AUTOMATIC PROGRAM SYNTHESIS FROM EXAMPLE PROBLEMS

L. Siklossy and D. A. Sykes
Computer Sciences Department
University of Texas at Austin
Austin, Texas, U.S.A.

Abstract

Desired algorithms to be synthesized are described implicitly by example problems that the algorithms should solve. The example problems are first solved by the problem-solver LAWALY. The obtained solution represents a trace of the desired algorithm. From the trace, the algorithm is synthesized by the synthesizer SYN. SYN writes recursive programs with repeat statements, and generates its own subroutines. Examples of functions synthesized include repetitive robot tasks and tree traversal algorithms.

1. Introduction

A program synthesizer transforms the implicit or explicit description of an algorithm into executable code. Although a compiler for a high level language might be considered a synthesizer, since it transforms an algorithm written in the language into executable machine code, generally discussions of program synthesizers are restricted to those systems which transform into code descriptions which are "far" from being executable. The concept of "far" is relative to a state of knowledge. For example, some synthesizers accept descriptions of algorithms in the predicate calculus. On the other hand, Kowalski (1) has argued in favor of a predicate calculus programming language. When a compiler for such a programming language is available, we can expect that some synthesizers will lose their "raison d'etre".

Approaches to program synthesis can be distinguished according to the descriptions of the algorithm to be synthesized. Properties of the algorithm can be given to the synthesizer, expressed in a programming language (2) or a logical calculus (4-7). A theorem-prover is used to extract the synthesized program. This approach suffers from difficulties in obtaining axiomatizations, and from the lack of power of present theorem-provers.

Another approach (8,9) starts from the traces that the algorithm would have produced-if it existed-on some particular test cases. The traces are generalized to the synthesized program. In this way, a variety of looping programs have been obtained. This approach suffers from the tedium experienced by human beings in producing traces, and from the errors that all too often manage to creep into the traces. The above approaches are described more fully in section 2.

We present here a novel approach to synthesis. The program to *bo* synthesized is described implicitly by sample problems that are assumed to be typical of the class of problems that the program should solve. The sample problems are first solved by the problem-solver LAWALY (10-12). The obtained solutions are traces which are generalized to the synthesized programs. Other efforts to synthesize programs given input-output pairs, but without the use of a problem-solver, arc described in (14,15).

Our synthesizer SYN has generated recursive programs with repeat statements. SYN writes its own subroutines. The synthesis process is totally automatic once sample problems, and the elementary operations necessary to solve the problems, have been given.

SYN is a running system programmed in LISP. The behavior of SYN is exemplified by several sample problems.

2. Approaches to Program Synthesis

2.1 Properties of Programs.

In many approaches to program synthesis, the synthesizer is given a description of properties of the desired program in some language, for example logical calculi or a programming language.

2.1.1 Properties in a Programming Language.

Siklossy (2) has described a synthesizer which accepts properties of algorithms expressed in LISP. Examples of such properties are: (EQUAL (TIMES A (PLUS B C)) (PLUS (TIMES A B) (TIMES AC))) and (EQUAL (APPEND A (APPEND B O) (APPEND (APPEND A B) C)). The synthesizer uses a property prover similar to that of Boyer and Moore (3).

2.1.2 Properties in Logical Calculi.

Several synthesizers accept descriptions of properties of algorithms in the predicate calculus (Waldinger, Lee and Manna, 4,5,6). A theorem prover is used to synthesize the algorithm. Other logical calculi have been used, for example by Luckham and Buchanan (7) in a synthesizer that permits man-machine interaction.

2.2 Traces of the Algorithm

The approaches described in section 2.1 suffer from several drawbacks: a) It is often difficult, if not downright painful at times, to give adequate axiomatizations of algorithms and their associated data structures, b) Among the major difficulties encountered in the approaches via logical calculi (section 2.1.2) is the requirement that the axiomatization be complete. If some property of the algorithm or data structure is not included, the theorem-prover used by the synthesizer cannot complete a proof, and no algorithm is synthesized. This difficulty does not exist in the approach via programming language description (section 2.1,1), since the synthesizer can produce partial syntheses which immediately result in demands for additional properties, c) Presently available theorem-provers in particular for logical calculi, appear too weak to synthesize difficult programs. Until the theorem-provers become more powerful, the logical calculi approach has been called "not pragmatic"

by its own supporters (6).

Human beings often understand a given program by executing (or simulating the executing of) the algorithm on some test cases, i.e. by considering traces of the algorithm. Similarly, human beings often prefer to have a desired algorithm (to be programmed) explained to them by a description of the behavior of the algorithm on some particular sample problem, i.e., in terms of traces. Biermann (8,9) has described a synthesizer which accepts the traces of an algorithm to be synthesized. The traces are assumed to be equal to the traces that the desired algorithm (if it existed) would have produced on the same test cases. The synthesizer generalizes the traces to a program which does produce the correct traces on the sample test cases. The programs synthesized include loops, and subroutines if these were specified in the traces.

2.3 Example Problems

Biermann *(personal* communication) has reported that inputting the traces by hand was not only tedious, but could not always be done without the inclusion of errors in the traces. We have therefore been led to automate the production of traces, both to avoid errors and to eliminate a tedious task.

In our synthesizer, SYN, the inputs are specific, sample problems belonging to the class of problems which the algorithm to be synthesized should solve. The human being is saved the tedium of producing traces. In addition, the description of the desired algorithm is not given under any form. The user of the synthesizer has the same relationship with the synthesizer as he has with a systems programmer: he only describes problems to be solved. It is the synthesizer's task, as it is the system programmer's,to develop an algorithm that solves the sample problems, and in addition a reasonable class of more general problems that include the sample problems.

The problems input to SYN are solved by the problem-solver LAWALY (10-12) which produces a solution which is interpreted as the trace of the desired algorithm. The trace is then generalized to an algorithm. The synthesizer proper, also called SYN, which produces programs from the traces, bears almost no resemblance to Biermann's synthesizer. While Biermann produced looping programs, SYN writes recursive programs with repeat statements in a language similar to LISP. SYN also generates its own subroutines.

Since the problem solver LAWALY has been described elsewhere in the literature, we shall concentrate on the synthesizer proper. Since, as we have argued, programs are best understood by describing their trace on a sample problem, we shall exhibit the behavior of SYN on a simple sample problem.

3. SYN on a Simple Sample

We consider a simple problem, illustrated in Figure 1. A robot is on ground X. She holds a flag FLAG. She can climb from X on boxes A, B, and C, one after the other. On the top box C she can plant the flag. We want the robot to plant the flag and return to the ground X.

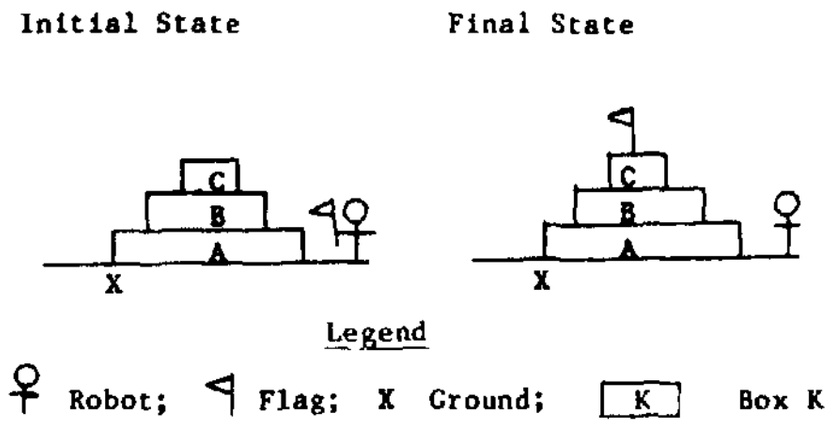

Figure 1. A Simple Sample Problem

As in (10-13), the world is described by a set of true facts, and the capabilities of the robot are given by elementary operators. The format of each operator is: (operator-name (list of arguments) (set of preconditions of the operator) (delete set of the operator) (add set of the operator)) For an operator to be applicable to the world OW, the preconditions of the operator must be satisfied in OW. As the result of the application of the operator, the world changes into a new world NW given by:

NW := ((OW - delete-set) + add-set), where - and + are set difference and union, respectively.

```
(climb (box1 box2)
     ((on box2 box1) (onrbt box1)) -preconditions
     ((onrbt box1))                -delete set
                                 onrbt = on robot
     ((onrbt box2))       )       -add set

(unclimb (box2 box1)  ((on box2 box1)(onrbt box2))
     ((onrbt box2))   ((onrbt box1))     )

(plantflag (flag box) ((top box) (holding flag)
                                      (onrbt box))
     ((holding flag)) ((planted flag box))    )
```

Figure 2. Operators for Simple Sample Problem

Figure 2 lists the operators relevant to the example task. In a more complex world, a more careful axlomatization could be necessary (13). Figure 3 shows the sequence of operators which constitutes the solution trace found by LAWALY to the problem. Each step of the solution is indexed by (Ei) for subsequent reference.

Operator	Reference
(climb X A)	(E1)
(climb A B)	(E2)
(climb B C)	(E3)
(plantflag FLAG C)	(E4)
(unclimb C B)	(E5)
(unclimb B A)	(E6)
(unclimb A X)	(E7)

Figure 3. Trace of solution to Simple Sample Problem.

3.1 *ODBOL*

The synthesized programs are written in ODBOL (no acronym), which has the flavor of LISP. The syntax of ODBOL is given in Figure 4. We intend to extend ODBOL as the power of SYN increases.

```
<fundefn> ::= (<identifier>(LAMBDA<idnlist><body>))
<idnlist> ::= (<identifier>*)      *denotes the
```

```
Kleene star.
<body>    ::= (COND<block>*)
<block>   ::= (<ifpart><opart>)
<opart>   ::=   BEGIN  <op>* END|
                BEGIN  <op>* REPEAT END
<ifpart>  ::= (TRUE) | (IF <precondition>*)|
              (EXISTS <idnlist><precondition>*)
<op>      ::= <robot operator>|<function call>
```

Figure 4. Syntax of ODBOL.

The semantics of ODBOL are as follows:

- When a function F is called, the formal parameters of F are bound to the actual parameters of the call.
- Having bound the parameters, each <block> in the <body> is scanned from left to right until an <ifpart> is found to be true. The corresponding <opart> is then executed. (If no <ifpart> is true and the <body> is exhausted, then an error has occurred.)
- The <op>s within the block are executed from left to right, as in the PROG feature in LISP. If the *<op>* is a function call, the indicated function is executed with the indicated arguments. Otherwise the <op> is executed and control passes to the next <op>. An END Is treated as a RETURN statement in a LISP PROG. A REPEAT statement is equivalent to a transfer to the top of the <block>, unless the block condition is now false, in which case control is passed to the next statement (always an END according to the syntax.)
- In the case of EXISTS, the identifiers in the <idnlist> are bound before entering the block, and these bindings hold throughout the rest of the block. If more than one possible binding is found for an identifier, then one of the bindings is chosen arbitrarily. (REPEAT statements or recursion will make the system consider several of the possible bindings.)

3.2 Arguments of the Function.

The function to be synthesized must be given a list of arguments. To select the arguments, we consider the preconditions of the first operator in the trace. The preconditions are ranked according to a hierarchy (see 10). The lowest ranked precondition, corresponding to the most restrictive precondition, is selected. Its arguments become the arguments of the function.

In our example, among the preconditions of the operator (climb X A), the most restrictive one is (onrbt X), hence X is selected as the argument of the function (call it FF) to be built.

If the first operator is to some extent incidental to the solution, this approach would appear unsatisfactory. However, as is shown in section 4.2, SYN will then generate a first function which accomplishes the Incidental work, and calls another function which performs the significant tasks.

3.3 Building the Function.

We shall show how the function FF is built. The total function FF is

ODBOL Code	Line Reference
(FF (LAMBDA (X) (COND	(L1)
((EXISTS (A) (on A X)) BEGIN (climb X A)	(L2)
(FF A)	(L3)
(unclimb A X)	(L6)
END)	(L7)
((EXISTS (FLAG) (top X) (holding FLAG)) BEGIN (plantflag FLAG X)	(L4)
END))))	

The line references correspond to the order in which the lines of code are generated.

So far, we have line LI only. The <ifpart> of the <block> comes from the preconditions for (climb X A), namely (on A X) and (onrbt X). The latter provided us with the variable X. (on A X) involves a new object. A, so an EXISTS construct is introduced. We obtain line L2. Having used EI, we consider E2. Since the operators in E2 and EI are the same, recursion is attempted, yielding line L3. The recursion is on A, since A occupies the same position in E2 as X did in EI.

The new program is executed to verify that it conforms with the trace. In fact, not only EI and E2 agree with the program, but E3 ae well. As we try to execute (FF C), the condition (EXISTS (alpha) (on alpha C)) is not satisfied in the world, hence the test fails, and we need to write new code. From the preconditions of plantflag in E4, we generate a second test in the CONDitional, line L4. Notice how the constant FLAG becomes a variable. Previously, the ground X also became a variable.

We next consider E5: (unclimb C B). The constants C and B In E5 are the name ones that were current in the call (FF B). (As the partial programs are executed, all contexts are saved for subsequent examination, if necessary.) Hence, it appears that a change of recursive level has taken place, since B is unknown in the call (FF C). Hence, we terminate the plantflag part of the program by adding END, line L5 -which forces a return to the previous recursive level- and by Inserting an unclilmh, line L6.

Looking at E6: (unclimb B A), we notice that the recursive context is changed again. Hence, an END is inserted after the unclimb, line L7. Ae before, new code must be checked against the trace. In fact, even E7 is generated, hence the synthesized program is found completely satisfactory.

3.4 Comparison with Human Programs.

The authors have asked several friends and colleagues to describe a generalization to the simple sample problem of Figure 2. Invariably, the solution was of the form:
-make a loop as the robot climbs to the top;
-plant the flag;
-make a loop as the robot unclimbs down to the ground.

It is remarkable that the program synthesized by SYN is in fact "better" than the version proposed by all humans (including ourselves!) as the most "natural" solution. The solution FF can be described as follows: a series of recursive calls leads the robot up to the top; she then plants the flag. Then, as the recursive calls are undone, the robot climbs down. So, in particular, the

operators (climb X A) and (unclimb A X) are paired in the same function call, and their complementary relationship is made transparent. By contrast, the "climb" and "unclimb" operators are separated in the two distinct, separate loops in the human program.

The above example does not illustrate all the capabilities of SYN. Some modifications of the simple sample problem will give a more complete picture of SYN.

3.5 REPEAT statements.

If the robot holds several FLAGS, and plants all of them on the top box, the second block of FF would be synthesized as:

```
((EXISTS (FLAG) (top X) (holding FLAG)) BEGIN
                        (plantflag FLAG X)
                        REPEAT END       )
```

Recursion could also have been used, but it would define (FF X) in terms of (FF X), the infinite recursion being avoided because of side effects. Hence, a REPEAT statement appears "cleaner" and easier to implement. Some synthesized programs contain several embedded REPEAT statements, see section 4.4.

3.6 Placement of <block>s within a <body>.

Let us assume that the robot can plant a FLAG while standing on the box just below the top box, with the help of the operator plantflag2. The trace would be:

```
(climb X A )                    (E1)
(climb A B)                     (E2)
(plantflag2 FLAG C B)           (E3)
(unclimb B A)                   (E4)
(unclimb A X)                   (E5)
```

Hence the robot climbs only to B, then plantflag2's (sic!). Synthesizing the procedure, we would generate lines L1, L2 and L3 of code. When X is bound to B, FF would make the robot climb on C, which is not in the trace. Hence, an <ifpart> is inserted to prevent the execution of (FF C). The partial definition of FF3 now becomes:

```
(FF3 (LAMBDA (X) (COND                          (L1)
((EXISTS (FLAG C)(on C X)(top C)(holding FLAG))
  BEGIN (plantflag2 FLAG C X)...)               (L4')
```

followed by L2 and L3. Of course, SYN must verify that this new insertion does not change the agreement of the program with the earlier parts of the trace.

3.7 Writing new functions.

ODBOL does not allow the binding of variables past the <ifpart> of a <block> . If we want to test whether there is a nice cube on A (which should be picked up If it is already melting) before calling FF again, we must have access to a variable that can be bound to nice cubes. SYN would call a new function GG with a definition similar to:

```
(GG (LAMBDA (A) (COND
    ((EXISTS (NICECUBE) (melting NICECUBE AT
    BEGIN (pickup NICECUBE A)...
```

Hence, we can see that SYN fights the increased complexity due to additional bindings within a <block> by creating new functions that absorb the bindings. The creation of new functions is also helpful when the original choice of function parameters is inappropriate (see sections 3.2 and 4.2).

4. Some Additional Synthesized Programs

In this section we describe some additional problems that were given to SYN, and the resulting synthesized programs in ODBOL. Each problem was synthesized independently and the actual computer outputs are given.

4.1 Returning to the First Step.

We modify the final state of the first example problem by letting the robot finish on step A instead of the ground X. The synthesized program is

```
(F00001 (LAMBDA (X)      (COND
       ((EXISTS (A) (ON A X))   BEGIN
                                (CLIMB X A)
                                (F00002 A)
                                END   )    )))
```

where F00002 is identical to FF except for variable names. In the solution F00001, a step is climbed and an auxiliary function is created, essentially identical to FF, which does the symmetric climbing and unclimbing, and plants the flag. The symmetry of much of the solution is apparent.

4.2 Picking up a backpack before planting the flag-

Before accomplishing the same task as in Figure 1, we require that a backpack be picked up as a first step. Hence this first step is irrelevant to the main problem to be solved. The synthesized program is F00006:

```
(F00006 (LAMBDA (BACKPACK X)       (COND
       ((TRUE)  BEGIN
                (PICKUP BACKPACK)
                (F00007 X)
                END   )     )))
```

Where F00007 is identical to FF. F00006 picks up the backpack (now a variable) and calls an auxiliary function F00007, essentially identical to FF, which does all the work.

This problem shows that SYN was not misled by incidental early actions in the trace.

4.3 Binary Tree Traversals.

Tree traversal algorithms are popular recursive algorithms. We start with a tree, and describe the order in which the nodes should be traversed. We shall describe results for postorder (inorder) and preorder traversals. The tree is shown in Figure 5.

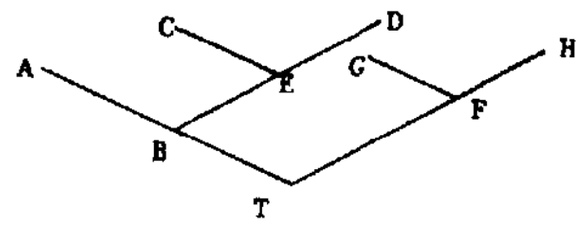

Figure 5. A Sample Tree

4.3.1 Postorder Traversal.

The input problem indicates that the branches should be MARKed in the order: A B C E D T G F H. LAWALY uses operators CLIMBLeft, CLIMBRight, and

MARK to indicate when the branch is actually visited. Predicates LBR and RBR test for Left and Right BRanches. The solution is F00001 and the output is:

```
(F00001 (LAMBDA (T)     (COND
  ((EXISTS (B) (LBR B T)) BEGIN (CLIMBL T B)
                                (F00001 B)
                                (UNCLIMB B T)
                                (MARK T)
                                (F00002 T) END   )
  ((TRUE) BEGIN (MARK T) END     )      )))
(F00002 (LAMBDA (B)   (COND
  ((EXISTS (E) (RBR E B)) BEGIN (CLIMBR B E)
                                (F00001 E)
                                (UNCLIMB E B)END))))
```

4.3.2 Preorder Traversal.

The branches are traversed in the order: T B A E C D F G H. The same operators as before are used by LAWALY to generate the trace of the solution on the sample tree. The synthesized solution by SYN is F00001 and the output is:

```
(F00002 (LAMBDA (T)     (COND
  ((EXISTS (B) (LBR B T)) BEGIN
                          (CLIMBL T B)
                          (F00001 B)
                          (UNCLIMB B T)
                          (F00003 T) END   )
  ((TRUE)  BEGIN
           END  )    )))
(F00003 (LAMBDA (B)     (COND
  ((EXISTS (E) (RBR E B)) BEGIN
                          (CLIMBR B E)
                          (F00001 E)
                          (UNCLIMB E B)
                          END  )    )))
(F00001 (LAMBDA (T)    (COND
  ((TRUE)  BEGIN
           (MARK T)
           (F00002 T)
           END  )    )))
```

F00002 and F00003 initiate traversal towards the left and right, respectively, and F00001 does the marking.

4.4 Embedded REPEAT Statements.

With a house as in Figure 6, the robot is asked to move boxes X, Y,Z and W into the HALL. She picks up one by one all the boxes in one room, moves them into the HALL and puts them down one by one, then repeats the same operations for all the rooms.

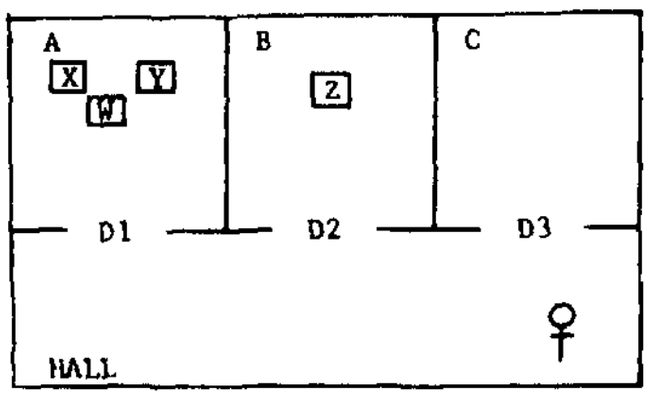

Figure 6. Moving Objects into the HALL.

The synthesized program is F00001:

```
(F00001 (LAMBDA (HALL) (COND
  ((EXISTS (A D1) (CONNECT HALL A D1)(UNVISITED A))
                              BEGIN (ENTER A D1 HALL)
                                    (F00002 A)
                                    (EXIT A D1 HALL)
                                    (F00003 HALL)
                                    REPEAT END) )))
(F00002 (LAMBDA (A) (COND
  ((EXISTS (X) (INROOM X A)) BEGIN (GONEXT X A)
                                   (PICKUP X)
                                   REPEAT END)
  ((TRUE) BEGIN
          END)  )))
(F00003 (LAMBDA (HALL) (COND
   ((EXISTS(X)(HOLDING X))BEGIN(PUTDOWN X HALL)
                               REPEAT END)
   ((TRUE) BEGIN
           END)  )))
```

F00002 does the picking up, F00003 the putting down. D1, D2 and D3 are doors.

5. Conclusions

In Our approach to program synthesis, a program is described implicitly by typical problems that it should solve. This description is sometimes very natural and compact, as in the case of tree traversals, for example. We have described only a few programs among the many that SYN synthesized. The principal difficulties of the approach are tied with axiomatizations and the problem-solver, syntheses from several problems, and the question of the correctness of the resultant programs.

In all our examples, the trace was obtained by our problem-solver LAWALY. If we insist that LAWALY (and not a human) generate the trace, we can synthesize only those programs which solve problems in a class typified by some problems that LAWALY can solve. Of course, we could always bypass LAWALY and give the trace of a solution by hand. Moreover, the trace should include the relevant operators and predicates. For example, the traces of the tree traversal algorithms must include the notions of left and right branches, and directions for climbing. The importance of a good axiomatization must once again be emphasized (13).

Until now, our syntheses all proceeded from a simple trace. Some programs might be so complex that no single problem would exemplify fully the class of problems to be solved. We are attempting to extend SYN to such cases.

Once the program has been synthesized, we know that it is a generalization of the trace input to SYN, but do not know whether it is "correct". To decide whether the program is correct, we must give additional properties of the program, besides typical problems that It should solve. The language ODBOL has a structure which would make proofs of correctness of ODBOL programs fairly easy.

We do not think that any single approach to program synthesis is fully satisfactory. As we mentioned, the problem-solver may be too weak to solve some problems, and it might be necessary to input traces by hand. Sometimes, some aspects of the program to be synthesized are best described as properties of the program, given either in a programming language or a logical calculus. Finally, the various approaches to program synthesis and correctness

should be embedded in a man-machine environment before significant advances in the automatic production of complex, quality software can be achieved.

6. References

1. Kowalski, R., "Predicate Logic as Programming Language," Proc. IFIP Congress 74, Stockholm, 569-574.

2. Siklossy, L., "The Synthesis of Programs from their Properties, and the Insane Heuristic," Proc. Third Texas Conference on Computing Systems, Austin, Texas, 1974.

3. Boyer, R. S., and Moore, J. S., "Proving Theorems about LISP Programs," Third International Joint Conference on Artificial Intelligence, Palo Alto, California, 1973, 486-493-

4- Waldinger, R. J., and Lee, R. C. T., "PROW: A Step Toward Automatic Program Writing," International Joint Conference on Artificial Intelligence, Washington, D.C., 1969, 241-252.

5. Manna, Z., and Waldinger, R. J., "Toward Automatic Program Synthesis," Comm. A. C. M., 14, 3, 1971, 151-164.

6. Lee, R. C. T., Chang, C. L., and Waldinger, R. J., "An Improved Program-Synthesizing Algorithm and its Correctness," Comm. A. C. M., 17, 4, 1974, 211-217.

7. Buchanan, J. R. and Luckham, D. C., On Automating the Construction of Programs, Technical Report STAN-CS-74-433 Computer Science Department, Stanford University, 1974.

8. Biermann, A. W., "On the Inference of Turing Machines from Sample Computations," Artificial Intelligence, 3, 1972, 181-198.

9. Biermann, A. W., Baum, R., Krlshnaswamy, R., and Petry, F. E., Automatic Program Synthesis Reports, Technical Report OSU-CISRC-TR-73-6, Computer and Information Sciences Research Center, Ohio State University, October, 1973.

10. Siklossy, L. and Dreussi, J., "An Efficient Robot Planner which Generates its own Procedures," Third International Joint Conference on Artificial Intelligence, Palo Alto, California, 1973, 423-430.

11. Siklossy L. and Roach J., "Collaborative Problem-Solving between Optimistic and Pessimistic Problem-Solvers," Proc. IFIP Congress 74, Stockholm, 1974, 814-817.

12. Siklossy, L., "Procedural Learning in Worlds of Robots," Proc. NATO Advanced Study Institute on Computer Oriented Learning Processes, Bonas, France, 1974, 423-436.

13. Siklossy, L. and Roach, J., "Model Verification and Improvement using DISPROVER," Artificial Intelligence, 6, 1975.

14. Green, C. C., et al., Progress Report on Program-Understanding Systems, Technical Report AIM-240, Computer Science, Stanford University, 1974.

15- Project MAC Progress Report X, Massachusetts Institute of Technology, Cambridge, 1973, 151-156.