Title	Computer Program Synthesis From Computation Traces (プログラムの基礎理論)
Author(s)	BIERMANN, A.W.
Citation	数理解析研究所講究録 (1973), 189: 89-100
Issue Date	1973-10
URL	http://hdl.handle.net/2433/107218
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

数理解析研究所講究録
189 巻 1973 年 89-100

COMPUTER PROGRAM SYNTHESIS FROM COMPUTATION TRACES

A.W. Biermann

presented at the
Symposium on
Fundamental Theory of Programming

October 9 to 11, 1972

Kyoto University
Kyoto, Japan

I. INTRODUCTION

A computer programmer with an algorithm in mind always goes through approximately the same process in obtaining an execution of the algorithm from a machine. That is, he writes a program in some language, debugs it by whatever means possible, and, at length, obtains a program which effectively executes his algorithm. This paper will propose a new method for obtaining the desired program and will show how that method can be implemented.

Specifically, we will propose that the programmer present the machine with one or several examples of the desired computation and have the machine synthesize the program from these examples. We know that if one human is to describe a process to another, he will very likely give an example rather than convey the idea with a series of general definitions and usages of those definitions. Thus a person would probably teach another to do long division by writing down several examples and working them out step-by-step with proper explanation. He would probably have a great deal more difficulty in conveying the idea if he restricted himself to X's and Y's, conditional statements, branches, and so forth. Even though the procedure is probably stored in each human's mind in a very general form, the communication of the procedure from one person to another is usually either partially or completely done using examples. The learner uses his inductive abilities to synthesize the general concept by observing the examples. This form of communication seems to be very natural and easy for humans and perhaps should be employed when communicating with machines as well.

A theory of inductive inference has begun to develop over the past several years (6, 7, 8, 9, 10, 11, 15, 16, 19) leading one to wonder whether machines, indeed, could conceivably construct their own programs from example computations. In order for such program synthesis to become practical, it is necessary to

(1) develop a system through which the programmer can naturally construct examples for transference to the machine and

(2) develop an algorithm which the machine can use to synthesize the program from the example.

We will now consider each of these problems.

II. PROGRAMMER SYNTHESIS OF EXAMPLES

Many hardware and software methods could be used to enable a person to synthesize example computations. The example might be worked and typed up on cards in the manner that programs are often written. Such an example computation might look like this: read the number to be divided, 230; read the divisor, 11; shift the 2 into register C; contents of register C is too small: shift the 3 into register C to get 23; put a zero in register D; subtract 11 from C to get 12 and increment D; subtract 11 from C to get 1 and increment D; contents of register C is too small: shift the 0 into register C to get 10; etc. This approach seems too cumbersome so we might propose a machine similar to a desk calculator with one or two keys corresponding to each operation. Thus to do the example, the programmer might simply push a sequence of keys: R_A,230,R_B,11 C←C&L(A),B≥C:C←C&L(A),D←D&"0",C←C-B,D←D+1,C←C-B,D←D+1--- etc. All of the registers would be visible to the user, and he would be able to see the complete computation progress as he sequentially touches the keys.

The best implementation would be a general purpose computer with a display unit and advanced input devices such as a light pen or touch sensitive screen. Here the user would be able to manipulate the data at least as easily as with paper and pencil, and would, hopefully, be able to go through an example with some facility.

The result of such a sequence of manipulations would be a computation trace, a complete record of the actions required to do the particular calculation. A computation trace consists of two types of elements, operation commands and conditions, and these concepts are illustrated in Figure 1.

Condition	Command	Results A	B	C	D
	R_A	230	-	-	-
	R_B	230	11	-	-
	C ← C&L(A)	30	11	2	-
B > C	C ← C&L(A)	0	11	23	-
	D ← D&"0"	0	11	23	0
	C ← C - B	0	11	12	0
	D ← D + 1	0	11	12	1
	C ← C - B	0	11	1	1
	D ← D + 1	0	11	1	2
B > C	C ← C&L(A)	-	11	10	2
	D ← D&"0"	-	11	10	20

Figure 1. A computation trace.

After one or several such traces are produced, the machine must produce a general program which is capable of doing all such computations. In this example, the program shown in Figure 2 might be constructed. We will consider in the next section exactly how such a program can be constructed from computation traces.

III. PROGRAM SYNTHESIS

We will model a computation trace with a string of condition-command pairs with the understanding that the first element will often be the trivial condition: no conditional test at all. Thus the problem is to find for a string of condition-command pairs, a program which is compatible with the string (See Figure 3). That is, the string must correspond to some path through the program. If a set of traces are given, each trace must correspond to a path through the program. There may be a large number of compatible programs for a given set of traces, and the strategy will be to find the simplest possible such program. Previous experience (4) with this type of synthesis indicates that only a few traces are usually needed in order to produce the desired program.

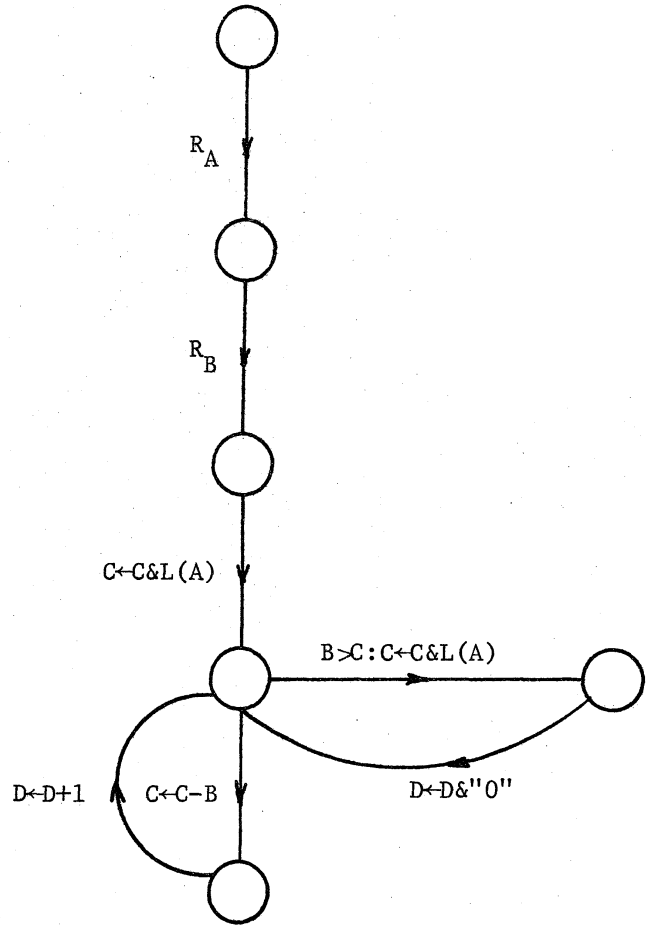

Figure 2. A synthesized program.

Condition	Command
A	Z
B	B
A	R
B	A
A	R
C	A
B	B
C	Z
C	Halt

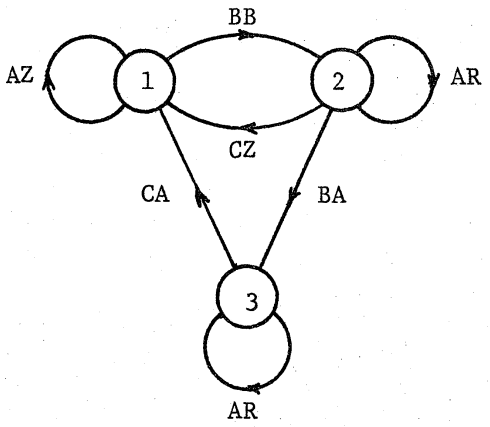

Figure 3. The problem: Find the program at right from the trace on the left.

An algorithm for solving this problem was given in (4) but that method was found to be inefficient for synthesizing complex flow diagrams with five or more nodes. In that paper, Turing machine controllers with three states could be constructed in several seconds of computer time, controllers with four states could be constructed in several minutes, and larger controllers often required even more time. During the past year, two methods have been under study for increasing the power and speed of the algorithm. The first technique involves the parallel usage of multiple traces and will be described here. The second technique involves a method for efficiently pruning the search tree, and this will be described in a later paper. Both techniques are capable of speeding up the synthesis process by orders of magnitude as shown by some recent experiments.

We will therefore describe a simple algorithm for synthesis of program flow diagrams from multiple traces which processes the traces in parallel. Consistent with our assertions above, we will communicate the algorithm by giving an example and avoid introducing innumerable definitions and abstract notations. The complete formalism for description of this algorithm does exist but it is not clear that the reader would like to see it.

The example is worked out in Figure 4 where the problem is to find the simplest possible flow diagram which is consistent with the seven given traces. Each trace consists of a string of six condition-operation pairs ending with the operation H, halt. A termination pair CH, for example, indicates that the program in its current state under condition C yields no operation or transition to another state. The problem is to discover the sequence of states traversed by each trace starting with state 1 and ending with the proper halting condition. Thus in Figure 4(a) the seven traces are shown with a 1 preceding the first transition of each trace indicating that each computation begins in state 1.

Studying Figure 4(a), we see that when the program is in state 1, condition A is followed by operation Z and condition B is followed by operation B.

The problem is to guess correctly which state the program will be in after each transition. We will always choose the first condition by some ordering technique (in this case alphabetical order), and thus we will attempt to guess what the state will be after transition AZ from state 1. We will always guess the lowest numbered state which has not been previously found to be in contradiction. Here we guess the next state will be 1 which means the program will have a transition 1 AZ 1 as shown in Figure 4(b). The parenthesized state in trace 2 indicates that it is in this state due to a transition constructed at an earlier time.

Now each trace is in state 1 with a B condition and a B operation indicated. If we again make the simplest possible guess that the next state is 1, we will have a contradiction in trace 1. Here it is apparent that if BB yields state 1, then state 1 under condition B must yield operation A which is in contradiction with the 1 BB 1 transition. So state 2 must be the next state as shown in Figure 4(c).

The reader may wish to follow each of the succeeding parts of Figure 4 and see what contradictions arise and why the resulting transitions are constructed. Before the algorithm is started, the maximum number k of states to be allowed is set. If at some point in the search, all of the possible state choices 1, 2, 3, ..., k are found to yield contradictions, the search is "backed up", a previous decision is changed, and the search is continued. The resulting program flow diagram is shown in Figure 3.

This algorithm and other similar algorithms have been programmed and tested extensively. These tests indicate that small but practical programs can be constructed by this technique with a few seconds of computer time on the basis of a small number of traces (usually only one). A simple sorting program (4), a program for finding the prime factors of an integer (4), and a program for multiplying n by n matrices (3) required 3, 8, and 10 seconds, respectively, to construct. Each was constructed on the basis of one short example computation. The technique for processing multiple traces in parallel is designed to make it possible to construct much more difficult programs by using information from many sources simultaneously. Contradictions not found with one trace will hopefully be discovered immediately from other traces without the necessity for executing long searches. For example, when the problem worked in Figure 4 was repeated using one trace of length 25, it took 42 steps to find the solution rather than the 7 steps shown here. The additional calculation was required because of wrong guesses made and the later backing up which resulted.

IV. DISCUSSION

This paper describes a method for constructing computer programs from computation traces. It is clear from this work that all phases of the method could be automated and that computer programs can be automatically generated. It is not clear whether such an implementation would actually be more desirable for constructing programs than other methods currently in use.

Two critical questions remain to be answered:

(1) Is it true, as we have suggested, that it is easier to work through one or several example computations than it is to write the program? We suspect that the answer to this question depends greatly on the program and on the facilities available for implementing the example.

Trace 1	Trace 2	Trace 3	Trace 4	Trace 5	Trace 6	Trace 7	Transitions
1	1	1	1	1	1	1	
B B	A Z	A Z	B B	B B	A Z	B B	
B A	A Z	B B	B A	B A	B B	A R	
A R	B B	A R	C A	A R	A R	B A	
A R	C Z	B A	B B	C A	A R	A R	
C A	B B	A R	C Z	B B	C Z	A R	
C H	D H	B H	C H	D H	C H	B H	

(a)

1	1	1	1	1	1	1	
B B	A Z	A Z	B B	B B	A Z	B B	1 AZ 1
	1	1			1		
B A	A Z	B B	B A	B A	B B	A R	
	(1)						
A R	B B	A R	C A	A R	A R	B A	
A R	C Z	B A	B B	C A	A R	A R	
C A	B B	A R	C Z	B B	C Z	A R	
C H	D H	B H	C H	D H	C H	B H	

(b)

1	1	1	1	1	1	1	
B B	A Z	A Z	B B	B B	A Z	B B	1 AZ 1
2	1	1	2	2	1	2	1 BB 2
B A	A Z	B B	B A	B A	B B	A R	
	(1)	2			2		
A R	B B	A R	C A	A R	A R	B A	
	2						
A R	C Z	B A	B B	C A	A R	A R	
C A	B B	A R	C Z	B B	C Z	A R	
C H	D H	B H	C H	D H	C H	B H	

(c)

Figure 4

Trace 1	Trace 2	Trace 3	Trace 4	Trace 5	Trace 6	Trace 7	Transitions
1	1	1	1	1	1	1	
B B	A Z	A Z	B B	B B	A Z	B B	1 AZ 1
2	1	1	2	2	1	2	1 BB 2
B A	A Z	B B	B A	B A	B B	A R	2 AR 2
	(1)	2			2	2	
A R	B B	A R	C A	A R	A R	B A	
	2	2			2		
A R	C Z	B A	B B	C A	A R	A R	
					(2)		
C A	B B	A R	C Z	B B	C Z	A R	
C H	D H	B H	C H	D H	C H	B H	

(d)

1	1	1	1	1	1	1	
B B	A Z	A Z	B B	B B	A Z	B B	1 AZ 1
2	1	1	2	2	1	2	1 BB 2
B A	A Z	B B	B A	B A	B B	A R	2 AR 2
3	(1)	2	3	3	2	2	2 BA 3
A R	B B	A R	C A	A R	A R	B A	
	2	2			2	3	
A R	C Z	B A	B B	C A	A R	A R	
		3			(2)		
C A	B B	A R	C Z	B B	C Z	A R	
C H	D H	B H	C H	D H	C H	B H	

(e)

1	1	1	1	1	1	1	
B B	A Z	A Z	B B	B B	A Z	B B	1 AZ 1
2	1	1	2	2	1	2	1 BB 2
B A	A Z	B B	B A	B A	B B	A R	2 AR 2
3	(1)	2	3	3	2	2	2 BA 3
A R	B B	A R	C A	A R	A R	B A	2 CZ 1
	2	2			2	3	
A R	C Z	B A	B B	C A	A R	A R	
	1	3			2		
C A	B B	A R	C Z	B B	C Z	A R	
	(2)				1		
C H	D H	B H	C H	D H	C H	B H	

(f)

Figure 4 (cont'd)

Trace 1	Trace 2	Trace 3	Trace 4	Trace 5	Trace 6	Trace 7	Transitions
1	1	1	1	1	1	1	
B B	A Z	A Z	B B	B B	A Z	B B	1 AZ 1
2	1	1	2	2	1	2	1 BB 2
B A	A Z	B B	B A	B A	B B	A R	2 AR 2
3	(1)	2	3	3	2	2	2 BA 3
A R	B B	A R	C A	A R	A R	B A	2 CZ 1
3	2	2		3	2	3	3 AR 3
A R	C Z	B A	B B	C A	A R	A R	
(3)	1	3			(2)	3	
C A	B B	A R	C Z	B B	C Z	A R	
	(2)	3			1	(3)	
C H	D H	B H	C H	D H	C H	B H	

(g)

1	1	1	1	1	1	1	1 AZ 1
B B	A Z	A Z	B B	B B	A Z	B B	1 BB 2
2	1	1	2	2	1	2	2 AR 2
B A	A Z	B B	B A	B A	B B	A R	2 BA 3
3	(1)	2	3	3	2	2	2 CZ 1
A R	B B	A R	C A	A R	A R	B A	3 AR 3
3	2	2	1	3	2	3	3 CA 1
A R	C Z	B A	B B	C Z	A R	A R	
(3)	1	3	(2)	1	(2)	3	
C A	B B	A R	C Z	B B	C Z	A R	
1	(2)	3	(1)	(2)	1	(3)	
C H	D H	B H	C H	D H	C H	B H	

(h)

Figure 4 (cont'd)

(2) Can a person produce a debugged example any more easily than he can produce a debugged program? Again, the answer is not obvious.

Our current research is aimed at increasing the power of the synthesis technique and discovering its properties. Furthermore, we are designing what we call an autoprogramming system which will include the features discussed above plus subroutine and macroprocessing facilities. We believe that a proper combination of these concepts may yield a useful programming system.

ACKNOWLEDGMENT

I am greatly indebted to Professor J.A. Feldman for many invaluable discussions during the period of this research.

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.

REFERENCES

1. Amarel, S. On the automatic formation of a computer program which represents a theory. Self Organizing Systems - 1962 (Yovits, Jacobi and Goldstein, eds.). Spartan Books, New York, 1962.

2. Amarel, S. Representations and modelling in problems of program formation. Machine Intelligence 6 (Meltzer and Michie, eds.). American Elsevier Publishing Company, Inc., New York, 1971.

3. Biermann, A.W. An Approach to the Design of Trainable Machines, Report to the National Science Foundation, Computer and Information Science Department, Ohio State University, June 1972, 6.35 - 6.39.

4. Biermann, A.W. On the Inference of Turing Machines, Artificial Intelligence 3 (1972), 181-198.

5. Biermann, A.W. and Feldman, J.A. On the synthesis of finite-state machines from samples of their behavior, IEEE Trans. Electron. Computers, C-21, No. 6 (1972).

6. Biermann, A.W. and Feldman, J.A. A survey of results in grammatical inference. Frontiers in Pattern Recognitions (Watanabe, M.S., ed.) Academic Press, New York 1972.

7. Enomoto, H., Tomita, E., Doshita, S. Synthesis of Automata that Recognize Given Strings and Characterization of Automata by Representative Sets of Strings, First USA-Japan Computer Conference Proceedings, Tokyo, Japan, 1972.

8. Feldman, J.A. First thoughts on grammatical inference. A.I. Memo No. 55, Computer Science Department, Stanford University, August 1967.

9. Feldman, J.A. Some decidability results on grammatical inference and complexity, Information and Control, 1972.

10. Feldman, J.A., Gips, J., Horning, J.J., and Reder, S. Grammatical Complexity and Inference. Technical Report No. CS125, Computer Science Department, Stanford University, June 1969.

11. Feldman, J.A. and Shields, P.C. Total complexity and the inference of best programs. A.I. Memo No. AIM-159, Computer Science Department, Stanford University, April 1972.

12. Gill, A. Realization of input-output relations by sequential machines. J.ACM 13, No. 1 (1966), 33-42.

13. Ginsburg, S. Synthesis of minimal-state machines. IRE Trans. Electron. Computers EC8 (1959), 441-449.

14. Ginsburg, S. An Introduction to Mathematical Machine Theory. Addison-Wesley, Reading, Mass., 1962.

15. Gold, M. Language identification in the limit. Information and Control 10 (1967), 447-474.

16. Horning, J.J. A study of grammatical inference, Technical Report No. CS 139, Computer Science Department, Stanford University, August 1969.

17. Manna, Z. and Waldinger, R.F. Toward automatic program synthesis. Comm. Ass. Computing Machinery 14, No. 3 (1971), 151-165.

18. Minsky, M.L. Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, N.J., 1967.

19. Solomonoff, R. A formal theory of inductive inference. Information and Control 7 (1964), 1-22; 224-254.

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