DEDALUS—The DEDuctive ALgorithm Ur-Synthesizer*

by ZOHAR MANNA
Stanford University
Stanford, California

and

RICHARD WALDINGER
SRI International
Menlo Park, California

INTRODUCTION

Program synthesis is the automatic construction of programs to meet given specifications. These specifications constitute a high-level description of the desired program which expresses the purpose of the program, without indicating the method by which that purpose is to be achieved.

The specifications are expressed in terms of many constructs which are endemic to the particular subject domain of the desired program (e.g., numbers, sets, lists). Because these constructs are only intended to describe the purpose of the program and need not be computed, they can be of a much higher level than the constructs of any programming language (e.g., they can include logical quantifiers, set constructors, and other noncomputable operations). The specification language can correspond closely with the concepts a programmer actually uses in thinking about the problem.

The techniques we are developing are independent of the choice of a target programming language. The particular language we use in our examples and in our experimental system is a simple LISP-like language containing only basic numerical and list-processing operations, conditional expressions, and recursion. In considering the formation of programs with side effects, we extend the language to include assignments to variables, array elements, and other data-structure components.

Our basic approach is to transform the specifications repeatedly according to certain rules; each rule replaces one segment of a program description by another, equivalent, segment. The process continues until a description is obtained that is entirely in terms of the primitive constructs of the target language; this description is the desired program. The entire sequence of descriptions leading from the specifications to the final program is called a *program derivation*.

The transformation rules are guided by certain strategic controls, which ensure that they are applied only at the appropriate time. Many of the transformation rules represent knowledge about the program's subject domain; some explicate the meaning of the constructs of the specification and target languages; a few rules correspond to basic programming principles, which are independent of the particular subject domain or programming language.

The programs constructed by these techniques are guaranteed to be correct and to terminate, and require no separate verification phase. Up to now, we have not been concerned with the efficiency of the target program.

The techniques we develop are tested in the implementation of an experimental system called DEDALUS (DEDuctive ALgorithm Ur-Synthesizer). This system is implemented in the QLISP programming language, an extension of INTERLISP that includes pattern-matching and backtracking facilities.

The identification of the basic programming principles, their codification as transformation rules, and their implementation in our experimental system have constituted a major component of our effort. Some of the principles we have identified so far are:

- *Conditional formation*—This principle causes a case analysis to be introduced into the derivation, yielding a conditional expression (or test) in the ultimate program.
- *Recursion formation*—This principle introduces a recursive call into the ultimate program by observing when a subgoal to be achieved is actually an instance of the desired top-level goal.
- *Well-founded ordering*—The termination of the recursive programs formed by the above technique is ensured by constructing a well-founded ordering with the property that the arguments of the program's recursive calls are all strictly less than the program's inputs.

* This research was supported in part by the National Science Foundation under Grants DCR72-03737 A01 and MCS76-83655, by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816, by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, and a grant from the United States-Israel Binational Science Foundation.

- *Procedure formation*—A subsidiary procedure is formed when a subgoal is found to be an instance, not of the top-level goal, but of a previously generated subgoal.
- *Generalization*—A generalized procedure is formed when two subgoals are found to be an instance of a third expression, which is somewhat more general than both.
- *Simultaneous goals*—In constructing a program to achieve two or more goals simultaneously, we first construct a program to achieve one goal, then modify that program to achieve the others as well, while protecting the condition that was already achieved.

Some of these principles are fairly well understood and have been incorporated into the DEDALUS system; others require further study. In the next sections we examine in some more detail our basic program-synthesis techniques, indicating which areas have already been implemented. Our treatment of these techniques will be extremely brief; further discussion of the same topics, at a more leisurely pace, appears in the authors' Stanford University-SRI International report, "Synthesis: Dreams⇒Programs" (November 1977).

GENERAL FRAMEWORK

Specifications

In designing the specification language, we have incorporated many constructs (e.g., the set constructor and the logical quantifiers) that facilitate the description of a program but that may not be included in the target programming language. We present below examples of specifications for simple programs using some of these high-level constructs.

A program *lessall(x l),* to test if a number x is less than every element of a list l of numbers, is specified as follows:

lessall(x l)⇐**compute** $x<all(l)$
where x is a number and
l is a list of numbers.

In general, the specification construct *P(all(l))* denotes that the property P holds for every element of the list l.

The specification for a program to compute the greatest common divisor $gcd(x\ y)$ of two nonnegative integers x and y is

$gcd(x\ y)$⇐**compute** $max\{z:z|x\ and\ z|y\}$
where x and y are nonnegative integers and
$x\neq 0$ or $y\neq 0$.

The set constructor $\{u: P(u)\}$ denotes the set of all elements u satisfying the property P.

The *all* construct $(P(all(l))$ and the set constructor $\{u: P(u)\}$ are specification constructs that are nonprimitive (i.e., they are not in the target programming language). The synthesis task is to transform a description of the desired program, such as the specifications presented above, into an equivalent description that employs only primitive constructs of the target language.

The specification language is not fixed: as we consider new subject domains, we introduce new specification constructs accordingly.

Transformation rules

We use the notation

$t \Rightarrow t'$ if P

to denote that a subexpression of form t may be replaced by the corresponding expression t', provided that the condition P is true.

For example, the rule

$Q\ and\ true \Rightarrow Q$

denotes the basic logical principle that an expression of form "Q *and true*" may be replaced by Q. This rule has no conditions; it can always be applied.

The rule

$P(all(l)) \Rightarrow P(head(l))\ and\ P(all(tail(l)))$ if *not empty* (l)

expresses the fact that a property P holds for every element of a nonempty list l if it holds for the first element *head*(l) and for every element of the list *tail*(l) of the other elements. This rule imposes the condition that the list l be nonempty.

In the DEDALUS system, transformation rules are represented as programs in the QLISP language. The full expressive power of the programming language may be brought to bear in representing each rule.

Derivation trees

In developing a program whose specifications are

$f(x)$⇐**compute** $P(x)$
where $Q(x)$,

we establish the output description as a goal to be achieved, viz.,

Goal: **compute** $P(x)$.

Subgoals are derived from this goal by application of the relevant transformation rules. For example, in deriving the *gcd* program, we form the top-level goal

Goal 1: **compute** $max\{z: z|x\ and\ z|y\}$

By applying a transformation rule

$u|v\ and\ u|w \Rightarrow u|v\ and\ u|w-v$

we obtain the subgoal

Goal 2: **compute** $max\{z: z|x\ and\ z|y-x\}$.

If a transformation rule imposes a condition P, which must be true if the rule is to be applied, a subgoal of the form

Goal: **prove** P

must be achieved before the rule can be applied. For example, in developing the program *lessall*(x l) to test if a number x is less than every element of a list l of numbers, we have the top-level goal

compute $x<all(l)$,

which is obtained directly from the specifications; in attempting to apply the rule

$P(all(l)) \Rightarrow true$ if $empty(l)$

to this goal, we derive the subgoal

Goal: **prove** $empty(l)$.

From each subgoal that is derived, we can generate further subgoals by the application of more transformation rules. We thus construct a tree of goals and subgoals, which we will call a *program derivation tree*.

A subgoal **compute** S is already achieved if S consists entirely of primitive constructs of the target language. A subgoal **prove** P is achieved if P is the logical constant *true*. Such goals are terminal nodes of the derivation tree.

Let us now discuss some of the basic programming principles used in developing the derivation tree.

BASIC PROGRAMMING PRINCIPLES

Conditional formation

Many of the transformation rules impose conditions (e.g., l is nonempty, x is nonnegative) that must be satisfied for the rule to be applied. Suppose that in attempting to apply a particular rule, we fail to prove or disprove a condition P, where P is expressed entirely in terms of the primitive constructs of the programming language; in such a situation, the conditional-formation rule is invoked. This rule allows us to introduce a case analysis, and consider separately the case in which P is true and P is false. Suppose we succeed in constructing a program segment S_1 that solves our problem under the assumption that P is true, and another program segment S_2 that solves the problem under the assumption that P is false. Then the conditional-formation principle puts these two program segments together into a conditional expression

if P *then* S_1 *else* S_2,

which solves our problem regardless of whether P is true or false.

If we happen to generate the program segment S_2, say, without using the case assumption that P is false, then S_2 solves our problem regardless of whether or not P is true. In this case, no conditional expression is formed, and the program constructed is simply S_2. Thus, conditional expressions are generated only for truly relevant conditions.

The conditional-formation rule is among the best-understood of our basic programming principles and was among the first to be incorporated into the DEDALUS implementation. Other program-synthesis systems that form conditional expressions by case analysis have been implemented by Buchanan[1] and Luckham and Warren.[2]

Recursion formation

Suppose, in constructing a program whose specifications are

$f(x)$⇐**compute** $P(x)$
where $Q(x)$

we encounter a subgoal

compute $P(t)$

which is an instance of our *output specification*, **compute** $P(x)$. Because the program $f(x)$ is intended to compute $P(x)$ for any x satisfying its *input specification* $Q(x)$, the recursion-formation rule proposes achieving the subgoal by computing $P(t)$ with a recursive call $f(t)$. For this step to be valid, it must ensure that the *input condition* $Q(t)$ holds when the proposed recursive call is executed. To ensure that the new recursive call will not cause the program to loop indefinitely, the rule must also establish a *termination condition*, showing that the argument t is strictly less than the input x in some well-founded ordering. (A *well-founded ordering* is one in which no infinite strictly decreasing sequences can exist.) This condition precludes the possibility that an infinite sequence of recursive calls might occur during the execution of the program.

For example, the DEDALUS system derived the program *lessall* (x l), which tests whether a given number x is less than every element of a given list l of numbers. The specifications for this program are

lessall (x l)⇐**compute** $x<all(l)$
where x is a number and
l is a list of numbers.

In deriving this program, we develop a subgoal

compute $x<all(tail(l))$

in the case that l is nonempty. This subgoal is an instance of our output specification, with the input l replaced by $tail(l)$; therefore, the recursion-formation principle proposes that we achieve the subgoal by introducing a recursive call *lessall* (x $tail(l)$). To ensure that this step is valid, the rule establishes an input condition, that

x is a number and
$tail(l)$ is a list of numbers,

and a termination-condition that the argument pair (x $tail(l)$) is less than the input pair (x l) in a well-founded ordering. This termination condition holds because $tail(l)$ is a proper sublist of l.

The final program we obtain is

lessall (x l)⇐*if* $empty(l)$
then true
else $x<head(l)$ *and lessall* (x $tail(l)$).

The recursion-formation principle is well-understood and is included in the DEDALUS implementation. The principle is the same as the "folding rule" of the Burstall and Darlington[3] program transformation system.

Procedure formation

Suppose in developing a program whose specifications are of the form

$f(x) \Leftarrow$ **compute** $P(x)$
where $Q(x)$

we encounter a subgoal

Goal B: **compute** $R(t)$,

which is an instance, not of the output specification **compute** $P(x)$, but of some previously generated subgoal

Goal A: **compute** $R(x)$.

The procedure-formation principle proposes that we introduce a new procedure $g(x)$ whose output specification is

$g(x) \Leftarrow$ **compute** $R(x)$.

In this way, we can achieve both Goals A and B by calls $g(x)$ and $g(t)$ to a single procedure. In the case that Goal B has been derived from Goal A, the call to $g(t)$ will be a recursive call; otherwise, both calls will be simple procedure calls.

For example, in constructing a program *cart* $(s\ t)$ to compute the Cartesian product s and t), of two sets, we are given the specification

cart $(s\ t) \Leftarrow$ **compute** $\{(x\ y): x \in s \text{ and } y \in t\}$
where s and t are finite sets.

In deriving the program, we obtain a subgoal

Goal A: **compute** $\{(x\ y): x = head(s) \text{ and } y \in t\}$

in the case that s is nonempty. Developing Goal A further, we derive the subgoal

Goal B: **compute** $\{(x\ y): x = head(s) \text{ and } y \in tail(t)\}$

in the case that t is nonempty. Goal B is an instance of Goal A; therefore, the procedure-formation rule proposes introducing a new procedure *carthead* $(s\ t)$ whose output specification is

carthead $(s\ t) \Leftarrow$ **compute** $\{(x\ y): x = head(s) \text{ and } y \in t\}$

so that we can achieve Goal A with a procedure call *carthead* $(s\ t)$ and Goal B with a (recursive) call *carthead* $(s\ tail(t))$.

The *carthead* procedure is constructed by the techniques we have already introduced. The final system of programs we obtain is

cart $(s\ t) \Leftarrow$ *if empty* (s)
then $\{\}$
else carthead $(s\ t) \cup$
cart $(tail(s)\ t)$,

where

carthead $(s\ t) \Leftarrow$ *if empty* (t)
then $\{\}$
else $\{(head(s)\ head(t))\} \cup$
carthead $(s\ tail(t))$.

The basic procedure-formation principle has been implemented, and the DEDALUS system can carry out this and other such derivations. However, our method for proving the termination of ordinary recursive calls does not always extend to the multiple-procedure case.

Generalization

Suppose in deriving a program we obtain two subgoals

Goal A: **compute** $R(a(x))$

and

Goal B: **compute** $R(b(x))$,

neither of which is an instance of the other but both of which are instances of the more general expression

compute $R(y)$.

Then the extended procedure-formation rule proposes that we introduce a new procedure, whose output specification is

$g(y) \Leftarrow$ **compute** $R(y)$,

so that we will be able to satisfy Goal A by a procedure call $g(a(x))$ and Goal B by a procedure call $g(b(x))$.

For example, in constructing a program *reverse* (l) to reverse a list l, we derive two subgoals

Goal A: **compute** *append*(*reverse*(*tail*(l))
cons(*head*(l)
nil))

and

Goal B: **compute** *append*(*reverse*(*tail*(*tail*(l)))
cons(*head*(*tail*(l))
cons(*head*(l) *nil*))).

Each of these goals is an instance of the more general expression

compute *append*(*reverse*(*tail*(l))
cons(*head*(l)
m));

therefore, the extended procedure-formation rule proposes introducing a new procedure *reversegen* $(l\ m)$, whose output specification is

reversegen $(l\ m) \Leftarrow$ **compute** *append*(*reverse*(*tail*(l))
cons(*head*(l)
m)).

This procedure reverses a nonempty list l and appends the result to m. Although the procedure solves a more general problem than the *reverse* program we actually require,

it turns out that the *reversegen* procedure is actually easier to construct. The final system of programs we obtain is

reverse (*l*)⇐*if empty* (*l*)
then nil
else reversegen (*l nil*)

where

reversegen (*l m*)⇐if *empty*(*tail* (*l*))
then cons(*head* (*l*) *m*)
else reversegen(*tail* (*l*)
cons(*head* (*l*) *m*)).

The generalization mechanism and the extended procedure-formation principle are just beginning to be formulated; the elaboration of these concepts is an important part of our projected effort. Generalization was proposed as a program-synthesis technique by Siklossy,[4] and is routinely performed by theorem-proving systems for proofs by induction (e.g., see Boyer and Moore[5]).

STRUCTURE-CHANGING PROGRAMS

In the discussion so far, we have been concerned with *structure-maintaining* (i.e., "side-effect-free") *programs,* which produce an output but effect no permanent change in the data objects of the programming environment. To produce *structure-changing programs,* which can effect such changes, we introduce structure-changing primitives, such as assignment statements, into our target language. To specify such programs, we admit new constructs into our specification language. Most notably, we employ **achieve** P to denote a program intended to make the condition P true.

All the principles we have applied to construct structure-maintaining programs may be adapted to construct structure-changing programs as well. However, certain new problems arise in the synthesis of structure-changing programs; among these is the *simultaneous-goal problem.*

In constructing a program to achieve two conditions P_1 and P_2, it is not sufficient to decompose the problem by constructing two independent programs to achieve P_1 and P_2 respectively. The program that achieves P_2 may in the process make P_1 false, and vice versa. Thus, the concatenation of the two programs will not achieve both conditions.

For example, suppose we want to construct a program to sort the values of three variables x, y, and z; in other words, we want to permute the values of the variables to achieve the two conditions $x \leq y$ and $y \leq z$ simultaneously. Assume that we are given the primitive instruction *sort*2(u v), which sorts the values of its input variables u and v. Then we can achieve each of our desired conditions independently by executing the program segment *sort*2(x y) and *sort*2(y z) respectively. However, the concatenation

*sort*2(x y)
*sort*2(y z)

of these two segments will not achieve both conditions simultaneously; in sorting y and z, the second segment *sort*2(y z) may make the first condition $x \leq y$ false.

To circumvent difficulties of this sort, we have introduced the following *simultaneous-goal principle*: To satisfy a goal of form

achieve P_1 *and* P_2,

first construct a program F to achieve P_1, then modify F to achieve P_2 while protecting P_1 at the end of F. The program-modification technique we employ is based on the "weakest-precondition operator" (Dijkstra[6]). A special "protection mechanism" (cf. Sussman[7]) ensures that no modification is permitted that destroys the truth of the protected condition P_1 at the end of the program.

To apply this principle to the goal

achieve $x \leq y$ *and* $y \leq z$

in the sorting problem, we first construct the program segment *sort*2(x y) that achieves the first condition. We then modify this program to achieve the second condition $y \leq z$. We cannot achieve this condition by inserting the instruction *sort*2(y z) at the end of the program, because (as we have seen) this modification violates the condition $x \leq y$, which we must protect.

However, our program-modification technique allows us to achieve a goal by inserting modifications at any point in the program, not merely at the end. In this case, the technique causes us to introduce the two instructions

if $y < x$ *then sort*2(x z)

and

if $x \leq y$ *then sort*2(y z)

at the beginning of the program segment. The modified program,

if $y < x$ *then sort*2(x z)
if $x \leq y$ *then sort*2(y z)
*sort*2(x y),

will achieve both conditions $x \leq y$ and $y \leq z$ simultaneously.

Similar techniques are employed by the system of Warren.[17]

The derivation of straight-line programs with simple side-effects is now fairly well understood; much work needs to be done on the derivation of structure-changing programs with conditional expressions and loops, and the derivation of programs that alter list structures and other complex data objects.

APPLICATIONS TO PROGRAMMING METHODOLOGY

Although the development of a practical program-synthesis system requires considerable research effort, certain applications of program-synthesis techniques to more restricted problems will be of more immediate practical value.

Various programming disciplines have been evolving that attempt to make the programming process simpler or more reliable. Let us consider several of these disciplines, to see where program-synthesis techniques may be applicable.

- *Structured Programming*—Like program synthesis, structured programming (cf. Dijkstra[8]) presents principles for deriving a program systematically from given specifications. However, the principles of structured programming are intended to guide a human programmer, whereas the principles of program synthesis are meant to direct a computer system. Nevertheless, we have found that some of the techniques we have developed for a program-synthesis system could well be employed by a human programmer. In particular, the recursion-formation principle is a better motivated guide for introducing a loop than the conventional structured-programming method for the same task.
- *Program Transformation*—In this approach (cf. Burstall and Darlington[9]), the programmer constructs a transparent program for his task, which is likely to be correct but which may be inefficient. This program is then transformed into an efficient equivalent program, which may be more difficult to understand. This transformation process is guaranteed to produce a program equivalent to the original. Program transformation may be regarded as a synthesis task in which the specifications are given in the form of a clear program in the target language. All the program synthesis techniques we have developed can be applied to program transformation as well.
- *Data Abstraction*—In this approach, the programmer expresses his program in terms of *abstract data types,* objects such as sets, queues, or graphs whose properties are well-defined but whose precise machine representation is left unspecified. When this program is complete, representations for its abstract data types are chosen and the program is transformed to replace the operations on the abstract data types by the corresponding concrete operations on the chosen representation (cf. Guttag et al.[9]).
- *Program Modification*—It is often observed that programmers spend more of their time extending programs that already perform some task correctly than they do in developing new programs. This process is particularly fraught with error, because in modifying a program, the programmer is likely to make some change that interferes with the program's original functioning. We have remarked that a program-modification technique was developed to support the simultaneous-goal principle. This technique can also be applied to perform independent program-modification tasks. The protection mechanism ensures that the modified program must still perform the task for which it was originally intended.

IMPLEMENTATION

It is difficult to develop or evaluate heuristic techniques without experimenting with an implementation. The DEDALUS system is a laboratory tool rather than a practical product. The system is implemented in QLISP (Wilber[10]), an extension of INTERLISP (Teitelman[11]) that includes pattern-matching and backtracking facilities. In this section, we will describe some of the special characteristics of our implementation without going into very much detail.

The specifications are expressed in a LISP-like notation. Thus, the output specification for the *lessall* program, which we wrote as

$$x<all(l),$$

is represented in the DEDALUS system as

```
(LESS X (ALL L)).
```

The output specification for the *gcd* program, which we wrote as

$$max\{z: z|x \text{ and } z|y\},$$

is represented as

```
(MAX (SETOF Z (AND (DIVIDES Z X)
                   (DIVIDES Z Y)))).
```

The target program is also expressed in LISP syntax.

The transformation rules are expressed as programs in the QLISP programming language. For example, the rule that we denoted by

$$Q \text{ and } true \Rightarrow Q$$

is represented by the QLISP program

```
(QLAMBDA (AND ←Q TRUE) $Q).
```

The rule

$$u|v \Rightarrow true \quad \text{if } u \text{ is an integer and } v=0$$

is expressed in QLISP as

```
(QLAMBDA (DIVIDES ←U ←V)
         (INSIST (PROVE ('(INTEGER $U))))
         (INSIST (PROVE ('(EQUAL $V 0))))
         TRUE).
```

Although the reader who is unfamiliar with the QLISP language may not understand all the details of the above programs, he may still observe that they are similar in form to the rules that they represent; the features of the QLISP language make this representation fairly direct. Because rules are represented as programs, we are allowed the full power of the programming language in expressing each rule.

The DEDALUS system currently contains more than a hundred such transformation rules. In expanding the system to handle a new subject domain, we simply introduce new rules.

The rules of the system are classified according to their *pattern,* their left-hand side. This pattern describes the class of subgoals to which the rule can be applied. Thus, the rules

$$u|v \Rightarrow true \quad \text{if } \ldots$$

and

$$u|v \Rightarrow u|v-u \quad \text{if } \ldots$$

both have pattern $u|v$, and can be applied to goals such as

compute $x|y+z$.

When a new goal is generated, the QLISP system retrieves those rules whose patterns match the form of the goal. This retrieval is facilitated by arranging the rules in a classification tree according to their patterns; thus the two rules above would be classified on the same branch of the tree. This mechanism allows us to avoid matching every rule in the system against each newly-generated goal.

If no rule matches the entire expression of a goal, its subexpressions are established as subgoals. If no rule matches any subexpression of a given goal, a *failure* occurs, and backtracking is invoked; the system attempts to find an alternate transformation that applies to a previous subgoal.

The QLISP pattern-matcher has special provisions for matching commutative functions. Thus, because the *and* operation is commutative, the rule

Q and true ⇒ *Q*,

represented as the QLISP program

(QLAMBDA (AND ←Q TRUE) $Q),

can be applied to goals of form "*true and Q*" as well as "*Q and true*". For this reason, commutativity rules such as

P *and* Q ⇒ Q *and* P

are not necessary in the DEDALUS system.

This kind of matching also occurs in the recursion-formation rule, in determining whether a new goal is an instance of some earlier goal. For example, in the actual synthesis of the *gcd* program, the top-level goal

compute $max\{z: z|x \text{ and } z|y\}$

is regarded as an instance of itself with the roles of x and y reversed, because the *and* function is commutative. The recursion-formation rule, therefore, is able to propose the recursive call $gcd(y\ x)$.

The final *gcd* program we obtain is

```
gcd(x y)⇐if y<x
          then gcd(y x)
          else if x≠0
               then gcd(x y−x)
               else y.
```

Currently, the DEDALUS implementation incorporates the principles of conditional formation, recursion formation (including the termination proofs), and procedure formation, but not generalization or the formation of structure-changing programs. The techniques for deriving straight-line structure-changing programs were implemented in a separate system (see Waldinger[12]).

Representative samples of the programs constructed by the current DEDALUS system are the following.

Numerical Programs:

- the subtractive *gcd* algorithm
- the Euclidean *gcd* algorithm
- the binary *gcd* algorithm
- the remainder of dividing two integers

List Programs:

- finding the maximum element of a list
- testing if a list is sorted
- testing if a number is less than every element of a list of numbers (*lessall*))
- testing if every element of one list of numbers is less than every element of another

Set Programs:

- computing the union or intersection of two sets
- testing if an element belongs to a set
- testing if one set is a subset of another
- computing the Cartesian product of two sets (*cart*).

COMPARISONS WITH AUTOMATIC PROGRAMMING

It has been claimed (e.g., see Balzer[13]) that, for a complex programming task, it is unrealistic to expect the user to formulate complete, correct specifications for the desired program. In specifying an airline-reservation system, an operating system, or a spacecraft-guidance system, for example, we are unlikely to anticipate the desired behavior of the system in every possible situation. In some systems, the specifications for the program are formulated gradually through an extended dialogue between the user and the system. (See, e.g., Green[14] and Balzer et al.,[15] or the survey of Heidorn[16].) The dialogue is continued during the program-construction process, so that the user can aid in the design of the algorithm and resolve any ambiguities or inconsistencies the system might discover. Typically, these systems attempt to play the role of an expert programmer-consultant, and they tend to rely more on built-in or acquired knowledge of a particular subject domain than on deductive processes. By concentrating on basic programming principles, we have focused on general techniques that apply to any subject domain. The ultimate automatic programming system, of course, will combine specific subject knowledge with general deductive methods.

REFERENCES

1. Buchanan, J. R. and D. C. Luckham, *On Automating the Construction of Programs,* Technical Report, Artificial Intelligence Laboratory, Stanford University, Stanford, CA., May 1974.
2. Warren, D. H. D., "Generating Conditional Plans and Programs," *Proceedings of the Conference on Artificial Intelligence and Simulation of Behavior,* Edinburgh, Scotland, July 1976, pp. 344-354.
3. Burstall, R. M. and J. Darlington, "A Transformation System for Developing Recursive Programs," *JACM,* Vol. 24, No. 1, January 1977, pp. 44-67.
4. Siklossy, L., "The Synthesis of Programs from Their Properties, and the Insane Heuristic," *Proceedings of the Third Texas Conference on Computing Systems,* Austin, TX., 1974.
5. Boyer, R. S. and J S. Moore, Proving Theorems about LISP Functions, *JACM,* Vol. 22, No. 1, January 1975, pp. 129-144.
6. Dijkstra, E. W., "Guarded Commands, Nondeterminancy and Formal Derivation of Programs," *CACM,* Vol. 18, No. 8, August 1975, pp. 453-457.

7. Sussman, G. J., *A Computer Model of Skill Acquisition,* American Elsevier, New York, NY., 1975.
8. Dijkstra, E. W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ., 1976.
9. Guttag, J. V., E. Horowitz and D. R. Musser, *Abstract Data Types and Software Validation,* Technical Report, Information Sciences Institute, Marina del Rey, CA., August 1976.
10. Wilber, B. M., *A QLISP Reference Manual,* Technical Report, SRI International, Menlo Park, CA., March 1976.
11. Teitelman, W., *INTERLISP Reference Manual,* Xerox Research Center, Palo Alto, CA., 1974.
12. Waldinger, R. J., "Achieving Several Goals Simultaneously," in *Machine Intelligence* 8: *Machine Representations of Knowledge,* (E. W. Elcock and D. Michie, eds.), Ellis Horwood Ltd., Chichester, England, 1977, pp. 94-136.
13. Balzer, R. M., *Automatic Programming,* Technical Report, Information Science Institute, University of Southern California, Marina del Rey, CA., September 1972.
14. Green, C. C., "The Design of the PSI Program Synthesis System," *Proceedings of the Second International Conference on Software Engineering,* San Francisco, CA, October 1976, pp. 4-18.
15. Balzer, R. M., N. Goldman, and D. Wile, "Informality in Program Specifications," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* Cambridge, MA, August 1977, pp. 389-397.
16. Heidorn, G. E., "Automatic Programming Through Natural Language Dialogue: A Survey," *IBM Journal of Research and Development,* Vol. 20, No. 4, July 1976, pp. 302-313.
17. Warren, D. H. D., *WARPLAN: A System for Generating Plans,* Technical Report, Department of Computational Logic, University of Edinburgh, Edinburgh, Scotland, June 1974.