j$		$1I_2$
6		IND ← IND + 1	3	
7		Give instruction I_j the name $1I_j$		$1I_2$
8		IND ← IND + 1	4	
9	Forced move	Indicate forced move parenthesized		$(1I_2)$
10		IND ← IND + 1	5	
11	Forced move	Indicate forced move parenthesized		$(1I_2)$
12		IND ← IND + 1	6	
13		Give instrution I_j the name $1I_j$		$1I_1$
14		IND ← IND + 1	7	
15	Forced move	Indicate forced move parenthesized		$(1I_2)$
16		IND ← IND + 1	8	
17	Forced move yields a contradiction	Decrease IND to last unparenthesized move	6	first backup
18		Increment name kI_j to $(k+1)I_j$		$2I_1$
19		IND ← IND + 1	7	
20		Give instruction I_j the name $1I_j$		$1I_2$
21		IND ← IND + 1	8	
22	Forced move yields a contradiction	Decrease IND to last unparenthesized move	7	second backup
23		Increment name kI_j to $(k+1)I_j$		$2I_2$
24	Allowed number of states L is exceeded	Decrease IND to last unparenthesized move	6	third backup
25		Increment name kI_j to $(k+1)I_j$		$3I_1$
26	Allowed number of states L is exceeded	Decrease IND to last unparenthesized move	3	fourth backup
27		Increment name kI_j to $(k+1)I_j$		$2I_2$
28		IND ← IND + 1	4	
29		Give instruction I_j the name $1I_j$		$1I_2$
30		IND ← IND + 1	5	
31	Forced move	Indicate forced move parenthesized		$(2I_2)$
32		IND ← IND + 1	6	
33		Give instruction I_j the name $1I_j$		$1I_1$
34		IND ← IND + 1	7	
35	Forced move	Indicate forced move parenthesized		$(1I_2)$
36		IND ← IND + 1	8	
37		Give instruction I_j the name $1I_j$		$1I_2$
38		IND ← IND + 1	9	
39	Forced move	Indicate forced move parenthesized		$(2I_2)$
40		IND ← IND + 1	10	
41	Forced move	Indicate forced move parenthesized		$(1I_1)$
42		IND ← IND + 1	11	
43		Give instruction I_j the name $1I_j$		$1I_3$
44	Halt instruction reached	Print the solution		
45		halt		

Figure 2. Working the example of Figure 1.

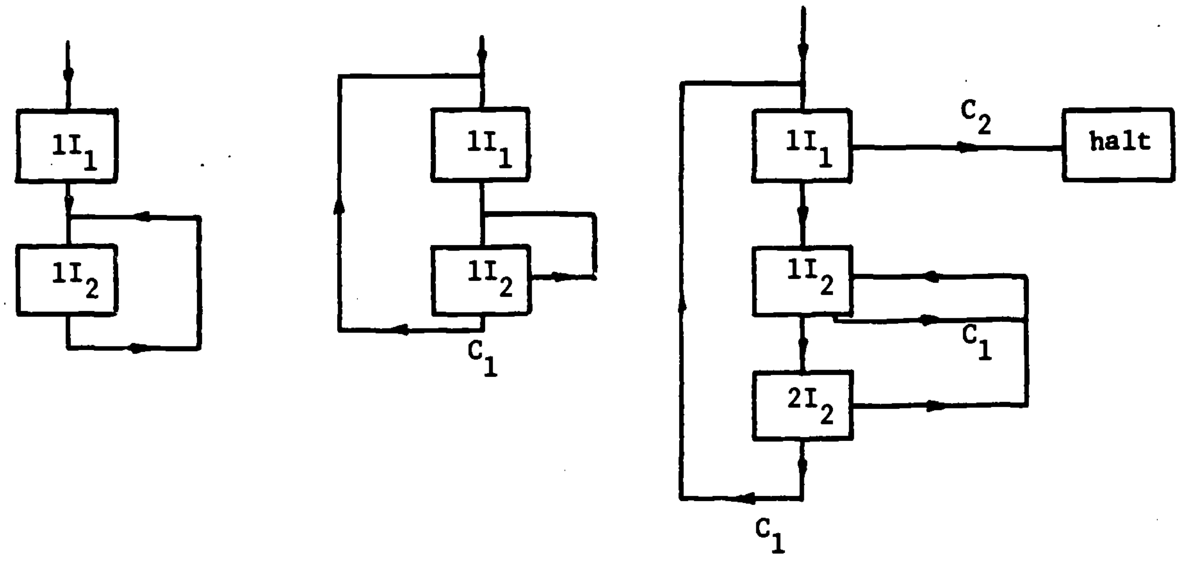


Figure 3. The construction of the program for the trace of Figure 1.

instruction I_1. (See Figure 1.) This instruction is assumed to be $1I_1$ in the program yielding the flow chart of Figure 3(b). This flow chart is compatible with the first seven steps of the trace but it yields a contradiction at step eight implying that a previous decision was incorrect. So the pointer IND is decreased to the last move which was not forced (IND = 6) and instruction I_1 is given the new name $2I_1$. This process is continued as detailed in Figure 2 until a compatible flow diagram is found as shown in Figure 3(c). If no four-instruction compatible flow diagram existed, backing up would eventually reach the beginning of the trace and the method could be tried again with L = 5.

The reader should note that the condition C_i before an instruction on a trace is included whenever it is checked and found to be true. It is omitted if either it is not checked or not true. Consequently, on a synthesized flow diagram, a particular instruction may have several transitions leading away, one blank and the others labelled $C_1, C_2, \ldots, C_k$. In this case, each C_i is assumed to be disjoint from C_j if $i \neq j$ and the unlabelled transition is taken if none of $C_1, C_2, \ldots, C_k$ is true.

The procedure described above is enumerative and will surely find a flow diagram compatible with the given trace if one exists with L instructions. If no solution is found, the method can be repeated as L is increased until one is found. After the first solution is discovered, the enumeration can be continued until additional solutions are found. This algorithm is similar to the one reported in [9] except that programs are modelled with Moore machines here rather than with Mealy machines[2]. This seems to be a more natural model and has led to many insights for improving performance. Sections III and IV will describe several techniques for pruning the search tree and greatly increasing the algorithm efficiency.

[2] In this paper we discuss flow charts and Moore machines interchangeably. Thus instructions represent states and flow lines represent transitions.

In order to make some of these ideas more precise, it is necessary to introduce additional notation. $[I_j]$ will be used to designate the ordered set $\{1I_j, 2I_j, \ldots, K_jI_j\}$ of occurrences of I_j in a synthesized program. For example, in Figure 3(c), $[I_2]=\{1I_2, 2I_2\}$. K will denote the number of different instructions in the program and K_j will denote the number of occurrences of the instruction I_j. The condition C_i and the instruction I_j at the kth level of the trace will be denoted N_k and O_k, respectively. Thus in the trace of Figure 1, we have N_1=(blank), $O_1=I_1, N_6=C_1, O_6=I_1$, and so forth. The cardinality of set S will be written $|S|$.

The function of the synthesis algorithm is thus to

(1) determine $|[I_j]|$ for $j=1,2,\ldots,K$ subject to the constraint that

$$\sum_{j=1}^{K} |[I_j]| \text{ is minimum, and}$$

(2) determine which member of $[O_j]$ corresponds to each specific occurrence of O_j in the trace.

To facilitate discussion of the above points, we introduce a selector variable (or state number) S^j which is associated with the j th level of the trace. The selector variable points to the element of $[O_j]$ that corresponds to the occurrence of O_j. Thus (2) can be restated: For each level j, determine the value of S^j. One can check that in the final solution of Figure 1, $S^1=1$, $S^2=1$, $S^3=2$, $S^4=1$, $S^5=2$, etc.. Since the synthesized program is to be deterministic, the assignment of a value to S^j is subject to this constraint: If there are levels i and j such that, $i<j$, $O_{i-1}=O_{j-1}$, $S^{i-1}=S^{j-1}$, $N_i=N_j$, and $O_i=O_j$, then S^i must be the same as S^j. If the previous condition is satisfied, we say that S^j is forced; otherwise it is arbitrary. Since the search changes the values of parameters dynamically, we will use the notation 'name'* to represent the current (perhaps transient) value of the entity 'name'.

The state count SK* is the current value of $\sum_{j=1}^{K} |[I_j]^*|$ and the state limit L gives the maximum value for SK* permitted during the synthesis. If S^1, S^2,...,S^j are incorrectly assigned values, then a contradiction will occur. Assignment at level j causes a contradiction if either

(1) there is a level i,i<j, such that $O_{i-1}=O_{j-1}$, $S^{i-1}=S^{j-1}$, $N_i=N_j$, and either $O_i \neq O_j$ or it is known that $S^i \neq S^j$, or

(2) SK* becomes greater than L.

An obvious synthesis technique is to simply try all possible assignments of S^1, S^2,...,S^M (assuming a trace of length M) for increasing values of L until a contradiction free (or compatible) assignment is found. Then the transition set of the synthesized (or compatible) machine is $\{(S^{j-1}O_{j-1},N_j,S^jO_j)|1<j\leq M\}$. The problem with a strictly enumerative technique is that as L-K becomes larger, the search space grows exponentially and the processing time for the synthesis becomes prohibitive. The next section will address the problem of limiting this search.

Before leaving this section, one can study a second example of the synthesis technique by applying the method described above to the computation trace in Figure 2. When this is done, the program of Figure 4 results illustrating that the synthesis technique can be employed to create its own program. The program of Figure 4 does, in fact, correctly express the synthesis algorithm discussed above and will be the starting point for the treatment in Section III. The reader who takes the trouble to work out this second example will discover that the desired program is created directly without any search or backing up. This always occurs when the program being constructed has no duplicated instructions, and we can even state a theorem: If the desired program has no duplicated instructions and if the given trace covers every transition in that program, then the program will be synthesized directly without any searching. Thus searching is only required to resolve the ambiguities which result from multiple copies of the same instruction in a program. This has important implications for practical synthesis problems because typical programs usually have few duplications.

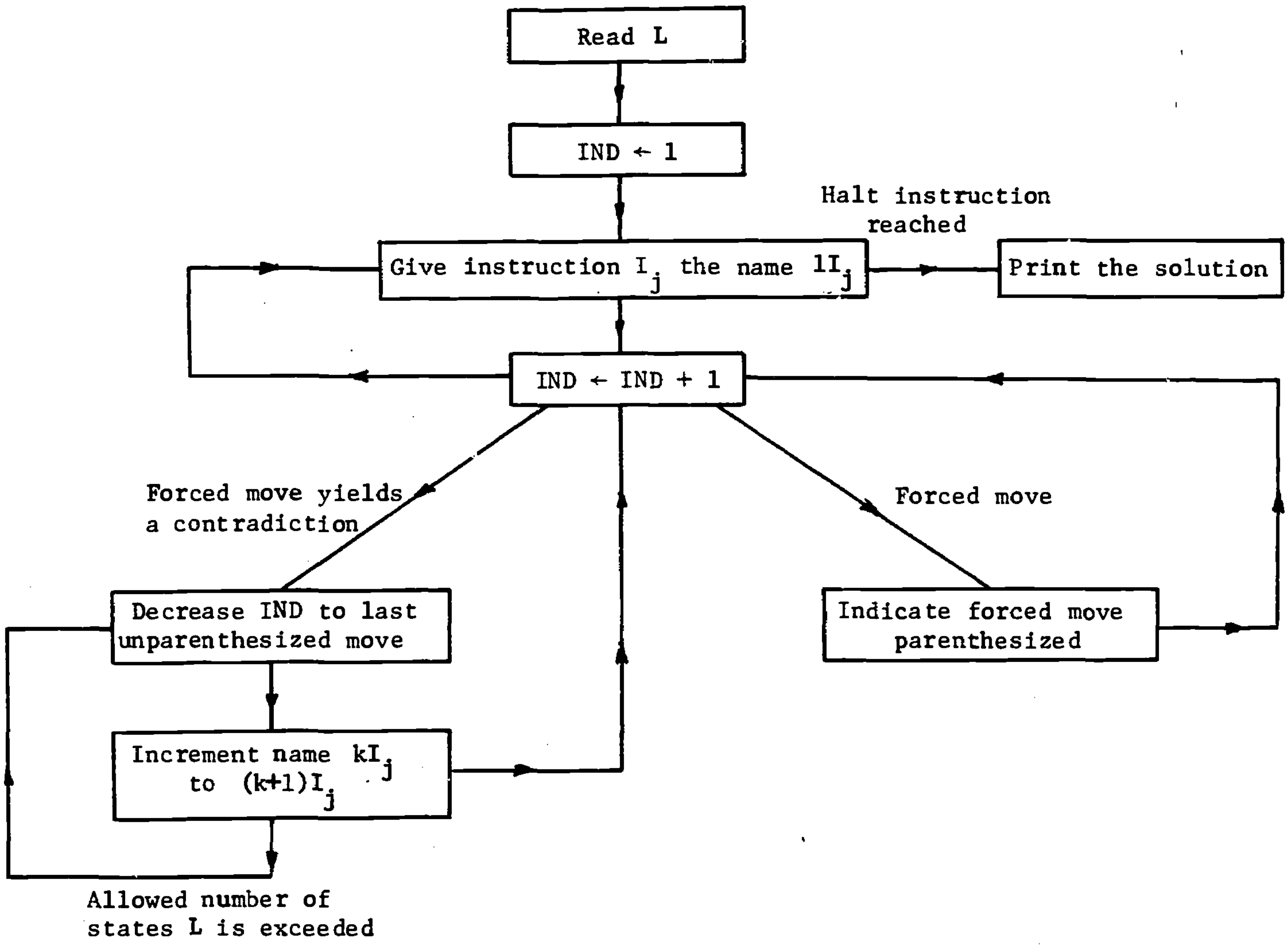

Read L
IND ← 1
Halt instruction reached
Give instruction I_j the name $1I_j$
Print the solution
IND ← IND + 1
Forced move yields a contradiction
Forced move
Decrease IND to last unparenthesized move
Indicate forced move parenthesized
Increment name kI_j to $(k+1)I_j$
Allowed number of states L is exceeded

Figure 4. Flowchart for the synthesis algorithm.

III. PRUNING TECHNIQUES

To make the synthesis process feasible it is necessary to find a way to reduce the search space. Pruning techniques fall into two categories: dynamic and static. Static processing performs a structural analysis of the trace without considering the effects of assigning specific values to the selector variables. Dynamic processing is performed while the search is in progress and uses all assigned selector variables as well as a record of potential contradictions to avoid mistakes.

Static processing, also called preprocessing, scans the trace and builds up nonequivalence information which is subsequently used by the dynamic phase of the synthesis.

Preprocessing makes use of distinguishing sequences found in the trace to differentiate 0-equivalent states. Let i and j be levels in the trace where $O_i=O_j$ (assume $i<j$). If $N_{i+p}=N_{j+p}$ for $p=1,2,\ldots,k$ and $O_{i+q}=O_{j+q}$ for $q=0,1,\ldots,k-1$ and $O_{i+k}\neq O_{j+k}$ then the states represented by S^iO_i and S^jO_j are k-nonequivalent. It then follows that $S^i\neq S^j$. This information is kept in a difference set DV.

associated with each level ℓ. DV_ℓ is defined as the set:

$$\{i \mid \ell < i \leq M \text{ and } O_i = O_\ell \text{ and } S^i \neq S^j\}.$$

We now use the difference sets to determine a lower bound or initial guess L_k for each $|[I_k]|$. A lower bound for L is then given by $\sum_{k=1}^{K} L_k$. L_k is the largest m such that there exist levels $\ell_1, \ell_2, \ldots, \ell_m$ $(\ell_1 < \ell_2 < \ldots < \ell_m)$ having the following properties:

(1) $O_{\ell_1} = O_{\ell_2} = \ldots = O_{\ell_m} = I_k$ and

(2) For $i = 2, 3, \ldots, m$: $\ell_i \in DV_{\ell_j}$ for $j = 1, 2, \ldots, i-1$

We will now introduce the parameters required to describe dynamic processing. The <u>free state limit</u> FL is defined as $L - \sum_{i=1}^{K} L_i$. If $FL > 0$ then $|[I_j]| > L_j$ for one or more values of j. Level i is called the <u>working</u> level WL* if $S^1, S^2, \ldots, S^i$ have been assigned values but $S^{i+1}, \ldots, S^M$ remain unassigned. The <u>free state count</u> F* is given by $F - \sum_{i=1}^{K} \max(|[I_i]^*| - L_i, 0)$. Most of the actions of dynamic processing are dependent on the contents of a <u>failure memory</u> FM. The failure memory may be conceptually visualized as an M by L matrix with each row FM_j corresponding to a level in the trace. Each matrix element FM_{ij} is a couple (W_{ij}, G_{ij}). W_{ij} is called the <u>structure factor</u> and G_{ij} is called the <u>free state factor.</u> The effect each FM_{ij} has on the search depends upon whether the element is <u>valid</u>, <u>latent</u>, or <u>null</u>. An element is valid if $W_{ij} > 0$ and $F^* < G_{ij}$, an element is latent if $W_{ij} > 0$ and $F^* \geq G_{ij}$, an element is null if $W_{ij} = 0$ and an element is <u>invalid</u> if it is null or latent. A latent or valid element will become null if WL* ever becomes less than W_{ij}. If FM_{ij} is valid then S^i may not be assigned the value j. Some of these definitions are summarized in Figure 5.

	$F^* < G_{ij}$	$F^* \geq G_{ij}$
$W_{ij} > 0$	valid	latent
$W_{ij} = 0$	null	

Figure 5.

This therefore outlines the basic failure memory technique. For example, suppose it is known from other considerations that if at working level 9 the selector variable S^9 is set to value 2, then a contradiction will later result. Then the failure memory entry at $FM_{9,2}$ will be valid, and during the search, S^9 may be set to 1,3,4, etc. but it should never be set to 2. Clearly if large numbers of failure memory entries are valid, the number of alternative settings for each S^i will be greatly reduced and so also will be the total size of the search. Suppose also that (3,4) is the entry in S^9 which is valid. This means that the above assertion ($S^9=2$ will cause a later contradiction) is known to be true if

(1) no changes have been made at level 3 or above since the entry (3,4) was made, i.e. S^1, S^2, S^3 are unchanged,

and

(2) there are fewer than 4 free states currently available, $F^* < G_{9,2} = 4$.

The entry (3,4) may be interpreted as an indicator of the cause of failure. The setting $S^9=2$ will yield a contradiction as long as S^1, S^2, S^3 are unchanged and the number of available free states is less than 4. The pruning technique therefore involves limiting the search by keeping the failure memory updated with as many valid entries as possible.

Dynamic processing makes use of static nonequivalence data by adding information to the failure memory. Whenever the selector variable at level ℓ is assigned the value j, the following procedure is performed:

For all $i \in DV_\ell$ do: if FM_{ij} is invalid then $FM_{ij} := (\ell, FL+1)$;

In other words, if S^ℓ is given a value j and S^i is known from preprocessing to be nonequivalent to S^ℓ, make an entry into FM_{ij} which will prevent the assignment $S^i=j$. Notice that the unconditional nature of static nonequivalence information is inherent in the failure memory entries made above since the free state factor FL+1 insures that FM_{ij} is either valid or null.

Since the failure memory is dynamically changing it is possible for two statically equivalent levels to be dynamically distinguished. For example, let i and j ($i<j$) represent levels in the trace where $O_i=O_j, N_{i+1}=N_{j+1}, O_{i+1}=O_{j+1}$ and $N_{i+2}\neq N_{j+2}$. Clearly, preprocessing is incapable of distinguishing S^i and S^j. Suppose that dynamic processing assigns S^{i+1} the value k and $FM_{(j+1)k}$ is or becomes valid; then $S^i\neq S^j$.

Dynamic nonequivalences are detected by using <u>couple classes</u>. A couple class $[I_pN_qI_s]$ is defined to be $[I_pN_qI_s]=\{i|1<i\leq M,O_{i-1}=I_p,N_i=N_q,O_i=I_s\}$. If i and j are members of the same couple class then we say that level i and level j belong to the same couple class. Suppose levels i and j ($i<j$) belong to the same couple class then (a) if $S^{i-1}=S^{j-1}$ then $S^i=S^j$ (i.e. S^j is forced) and (b) if it is known that $S^i\neq S^j$ then $S^{i-1}\neq S^{i-1}$. Hence, couple classed give dynamic processing an efficient way to detect forced moves and to propagate dynamic nonequivalence failure memory information. Selector variable S^j is forced if and only if

(1) $|[O_{j-1}N_jO_j]|>1$,

(2) there is a level $i, i<j$, such that $i\in[O_{j-1}N_jO_j]$

and (3) $S^{i-1}=S^{j-1}$.

In order to implement information from (b) above, $FM_{(j-1),S^{i-1}}$ can be updated as follows if it is currently invalid: $FM_{(j-1),S^{i-1}}$ is set to (WL^*,q) if (1) levels i and j are in the same couple class,

(2) $S^i=n$

and (3) FM_{jn} contains the nonnull couple (p,q).

We will now turn our attention from nonequivalence induced contradictions to contradictions arising from violation of the state limit L. The assignments to $S^1,S^2,\ldots,S^{WL^*}$ are said to violate the state limit if they collectively imply that the number of states in the final synthesized machine will be greater than L. If the state limit L is violated then a <u>fence</u> will occur at some level in the trace. An <u>incipient fence</u> exists at level ℓ if for all $i\leq|[O_\ell]^*|$,

$FM_{\ell i}$ is valid. An incipient fence implies that a new element must be added to $[O_\ell]^*$ (or in other words, that a new state must be created). If $O_\ell = I_k$ and $|[I_k]^*| \geq L_k$ and $F^*>0$, then F^* must be decremented by one since the new state has "committed" a "freestate" to be an element of $[I_k]^*$. A fence exists at level ℓ if an incipient fence exists there and $|[I_k]^*| \geq L_k$ and $F^*=0$. A contradiction immediately follows from a fence since it implies that a needed new state cannot be created.

Dynamic processing uses the concept of the pseudo-assigned selector variable to further enhance its contradiction detection mechanism. Selector variable S^ℓ which is to be assigned a value later in the search is said to be pseudo-assigned the value n if validation of $FM_{\ell n}$ causes a fence at level ℓ. That is, $FM_{\ell i}$ is valid for all possible i except i=n so $S^\ell=n$ will be the only possible assignment at the later time when $WL^*=\ell$. Therefore, if there is a j with the properties $WL^*<j<\ell$ and ℓ is in DV_j, then the assignment $S^j=n$ would result in a fence at level ℓ. Therefore failure memory entries are entered in column n for all such j: $FM_{jn}=(WL^*,1)$ if $WL^*<j<\ell$ and ℓ is in DV_j. The free state factor 1 insures the entries will be valid since the existence of the potential fence implies $F^*=0$. In addition to these failure memory entries, pseudo-assignment at level ℓ also causes entries at levels greater than ℓ in the same way as an ordinary assignment at level ℓ.

If the assignment of the value n to S^ℓ causes a contradiction then dynamic processing must perform a fixup or a backup. The fixup operation unassigns S^ℓ (thus causing WL^* to be momentarily decremented) and then reassigns the value n+1 to S^ℓ. The fixup operation cannot take place if S^ℓ is forced or if the last assignment to S^ℓ created a new state or if validation of $FM_{\ell n}$ causes a fence. When S^ℓ is reassigned n+1 a failure entry must be stored in $FM_{\ell n}$. Before we describe how this entry is built, we must introduce the concept of usage information.

Every decision made during dynamic processing is dependent on the assignment of values to certain selector variables. For example, a forced assignment to S^ℓ

depends on the values $S^{\ell-1}, S^{j}$, and S^{j-1} where ℓ and j are in the same couple class. By maintaining a record of exactly what selector variables contributed to every assignment or failure memory entry it is possible to update the failure memory during fixup and backup. Now we are ready to give an intuitive definition of the structure factor. The structure factor W* is the greatest j less than or equal to WL* such that S^{j} contributed to the decision presently being consumated. Whenever WL* is decreased to j all failure memory entries (p,q) with p>j become (0,0) and all side effects (except usage information) of nulled failure memory entries and previously assigned selector variables at levels greater than j are removed. Thus the states of all entities other than usage are restored to the way they were when WL* was last equal to j. Therefore whenever a fixup occurs at level ℓ as described above, the entry $FM_{\ell n}$ is set to (W*,F*+1) .

Backup is initiated from level ℓ if:

(1) S^{ℓ} is forced and $FM_{\ell,S^{\ell}}$ is valid or

(2) S^{ℓ} causes a contradiction and S^{ℓ} represents a new state or

(3) validation of $FM_{\ell,S^{\ell}}$ will create a fence .

The function of backup is to find the greatest j such that (1) S^{j} contributed to the contradiction causing the backup and (2) it is possible to apply a fixup operation on S^{j}. Backup operates by repeatly setting WL* to W* until a fixup operation can be applied. Failure memory entries may be created at levels bypassed by backup. The entries are of the form (W*,F*+1) and are located on all levels i (in the appropriate column) such that W*>j. If backup causes WL* to become zero then there is no L-state machine compatible with the trace. In this case, all limit variables are increased by one and the search begins again.

In order to illustrate the operation of the algorithm, we will give a "blow by blow" account of an example synthesis by tagging all dynamic entities with an event number. These tags indicate the creation order of the entities. When an entity is redefined, we will simply write the new value to the right of the old one so that a complete history of the synthesis is available when the algorithm terminates.

This example and a second example in Appendix 1 each consists of two tables. The EVENT TABLE provides commentary of each event. The TRACE TABLE provides a representation of all entities, static and dynamic, used by the algorithm. The TRACE TABLE consists of the following columns:

Level - - - -	a level number
O_i - - - - -	an output symbol
N_i - - - - -	an input symbol
Couple-class - -	the couple class that contains the level
Difference set -	the set DV_{level}
Selector values -	the integer values that S^{level} takes during the synthesis
FM_{*j} - - - -	this column represents column j of the failure memory matrix. An entry (W_{ij},G_{ij}) is represented by the notation ${W_{ij}}_{G_{ij}}$.

Note that the string 'entity' is tagged by writing it as 'entity' (event number). The difference vector and couple class information is set up during the preprocessing and the numbered events refer to the dynamic phase.

Event(s)	Level	Discussion
0		Initiate synthesis, F=0, FL=0
1	1	S^1 is assigned value 1 (assignment)
2-10	35,32,26,23 20,11,9,6,4	Failure memory entry created at a level in DV_1 (DV_1 resolution)
11,12	2,3	assignment
13	4	S^4 is assigned 2 since FM_{41} is valid
14	32	DV_4 resolution.
15	6	FM_{63} is made valid since $S^6=3 \rightarrow FM_{32,3}$ is valid $\rightarrow$ fence at level 32. The first implication follows from DV_6 (look-ahead fence prevention at 32 from 6)
16,17	29,23	DV_4 resolution
18	20	look-ahead fence prevention at 23 from 20
19	11	DV_4 resolution, since S^{11} is now pseudo-assigned the value 3, since $FM_{11,1}$ and $FM_{11,2}$ are valid and $L_A=3$ and F=0
20-22	35,29,26	DV_{11} resolution performed since S^{11} is pseudo-assigned
23	9	DV_4 resolution
ˀ4,25	5,6	assignment
26	7	S^7 must be assigned value 1 since $S^5=1$ and 5,7 CC_3 (forced move (denoted by dot))
27	8	forced move
28-33	9-14	Assignment/forced move
34	34	$FM_{34,3}$ is validated since 14 and 35 are both members of same couple class but $S^{35} \neq 1$ (dynamic nonequivalence via CC_8)
35	33	dynamic nonequivalence via CC_7. This implies that $\|[S]\|>1$, but this is a contradiction since F=0 and $L_S=1$, so .....
36	33,34	the assignment $S^{14}=1$ is incorrect. Invalidate $FM_{33,1}$ and $FM_{34,3}$
37,38	14,15	assignment
39	19	dynamic nonequivalence via CC_9
40	16	assignment
41-42	19-18	dynamic nonequivalence via CC_9
43	18	dynamic nonequivalence via CC_9, since S^{19} cannot equal S^{20} (they're both pseudo assigned) and since level 19 and level 20 are in the same couple class it follows that $S^{18} \neq S^{19}$. Since S^{18} and S^{19} cannot be assigned 1 or 2 it follows that $\|[A]\|>3$, but this is a contradiction since F=0 and $L_A=3$, so .....
44	18,18,19	the assignment $S^{16}=1$ is incorrect. Invalidate $FM_{19,1}$, $FM_{18,1}$, $FM_{18,2}$
45	16	assignment
46	18	dynamic nonequivalence via CC_9
47	17	assignment
48	37	DV_{17} resolution
49	18	assignment
50	31	dynamic nonequivalence via CC_{10}
51	30	dynamic nonequivalence via CC_2, This implies that $\|[P]\|>1$, this is a contradiction since F=0 and $L_P=1$
52	30,31	the assignment S^{18} is incorrect. Invalidate $FM_{31,2}$ and $FM_{30,1}$
53	18	assignment
54	19	dynamic nonequivalence via CC_9
55-74	19-38	assignment/forced move, synthesis complete

Figure 6. Event table for example synthesis.

I = {A,P,S,R,T}

Level	N_i	a_{i-1}	Couple-class	Difference Set
1	-	A	.+	{35,32,26,23,20,11,9,6,4}
2	N	P	CC_1	
3	R	A	CC_2	
4	A	A	.	{32,29,23,11,9}
5	N	R	CC_3	
6	R	A	CC_4	{32,29,23,11,9}
7	N	R	CC_3	
8	R	A	CC_4	
9	B	A	CC_5	{35,29,26,20}
10	N	S	CC_6	
11	R	A	CC_7	{35,29,26,20}
12	N	S	CC_6	
13	R	A	CC_7	
14	D	A	CC_8	
15	X	A	CC_9	
16	X	A	CC_9	
17	X	A	CC_9	{37}
18	Y	A	CC_{10}	
19	X	A	CC_9	
20	X	A	CC_9	{32,29,23}
21	N	R	CC_3	
22	R	A	CC_4	
23	B	A	CC_5	{35,29,26}
24	N	S	CC_6	
25	R	A	CC_7	
26	D	A	CC_8	{32,29}
27	N	R	CC_3	
28	R	A	CC_4	
29	G	A	.	{35,32}
30	N	P	CC_1	
31	R	A	CC_2	{37}
32	Y	A	CC_{10}	{35}
33	N	S	CC_6	
34	R	A	CC_7	
35	D	A	CC_8	
36	N	R	CC_3	
37	R	A	CC_4	
38	Y	T	.	

Couple class naming for couple classes with cardinality greater than 1.

[A-N-P] = CC_1 = {2,30}
[P-R-A] = CC_2 = {3,31}
[A-N-R] = CC_3 = {5,7}
[R-R-A] = CC_4 = {6,8,22,28,37}

[A-B-A] = CC_5 = {9,23}
[A-N-S] = CC_6 = {10,12,24,33}
[S-R-A] = CC_7 = {11,13,25,34}
[A-D-A] = CC_8 = {14,26,35}

+ a · means the couple class has cardinality 1

Figure 7. Trace Table for Example Synthesis

Selector Values	FM_{*1}	FM_{*2}	FM_{*3}	Level
1(1)				1
1(11)				2
1(12)				3
2(13)	1_1(10)			4
1(24)				5
2(25)	1_1(9)		4_1(15)	6
.1(26)				7
.2(27)				8
3(28)	1_1(8)	4_1(23)		9
1(29)				10
3(30)	1_1(7)	4_1(19)		11
.1(31)				12
.3(32)				13
1(33),2(37)				14
1(38)				15
1(40) 2(45)				16
.1(47)				17
2(49) 3(53)	16_1(42),0(44),16_2(46)	16_1(43),0(44)		18
1(55)	16_1(41),0(44)	15_1(39)	18_1(54)	19
.2(56)	1_1(6)		4_1(18)	20
.1(57)				21
.2(58)				22
.3(59)	1_1(5)	4_1(17)		23
.1(60)				24
.3(61)				25
.2(62)	1_1(4)		4_1(22)	26
.1(63)				27
.2(64)				28
1(65)		4_1(16)	4_1(21)	29
.1(66)	18_1(51),0(52)			30
.1(67)	18_1(50),0(52)			31
.3(68)	1_1(3)	4_1(14)		32
.1(69)	14_1(35),0(36)			33
.3(70)			14_1(34),0(36)	34
.2(71)	1_1(2)		4_1(20)	35
.1(72)				36
.2(73)	17_1(48)			37
1(74)				38

[A-X-A] = CC_9 = {15,16,17,19,20}
[A-Y-A] = CC_{10} = {18,32}

$L_A = 3$
$L_P = 1$
$L_S = 1$
$L_R = 1$
$L_T = 1$

Figure 7. (Continued)

IV. A PROGRAM FOR COMPARISON

Theoretical methods do not exist for evaluation of our pruning techniques and so it was necessary to do experimental comparisons with other algorithms. A program was written to execute a basically enumerative algorithm which included no sophisticated preprocessing or pruning features at all. The program was written largely in the spirit of the diagram of Figure 4 except that a technique for processing multiple traces in parallel was included. Whereas, the algorithm of the previous section handles multiple traces by concatenating them head-to-tail to make one long trace, a simpler algorithm can utilize the several traces better by conceptually laying them side by side and working on them simultaneously.

Parallel processing is achieved by using the algorithm already given applied to a subset of the traces. The choice of the subset for a given stage is illustrated in Figure 8. The current levels of the traces in that figure are 2,2,1 where the instructions are I_2,I_2,I_1 followed by C_2,C_2,C_2, respectively. The lowest instruction followed by the lowest condition (by some ordering) is chosen along this frontier and the next guess is made for the associated traces. In this case, I_1 followed by C_2, is lowest indicating that the next guess will be for trace 3, the second level. At the next stage all the levels happen to be 2 and the subset chosen for consideration consists of traces 1 and 2. (The frontier instructions are I_2,I_2,I_2 and the next conditions are C_2,C_2,C_3.)

After a subset of the traces is chosen, the algorithm is then used to add the appropriate transition and perform as many forced moves as possible on these traces. As before if a contradiction occurs while forcing or attempting to add a transition, then a backup is done until some transition can be changed. Thus again in Figure 8, at stage 3 when traces 1 and 2 are considered, a contradiction occurs when a transition is attempted to be found for I_2 followed by C_2 and backup must occur.

The advantage of parallel processing is that if a transition which has been added would lead to a contradiction it would likely be discovered sooner than by the sequential processing. The latter might go through several traces completely before the same contradiction could be found. Consider, for example, the contradiction mentioned above between traces 1 and 2. An experiment measuring the usefulness of this parallelism is described in Appendix 2.

	Trace 1			Trace 2			Trace 3	
	I_1	$1I_1$		I_1	$1I_1$		I_1	$1I_1$
C_1	I_2	$1I_2$	C_1	I_2	$1I_2$	C_2	I_2	
C_2	I_1		C_2	I_2		C_3	I_1	
	.			.			.	
	.			.			.	
	.			.			.	

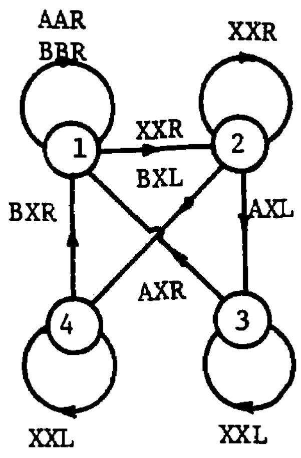
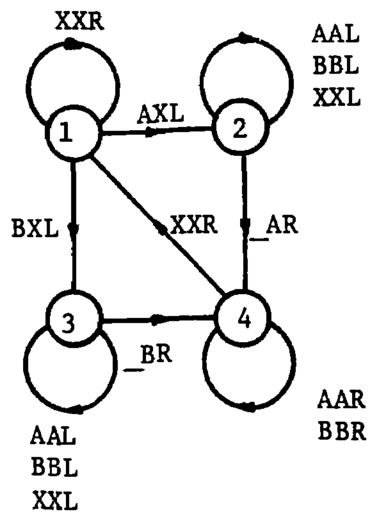
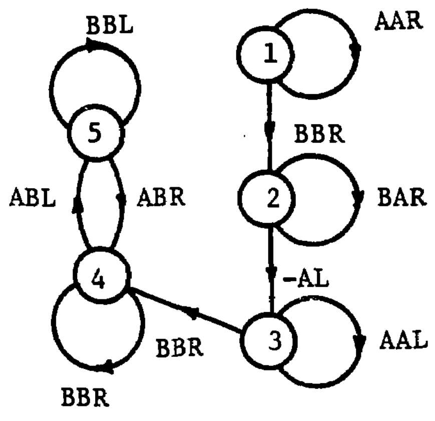
Figure 8. Parallel Processing Example

The comparison algorithm should then be thought of as probably the best possible algorithm that can be obtained without much sophistication.

V. THE EXPERIMENTS

The performance of the two algorithms described here was compared with that of the original paper, Biermann [9]. Turing machines were to be learned as shown in column one of Figure 9 where the three letter triples on each transition indicate the symbol read, the symbol printed, and the head move direction, respectively. As in the original paper, Mealy machines are shown although in our recent experiments their Moore equivalents were used. Column two indicates the sets of traces input to the algorithms by giving the initial tape associated with each trace. Each trace can be reconstructed by applying the proper Turing machine to the given initial tape with the head initially placed at the leftmost nonblank symbol. In terms of the notation of this paper, a transition ABL, for example, is transcribed as a tested condition N_i=A and a resulting executed instruction O_i=BL, print a B and move left. An examination of these results indicates that the recent programs are much faster than the original system but part of this improvement may be due to a switch from compiled Stanford LISP 1.6 to FORTRAN. (The machine in each case was a DEC PDP-10.) The second conclusion is that on these relatively easy problems, the enumerative parallel algorithm is approximately as fast as its more sophisticated counterpart. This is largely because the enumerative algorithm wasted no time on preprocessing or failure memory overhead and the amount of backing up was not large enough to become catastrophic.

As a more serious test of our system, it was decided to attempt to synthesize a universal Turing machine of the style of Minsky[24], Chapter 7. In particular, it was proposed that if our system were led through a simulation of the two state Turing machine in Minsky's example 7.3, it should type out a copy of Minsky's universal machine. After some experimentation, it was found

Turing Machine	Set of traces	Enum. Algor.	With Pruning	From Ref.[9]
 AAR BBR XXR XXR BXL BXR AXL AXR XXL XXL	{AXA, AXB, BXA BXB, AAXA, AAXB, ABXA ABXB, BAXA, BAXB BBXA, BBXB, AXAA}	.15*	.23*	13.13
	{ABXBA, AXB, BAXAB}	.05*	.07*	22.58
 XXR AAL BBL XXL AXL BXL XXR _AR _BR AAL BBL XXL AAR BBR	{AAA, AAB, ABA, ABB BBB, BBA, BAB, BAA}	1.15	.45*	
	{AAA, AAB, ABA, ABB BAA, BAB, BBBA}	1.12	.42*	** 317.65
	{BAAABBBAABAB, AABABBABAAB}	3.08	2.18*	
	{A, B, AA, AB, BA, BB, AAA, AAB, ABA, ABB, BAA, BAB, BBB}	1.70	.70*	** 1106.60
	{ABA, BAA, ABB, BAB, BBA, AAB, AAABBB}	1.30	.63*	
 BBL AAR BBR ABL ABR BAR -AL BBR AAL BBR	{A, B, AA, AB, BB, AAA, AAB, ABB, BBB, AAAA, AAAB}	.20*	.22*	151.9*
	{AABB, B, AAABBB}	.10*	.17*	** 96.0*

* Alternate but correct solution found.
** Input traces shown are slightly different from those in [9].

Figure 9. Synthesis times for several Turing machines in seconds.

that this can be accomplished if the universal machine is separated into four subroutines (as Minsky has done) which can each be synthesized individually. The first experiment of this type yielded routine 4 synthesized properly, routines 2 and 3 synthesized with errors, and an unending search (more than several hours) for routine 1. A study of the results of this experiment indicated that the example was too simple to yield a completely general universal machine and that the two leftmoving states in routine 1 were too similar to be distinguished by our system in a reasonable amount of time. Consequently, a slight modification was made to Minsky's example to force the system to produce a completely general machine, and one of the leftmoving states in routine 1 was moved into routine 4 so that it could be synthesized independently. With these changes the universal machine of Figure 10 was constructed easily by our best algorithm as shown in Figure 11. An instruction L or R means move left or right while any other instruction I means "print I". All transitions of the form iL-C-iL or iR-C-iR have been omitted from Figure 10 for the sake of readability. Thus if the head is moving left when a symbol C is read and no corresponding transition is shown, then the same instruction L is executed again. The only item missing from Figure 10 is the transition to the halt instruction which is never encountered in this infinitely long example computation.

This universal Turing machine was constructed from a computation simulating the two state Turing machine shown in Figure 12. The transitions of the machine to be simulated are coded in the form Q S Q' S' D where

Q is the current state,

S is the symbol read,

Q' is the next state,

S' is the symbol printed, and

D is the direction of the move,

0 for a left move and 1 for a right move. Thus the transition in Figure 12 from

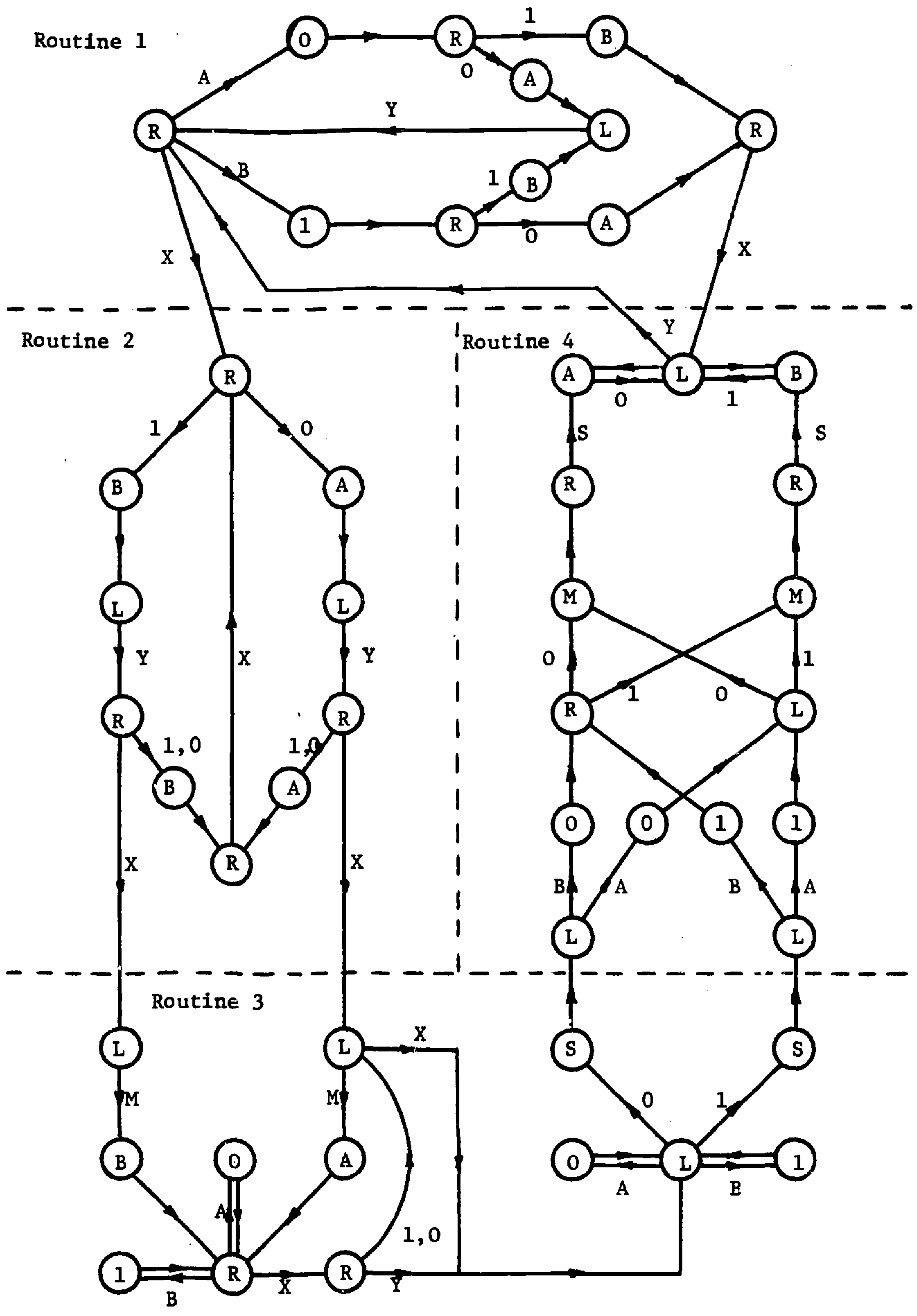
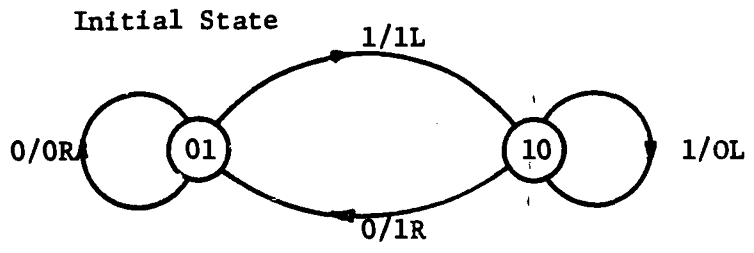


Figure 10. The Minsky universal Turing machine as constructed from traces

Routine	Number of States	Number of Transitions	Number of Simulated moves Required	Total Length of Input Trace	CPU time to synthesize routine (Seconds)
1	11	30	6	846	25.23
2	10	34	4	593	9.43
3	13	36	6	455	9.78
4	15	36	6	321	11.93

Figure 11. Times for synthesis of the four routines of the universal Turing machine



Simulated Machine

$$\underbrace{0000M000}_{\text{Simulated tape}}Y\overbrace{011}^{\text{current state of simulated machine followed by symbol being currently read}}\underbrace{X0111010X1000111X0100101X1011000Y}_{\text{Transitions of the simulated machine}}$$

Initial tape for the simulation

Figure 12. The universal Turing machine was constructed on the basis of six simulated moves of this machine.

state 01, reading symbol 1, going to state 10, printing symbol 1, and moving left is coded 0111010 and given to the universal Turing machine in that form. The initial tape for the example computation which yielded the universal machine was as shown in Figure 12 with the tape to be simulated, the current state and symbol read, and the transitions to be used included. M marks the location of the simulated head and the X's and Y's serve as markers to separate the sequential items on the tape. We can conclude from this experiment that the four routines of the universal Turing machine can be constructed from one sample computation involved six simulated moves and the total CPU time required for the synthesis is less than one minute. This synthesis was completed with our best algorithm using preprocessing and pruning techniques, and none of the four routines could ever be found by our purely enumerative program even after searches of 45 minutes or more.

The performance of a system can sometimes be best judged if it is studied over a class of examples rather than a few individual cases as we have above. Toward this end, we will introduce a grammar-generated class of schema and examine the performance of our programs over a range of problems.

Definitions:

C_i	is an operator which concatenates a program with itself i times. C_i operation on the null program yields a sequence of i copies of the instruction A.
L	is an operator which adds a transition to a program leading from the last state to the first state. The condition associated with this transition is different from any other associated with the last state.
Q	is an operator which works on instructions. If instruction A has any labelled transitions leading away, then Q(A)=Bi where Bi is an instruction different from any other instruction in the program. Otherwise Q(A)=A.

$S(i,j,k)$ is the program which results if Q is applied to every instruction in $\left(L^jC_i\right)^k$.

One of the members of this class, S(3,2,2), appears in Figure 13, and the synthesis times for a whole range of these programs are given in Figure 14. Each solution was constructed from one trace which was generated as if each loop was an ALGOL FOR-loop with an index advancing through values i=1,2. If each loop is executed n times every time it is entered, the length of the trace will be $\left(in^j\right)^k$. Omissions in Figure 14 occur because generated traces were long enough to cause a memory overflow.

Figure 14 seems to indicate that problems of the type S(i,j,k) should be done enumeratively, and that the large overhead associated with preprocessing and maintenance of a failure memory is not worthwhile. However, if we make the problem more difficult by equating all of the B_i's in each S(i,j,k), the conclusion is reversed. ($B=B_1=B_2=$---) Figure 15 shows the results of this modified experiment where the more sophisticated program produced some answers easily that could not be found otherwise after even 45 minutes of searching.

In conclusion, we offer one example of the type of operation that we will expect from our autoprogrammer system. A user seated at the display terminal wishes to sort the sequence of N=4 integers 3,2,1,4 which appears on the screen. He wishes to use a selection sorting method, and he has a number of commands available on the screen which can be referenced with the touch of a light pen. He sets a pointer I to the first item, 3, a pointer J to the second item, 2, notes the two are out of order,interchanges them, increments J, and so forth. With proper hardware and software, we are hoping he will be able to complete the sort in a minute or two. As he goes through these steps, the autoprogrammer will store the sequence of actions shown in Figure 16 and then will output the indicated program. A report on this system will be soon forthcoming by Biermann and Krishnaswamy [13].

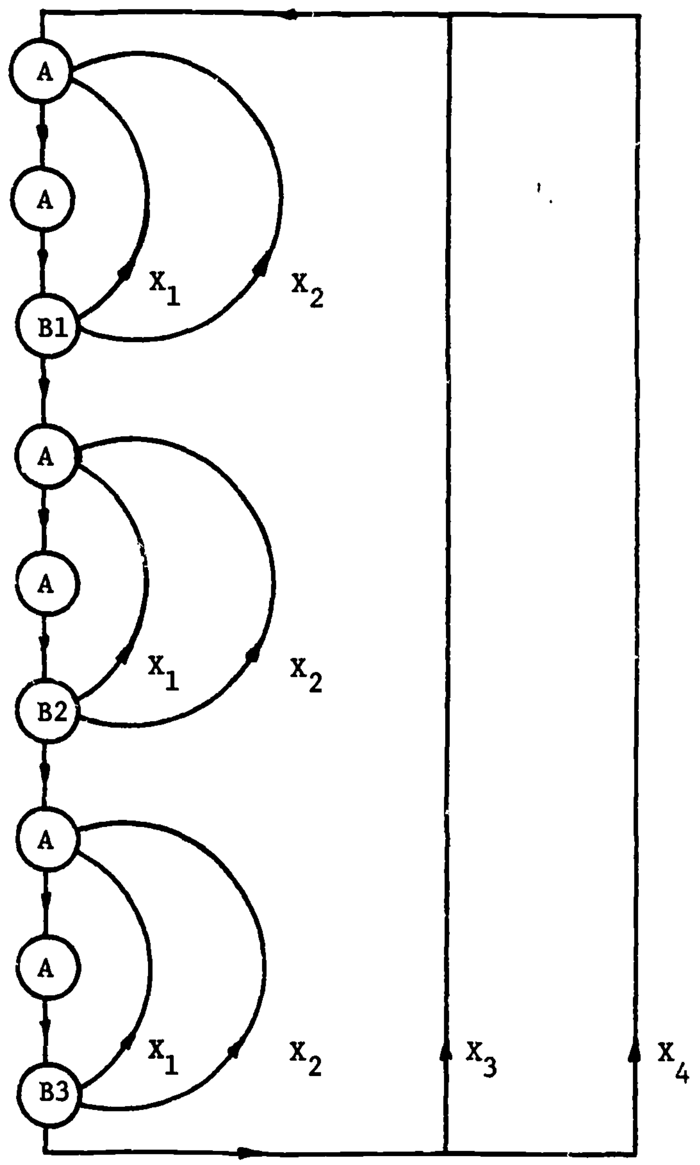


Figure 13. S(3,2,2)

i	j	k=1	k=2	k=3
1	1	.00(.02)	.02(.00)	.00(.00)
1	2	.00(.00)	.02(.02)	.09(.03)
1	3	.00(.02)	.09(.03)	2.32(.23)
1	4	.02(.00)	.67(.12)	
2	1	.00(.00)	.04(.02)	.34(.15)
2	2	.00(.00)	.22(.03)	16.39(1.22)
2	3	.03(.02)	3.04(.15)	
2	4	.02(.02)		
3	1	.00(.00)	.17(.23)	5.92
3	2	.04(.02)	2.49(.73)	
3	3	.07(.02)	(2.65)	
3	4	.20(.03)		
4	1	.00(.02)	.65(7.48)	
4	2	.06(.02)	(76.70)	
4	3	.17(.03)		
4	4	.52(.03)		

Figure 14. Synthesis times in seconds for schema S(i,j,k) using the program of Section III. Times for the purely enumerative algorithm are parenthesized.

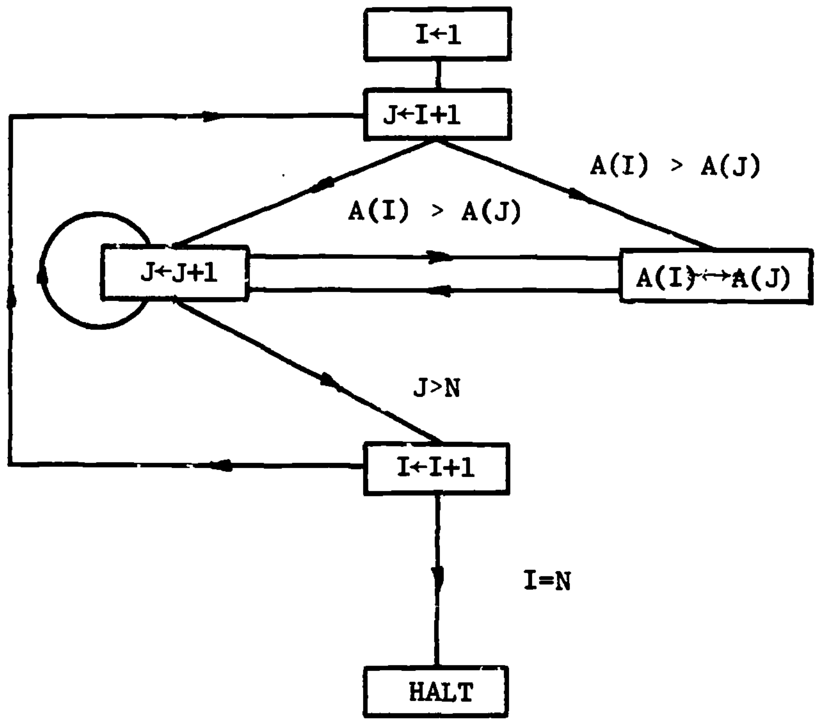
i	j	k=2	k=3	
2	1	.05(.08)	.62(45.55)	*
2	2	.28(1.25)	6.15	*
2	3	2.58(20.22)		
3	1	.73(16.3)		
3	2	122.95(> 45 min.)		
4	1	8.08(> 45 min.)		

* Alternate solution found because of inadequate trace.

Figure 15. Synthesis times in seconds for schema S(i,j,k) after all B_i's have been equated. Times for the purely enumerative algorithm are parenthesized.

	I←1
	J←I+1
A(I)>A(J)	A(I)↔A(J)
	J←J+1
A(I)>A(J)	A(I)↔A(J)
	J←J+1
	J←J+1
J>N	I←I+1
	J←I+1
A(I)>A(J)	A(I)↔A(J)
	J←J+1
	J←J+1
J>N	I←I+1
	J←I+1
	J←J+1
J>N	I←I+1
I=N	HALT

The Trace



The Program

Figure 16. An autoprogrammer example. Constructing a selection sort program.

Finally it should be said that the computation times given in this section should be considered to be indicators of comparative algorithm efficiency and not absolute measures of any kind. Any of the programs discussed here could obviously be improved and the times given indicate what can be attained with moderately careful programming.

VI. CONCLUSION

The experiments described here demonstrate that programs of significant complexity can be synthesized in reasonable time from instruction traces. We do not expect that our autoprogramming system will be greatly burdened by long waits for the completion of a synthesis. However, it may be that this work will have broader significance because it provides a benchmark for those individuals who are interested in program synthesis from much weaker information. If our system with its very complete input information has difficulty in doing a construction, one can expect a vastly more difficult task in doing the synthesis from simply input-output pairs, sparse traces, or formal specifications on desired performance.

VII ACKNOWLEDGMENT

The first author is greatly indebted to Professor J. A. Feldman for comments and criticisms of this research during the summer of 1972. We are also similarly indebted to Professor C. Green.

The research reported here was done partly in the Computer Science Department, Stanford University and was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183) and the National Science Foundation Grant No. GJ-776. Part of the work was done in the Computer and Information Science Department at The Ohio State University and was supported by Grant Numbers GN-534.1 and GJ-34739X from the National Science Foundation.

This paper was typed by Roberta Harrison, Cynthia Karr, and Sarah Sieling.

REFERENCES

[1] S. Amarel, "On the Automatic Formation of a Computer Program which Represents a Theory" in Self Organizing Systems - 1962 (Yovits, Jacobi, and Goldstein, Eds.) Spartan Books, New York, 1962.

[2] S. Amarel, "Representations and Modelling in Problems of Program Formation" Machine Intelligence 6 (Meltzer and Michie, Eds.) American Elsevier Publishing Company, Inc. New York 1971.

[3] D. D. Aufenkamp, "Analysis of Sequential Machines, II," IRE Transactions on Electronic Computers, Vol. EC-7, pp. 299-306, December, 1958.

[4] D. D. Aufenkamp and F. E. Hohn, "Analysis of Sequential Machines," IRE Transactions on Electronic Computers, Vol. EC-6, pp. 276-285, December, 1957.

[5] R. Balzer, "A Global View of Automatic Programming", Third International Joint Conference on Artificial Intelligence, Stanford, California, August 20 to 24, 1973.

[6] J. Barzdin, "On Synthesizing Programs Given by Examples", Computing Center, Latvian State University, Riga, U.S.S.R.

[7] J. Barzdin and R. V. Freivald, "On the Prediction of General Recursive Functions", Soviet Math. Dokl., vol. 13, no. 5, 1972.

[8] L. Blum and M. Blum, "Inductive Inference: A Recursion Theoretic Approach", Recursive Function Theory Newsletter, no. 6, July, 1973.

[9] A. W. Biermann, "On the Inference of Turing Machines from Sample Computations", Artificial Intelligence 3, 1972.

[10] A. W. Biermann, "An Interactive Finite-State Language Learner", USA-Japan Computer Conference, Tokyo, Japan, October 3-5, 1972.

[11] A. W. Biermann, "Computer Program Synthesis from Computation Traces", Symposium on Fundamental Theory of Programming, Kyoto University, Kyoto, Japan, October 9-11, 1972.

[12] A. W. Biermann and J. A. Feldman, "A Survey of Results in Grammatical Inference", Frontiers in Pattern Recognition (Watanabe, ed.), Academic Press, New York, 1972.

[13] A. W. Biermann and R. Krishnaswamy, "A System for Program Synthesis from Examples," in preparation.

[14] S. Crespi-Reghizzi, M. A. Melkanoff, and L. Lichten," The Use of Grammatical Inference for Designing Programming Languages," Communications of the ACM, 16 No. 2, Feb. 1973.

[15] H. Enomoto, E. Tomita, and S. Doshita, "Synthesis of Automata that Recognize Given Strings and Characterization of Automata by Representative, "USA-Japan Computer Conference, Tokyo, Japan, October 3-5, 1972.

[16] J. A. Feldman, "Automatic Programming" Computer Science Dept. Report no. 255, Stanford Univeristy, Feb. 1972.

[17] J. A. Feldman, "Some Decidability Results on Grammatical Inference and Complexity", Information and Control, 1972.

[18] J. A. Feldman and P. C. Shields, "Total Complexity and the Inference of Best Programs", AIM-159, Computer Science Department, Stanford University, April, 1972.

[19] Arthur Gill, "Realization of Input-Output Relations by Sequential Machines," Journal of the Association for Computing Machinery, Volume 13, pp. 33-42, January, 1966.

[20] Seymour Ginsburg, "A Technique for the Reduction of a Given Machine to a Minimal-State Machine," IRE Transactions on Electronic Computers, Vol. EC-8, pp. 346-355, September, 1959.

[21] Seymour Ginsburg, "Synthesis of Minimal-State Machines," IRE Transactions on Electronic Computers, Vol. EC-8, pp. 441-449, December, 1959.

[22] A Grasselli and F. Luccio, "A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks," IEEE Transactions on Electronic Computers, Vol. EC-14, pp. 350-359, June, 1965.

[23] Z. Manna and R. J. Waldinger, "Toward Automatic Program Synthesis", Communications of the ACM 14, no. 3, 1971.

[24] M. Minsky, Computation: Finite and Infinite Machines, Prentice-Hall Inc. Englewood Cliffs, N. J., 1967.

[25] M. C. Paull and S. H. Unger, "Minimizing the Number of States in Incompletely Specified Sequential Switching Functions," IRE Transactions on Electronic Computers, Vol. EC-8, pp. 356-367, September, 1959.

[26] P. Raulefs, "Autômatic Synthesis of Minimal Algorithms from Samples of the Behavior", Institut fur Informatik, Report INF I-73-2, Universitat Karlsruhe, Germany, Jan., 1973.

[27] H. A. Simon, "Experiments with a Heuristic Compiler", Journal of the ACM, Oct., 1963.

[28] A. A. Tal, "Questionaire Language and Abstract Synthesis of Minimum Sequential Machines," Avtomatika i Telemekhanika 25, pp. 946-962, 1964.

[29] R. J. Waldinger and R. C. T. Lee, "PROW: A Step Toward Automatic Program Writing", Proceedings of the Internation Joint Conference on Artificial Intelligence, Washington, D. C. 1969.

APPENDIX 1

We will give here another example in the style of Section III. In this synthesis, we see the algorithm search for and fail to find a solution. Then it creates a free state and succeeds.

Event	Level	Discussion
0		initiate synthesis, F=0 and FL=0
1-3	1,2,3	assignment
4,5	17,14	DV_3 resolution, S^{14} and S^{17} are now pseudo assigned
6	8	look ahead fence prevention at 17 from 8
7,8	4,5	assignment
9	12	DV_5 resolution
10	11	dynamic nonequivalence via CC_1, this implies that $\|[A]\|>1$, this is a contradiction, so .....
11	11,12	the assignment $S^5=1$ is incorrect. Invalidate $FM_{10,1}$ and $FM_{11,1}$
12	5	assignment
13	12	DV_5 resolution
14,15	6,7	assignment
16	8	S^8 is forced to be 2, this is a contradiction since $FM_{8,2}$ is valid, so .....
17	8,12,14,17	Initiate backup: S^7 and S^6 cannot be changed since $L_E=L_C=1$ and F=0; S^5 cannot be changed since $L_B=2$ and F=0; S^4 cannot be changed since $L_C=1$ and F=0; S^3 cannot be changed since this level represents the first occurrence of [B] and so changing S^3 to 2 would simply be a renaming of an existing state. S^2 and S^1 cannot be changed since $L_A=L_S=1$ and F=0. At this point we have backed up to the first level of the trace so Invalidate all failure memory entries, set FL=1, F=0 and initiate synthesis again
18-20	1,2,3	assignment
21,22	17,14	DV_3 resolution
23,24	4,5	assignment
25	12	DV_5 resolution
26	11	dynamic nonequivalence via CC_1, since this implies $\|[A]\|>1$ we must use available free state to allow $\|[A]\|=2$. Now F=0.
27	8	since F is now zero level 17 (or 14) can be pseudo assigned the value 2, thus $FM_{8,2}$ is made valid via the look ahead fence prevention mechanism.
28-42	6-20	assignment/forced move, synthesis complete

Figure A1. Event table for example synthesis.

$I = \{S,A,B,C,D,E,F\}$

Level	N_i	O_i	Couple-Class	Difference Set	Selector Values		FM_{*1}	FM_{*2}	Level
1	-	S	.		1(1)	1(18)			1
2	X	A	.		1(2)	1(19)			2
3	X	B	CC_1	{17,14}	1(3)	1(20)			3
4	X	C	CC_6		1(7)	1(23)			4
5	X	B	CC_2	{12}	1(8),2(12)	1(24)			5
6	Y	E	.		1(14)	1(28)			6
7	X	C	.		2(15)	1(29)			7
8	X	B	CC_2	{17,14}	.2(16)	.1(30)		$3_1(6),0(17),5_2(27)$	8
9	X	C	CC_6			.1(31)			9
10	Y	S	.			1(32)			10
11	Y	A	.			2(33)	$5_1(10),0(11),5_2(26)$		11
12	X	B	CC_1			2(34)	$5_1(9)\ ,0(11),5_2(25)$	$5_1(13),0(17)$	12
13	Y	D	.			1(35)			13
14	X	B	CC_3			2(36)	$3_1(5),0(17),3_2(22)$		14
15	X	F	CC_4			1(37)			15
16	X	D	CC_5			1(38)			16
17	X	B	CC_3			.2(39)	$3_1(4),0(17),3_2(21)$		17
18	X	F	CC_4			.1(40)			18
19	X	D	CC_5			.1(41)			19
20	Y	S	.			1(42)			20

$[A\text{-}X\text{-}B] = CC_1 = \{3,12\}$

$[C\text{-}X\text{-}B] = CC_2 = \{5,8\}$

$[D\text{-}X\text{-}B] = CC_3 = \{14,17\}$

$[B\text{-}X\text{-}F] = CC_4 = \{15,18\}$

$[F\text{-}X\text{-}D] = CC_5 = \{16,19\}$

$[B\text{-}X\text{-}C] = CC_6 = \{4,9\}$

$L_S = 1$	$L_D = 1$
$L_A = 1$	$L_E = 1$
$L_B = 2$	$L_F = 1$
$L_C = 1$	

Figure A2. Trace Table for Example Synthesis

APPENDIX 2

In order to study the effect of parallel processing in our enumerative algorithm, a large number of tests were run on the second problem in Figure 9. Figure A3 shows the results of these experiments where the number of modes expanded on the search tree are plotted versus the number of input traces with the total of the lengths of the traces indicated beside each point. The times for most of the runs (including input-output operations) were in the range of 2 to 5 seconds on a IBM 370/165 with the longest run requiring 9.5 seconds to do a synthesis from 60 traces. Thus the overhead involved in the parallel computation becomes significant indicating that there is probably an optimum number of traces for synthesizing this machine. It can be shown that the Moore form of this machine cannot possibly be constructed from only one trace, and considering the uncircled points only, two or three traces did not yield a quick synthesis. However, four or five traces seemed to be quite sufficient and additional traces helped none whatsoever.

The circled points indicate that a second (equivalent) version of the Turing machine was synthesized and this occurred if most of the traces processed two or more B's before processing A's. Thus the performance was heavily dependent on the first few transitions of each trace.

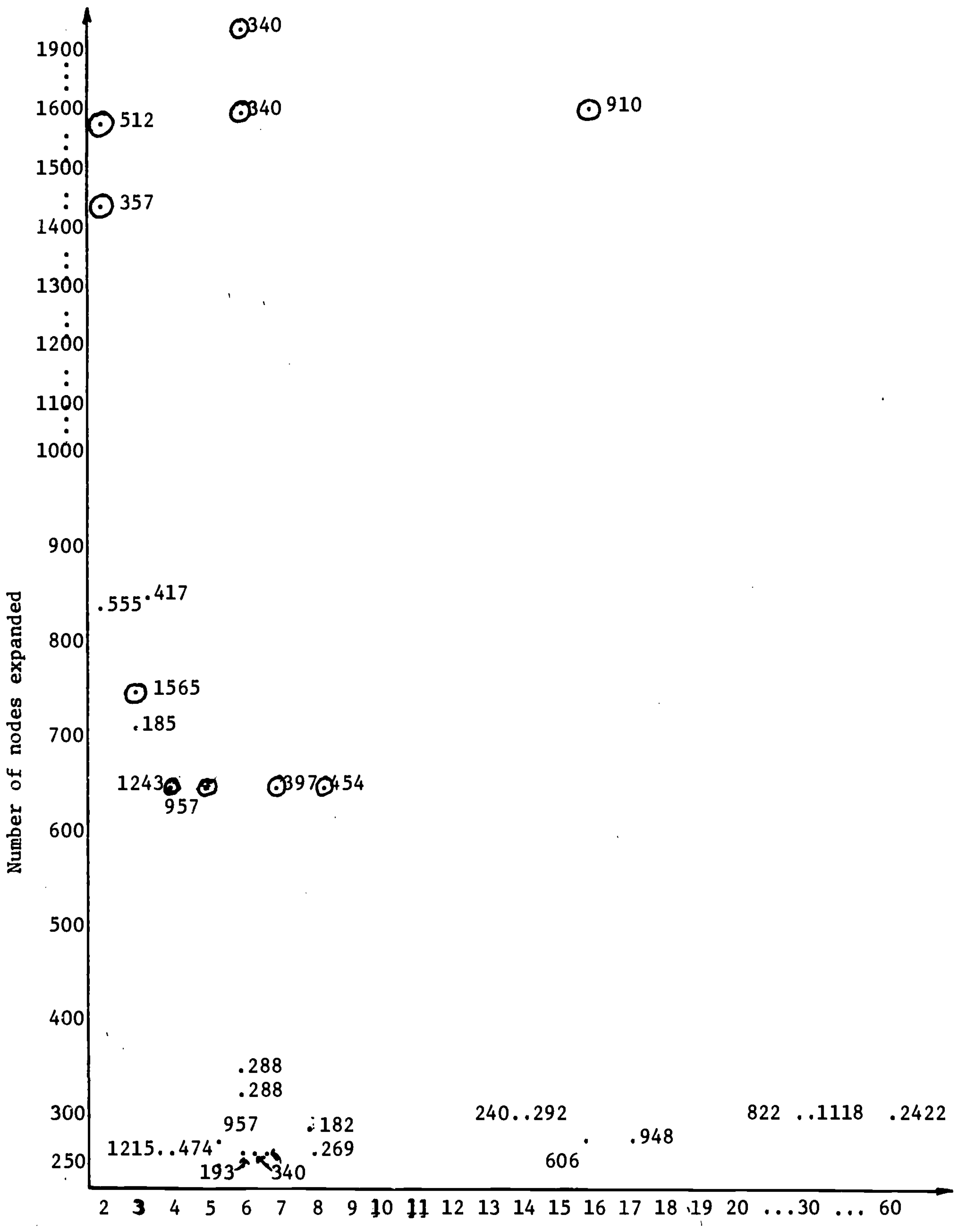


Figure A3. The usefulness of parallelism in enumerative processing.

A System for Program Synthesis from Examples

A. W. Biermann and R. Krishnaswamy

Department of Computer and Information Science
The Ohio State University

August 1973

I. INTRODUCTION

An autoprogrammer is an interactive computer programming system which accepts as input example calculations and which yields computer programs for doing those calculations. Such a system provides the user with a sort of scratch pad and command system for executing the examples, and it synthesizes programs on the basis of a recorded history of the steps required to do those computations. This paper will briefly describe an autoprogramming system which is currently under development by the authors. The reader will find more details in [1], [2], and later reports.

II. THE LANGUAGE

Every program created by the system is a subroutine with a name which can be called from any subroutine including itself or which can act as a top-level main program. A subroutine is created by

(1) declaring the name of the subroutine,

(2) declaring the data structures (arrays and variables) to be referenced including information about which are arguments for the subroutine, which will be held in common with other subroutines, and which variables will be pointers into arrays, and

(3) executing one or more example calculations using the autoprogramming language.

After steps (1) and (2) above are completed, the declared arrays and variables appear on a display screen properly labelled and with all entries set to zero. Labelled arrows appear in the arrays resulting from pointer declarations in (2). At this point, a sample computation can be performed. A light pen at the display screen is used to reference the commands of the language which also appear on the screen and their operands among the data structures.

Let p_i stand for the i-th operand touched by the light pen after a command is referenced. Then the commands of the language are:

S	start - the first instruction in a subroutine.
H	halt - terminate computation and return to the calling routine.
+	add - $p_3 \leftarrow p_1+p_2$ or $p_2 \leftarrow p_1+p_2$.
-	subtract - $p_3 \leftarrow p_2-p_1$ or $p_2 \leftarrow p_2-p_1$
*	multiply - $p_3 \leftarrow p_1*p_2$ or $p_2 \leftarrow p_1*p_2$.
/	divide - $p_3 \leftarrow p_2/p_1$ or $p_2 \leftarrow p_2/p_1$.

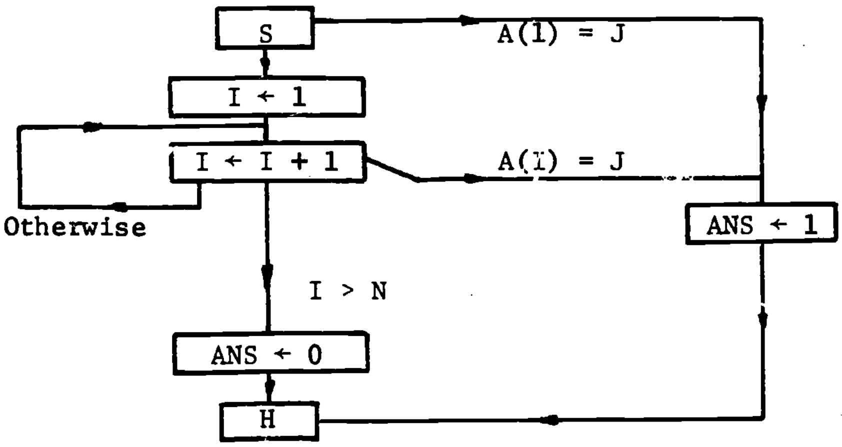
M	move - $p_2 \leftarrow p_1$.
R	read - p_1 gets the next integer typed on the teletype. If two operands in an array $e_{i,j}$ and $e_{m,n}$ are touched, all entries $e_{x,y}$ in the array are read such that $i \leq x \leq m$ and $j \leq y \leq n$.
W	write - outputs integers to the teletype using conventions as in read, R.
T	type - outputs a specific character string to the teletype.
C	call subroutine - p_1 is the subroutine name being called and p_i for $i > 1$ are the arguments to be referenced.
?	conditional - note that the relationship $p_1 p_2 p_3$ holds where p_2 is either =,>, or<.

The only data type currently available is "integer". Some of the commands may take varying numbers of operands and their interpretation depends on how many are given. For example, the result of an arithmetic instruction is left in p_3 if three operands are given and in p_2 if only two are given. The digits 0,1,2,.....,9 appear on the screen for the generation of literals. Multiple digit literals are generated by a sequence of light pen hits on these digits.

One can best understand the operation of the system by considering how a few simple examples of a linear search might be converted into a program. Suppose we search the array A=(1,4.2) with N=3 elements for the elements J = 1,2, and 3. Then we declare the array, enter some typical values (1,4,2), and use the light pen commands to advance the pointer across the array until the item is found. If it is found, we return ANS $\leftarrow$ 1, otherwise ANS$\leftarrow$ 0. As each calculation proceeds, the system stores the sequence of actions performed as shown in Figure 1 and then

S	S	S
(A(1)=J)	$I \leftarrow 1$	$I \leftarrow 1$
ANS $\leftarrow$ 1	$I \leftarrow I + 1$	$I \leftarrow I + 1$
H	$I \leftarrow I + 1$	$I \leftarrow I + 1$
	(A(I) = J)	$I \leftarrow I + 1$
	ANS $\leftarrow$ 1	(I > N)
	H	ANS $\leftarrow$ 0
		H
J=1	J=2	J=3

The commands generated in searching array A for J = 1,2,3



The shortest program compatible with the traces.

Figure 1. Constructing a program for a linear search.

constructs the shortest program which is compatible with these traces. It can be shown that every possible program (or its equivalent) can be so constructed from a finite set of traces, and the art of such synthesis is described in some detail in [1]. In this case, the program appearing in Figure 1 is correct for linear searching and can be constructed from traces in a fraction of a second.

If a program synthesized on the basis of one or several traces is tested and proven to be incorrect, one or more additional sample calculations may be input to remove the bugs. The synthesis algorithm is capable of program construction from pieces of traces as well as complete traces. This means that if a program is correct except for some small part, a portion of the calculation involving just that part may be completed and input as a correction to the program. The synthesized program is the shortest program which is compatible with the given traces and so is guaranteed to execute those examples correctly. Thus if the examples are correct, a program with errors will be constructed only if one exists which is compatible with the traces and has length less than or equal to the correct program.

The fact that subroutines can be called while executing a sample calculation means that large programs can be constructed using many smaller building blocks. A subroutine called in this way presumably exists from an earlier synthesis and is easily available. If it does not exist, the user is asked to supply the result that it would have yielded if it did exist! Thus the sample calculation can proceed whether or not the subroutines upon which it depends have already been constructed. Clearly, this leaves the possibility open for so-called "top-down" programming.

All arrays and variables for all subroutines are stored in a hash table where the keys are computed from a combination of the name, the associated subroutine name, and the level of the subroutine call. This means that

(1) arrays are effectively infinitely large as long as the hash table does not overflow so that no dimensions are associated with arrays, and

(2) subroutines may be called recursively.

A "common" feature is available for transmittal of information between subroutines.

Another feature to be included will be the "continue" feature. If a sample calculation is quite long and repetitive, there is a good chance that part or all of the program can be correctly constructed before the sample calculation is completed. Then a "continue" key may be touched midway through the sample causing a program synthesis on just the portion of the trace which is available. Subsequent inputs through the "continue" key will each cause another instruction from the synthesized program to be executed. If the program yields an error, a "backup" key will be available allowing the last one or several instructions to be "unexecuted", the corrected instructions may be inserted with the light pen, and the calculation can continue. As an example, if a column of ten integers is to be added and then printed, the user may add two or three with the light pen to cause a program with a loop to be constructed.

Then he may hit the continue key repeatedly until all are added, and finally insert (again with the light pen) the instruction to do the printing. The process of trace construction should not, therefore, be thought of as simply a long process of light pen work. The light pen will be used alternately with the continue key and backup key to allow the easiest and most carefree possible completion of examples.

A great deal of effort has gone into making the system easy and comfortable to use. The user may input instructions at any rate he desires without significant delays for internal processing. Each hit of the light pen is acknowledged by a momentary (one second) disappearance of the designated image. Erroneous light pen hits which result in illegal instructions cause a message to be output at the teletype, the illegal instruction is deleted, and the user may repeat the instruction without any special action. The sensitized points on the screen are well separated so that incorrect hits are not common. Subroutines may be created or called at any time in a carefree manner. If in the process of doing a computation it is desired to declare a new array or pointer, this can be done at any time without interruption of the current process. It is hoped that the system will be so easy to use that anyone will be able to run it with less than an hour instruction.

III. THE CURRENT STATUS OF THE SYSTEM

As already mentioned, this system is still under development and not all of the features given above are operative. The three major parts of the program, the light pen and display routines, the language interpreter, and the program synthesizer are all running individually and debugged. The program synthesizer has been extensively tested for several months as described in [1]. The coordinator of these programs and the interface routines are still only partially operative but should be completed within a few months.

The photograph of Figure 2 shows a user with light pen referencing an array on the display screen. The display and keyboard at right are used for array declarations, input-output for the synthesized program, error messages, and so forth. Figure 3 shows the display screen after the declarations have been made for a matrix multiply calculation. The autoprogrammer instructions appear in a list at the right, and the ten digits for literal generation are at the bottom of the screen. Figure 4 shows the screen at the end of the computation, and Figure 5 gives the synthesized program produced after about 50 milliseconds of internal synthesis time. Many other programs of similar complexity have also been synthesized by the system.

Figure 2. The second author prepares to execute a matrix multiplication.

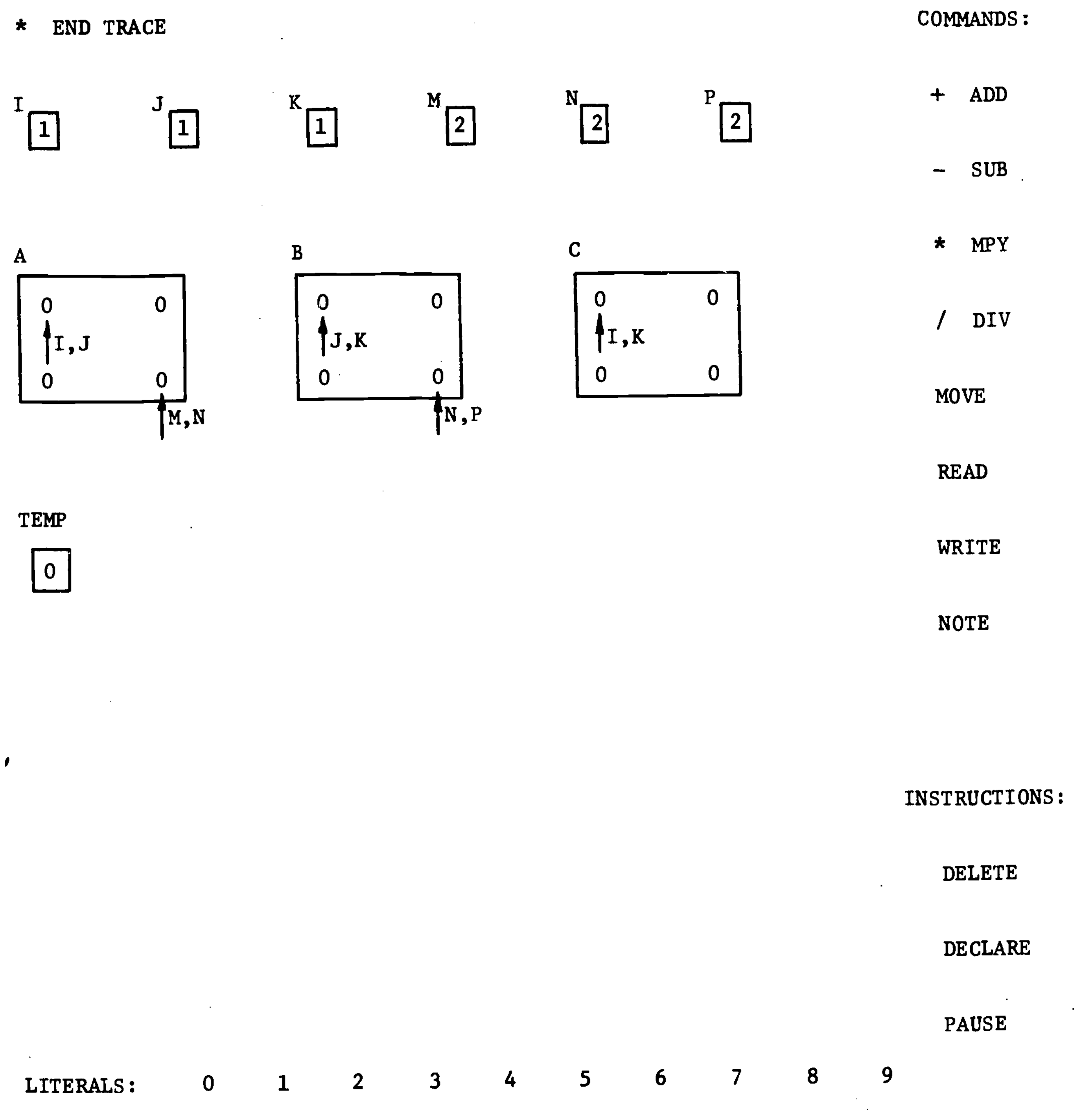


Figure 3. The autoprogrammer display before executing a matrix multiply calculation

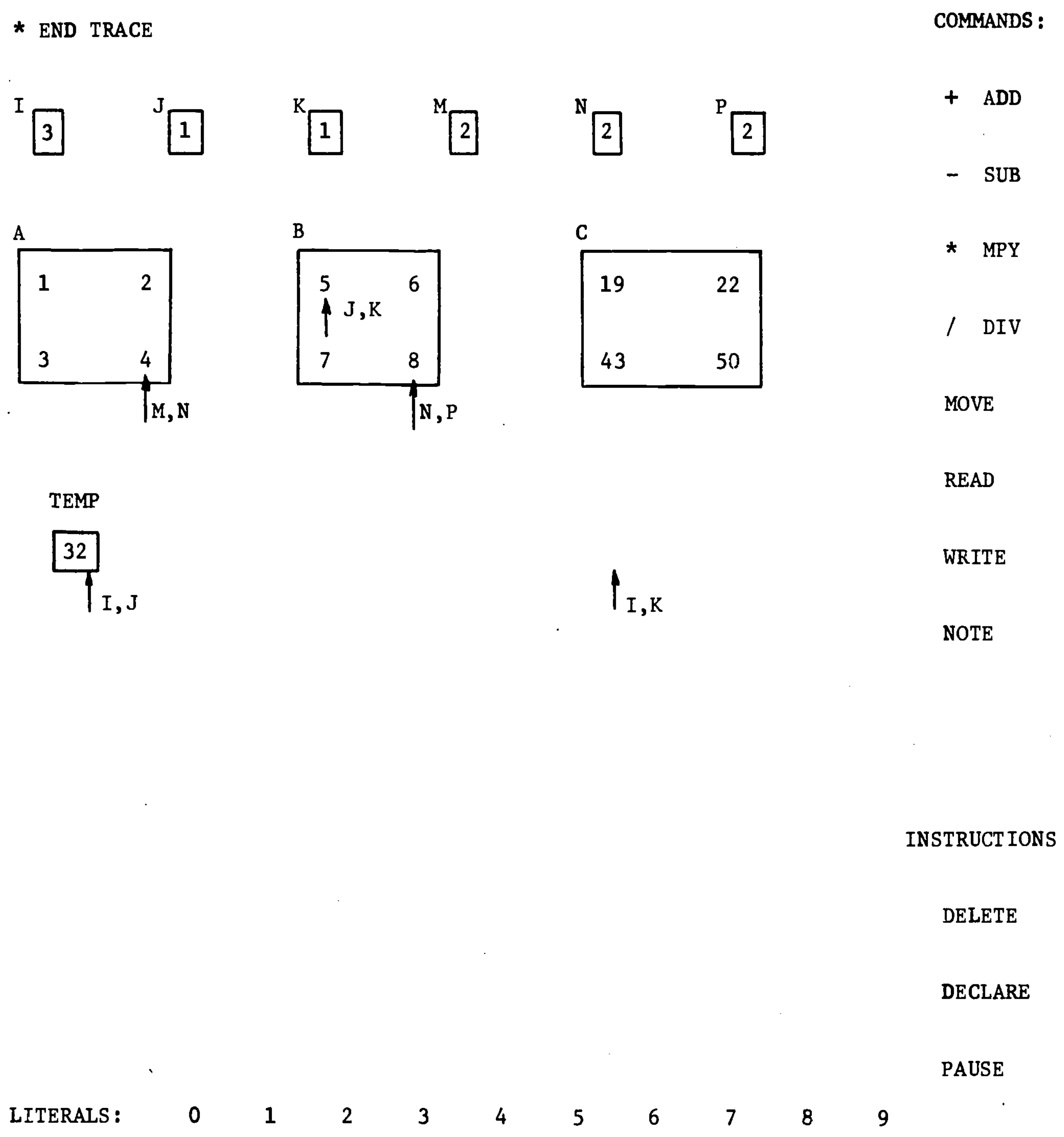


Figure 4. The display after the calculation has been completed

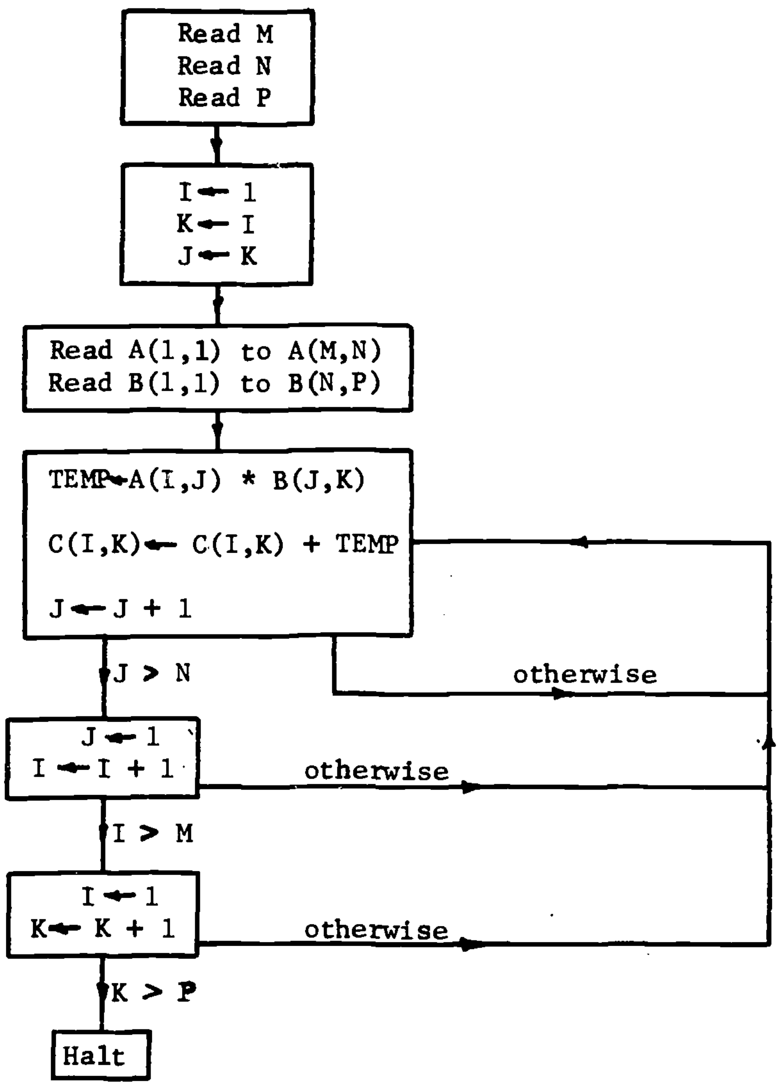


Figure 5. The synthesized program for multiplcation of an M by N matrix times an N by P matrix.

IV. SOME PROBLEMS

Light pens are inherently very slow devices and will never be adequate input devices for autoprogrammer systems. A more ideal hardware arrangement would be to have a touch-sensitive display screen mounted down flush with a desk top. Then the user's fingers could leap around on the screen at the speed possible on a modern desk calculator inputting individual hits at the rate of several per second. There is every reason to believe that a person can think fast enough to achieve that rate which is nearly an order of magnitude faster than possible with a light pen. However, we feel that our light pen driven system will clearly show the value of the autoprogrammer idea and will constitute a high quality basis for further work.

The size of the display screen is also an important limiting factor. It is desirable to have all the current data structures visible at one time but our screen is too small for the display of many arrays. This can be solved with windowing features, the use of multiple displays, and the dependence on subroutines, but all of these alternatives have difficulties. Windowing features are a bother for the user, multiple displays are expensive, and it is not always easy to hide a significant number of arrays in subroutines. Fortunately, programs can be synthesized on the basis of relatively small samples so that the size of displayed arrays need not be large.

Finally, we are concerned about the ability of humans to produce correct example computations. Our system moves the debugging problem from the domain of language syntax and semantics to the domain of examples, but the possibility for human error remains. If the example calculation cannot be generalized into a correct program, certainly our system cannot succeed. For example, if we want a program which inputs a prime number and outputs the next sequential prime number, we cannot offer as a sample computation: input the integer 11, add 2, print the result. The example calculation must execute the instructions of some correct program in the order that the correct program would execute them. Only if this constraint is met will that program be found. A central theme of our current research is to weaken this requirement.

V. ACKNOWLEDGMENT

Messrs. Richard Baum and Frederick Petry have been in charge of developing our synthesis algorithms. We are greatly indebted to Mr. Serge Fournier of our PDP-10 staff for help with our systems programming problems. We would like to thank Dr. Robert Mathis for photographing our display output for inclusion in this report.

This work was supported by grant number GJ-34739X from the National Science Foundation and by the Department of Computer and Information Science, The Ohio State University.

VI. REFERENCES

[1] A.W. Biermann, R. Baum, and F.E. Petry, "Speeding Up a Program Synthesizer," In preparation.

[2] A.W. Biermann, "On the Inference of Turing Machines from Sample Computations, Artificial Intelligence, Volume 3, 1972.