STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-236

STAN-CS-74-433

ON AUTOMATING THE CONSTRUCTION OF PROGRAMS

BY

JACK R. BUCHANAN AND DAVID C. LUCKHAM

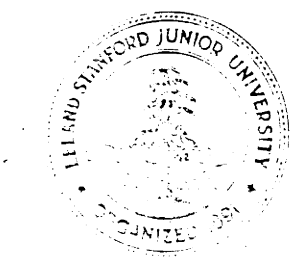
SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 2494

MAY, 1974

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences
STANFORD UNIVERSITY



STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM- 236

COMPUTER SCIENCE DEPARTMENT
REPORT STAN-CS-74- 433

ON AUTOMATING THE CONSTRUCTION OF PROGRAMS

by

JACK R. BUCHANAN and DAVID C. LUCKHAM

Artificial Intelligence Laboratory
Stanford University

May 1974

ABSTRACT

An experimental system for automatically generating certain simple kinds of programs is described. The programs constructed are expressed in a subset of ALGOL containing assignments, function calls, conditional statements, while loops, and non-recursive procedure calls. The input is an environment of primitive programs and programming methods specified in a language currently used to define the semantics of the output programming language. The system has been used to generate programs for symbolic manipulation, robot control, every day planning, and computing arithmetical functions.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract [DAHC15-73-C-0435]. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ARPA, NASA, or the U.S. Government.

1. INTRODUCTION.

We present an experimental system for writing certain simple kinds of programs automatically. The system requires as input a programming environment consisting, roughly speaking, of primitive functions and procedures, rules of composition and logical facts. If it is then given a problem it attempts to find a method of solution in terms of these rules and primitives. It will take account of certain kinds of advice from the user. Some of the techniques it uses are most decidedly "heuristic", If successful, the system will output the method of solution in the form of a plan or program in a language somewhat similar to a subset of Algol containing assignments, function calls, conditional branches, while loops, and non-recursive procedure calls. We call this language the OUTPUT (or PROGRAM) language. The forms of the definitions of the elements of the programming environment (i.e. the primitive procedures and rules of composition) correspond to axioms and rules of inference in a logic of programs currently used to define the semantics of the programming language Pascal [Hoare 1969, Hoare and Wirth 1972; see also Igarashi, London, Luckham 1973]. For example rules for constructing while loops have a form corresponding to the iteration rule. The contents of these definitions vary with the actual environment. Thus, the system can be used to generate simple Algol-like programs for robot control problems, for every-day planning, or for computing arithmetical functions.

Given a programming environment (from now on, often called a FRAME), problems to be solved are stated as pairs of conditions, the initial input condition and the goal output condition. We may regard these pairs as the input-output assertions of formulas in the logic of programs referred to above. The system is presented with an incomplete formula (i.e. a program part that satisfies the input-output assert ions is missing), and its job is to complete the formula. The construction of a solution program may therefore be formulated as a search for a proof in the logic of programs of a theorem whose input-output assertions match those of the incomplete problem formula. This enables us to justify the formal methods of the system {as opposed to the actual implementation) by showing that the formal methods will always construct correct programs.

The basic component that does most of the searching is a very simple backtrack problem reduction algorithm. It recursively applies to a given goal the primitives and rules of the programming environment to generate subgoals whose solution will imply a solution to the goal. It proved necessary to use some of the logical facts of the programming environment in special ways to evoke procedures for restricting the growth of the subgoal tree. This is often referred to as "building in" knowledge. In this case, this led to a few rather unusual complexities in the primitive language we have for defining the environment, which we call the FRAME language, The choice of special facts, as it stands at the moment, was very much influenced by our original aim to study autonomous robot planning. The set of these facts is not dependant on the environment but it probably should be. The point is that the definition of a programming environment requires not only the definitions of primitive procedures, rules of composition, and logical facts, but also some additional information about the relations in the environment as well, This

information to some extent guides the problem-solving behavior. The basis of the frame language is a free variable first order logic in which statements may have one of three truth values (TRUE, FALSE, and UNDETERMINED).

In addition to the special logical facts, certain statements about the action of the problem solver itself are useful in reducing the search. These are statements such as "when an attempt at goal A fails, do goal B before reattempting A" or "try the procedure FLY before the procedure WALK"; their usefulness usually varies from problem to problem within a given frame. We have therefore chosen not to allow such statements within the frame language, but to develop a separate ADVICE language for them. Advice can be given to the system interactively while it is attempting to produce a program. The kind of advice that can be expressed at the moment is very elementary and is not specialized towards any particular domain of program generation. The function of advice is to impose structure on the frame (more accurately, preference and relevance connections between the rules and axioms).

Certainly the class of programs that this system will construct given only input-output specifications depends on the extensiveness of the frame. If the frame contains enough primitives and rules (one might call these programming methods) and logical facts, the system ought to enable a user to program a solution to a problem without having to give much thought in advance to detailed methodology. Thus one of our examples of generated programs (Section 3) is the very simple Fibonacci program suggested in [Balzer 1972] as an example of what automatic programming systems ought to try to do. Admittedly, our frame input isn't quite so informal, but it could easily be extended to accept the recurrence equation input suggested in [Balzer 1972); this could be translated into an iterative rule in the frame by straightforward met hods (even the standard algorithm for translating linear recursive definitions to iterative form would do).

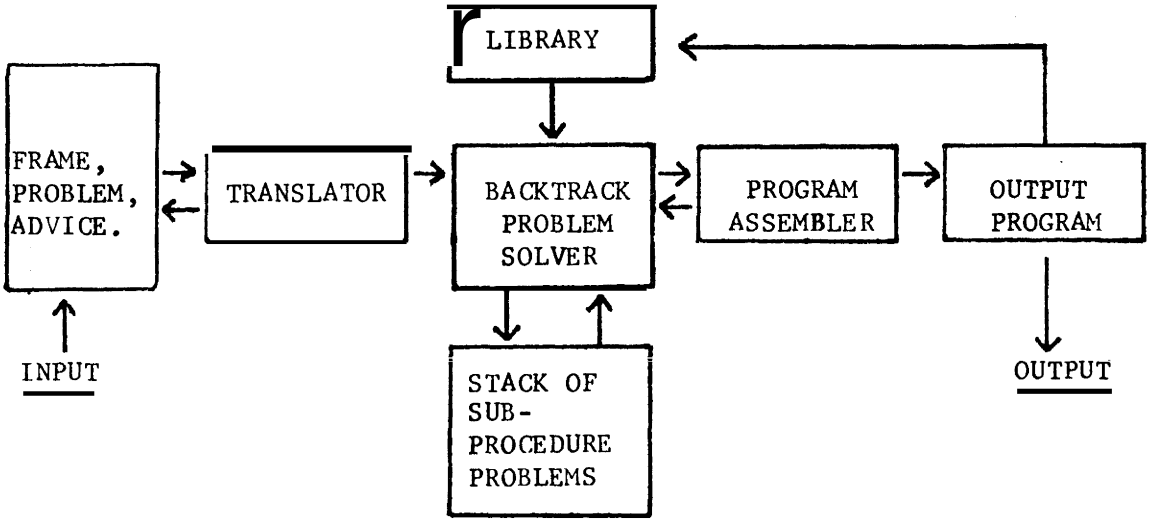


Figure 1. Main System Components

At run time the first action of the system is to translate a given frame into a backtrack problem solver augmented by special search procedures. If advice is given during a search for a solution (i.e. during the program generation phase} the translator is called and the problem solver is modified. If a solution program is found, the user is faced with a number of choices. He can ask for another program which takes the output conditions of the solution 'as its input conditions; programs can thus be constructed in segments that "fit toget her". He can choose to have the solution optimized according to some very trivial criteria, or generalized and placed on a library of nonprimitive procedures. If the solution program cont ains conditional branches calling other procedures, he can choose to have those secondary procedures constructed. Eventually he may choose to stop. Figure 1 shows the main components of the system and how they interact. We have begun to make some other additions, for example, the ability to assume the existence of non-primitive procedures, in order to try the system as an interactive aid to structured programming. The system is implemented in LISP using the primitives and backtracking facilities of MicroPlanner [Hewitt 1971, Sussman and Winograd 1972.1 In the following sections we have tried to say what the various components of the system do without going into too many details of how. Most of the algorithms are -quite straightforward so it does seem possible to do this. Wherever we omit discussion of special tricks, or inadequacies in the implementation languages force restrictions upon us, we try to leave a warning. Details of the actual implementation are given in [Buchanan 1974].

. We assume that the reader is familiar with the usual notation and terminology of first order logic and also with some straightforward concepts from the theory of subgoaling and tree searching that are explained in [Nilsson 1971]. In addition we rely on (i.e. use without defining) some of the concepts of backtrack programming which have attained fairly standard usage in many papers, and may be found in [Hewitt 1971, Sussman and Winograd, 1972]. The interest in applications to robot planning is manifest in our use of concepts such as FLUENT and NON-FLUENT etc., to be found in [McCarthy and Hayes 1969].

Section 2 presents an overview of the program generation system, and introduces some of the questions dealt with in later sections, A brief outline of the logic of programs is given and it is shown how frame definitions and the program construction rules of the system may be formulated within this logic. An example of a frame and problem is given. We indicate how a successful subgoal search for a solution may be converted into a proof within the logic of programs that the output program solves the given problem. At this point we give a sketch of how correctness proofs may be constructed in general.

Section 3 describes the language for frame definitions, the advice language and the output program language. Details of features of the system are given in the following sections: Section 4 provides a brief description of how the various problem solving and program generation processes use the extra facts provided in a frame definition, evaluation of LISP functions, and advice from the user. The methods for constructing conditional statements are given in Section 5, and for constructing iterative loops in Section 6. Section 7 illustrates how simple facilities of this

present system can be used to develop complicated programs in structured steps. Illustrative examples of frames and generated programs are given in Sections 3, 5, 6 and 7, and the appendix contains a complete interactive session.

This present system can be extended at many points. These include adding new kinds of frame rules (for constructing recursive procedures, co-routines etc.), and improving the implementation facilities, the interactive system, and the problem solver. There are many other problem domains beyond those presented in this paper where the possibility of using the present system to generate procedures for solving problems exists. For example, its application to generating assembly and repair programs for simple machinery is illustrated in [Luckham and Buchanan, 1974]. At some point in these developments it will certainly pay to construct specialized systems for particular classes of frames. Additional special features common to frames in each class can be then used as built-in assumptions to speed up the problem solver, make the frame and advice languages more natural, and build up the program library.

What has been demonstrated thus far by the system presented here is (i) the current axiomatic theory of defining the semantics of programming languages can be used with slight modifications to define many other simple but useful problem environments; (ii) there are straight-forward techniques for translating declarative descriptions into procedural descriptions for problem solving; (iii) standard problem-solving met hods can be used to synthesize programs in a structured way on the basis of given specifications, and to handle some burdensome details.

2. LOGICAL BASIS AND OVERVIEW

We begin by describing how frames and the program construction methods of the system can be formulated within the Logic of Programs. The soundness of frames and correctness of programs are discussed. A brief description of the underlying problem-solving algorithm of the system is given. We then outline proofs that under certain assumptions the programs constructed by the system will be correct. The presentation here is intended to be informal and to serve as an introduction to the later sections;many details are left unmentioned until later, and statements of the correctness results are weaker and more restricted than they need be. Extensions of the correctness proof are discussed in later sections.

NOTATION: x,y,z,u,v,w...variables,
X,Y,Z,... lists of variables,
f,g,h.... functions,
s,t... functional terms,

G,I,P,Q,R,S,... Boolean expressions (essentially formulas of first order logic with standard functions and predicates for equality, numbers, lists and other data types),

P(X) denotes the formula obtained by replacing each free variable in P by a new variable from X,

$(\exists X)P(X)$ denotes existential quantification over all X-variables in P(X),

A,B,C,... programs and program parts in an Algol-like plan language (details in Section 3),

p,q,... procedure names,

$\alpha,\beta,\lambda,...$ substitutions of terms for variables, also denoted by $(\langle x \leftarrow t \rangle)$.

P(t) denotes the result of replacing x by t everywhere in P(x),

$\alpha\beta$ denotes the COMPOSITION of α and β; $E\alpha\beta = (E\alpha)\beta$ for all expressions E.

We assume the existence of a fixed arbitrary ordering of literals (atoms and negations of atoms).

2.1 LOGIC OF PROGRAMS

We review briefly the elements of an inference system for proving properties of programs [Hoare 1969]. Further details may be found in [Igarashi, London, Luckham 1973].

STATEMENTS of the logic are of three kinds:

(i) Boolean expressions, (henceforth often called ASSERTIONS)

(ii) statements of the form P{A}Q where P,Q are Boolean expressions and A is a program or program part.

P{A}Q means "if P is true of the input state and A halts (or halts normally in the case that A contains a GO TO to a label not in A) then Q is true of the output state".

(iii) Procedure declarations, p PROC K where p is a procedure name and K is a program (the body of p).

A RULE OF INFERENCE is a transformation rule from the conjunction of a set of statements (premisses, say $H_1,...,H_n$) to a statement (conclusion, say K) of kind (ii). Such rules are denoted by

$$\frac{H_1,...,H_n}{K}$$

The concept of PROOF in the logic of programs is defined in the usual way as a sequence of statements that are either axioms or obtained from previous members of the sequence by a rule. A proof sequence is a proof of its end statement.

NOTATION: We use H ||- K to denote that K can be proved by assuming H. H |- K denotes the same thing for first order logic. It is sometimes helpful to denote statements that are problems or subproblems for the program generator to solve by P{?}Q.

2.2 FRAMES AND PROBLEMS

We restrict our discussion to problems that can be represented in the following general form.

The problem representation consists of two elements:

(1) F - a set of rules (or laws) called the ENVIRONMENT (or FRAME)

(2) The problem, which is a pair <I,G>:

I - an input assertion (or initial state).

G - output assertion (or goal).

The RULES in F are of at least three kinds:

(a) PROCEDURES: transforming states into states;

(b) SCHEMES: methods for constructing programs;

(c) RELATIONAL LAWS: definitions and axioms which hold in all states and serve to "complete" incomplete state descriptions by permitting deduction of other elements of a state from those given.

The PROBLEM is the problem of transforming I into G using the rules of F. A SOLUTION is a sequence of rules that transforms I to G.

REMARKS:

1. For the purposes of discussing the present system we can make the following restrictions:

(i) The language of assertions is very similar to Aigoi Boolean Expressions (as referred to above).

(ii) Procedure rules and schemes are expressed as statements and as rules of inference (respectively) in the logic of programs.

(iii) The underlying logic of the relational laws is first order logic,

(iv) The logic of the procedures and schemes is the logic of programs,

2. We probably ought to permit other kinds of rules in F, e.g. rules for evaluating stat es, comparing stat es etc.

NOTATION and RESTRICTIONS: Q U F $\supset$ R denotes that R is a logical consequence of Q and the axioms of F. Assertions describing states are denoted by I,I',...,G,G',... These assertions (but not the assertions in rule definitions) are restricted to be conjunctions of atomic assertions. We write R∈I to denote that R is a conjunct in I. L(F) denotes the logic of F,i.e. the set of consequences of the rules of F. Substitutions α do not replace any variable that occurs in the initial state I. Expressions, ail of whose variables occur in the initial state are called "fully instantiated".

STANDARD FRAME RULES: A set of standard rules are assumed to be part of every frame. These are rules implemented in the program construction methods of the problem solving aigorit hm:

RO. Assignment Axioms:

(i) Simple Assignment: $P(t)\{x \leftarrow t\}P(x)$

(ii), Conditional Assignment: $(\exists Z)P(Z)\{\text{IF } P(W) \text{ THEN } Y \leftarrow W\}P(Y)$
$\neg(\exists Z)P(Z) \wedge Q(Y)\{\text{IF } P(W) \text{ THEN } Y \leftarrow W\}Q(Y)$

where Y-variables in P(Y) do not occur in P(W), W-variables are special variables occurring only in conditional assignments, and $Y \leftarrow W$ denotes the sequence of simple assignments between members of Y and W that occur in the same argument positions in P(Y) and P(W).

R1. Rule of Consequence: $\dfrac{P \supset Q, Q\{A\}R \quad P\{A\}Q, Q \supset R}{P\{A\}R \qquad P\{A\}R}$

R2. Rule of Composition: $\dfrac{P\{A\}Q, Q\{B\}R}{P\{A;B\}R}$

R3. Rule of Invariance: if $P\{A\}Q$ and $I \cup F \supset P$ then $I\{A\}Inv(Q,I)$
where if $R_1, R_2, \ldots, R_n$ are the conjuncts of I
in the fixed order, then $I_1 = Q$,
for $0 \leq m < n$, $I_{m+1} = I_1 \wedge R_m$ if $\neg(I_m \cup F \supset \neg R_m)$
$I_{m+1} = I_1$ otherwise,
and $Inv(Q,I) = I_1$.

R4. Change of Variables: $\dfrac{P(x)\{A(x)\}Q(x)}{P(y)\{A(y)\}Q(y)}$ where y is not a special variable,

R5. Conditional Rule: $\dfrac{P \wedge Q\{A\}R, P \wedge \neg Q\{B\}R}{P\{\text{IF } Q \text{ THEN A ELSE } B\}R}$

R6. Undetermined values: If $I'\{?\}G$ cannot be solved and $\neg(I' \cup F \supset \neg G)$ then G is UNDETERMINED in I'.

STANDARD RULES

REMARKS: The axioms R0(ii) define the semantics of conditional assignment statements, The occurrence of P(W) within the IF statement is interpreted as a call to a procedure with variable parameters W, the result of which is to bind those W-parameters to values that make the Boolean statement P(W) true, if such values exist. We have adopted a convention on W-variables, $w_1, w_2, \ldots$ whereby they occur only in conditional assignments as above, and indicate the use of an atomic assertion as a procedure call (we call them "special variables"). This eliminates the need for explicit Skolem "successor" functions for each relation in the frame. Note that if $\neg(\exists Z)P(Z)$ is true of the input, then the rule "says" that the THEN part of the IF statement is not executed.

invariance states that things stay the same unless it can be proved that they conflict. This is a way of dealing with the "frame problem" [McCarthy and Hayes 1969], but it does force the user into being careful about stating what does change. Invariance can be derived within the logic of programs from a rule which states that procedures do not have side effects. Undetermined values is a rule for deciding when to construct conditional statements (section 2.4). The change of variables rule is an instance of the rule of substitution (see [Hoare 1969] for this and the remaining rules). Usually, restrictions are placed on R4 to maintain consistency. In this system the use of the assignment axioms R0 is restricted. However, the user can introduce a primitive assignment procedure (see below) which would not be restricted in its use; in this case he should use a formulation which distinguishes between a variable and its value.

INPUT FRAME RULES; In addition to the standard rules, a frame may contain rules of the following types (these constitute the user defined elements of the frame):

S1. Primitive procedures (or operators): the rule defining procedure p is of the form P{p}Q. The assertions P and Q are the pre- and post-conditions of p. p must contain a procedure name and parameter list.

$2. Iterative rules: an iterative rule definition containing the Boolean expressions P(basis), Q(loop invariant), R(iteration step goal), L(control test) and G(rule goal) is a rule of inference of the form:

(a) $P, \vdash Q,\ Q \wedge L\{?\}R,\ R\{??\}Q \vee \neg L$

$$P\{\text{while } L \text{ do } ?;??\}G$$

where the free variables of R and L occur in Q. Such rules are permitted not to contain P or L, in which case they correspond to inferences of the form:

(b) $Q,\ Q \wedge \neg G\{?\}R,\ R\{??\}Q \vee G$

$$Q\{\text{while } \neg G \text{ do } ?;??\}G$$

S3. Definitions. A definition of G in terms of P is a logical equivalence $\vdash P \equiv G$.

S4. Axioms. A frame axiom P is a logical axiom $\vdash P$.

Terms and predicates in assertions may contain calls to LISP functions. If the frame definition contains functional terms or predicate tests that are evaluated by calls to LISP functions, the set of premisses must be expanded to include both the input-output assertions for these function calls and the logical axioms for the relevant data types.

REMARKS (i) The iterative schemes S2 permit the definition of methods for constructing loops; they are instances of:

WEAK ITERATION RULE: $\frac{Q \wedge L\{B\}Q \vee \neg L}{Q\{\text{WHILE } L \text{ DO } B\}\neg L}$

where Q is the invariant of the loop. The meaning of |-Q in the premiss is that the rule may only be applied in states where Q is -a first order consequence of the state description. The program part ?? is restricted to be a sequence of assignment statements (see Section 6). (ii) Inconsistencies may arise in several different ways in frames. The axioms can be inconsistent, or the post conditions of a rule can be inconsistent with the axioms. Also the elements of iterative schemes must satisfy some simple consistency criteria (section 6). (iii) Note that each frame rule has a goal. The goal of a procedure is its postcondition; the goal of an axiom or definition is its consequent. If invariance (R3) is applied to program part A constructed from applying a single frame rule,then Q is the goal of that rule.

The following lemma is useful in proving properties of conditional assignments [Igarashi,London,Luckham 1973]:

OR-LEMMA __ $\frac{P\{A\}Q,\ R\{A\}S}{P \vee R\{A\}Q \vee S}$

EXAMPLE: Next, we show how a rather simple problem can be stated within our frame formalism. This leads us very quickly into the further questions of (i) defining simple general methods of finding solutions, (ii) formulating the correctness of solutions, and (iii) the correctness of solutions obtained in frames that have unintended or nonstandard interpretations.

Consider the following frame and problem:

INPUT FRAME RULES:

1. Procedure: st andon

$AT(x,y) \wedge AT(z,y) \wedge ROBOT(x) \wedge BOX(z)\{standon(x,z)\}ON(x,z).$

F2. Procedure: step-up

$ROBOT(x) \wedge ON(x,y) \wedge STACKED(z,y)\{step\text{-}up(x,y,z)\}ON(x,z).$

F3. Iterative Rule: climb

$$\frac{ROBOT(M) \wedge ON(M,y) \wedge STACKED(u,y) \wedge \neg ONTOP(M)\{?\}ON(M,u)}{ROBOT(M) \wedge ON(M,y) \wedge STACKED(u,y)\{\text{WHILE } \neg ONTOP(M) \text{ DO BEGIN } ?;?? \text{ END}\}ONTOP(M)}$$

F4. Axiom: ROBOT(x)∧∃y(ON(x,y)∧∀z¬STACKED(z,y))↔ONTOP(x).

PROBLEM:

I: ROBOT(M)∧BOX(B 1)∧BOX(B2)∧BOX(B3)∧AT(B1,L)∧AT(M,L)
∧STACKED(B2,B 1) A STACKED(B3,B2).

G: ONTOP(M)

PROBLEM 1: CLIMBING

COMMENTS ON PROBLEM 1:

i. The iterative rule says "A solution to the problem of climbing one box at a time, can be used to construct a WHILE loop that solves the problem of climbing a stack of boxes". The rule defines the meaning of WHILE in the environment. Or, if we regard WHILE as a primitive constructor whose meaning we understand, the rule is an induction principle for the environment.

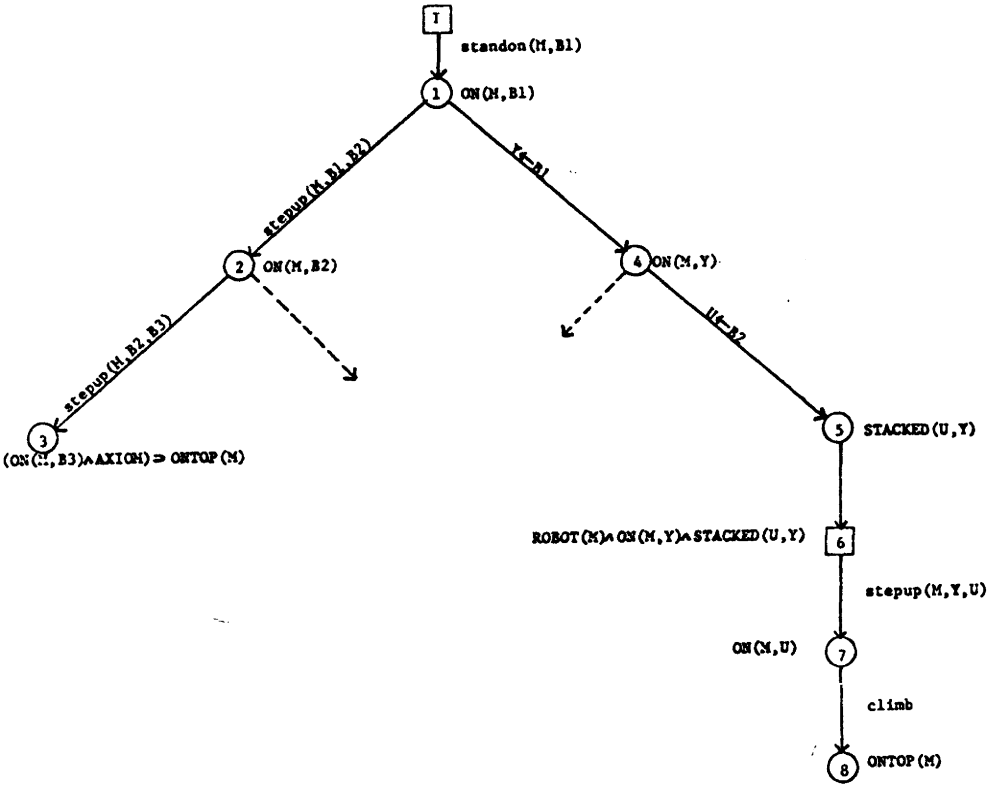
ii. The program part ?? in the conclusion of the iterative rule transforms the situation after the execution of the loop body (?) back into one in which the invariant is again true if the test is true;

ON(x,u){??}ROBOT(x)∧ON(x,y)∧STACKED(u,y).

We restrict ?? to be a sequence of assignments.

iii. The goal of climb is ONTOP(M), the negation of the control test in this example,

Steps taken by a search procedure in solving this problem are shown in Figure 2. It starts with state situation I and determines by logical reasoning from I and the axioms which operators have pre-conditions that are true in I . It applies these operators and updates the state to the new state using the rule of invariance. It repeats this process on the new stat es. Node 6 indicates the initiation of a subproblem (the premiss of the iterative rule) with a new initial state (the invariant) which is a subset of the state above it at Node 5.



SEARCH FOR SOLUTIONS TO THE CLIMBING PROBLEM
Figure 2

The solutions corresponding to the paths shown in figure 2 are:

(i) I{standon(M,B1);stepup(M,B1,B2);stepup(M,B2,B3)}ONTOP(M).

(ii) I{standon(M,B1);y←B1;u←B2;
WHILE ¬ONTOP(M) DO BEGIN
stepup(M,y,u);
y←u;
IF STACKED(w,y)THEN u←w;
END}ONTOP(M)

where the assignments within the WHILE loop correspond to the ?? of the iterative rule. The variable w is a special variable,

NOTE: It looks as though solution (ii) is more general than solution (i).

Using the frame rules we can now construct a proof of the statement I{solution}G within the logic of programs.

1 . I⊃(ROBOT(M)∧AT(M,L)∧AT(B1,L)∧BOX(B1))

2 . I{standon(M,B1)}ON(M,B1)∧STACKED(B2,B1)∧ROBOT(M) 1,F1,R4,R1,R3

3 . ON(M,B1)∧STACKED(B2,B1)∧ROBOT(M){y←B1;
u←B2}ROBOT(M)∧ON(M,y)∧STACKED(u,y) R0(i),R2,R3

4. I{ st andon(M,B 1);y←B1;u←B2}ROBOT(M)∧ON(M,y)∧STACKED(u,y) 2,3,R2

5 . ROBOT(M)∧ON(M,y)∧STACKED(u,y){stepup(M,y,u)}ON(M,u)∧ROBOT(M) F2,R4

6. ROBOT(M) ∧ON(M,u) {y←u}ROBOT(M)∧ON(M,y) R0,R3

7. ON(M,y)∧∃zSTACKED(z,y){IF STACKED{ w,y)THEN u←w}ON(M,y)∧STACKED(u,y) R0,R3

8 . ¬∃zSTACKED(z,y)∧ONTOP(M){IF STACKED(w,y)THEN u←w}ONTOP(M) R0

9. (ON(M,y)∧∃zSTACKED(z,y))∨(¬∃zSTACKED(z,y)∧ONTOP(M))
{IF STACKED(w,y)THEN u←w}(ON(M,y)∧STACKED(u,y))∨ ONTOP(M) OR-Lemma 7,8.

10. ROBOT(M)∧ON(M,y)∧¬(∃z)STACKED(z,y) ⊃ ONTOP(M) F4,
⊃(ON(M,y)∧∃zSTACKED(z,y))∨ONTOP(M)
ROBOT(M)∧ON(M,y)∧ ∃zSTACKED(z,y) ⊃ (ON(M,y)∧∃zSTACKED(z,y))∨ONTOP(M)
ROBOT(M)∧ON(M,y) ⊃ (ON(M,y)∧∃zSTACKED(z,y))∨ONTOP(M)

11. ROBOT(M)∧ON(M,y)∧STACKED(u,y){stepup(M,y,u);y←u;
. IF STACKED(w,y)THEN u←w}(ON(M,y)∧STACKED(u,y))∨ ONTOP(M) 5,6,10,9,R2,R1

12. ROBOT(M)∧ON(M,y)∧STACKED(u,y){ WHILE¬ONTOP(M) DO . ..}ONTOP(M) 11 ,R1 ,F3

13. I{solution (ii)}ONTOP(M) 4,12,R2

PROOF of I{solution (ii)}G

We refer to a formal proof of L(F)||-I{A}G as a correctness proof. The existence of such a proof implies only that the program is correct relative to the frame. Thus it is easily seen that the final state implies (∀x)(BOX(x)⊃ON(M,x)), hardly a situat ion we had intended, but which arises from the invariance rule owing to our not having axioms such as,

ON(M,x) ∧ON(M,y)⊃x=y.

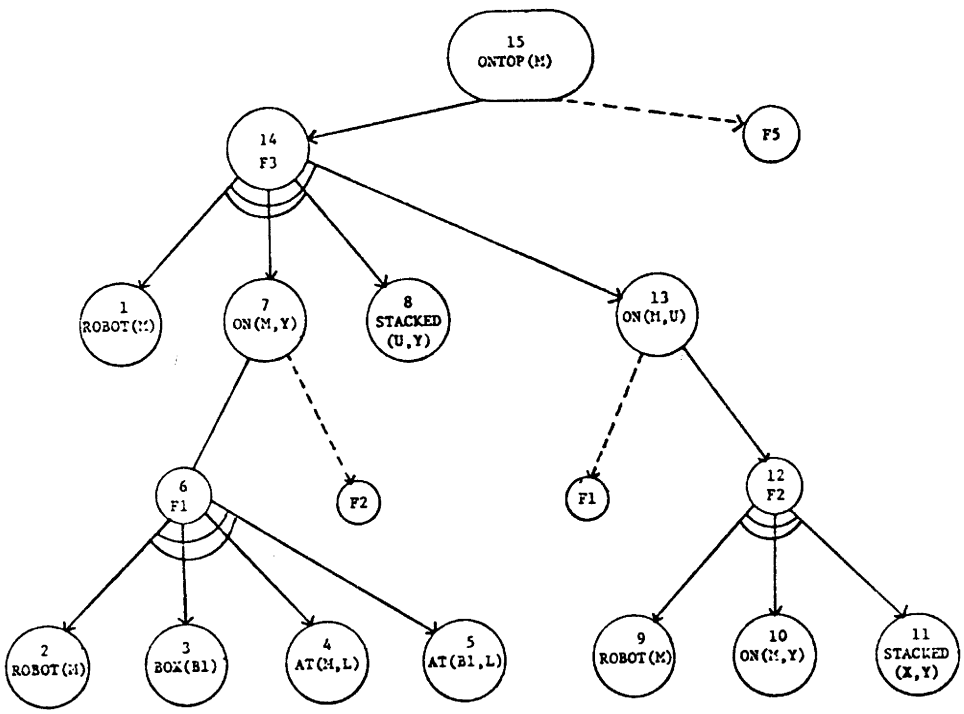
In other words, our frame admits non-standard models.
We could extend the frame by adding this additional logical axiom and go back to solving the problem all over again. But this would have to be repeated if some other non-standard model was discovered still later. We ought to be able to do better than that!

Now, solution (ii) may still be "correct" (or solve the problem) in the extended frame. And we can determine this from the proof of I{solution (ii)}ONTOP(M) by checking to

see if any step uses facts from an intermediate state situation I' that contradict the extra logical rule. In other words, we can "run" the proof on the new world with a special consistency check against the additional facts. This ought to be much easier than solving the problem again from scratch.

The proof above formalizes (i.e. provides a description for the purposes of analysis) WHAT it is the problem solver has finally done when it has solved the problem. It is a record of those features of the frame and initial state that were essential in constructing the solution. For example, we have actually proved
ROBOT(M)∧BOX(B1)∧STACKED(B2,B1)∧AT(M,L)∧AT(B1,L){Solution(ii)}ONTOP(M)
within L(F). This proof did not use BOX(B2),BOX(B3),or STACKED(B3,B2). If there was a stacking operator in the environment, we could alter the proof--without having to resort to the problem solver again -- to eliminate the hypothesis "Stacked (B2,B1)". It will be noticed that a similar proof for solution (i) uses more properties of I; solution (i) IS less general.

It is therefore plausible that a correctness proof for a solution program will be useful in answering further questions about that program such as: Does it solve this new problem? Can it be altered to solve a given new problem? Are there problems it will work on that another program won't?



PROBLEM 1: THAND-OR-AND TREE SEARCH
Figure 3

2.3 THE FORMAL PROBLEM SOLVING ALGORITHM

To automate solving simple problems of this kind it is sufficient to use a straightforward problem reduction search [Nilsson]. Figure 3 illustrates the depth first reduction of goals to subgoals using the input frame rules (as described below) until subgoals are reached that are true in the current state. In figure 3, there are two kinds of nodes, Goal nodes and Rule nodes corresponding to the separate steps of (1) choosing a rule to use, and (2) generating the subgoals necessary to apply that rule. Goal nodes may be any combination of THAND,(defined below) OR, AND, but Rule nodes are always OR nodes [Nilsson1971]. The arrows from each rule node point to its immediate subgoals. If a node reduces to an OR of its subgoals (which are thus OR- nodes), it has no angle mark; if it reduces to a THAND of its subgoals the relevant arrows are connected by one angle mark; an AND of subgoals is denoted by two angle marks. Each rule node is labelled <n,Fm> where n is the order in which it was achieved (omitted if it was not) and Fm is the frame rule used; similarly goal nodes are labelled <n,Gm>.

We give an informal description of the reduction algorithm (or subgoaler) in the simple case where it does not contain the rule of undetermined values, as follows:

The subgoaler computes on a triple, <G',I',A>, where G' is the subgoal to be attempted next, I' is the description of the current state, and A is the current partial answer. Let $\propto$ be a substitution that replaces variables by terms from I (the initial state). Nodes in the subgoal tree are developed by using input rules in F: if a rule of F has a conclusion or postcondition Q such that $Q\propto$ = G' then the rule is USED to develop the node by appending its premisses or preconditions $H_1\propto,...H_n\propto$ as subgoals of G'. Q is said to match G'.

A goal G' is ACHIEVED in one of four ways:

(a) if there is an $\propto$ such that I' U F $\supset$ G'$\propto$,

(b) if not (a), then G' is developed using an instance of a frame rule with post-condition (or goal} $Q\propto$. Let the immediate subgoals of G' be G1*G2 where * is the principle connective in the preconditions of the frame rule, so that G1 and G2 are *-nodes. In this case, G' is ACHIEVED if:

(i) one of G1 or G2 is achieved (in the case * is OR),

(ii) both G1 and G2 are achieved (in the case * is THAND),

(iii) both G1 and G2 are achieved (in that order, say) and the updated state {defined below) that results from achieving G2 also satisfies G1 (in the case * is AND).

If G' is achieved under (a) there is no change in the current state and answer. However, in case (b), both are UPDATED as follows: let I' be the current state resulting from achieving G 1 *G2; the state resulting from achieving G' is Inv(Q∝,I'). A is composed (by R2) with the procedure call or while statement corresponding to the rule that was used to develop G'.

A node in the THAND-OR-AND tree FAILS when the goal associated with the node cannot be achieved - essentially because it is not true of the associated state and either no rule can be applied to reduce it or one of its subgoals is not achievable. Whenever a goal node fails, the search procedure (simplest form) "BACKS UP" to the goal node immediately PRECEDING it and attempts the next OR-possibility for that goal. The search is DEPTH FIRST.

Thus, an AND assertion is achieved when all of its elements (subgoals) have been achieved simultaneously in the same state; a THAND assertion requires only that its subgoals be achieved in some order but not necessarily simultaneously.

This simple kind of search algorithm can be implemented quite easily using the goal tree generation, automatic backtrack and data base access functions of MICRO PLANNER [Hewitt 1971, Sussman and Winograd 1972], or any of the other current problem solving languages. However,it is necessary to distinguish between the formal algorithm and the implementation since the latter can only approximate some of the formal rules.

THE UPDATE PROBLEM. The updating of a state to the new state resulting from the application of an inpyt rule is formulated by invariance. In general the rule of invariance is not computable, but even in cases where it might be, it is IMPRACTICAL. The implementation of this rule has to fall short of its formulation. Inconsistencies in the state description are almost certain to arise eventually. We can try to delay this by paying special attention to those axioms that are most likely to be transgressed (e.g. uniqueness and single-valuedness properties). The case of ITERATIVE rules provides a particular difficulty since the rule goal G may not provide enough information about what went on during the iterations of the loop body to continue planning after an application of such a rule. We allow the user to specify an output assertion as part of an iterative rule, in which case invariance is applied using this assertion in place of the usual rule goal (see section 6).

2.4 CONDITIONALS.

Extending the description of the goal reduction algorithm to include the rule of undetermined truth values follows closely the actual system implementation discussed in Section 5. Here we give some motivation for rules R5 and R6.

Conditional statements are constructed whenever an undetermined goal occurs. The notion of undetermined truth value used here is an operational one. The problem solver wants G' to be true in I', G' is not true in I', no way of making G' true can be found, and G' is not false in I'. In such cases, the algorithm continues by splitting its

problem into two subproblems: to solve a more global problem G* say, (a) assuming G' is true and (b) assuming G' is false.

For example, relative to the frame in problem 1 we can pose a second problem, I1{?}ONTOP(M) where I1 differs from I only in not containing the assertion AT(M,L). Our solution (ii) above is no longer a solution to this new problem since AT(M,L) is not true in I1 (neither is it known to be false!) and there is no way of achieving it, Using R6 and R5,t he extended algorithm can construct the solution;

(iii) I1{IF¬AT(M,L) THEN CALL PROC1(M,L) ELSE
BEGIN
st andon(M,B 1);y←B 1 ;u←B2;
WHILE¬ONTOP(M) DO
BEGIN st epup(M,y,u); y←u;
IF STACKED(w,y) THEN u←w;
END
END}ONTOP(M).

and the proof of correctness of solution (ii) can be extended to a proof of I1 (solution (iii) }ONTOP(M).

The implement at ion of these rules is complicated by considerations such as the following.

{a} A stack is required for the subproblems for cases when undetermined subgoals are assumed false, i.e. subproblems for the form I'∧¬G'{PROCN}G*.

(b) Criteria for the choice of G* are required. For example, the contingency problem above is I1∧¬AT(M,L){PROC1(M,L)}ONTOP(M). Although the problem solver has found that it cannot solve I1{?}AT(M,L), there is no reason to suppose that this is a good choice, or indeed that it can be solved. We might have chosen I1∧¬AT(M,L){PROC1}ON(M,B1) instead.

(c) The order in which goals are attempted may affect not only whether a solution can be found, but also whether the solution is sensible.

(d) Undetermined truth values can also arise as a result of applying unreliable operators, for example;
AT(hand,x)∧AT(object,x){lift(hand,object)}HAS(hand,object)∨ DROPPED(hand,object).

We shall consider these problems in detail in Section 5.

2.5. CORRECTNESS OF SOLUTIONS

In the previous examples we showed that if the frame rules were taken as assumptions then the solutions could be proved within the logic of programs to solve the problems.

This is what we mean by the CORRECTNESS of the solutions. The proofs require the standard rules, but these are all rules of the logic of programs, with the exception of invariance and undetermined values. A proof of correctness of a solution generated by the formal problem solving algorithm, based on the frame in which the problem was posed,can be given in every case. This does not guarantee the correctness of every actual solution since, as we have seen, the implementation only approximates certain rules of the formal algorithm. It is a justification of the formal methods. In addition it provides a measure of confidence in actual solutions relative to the soundness of the frame (which is the user's responsibility) and to the degree to which unsound heuristics in the implementation have been invoked in finding a solution. In fact, the result allows us to state sufficient conditions under which actual solutions will be correct, but we will not do that here.

To establish this result it is necessary to prove (a) a successful search tree of the formal algorithm has certain properties, and (b) a tree with those properties can be transformed into a correctness proof of the solution. We shall state without proof the properties of successful searches, and then give the details of step (b).

Let us first consider the very restricted case where (a) no calls to LISP functions take place, (b) no undetermined goals occur, and (c) no iteration rules are used. We assume that the problem is stated in the form I{?}G where G contains only variables occurring in I.

The subgoaling algorithm treats $\vee$ (or) as exclusive; in order to achieve $P(x) \vee Q(x)$ it tries to achieve P(x) and if this fails it tries Q(x). When the subgoaler completes a successful computation it has constructed a goal tree, Tr say, and a substitution $\propto$. Tr consists solely of goal nodes (the single rule node between a goal and its subgoals in the completed search tree can be eliminated and the arrows leading directly from the goal to its subgoals labelled by the rule name). Tr and $\propto$ have the following properties;

(1) each node of Tr has associated with it the number n if it was the nth node to be achieved, a Boolean expression G(n) (its goal), a program part A(n), and a state condition I(n),

(2) $\propto$ substitutes terms from I for variables in Tr,

(3) $I \cup F \vdash G(1)\propto$,

(4) if G(n+1) is at a leaf node then $I(n) \cup F \vdash G(n+1)\propto$,

(5) if G(n+1) is not at a leaf node then it is related to its immediate subbgoals G(k),...,G(n) by a procedure P{p}Q or a definition P≡Q such that $Q\propto = G(n+1)\propto \wedge Q'\propto$ and $P\propto = G(k) * \ldots * G(n)$, where * is either AND or THAND. G(n+1)is achieved from I(n).

(6)In cases 3 and 4,and where a definition was used to develop G(n+I), I(n+1)=I(n) and A(n+1)=A(n); in the case of a procedure call of the form $P\propto\{p\propto\}Q\propto$, I(n+1) is $In \vee (Q\propto, I(n))$ and A(n+1)=A(n);$p\propto$. Finally, the property that G(n+1) is achieved from I(n) implies that $I(n) \cup F \vdash P\propto$. (NOTE: this use of "|-" is an extension of the usual not ion of

first order proof in the case when $P\alpha$ is a THAND; however it is easily seen that THAND connectives may be eliminated from frames by introducing extra definitions, so the extension is not essential.)

Let the root of Tr be the m_{TH} node. We may prove that the output program A(m) solves the problem, i.e., $L(F) \vdash I\{A(m)\}G$, (here $G(m)=G$) by proving a similar result for each intermediate goal and partial answer. Namely, for each $n \leq m$, $L(F) \vdash I\{A(n)\}I(n)$ and $I(n) \supset G(n)\alpha$ can be proved by induction on n. The cases are as follows.

First, $L(F) \vdash I \supset G(1)\alpha$ by property (3) above

Now assume $L(F) \vdash I\{A(n)\}I(n)$.

If G(n+1) is at a leaf node then $I(n) \cup F \supset G(n+1)\alpha$, $I(n+1)=I(n)$, and $A(n+1)=A(n)$. Thus $L(F) \vdash I\{A(n+1)\}I(n+1)$ and $L(F) \vdash I\{A(n+1)\}G(n+1)\alpha$ by the rule of consequence R1.

If G(n+1) is not a leaf node then $I(n) \cup F \vdash P\alpha$ by property (5) above. If G(n+1) is related to its immediate subgoals by a procedure, say $P\{p\}Q$, then $P\alpha\{p\}Q\alpha$ is derivable by the change of variables rule R4. The rule of consequence implies $L(F) \vdash I(n)\{p\alpha\}Q\alpha$ and invariance implies $L(F) \vdash I(n)\{p\alpha\}I(n+1)$. Rule R2 allows the composition of this with the inductive assumption so that $L(F) \vdash I\{A(n);p\alpha\}I(n+1)$. Finally $I(n+1) \vdash G(n+1)\alpha$ since $Q\alpha = G(n+1)\alpha \wedge Q'\alpha$. The case when G(n+1) is related to its subgoals by a frame definition is straightforward.

Thus, by induction on n we can prove $L(F) \vdash I\{A(m)\}I(m)$ and $I(m) \supset G\alpha$. Finally we note that if G contains only variables occurring in I then $G\alpha=G$. Therefore, we have proved $L(F) \vdash I\{A\}G$.

The extension of this proof for the case when there are undetermined goals is given in Section 5, and for the case when iterative rules are used in Section 6.

3. DEFINING THE PROGRAMMING ENVIRONMENT

In this section the Frame definition formalism is presented. This includes the Frame language the Advice language, and the output Program language. A complete example of an input frame, together with advice, and the resulting output program is given.

3.1 FRAME LANGUAGE

3.1.1 ASSERTIONS: The syntax for assertions used in definitions of rules, axioms and state descriptions is shown in Figure 4.

<variable> ::= <identifier>
<function symbol> ::= <identifier>
<predicat e symbol> ::= <identifier>
<t erm> ::= <variable>|(<function symbol>)|
(<function symbol><argument list>)
<argument list> ::= <term>|<term>,<argument list>
<functional term> ::= (EV<term>)|(EVN<term>)|<term>
<atomic formula> ::= <predicate symbol>(<predicate argument list>)
<predicat e argument list> ::= <funct ional t erm>|<funct ional term>,
<predicate argument list>
<lit eral> ::= <at omic formula>|¬<at omic formula>
<literal element> ::= <lit eral>|REQUEST(<literal>)|{<assertion>}
<disjunction> ::= <literal element>|<literal element><or><disjunction>
<assert ion> ::= <disjunction>|<disjunction><and><assertion>
<and> ::= ∧|&
<or> ::= ∨|⊗

SYNTAX OF ASSERTIONS
Figure 4.

Ident ifiers are strings of characters not containing the negation symbol, "¬", nor the usual LISP delimiters, e.g., blanks, commas or parentheses. The <or> connectives have higher precedence than the <and> connectives and a logical condition is terminated by a semicolon, ";".

The only constructs whose meaning requires special explanation are <functional term>, <literal element>, and the connectives "&" and "⊗".

If a term is in the scope of the modifier "EV" then all functions in that term are applied to their arguments (i.e. evaluated as LISP functions) when that literal is used in the problem-solving process. "EVN" further specifies that the functions to be evaluated have numerical values. The default convention is that the term is manipulated as an unevaluated symbolic expression. The "REQUEST" modifier, which takes a literal as its argument, alters the way that literal is treated by the problem solver. This is discussed in Section 4.

The AND connective is denoted by "∧". Thus a state satisfies the assertion A∧B if it

satisfies both A and B. The weaker THAND connective is denoted by & (Section 2). Exclusive OR is denoted by "⊗".

3.1.2 STATE DESCRIPTIONS: Assertions specifying states are restricted to be conjunctions of literals.

3.1.3 AXIOMS: Axioms are stated in either of the forms $P \supset Q$ or P, where P and Q are assertions. They hold in all states and are used to complete a given state description by deduction of other elements of a state from those given.

3.1.4 RULES: There are three types of rules: primitive procedures, definitions, and iterative rules.

(a) A primitive procedure is specified by a name, an argument list, and its pre and post - conditions, i.e.

> $P\ \{f\ (x_1,...,x_k)\}Q$ where P and Q are assertions in which $x_1,...,x_k$ are free, and f is the procedure name.

The variables are formal parameters of the procedure. They may be "bound" by substitution of actual parameters when the procedure is applied to a state.

When a primitive procedure is defined it may be declared to be an ASSUMPTION. If it is used in a successful program construction, then the user is informed and is given the opportunity to carry out a structured program development of this non-primitive operation. This is described in Section 7.

(b) A definitional rule is of the form $R \equiv S$ where R and S are assertions. The relation, S, is given as the post- condition of the rule. The meaning of a definition is that whenever it is desired that S be true it is equivalent to establish the truth of R. A definition is often used to shorten assertions in rules by defining a single relation as equivalent to an often used condition.

(c) iterative rules specify conditions that if satisfied justify the assembly of a "while" loop to achieve the associated goal. They are instances of the iterative rule S2 in Section 2.2, and are defined by giving:

- (i) A name, e.g. TLOOP, (without parameters).
- (ii) A basis assertion P.
- (iii) A loop invariant assertion Q that specifies relations that must be true in the state prior to each iteration.
- (iv) An iteration step assertion R that specifies the goals to be achieved during an execution of the loop body.
- (v) An iterative goal G, the assertion considered achievable by the iterative process.
- (vi) The format of iterative rules also allows the specification of a loop control test L and an output assertion S if they differ from G.

The rule,

TLOOP
P;Q;R;G;L;S;
where P,Q,R,G,L and S are assertions,
defines the iterative rule **"TLOOP"**
associated with the goal G.

3.1.5 SPECIAL AXIOMS: After the rules and initial state have been defined the system requests the following information for each predicate symbol P that has been mentioned. The system use of this information is discussed in Section 4.

a) "Is P a function of the state?" The intent of this classification is to separate those relations whose truth value may be affected by a state transformation, i.e., FLUENT relations,from those whose truth value is constant over all achievable worlds, i.e., NON-FLUENT relations such as "ROBOT(X)", "INTEGER(Y)".

b) "Is knowledge represented using P partial?" A partial relation may have truth values TRUE, FALSE, or UNDETERMINED. Partial relations may be used to represent incomplete knowledge of the world which may cause conditional statements to be generated as explained in Section 5. A relation may be declared "uncertain" which implies an absence of knowledge about it so that it is assigned a truth value of undetermined a priori. If P is not "partial" it is "total" and can only have truth values of either true or false. Thus rule **R6** applies to partial predicates only.

c) "Does P have a uniqueness property in certain argument positions?" A "yes" answer indicates that P cannot be true for two sequences of argument values that differ only at one of those positions that are unique. The unique positions are given using the notation, **(X1,*,X3,*,...,Xn),** for example, to designate the second and fourth argument positions. For each unique argument position in relation **P(a1,...,an),** an axiom is "built-in" from which a contradiction may be established with **P(b1,...,bn)** that differs in a **unique** posit ion and matches elsewhere.

For example the statement, "an object can only be in one piece at one time", is expressed by, **AT(X1,*)**. If we add, "and only one object can be at any place", then **we** use **AT(*,*)**.

3.1.6 SIMPLIFICATION: Algebraic simplification rules may be given to simplify the terms that may occur in subgoals during the problem solving phase. The simplification is driven by a table of rules of the form **s=t** where s and **t** are terms; occurrences of $s\alpha$ are replaced by $t\alpha$ for any substitution α.

The output format of any functional term may be specified by the user by giving a **rule** in which its input prefix form is on the left, e.g., (PLUS X Y) **≡ (X+Y).**

3.2. ADVICE LANGUAGE

The advice facility is intended to enable the user to impose structure relevant to solving a particular problem upon an already defined frame, This additional structure includes preference orderings among goals and rules, and restrictions on the search space. The preferences may also reflect the kind of solution the user wants,

Advice is given during program generation by means of an interactive facility. The advice subsystem may be entered by responding to a system query, “DO YOU HAVE ADVICE?” , or by typing any key during program generation. The user may request to see the current path in the subgoal tree i.e. rules entered and goals pending, and receive a diagnosis of the cause of any failure. This is useful in deciding what advice to give.

The advice system enters a read loop recognizing and numbering commands from the language, shown in Figure 5. In the language syntax, optional symbols are enclosed in "[" and "]"; enclosing a list of symbols in "{" and "}" indicates that one must be chosen; <rule> is a rule name; <rule list> is a list of rule names; <proc> is a primitive procedure name; <advice num> is of the form " n”, where n is an integer; and Q denote8 the pre-condition of <rule>.

After advice has been given the system may be directed to reject the rule it is currently using, if any, or to try (perhaps re-try) the current rule.

The advice facility is an important tool for experimenting interactively with different frames to determine their adequacy and soundness. At present, the language is rudimentary and should be extended.

3.3 PROGRAMMING LANGUAGE

The generated programs are expressed in an elementary ALGOL-like language which includes block structure, assignment statements, conditional statements, while loops, and non-recursive procedures calls. The procedures may be typed, including Boolean, and may have side effects in addition to the value returned. The procedure parameters are normally called by value except in the case of special W-variable8 in conditional assignments (rule RO, Section 2).

ADVICE LANGUAGE

COMMAND SYNTAX	ACTION PERFORMED
TRY <rule1> BEFORE <rule2>	Use <rule1> before <rule2> to develop a subgoal.
FOR <rule> [FIRST] TRY <literal>	Change the precondition Q of <rule> to <literal> & Q if "FIRST" is given otherwise Q ∨ <literal>.
DELETE {<rule>,<literal>, <advice num>}	If <rule> is given, remove that rule. If <literal> then alter state to make <literal> not true. If <advice num> then delete the associated advice and undo its effects on the system.
ADD{<rule>,<literal>}	If <rule> is given then accept a new rule. If <literal> then alter state to make <literal> true.
ALTER <rule>	<rule> may be modified.
ASSUME {<rule>,<literal>}	If <rule> is given then an assumed rule may be defined. If <literal> then alter state to make <literal> true and mark it as an assumption.
RESTRICT <rule>{TO,FROM} <rule list>	For any goal in Q, if "TO" is given then only rules in <rule list> may be used, if "FROM" then no rule in <rule list> will be used.
ADVICE	All advice given that session is displayed.
STATUS	The following information is displayed: -rules entered and goals pending in current subgoal tree, -rules and goals in longest path obtained so far, -currently constructed program segment -longest program segment constructed so far.
PAIRWISE INEQUALITIES <proc>	Pairwise equality is prohibited in primitive procedure argument positions containing "*".
RECURSIVE <rule>	The rule may be used directly to achieve a goal in its pre-condition, otherwise it may not.

3.4 AN EXAMPLE

Consider the task of writing a program to compute the nth Fibonacci number for some integer n. This task has been posed in [Balzer 1972]. The basic information required is the recursive definition and the basis values. One way to express this in the Frame language uses the following predicates with the indicated meanings:

VFIB(X,Y): "The value of the X Fibonacci number is Y",
C(X,Y): "The contents of the variable X is Y",
FIB(X,Y): "The variable X contains the Y Fibonacci number,
INTEGER(X): "X is an integer",
ISVAR(X): "X is a variable",
>(X,Y): "X is greater than Y"
NEWVAR(X,Y): "X and Y are local variables".

The problem is ISVAR(X3)∧INTEGER(N){?}FIB(X3,N).

The frame contains:

1. Axioms VFIB(1,1) and VFIB((ADD1 1),2) (these define initial values),

2. Axiom
   TAFIB
   VFIB((SUB1 V1),V2)∧VFIB((SUB1(SUB1 V1)),V3)∧ =(V4,(PLUS V2 V3));
   VFIB(V 1 ,V4);
   (defines VFIB(V1,V4) for terms beyond the initial values).

3. An iterative rule (named TFIB) with goal FIB(V3,V8); this rule defines the conditions to be satisfied during an iterative upward computation. The basis condition (to initialize the counter and program variables) is:

   NEWVAR(V1,V2)∧INTEGER(V8)∧C(V1,(ADD1 1))∧C(V2,1)∧C(V3,(ADD1 1));.

- The loop invariant condition is:

   C(V1,V5)∧C(V2,V9)∧C(V3,V10)∧VFIB(V5,V10)∧VFIB((SUB1 V5),V9);.

This states that at each entry to the loop body, if the value in the counter is i and the values in the program variables are j and k then j is the ith Fibonacci number and k is the (i-1)st Fibonacci number.

The iteration step condition

   C(V 1,(ADD 1 V5))∧FIB(V2,V5)∧FIB(V3,(ADD 1 V5));

specifies what the iteration step is to accomplish. The control test, >(V5,V8) and an out put assert ion FIB(V3,V8) are given.

4. A definition of FIB in terms of VFIB and C

```
TDFIB
VFIB( V2,V3)∧C( V4,V3); FIB( V4,V2);
```

5. A simple primitive procedure for assignment is also given, i.e.

```
←(V1,A1)
ISVAR(V1); C(V1,A1);.
```

No rules are declared a8 assumptions. The additional information given to complete the Frame specification is shown in Figure 6, and a program generated from this Frame is shown in Figure 7.

PREDICATE SYMBOL	FLUENT	PARTIAL	UNIQUENESS
C	TRUE	FALSE	C(X, *)
FIB	TRUE	FALSE	FIB(X,*)
>	TRUE	FALSE	FALSE
VFIB	TRUE	FALSE	VFIB(*,*)
INTEGER	FALSE	FALSE	FALSE
=	TRUE	FALSE	FALSE
ISVAR	FALSE	FALSE	FALSE

SIMPLIFICATION RULES:

(ADD1 (SUB1 x)) → x

(SUB1 (ADD1 x)) → X

FUNCTION OUTPUT SYNTAX:

(ADD1 x) = (X+1)

(SUB1 X) = (X-1)

(PLUS x Y) = (X+Y)

ADVICE: TRY TFIB BEFORE TDFIB

RECURSIVE TAFIB

Figure 6

****

```
PROC1 (X3,N)
ISVAR(X3);INTEGER(N);
COMMENT
INPUT  ASSERTION
NONE
OUTPUT  ASSERTION
FIB(X3,N)
   BEGIN
   Y1 ← (1+1);
   Y2 ← 1;
   X3 ← (1+1);
   WHILE ¬>(Y1,N) DO
      BEGIN
      Y1 ← (Y1 + 1);
      Z2 ← X3;
      X3 ← (X3 + Y2);
      Y2 ← Z2;
      END
   END
```

Figure 7

4. PROBLEM SOLVING PROCESSES

During the process of problem solving and program generation, information is needed at many points to reduce the search space or to produce reasonable programs. Some of the information is provided in the frame specification by statements about the rules end predicates; other useful facts are provided to the problem solver in the form of rather simple advice. Roughly speaking, there are six basic processes in the problem-solving system where extra facts can help: (a) pattern matching, (b) development of nodes in the subgoal tree, (c) updating the state description (i.e. implementing invariance), (d) backtracking in the subgoal tree, (e) conditional branching, (f) assembly of programs. Each fact (as opposed to a rule or axiom) in a frame specification and each sort of advice has at least one function in speeding up a basic process. Below we describe some of the ways in which the present variety of fact8 and advice is used (full detail8 are given in [Buchanan 1974]).

(1) OR-Node Se lec ion. When more than one rule can be applied to reduce a given goal, some selection and preference criteria are needed. By using the advice syst em,t he rules and axioms that may be applied to achieve goals within the precondition of a rule or axiom may be restricted to or excluded from a given list. Also, a preference ordering may be specified among rule8 and axioms with common post-conditions. Goals within the precondition8 of axioms are always restricted to deduction within the current state, i.e. can be reduced only by use of other axioms, and do not cause a state transformation nor add any construct to the generated program.

(2) Predicate Classification. A predicate P is classified according to the kind of subgoaling permitted to achieve a goal of the form P(t). If P is declared to be NON-FLUENT, then any goal literal containing P can be achieved only by deduction from the current state. No rules (procedure, iterative or definitional) are applied. FLUENT goals are attempted by deduction and state transformation. If a fluent predicate occur8 in a lit eral which is the argument of the REQUEST modifier, then it is treated as a non-fluent.

(3) Goal Ordering. The achievement of a condition (and the efficiency of the output program) is strongly influenced by the ordering of its subgoals In particular, the bindings of variables occurring in goals may be determined by earlier achieved instances. In some cases only certain orderings will permit achievement. An objective of an automatic problem solving system is to determine the optimal subgoal ordering, but at present this is provided by the user when the Frame is defined and may be altered by advice. However, the system automatically orders non-fluent goals first in a condition; this relatively short achievement search is used both a8 a quick rejection strategy and to get variable bindings of the correct type for the remaining fluent goals.

(4) Recurring failures. When failure occurs in some subtree prior to successfully solving a subproblem, its causes should be used to avoid repeating the same failure in the continued search if possible. At present this must be handled using the interactive advice system. This informs the user of the current path in the subgoal tree, current program generated, and goals that fail, thus allowing interactive correction when a

repetition occurs. These situations can also be eliminated by placing the (eventual) successful subprograms on the program library for use as MACROS.

(5) Repetition. Certain types of looping behavior in the subgoaler are prevented using the feature of the Frame language that allows a rule to be declared recursive or non-recursive. If declared non-recursive, then that rule will not be used directly to achieve a goal in its pre- condition and it will not be entered twice to achieve the same instance of its post-condition within the same subgoal tree. A more general mechanism should consider not only the current goal and rule but also the current state as well.

(6) Truth Values. Though the underlying semantics is three valued, search efficiency is gained by restricting relations involving certain predicate symbols to be two valued. If a predicate P is declared to be TOTAL, then failure to achieve P indicates that ¬P is true. Only true positive instances of total predicates are stored in the state. The rule of undetermined values is not applicable to literals involving total predicates. The additional processing required for PARTIAL predicates is described in Section 5.

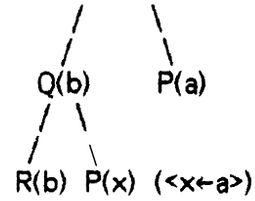
(7) Useless Procedure Calls In some cases, the application and generation of redundant or trivial procedure calls are detected and avoided. At the moment this is done by placing restrictions in the frame on the actual parameters of primitive procedures. The system will not use an instance of a primitive procedure that contains pairwise equality between its actual parameters that has been prohibited by the user, For example, the advice "PAIRWISE EQUALITY MOVE(x1,x2,*,*)" will cause the rejection of the procedure call "MOVE(MAN,CHAIR,P,P)".

(8) Uniqueness Properties. Uniqueness or single-valuedness in argument positions of certain predicates is sufficiently important to justify a special mechanism rather than to rely on deduction using axioms. The designation of certain argument positions as unique is equivalent to efficiently building in axioms of a particular form, e.g. P(x1,*) represents the axiom,

$$P(x1,x2) \wedge x2 \neq x3 \rightarrow \neg P(x1,x3).$$

These special axioms are used for consistency checking (in the implementation of the rule of invariance) when the state is updated.

(9) Context Linking. The context, which includes the state and bindings on subgoals currently pending at a node, should be available to aid search decisions, e.g. instantiations of subgoals or choice of rule, at descendent nodes in the subgoal tree. The system has a mechanism that if requested will keep track of the instantiated goals at each level of the subgoal tree so that their variable bindings are available when attempting lower level goals that precede them in the depth first ordering, This is used to instantiate the lower level goals. For example, suppose Q(b) ∧ P(a) is a condition to be achieved and a primitive procedure $R(y) \wedge P(x) \{p(x,y)\} Q(y)$ is applied to achieve Q(b), then for the P(x) in the precondition of p, P(a) will be used since it must be achieved at the higher level anyway, i.e.,



This heuristic may be viewed as the opposite of subsumption, the strategy being to get ground instances as soon as possible to help avoid long searches using rules. This is a rather restrictive strategy that may exclude solutions and is only used when requested by the user.

(10) Evaluation of Predicates and Functions. For certain predicates occurring in subgoals, achievement is most efficient by direct evaluation. If a literal occurring in a goal is formed with a predicate that has a LISP definition, then that literal is evaluated as a LISP statement., Special processes or even subsystems can thereby be linked into program generation. Evaluation of arbitrary functions occurring in terms in arguments of goal literals is done if the function occurs in the scope of an **EV** 'modifier, These evaluations assume the soundness of implicit axioms describing the LISP definitions, and the consistency of these axioms with the Frame. For example, the equality predicate, "=", is evaluated using the LISP "EQUAL", and the predicate **NEWVAR(x1,x2,...,xn)** takes an arbitrary number of arguments and binds each Frame variable xi to a new program variable (for use perhaps as a local variable in a block).

(1 1) Simplification rules. Rules of the form s → t where s and t are terms, may be included in the Frame. Such rules are applied to simplify terms in goals by replacing occurrences of $s\alpha$ by $t\alpha$. This not only reduces the complexity of terms in the subgoal tree, but it also modifies the pattern matching process and the set of rules that can **be** applied to reduce a goal.

(12) Computing Input/Output Assertions. In Section 2 primitive procedures were viewed as Frame rules of the form **P{p}Q,** where P and Q are the pre and postconditions for p. The conditions P and Q may also be viewed as sufficient input and out put assertions for p , that must be satisfied by the actual parameters of p. For any generated program segment A, the input assertion I_A is computed as the conjunction of all literals, l, from a state that were used in achieving subgoals encountered during the generation of A and did not occur in that state as a result of a postcondition of **a** procedure whose generation in A preceded the addition of l to I_A. The output assertion O_A is the conjunction of literals added to a state during the generation of A that are true in the final state.

The usefulness of computing sufficient input and output assertions for a program or segment thereof will become apparent when we discuss program generalization and the construct ion of conditional stat ementa.

All of these applications of facts and advice with the exception of **(12),** are intended to have a direct effect on reducing the growth of the subgoal tree (process (b)). In

addition, the pattern matching process (a) is extended by (11); (c) is aided by the restriction of truth values and the special axioms (6,8); (e) is dependent on (6 and 12); (f) is aided by (3,7,11, 12). There are other techniques, mainly details of the implementation, some of them heuristic, that affect problem solver, particularly the backtrack (d), the updating (c) and assembly of programs (f) (e.g. the implementation of the A connective by software interrupts that protect already achieved goals, includes certain assumptions about backtracking when an AND-node fails). Details of these will be found in [Buchanan 1974].

5. GENERATION OF CONDITIONAL STATEMENTS

Conditional statements are generated in situations where the rule of undetermined values applies or when the outcome of a primitive procedure is uncertain. In this section the system methods for constructing conditionals will be described and an example given. The question of extending the formal algorithm and the correctness proof is considered.

5.1 UNCERTAIN PRECONDITIONS. As previously mentioned, relations involving partial predicates may have truth values of TRUE, FALSE, or UNDETERMINED, whereas all other relations must be either TRUE or FALSE. Partially valued predicates are intended to express the possibility of an uncertainty or lack of knowledge about a state arising during the problem solving and program generation phase of the system. The formal algorithm for deciding when an uncertainty has arisen is rule R6 (the "I give up" criterion of the system). As with invariance, the implementation of R6 is only an approximation to the formal rule. The system may give up too early, but this, in itself, does not lead to--incorrect programs, merely redundant ones.

5.1.1 UNDETERMINED VALUES. During the generation of a program, uncertainty may arise when a precondition for the application of a rule is UNDETERMINED with respect to the current state. The implementation of the rule R6 is described by the following definitions:

DEFINITION A literal l is UNDETERMINED in a state S if the following conditions hold:

(i) pred(l) is partial,
and (ii) the system halts without solving S{?}l,
and (iii) the system cannot prove SUF⊃¬l.

Condition (ii) means that l is not true in S nor can S be transformed into a state in which l is true. If condition (ii) is true and ¬l is true in S then l must retain a truth value of FALSE and the precondition subgoal l must fail. Failure to prove ¬l from S establishes a truth value of UNDETERMINED for l with respect to S. This definition applies to fluent and nonfluent literals but since the truth value of a "nonfluent" cannot be changed by a state transformation, for them, it is sufficient to use only the logical axioms in deciding condition (ii).

For the more general case in which the precondition may be a disjunction of literals we have the definition,

DEFINITION A disjunction of literals $\{l_i\}_{i=1}^{n}$ is UNDETERMINED in a state S if at least one literal is UNDETERMINED and no literal can be achieved from S.

5.2 CONDITIONAL STATEMENTS: When a pre-condition P is UNDETERMINED in a state S, a conditional branch is inserted in the solution program. If P is a single literal l, then

program generation may continue either along the path in which l is assumed to be TRUE and in which future goals are attempted with respect to state S U{l}, or along the path in which ¬l is assumed to be TRUE using state S U{¬l}. The system convention has been to generate a call to a yet ungenerated procedure for the latter case. The tasks of generating such contingency programs are placed in a subproblem stack for later attention (see section 5.3). Program generation continues, by convention, along the path using state S U {l}. This path is referred to as the "trunk" program of the tree of contingency programs generated while attempting to achieve the main goal. The, path selection at present is rather ad hoc since no assignments of probability are made at the Points of uncertainty and no path is considered more likely to be successful in general.

If an undetermined disjunctive precondition $\{l_i\}^n_{i=1}$ occurs in which literals $\{l_i\}^m_{i=1}$ $m \leq n$ are UNDETERMINED in S, then a nested conditional of the following form will be generated:

if l_1 then
  if $\neg l_2$ then

    .
    .
    .
    if l_m then p_m
    else p_{m-1}

    .
    .
    .
  else p_1
else p_0

where each p_j is a call to a program to achieve a selected goal G from state S, = S U $\{l_i : i=j+1 \ \& \ i<m\}$ U $\{\neg l_i : 1 \leq i \leq j\}\}$ and p_0 is the trunk program segment which satisfies $S \wedge l_1 \{p_0\} G$ and forms the else-statement in the main-clause of the conditional. Each member of the set of triples $\{<p_j, S_j, G>: 1 \leq j \leq m\}$ is placed in the stack of contingencies and program generation continues for p_0. The assumed literal, l_1, is removed from the state following the generation of the ELSE clause in the trunk program if it is not in the output assertion.

5.3 SELECTION OF CONTINGENCY GOAL: The goal G to be achieved by the contingency programs is selected from the set of goals in the subgoal tree that are global to the undetermined precondition. Let us refer to the set of goals which are below G in the subgoal tree, as the SCOPE of G.

The particular G chosen and its associated scope affect the length of p_0 , duplication among cont ingency programs, degree of difficulty in generating cont ingency programs and validity of their use. If the structure of the trunk program is to remain fixed during *contingency* program generation then the choice of G cannot be deferred, The block

structure of our program language imposes the restriction that for any conditionals in p_0, a contingency goal G' must not have a greater scope than G. There is also the problem that if G is not fully instantiated (i.e. some of its variables are not in the initial state) then inconsistent instantiations may occur in different contingency programs which must validly rejoin the main program following the ELSE clause. The present system selects the least global fully instantiated goal thereby satisfying the block nesting constraint and minimizing the scope while avoiding the problem of handling deferred instantiation. This selection process is always effective in the present system since the top level goal is fully instantiated.

5.4 REJOIN CONDITIONS: When a contingency program is generated its output state must satisfy certain conditions, hereafter called the rejoin condition, for return of control to the trunk program to be correct. Consider the case of an undetermined goal L in state S and a contingency goal G in Figure 8 .Let A and B be program segments that satisfy $S \wedge L\{A\}G$ and $S \wedge \neg L\{B\}G$ and let C be the rest of the trunk program.

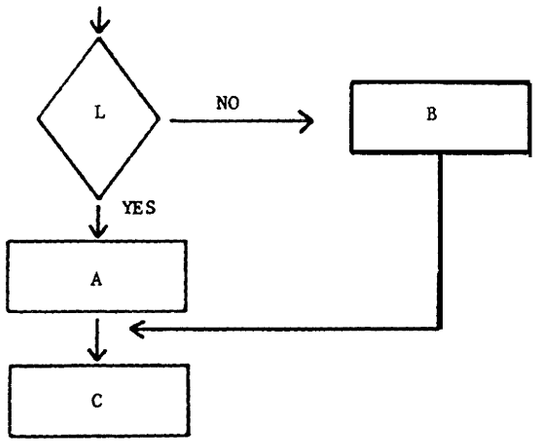


Figure 8

Let R be the output state of B obtained by applying invariance; thus $S\wedge\neg L\{B\}R$ and $R\supset G$. Similarly, let $S\wedge L\{A\}P$ where $P\supset G$, and let Q be the minimal subset of P required as input to C (section 4(12)). Then, the REJOIN CONDITION for B is $R\supset Q$. B is said to have BAD SIDE EFFECTS if in fact $R\supset Q$ cannot be established.

5.5 SUBPROBLEM STACK: The task of generating a contingency procedure is specified by the quadruple;

(<procname> <state> <goal> <rejoincond>)

where,

<procname> is the name of the yet ungenerated procedure that must satisfy <state>{<procname>}<goal> A <rejoincond>.

At the point in the planning when the uncertainty is encountered, the first three elements of the quadruple are placed in a stack. The rejoin condition is not known at this time since it involves the input assertion for the trunk segment C following the point where cant rol returns from the contingency plan to the trunk plan. After C is generated, the rejoin condition is computed and stored as the fourth element of the quadruple.

When planning has been completed for a trunk procedure, if the subproblem stack is not empty then contingency planning may be done by removing a quadruple from the stack and posing this as a program generation task. The state of the system is initialized to the specified contingency state and the subgoaling system is given <goal> as its main goal. If it is successful in achieving a state in which the main goal is true then a test is made--to see if the rejoin condition is true in that state. If it is then the procedure declaration is adjoined to its trunk program. If the condition cannot be proved, the system allows the user two alternatives: (i) Mark the call to the program as an error exit in the trunk program, or (ii) "Fit" the program to the trunk program by posing the currently untrue rejoin condition as a new goal, constructing a new program segment that achieves it, and appending this segment to the end of the contingency program.

This process of generating a trunk procedure which may create new contingency tasks then generating contingency procedures as directed by the user may continue until all contingencies have been processed and the stack is exhausted,

5.6 COMPUTATION OF INPUT/OUTPUT ASSERTIONS The computation of input /out put assertions for programs not containing conditionals is described in Section 4(12). The uncertainty as to which path computation will follow in a program containing conditional statements complicates these assert ions. The input/output assertions in this case must be computed incrementally as each contingency program is generated.

In the conditional statement shown in Figure 8, suppose we know the minimal input and output assertions for A and B, say P{A}Q and R{B}S. then the input and output assertions for the conditional statement are

$$(L \wedge P) \vee (\neg L \wedge R)\{\text{if } L \text{ then } A \text{ else } B\}Q \vee S.$$

To reduce computation, We use the simpler sufficient input assertion $P \wedge R$, (Note that $P \wedge R$ should be consistent since it is a subconjunct of a previous state). There doesn't appear to be a simplifying approximation for output assertions ,

5.7 UNCERTAIN PRIMITIVE PROCEDURES: A primitive procedure q defined by **P{q}Q** has an uncertain outcome if Q is a disjunction, In the present system, disjunctive post - conditions use the exclusive OR connective, **"⊗"**. This allows us to define frame procedures that have an intended result but may be unreliable. It is assumed that exactly one of the possible outcomes will be true in the output state, At the point where an uncertain operator is applied, the problem solver has no knowledge of what the outcome will be and a conditional statement must be generated. Let Q be the disjunction of literals $\{l_i\}^{\vee}_{i=1}$. The first outcome l, is considered to be the normal (goal) result of executing q. Following the inclusion of q in the program in state S, a conditional statement of the following form is generated:

if ¬ l, then
 if ¬ l_1 A l_2 A ¬ l, ∧...∧ ¬ l, then p_2
  else if ¬ l_1 A ¬ l_2 A l, A ¬ l_4 ∧...∧ ¬ l, then p_3

  else if ¬ l, A ¬ l, ∧...∧ ¬ l,,, A l, then p_n
   else p_{n+1}

where each p_i, $2 \leq i \leq n$, is a call to a program to achieve l_1 from state $S_i = S \cup \{l_i\} \cup \{\neg l_j : j \neq i \ \& \ 1 \leq j \leq n\}$, and p_{n+1} is an error exit. The contingency stat es **will** correspond to the n ways of assigning exactly one literal true and the remaining **literals** false.

5.8 AN EXAMPLE Suppose a procedure is to be generated for a man to travel from San Francisco to New York given three modes of travel, i.e., flying, driving, or walking. This is similar to the "airport problem" discussed in [McCarthy 1959]. A FRAME for this problem consists of defining a primitive procedure for each mode of travel, an initial state, and relation information as shown in Figure 9. A few of the contingency programs generated are shown in Figure 10.

RELATIONS	DEFINITION	FLUENT	PARTIAL	UNIQUENESS
ROB(X)	"X is a robot"	FALSE	FALSE	FALSE
AUTO(X)	"X is an automobile"	FALSE	FALSE	FALSE
PLANE(X)	"X is an airplane"	FALSE	FALSE	FALSE
AIRPORT(X)	"X is an airport"	FALSE	FALSE	FALSE
AT(X,Y)	"X is at location Y"	TRUE	FALSE	AT(X,*)
WALKABLE(X,Y)	"A walkable path exists between X and Y"	TRUE	TRUE	FALSE
CLEAR(X,Y)	"The sky is clear between X and Y"	TRUE	TRUE	FALSE
DRIVABLE(X,Y)	"A drivable road exists between X and	TRUE	TRUE	FALSE
H&UMBRELLA(X)	"X has an umbrella"	TRUE	TRUE	FALSE
CRASHED(X,Y,Z)	"X crashed between Y and Z"	TRUE	FALSE	FALSE
KILLED'X'	"X has been killed"	TRUE	FALSE	FALSE
RUNS(X)	"X will run properly"	TRUE	TRUE	FALSE
FLIES(X)	"X will fly properly"	TRUE	TRUE	FALSE

****

PRIMITIVE PROCEDURE	PRE-CONDITIONS	POST-CONDITIONS
walk(R1,L1,L2) "R1 walks from L1 to L2"	ROB(R1)∧¬ KILLED(R1)∧AT(R1,L1) ∧CLEAR(L1,L2)∨HASUMBRELLA(R1) ∧WALKABLE(L1,L2);	AT(R1,L2)
drive(R1,C1,L1,L2) "R1 drives C1 from L1 to L2"	ROB(R1)∧¬ KILLED(R1)∧AUTO(C1) AT(C1,L1)∧RUNS(C1) ∧DRIVABLE(L1,L2)∧AT(R1,L1);	AT(R1,L2) ∧AT(C1,L2)
fly(R1,A1,L1,L2) "R1 flies A1 fran L1 to L2"	ROB(R1)∧¬ KILLED(R1)∧PLANE(A1) ∧AIRPORT(L2)∧AT(A1,L1) ∧FLIES(A1)∧CLEAR(L1,L2) ∧AT(R1,L1);	[AT(R1,L2)∧ AT(A1,L2)] ∀[CRASHED(A1,L1,L2) ∧KILLED(R1)]

*******x*x* **********

INITIAL STATE

ROB(MAN)∧AUTO(BMW)∧PLANE F111)∧AIRPORT(SFO)∧AIRPORT(NYC)∧AT(MAN,HOME)∧AT(BMW,GARAGE)∧AT(F111,SFO);

****

ADVICE

PAIRWISE INEQUALITIES walk(R1,*,*),drive(R1,C1,*,*),fly(R1,A1,*,*)

TRY FLY BEFORE DRIVE, TRY DRIVE BEFORE WALK

Figure 9

```
PROC1 MAN NYC
ROB MAN ;AUTO BMW ;PLANE F111 ;AIRPORT SFO ;
COMMENT
INPUT ASSERTION:
AT MAN HOME ^CLEAR HOME GARAGE ^AT BMW GARAGE ^AT 1111 SFO
^FLIES F111 ^CLEAR SFO NYC ^RUNS BMW
RDRIVABLE GARAGE SFO ^WALKABLE HOME GARAGE
OUTPUT ASSERTIOS:
AT BMW SFO ^AT F111 NYC ^AT MAN NYC ;
COMMENT
PROC11 ATTEMPTS_TO_ACHIEVE_ AT MAN NYC
PROC1 ATTEMPTS-TO-ACHIEVE- AT MAN GARAGE
PROC ATTEMPTS_TO_ACHIEVE_ AT MAN GARAGE
PROC ATTEMPTS_TO_ACHIEVE_ AT MAN GARAGE
PROC ATTEMPTS_TO_ACHIEVE_ AT MAN SFO
PROC- ATTEMPTS-TO-ACHIEVE- AT MAN SFO
PROC ATTEMPTS_TO_ACHIEVE AT MAN NYC,
PROC ATTEMPTS-TO-ACHIEVE- AT MAN NYC, ;
   BEGIN
   IF ¬FLIES F111 THEN
      PROC MAN NYC;
   ELSE
      BEGIN
      IF --CLEAR SFO NYC THES
         PROC MAN NYC
      ELSE
         BEGIN
         IF -RUNS BMW THEN
            PROC- MAN SFO
         ELSE
            BEGIN
            IF ¬DRIVABLE GARAGE SFO THEN
               PROC MAN SFO
            ELSE
               BEGIN
               IF ¬CLEAR HOME GARAGE THEN
                  IF ¬HASUMBRELLA MAN THEN
                     PROC MAN GARAGE,
                  ELSE PROC MAN GARAGE
               ELSE
                  BEGIN
                  IF ¬WALKABLE HOME GARAGE THES
                     PROC1 MAN GARAGE
                  ELSE
                     BEGIN
                     WALK MAN HOME GARAGE
                     END
                  END
               DRIVE MAN BMW GARAGE SFO;
               END
            END
         FLY MAN F111 SFO NYC:
         IF ¬AT(MAN NYC) THEN
            IF ¬AT(MAN NYC) A CRASHED (F111 SFO NYC)
               PROC11(MAN NYC)
            ELSE PROCER(MAN NYC)
      END END
   t: NO

PROC(MAN NYC)
ROB(MAN);AUTO BMW);
COMMENT
INPUT_ASSERTION:
AT(MAN HOME ^CLEAR HOME GARAGE ^AT BMW GARAGE)^RUNS BMW
^DRIVABLE(GARAGE NYC ^WALKABLE HOME GARAGE)
```

Figure 10a

```
OUTPUT-ASSERTION:
AT(BMW NYC)^AT(MAN NYC);
COMMENT
PROC16 ATTEMPTS-TO-ACHIEVE- (AT MAN GARAGE)
PROC15 ATTEMPTS_TO_ACHIEVE_ (AT MAN GARAGE)
PROC14 ATTEMPTS_TO_ACHIEVE_ (AT MAN GARAGE)
PROC13 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC)
PROC12 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC);
   BEGIN
   IF ¬RUNS(BMW) THEN
      PROC12(MAN NYC)
   ELSE
      BEGIN
      IF ¬DRIVABLE(GARAGE NYC) THEN
         PROC13(MAN NYC)
      ELSE
         BEGIN
         IF ¬CLEAR(HOME GARAGE) THEN
            IF ¬HASUMBRELLA(MAN) THEN
               PROC14(MAN GARAGE)
            ELSE PROC15(MAN GARAGE)
         ELSE
            BEGIN
            IF¬WALKABLE (HOME GARAGE) THEN
               PROC16(MAN GARAGE)
            ELSE
               BEGIN
               WALK(MAN HOME GARAGE
               END
            END
         DRIVE(MAN BMW GARAGE NYC)
         END
      END
   END


PROC4(MAN SFO)
ROB(MAN);
COMMENT
INPUT_ASSERTION:
AT(MAN HOME)^CLEAR(HOME SFO)^WALKABLE(HOME SFO)
OUTPUT_ASSERTION:
AT(MAN SFO);
COMMENT
PROC25 ATTEMPTS-TO-ACHIEVE- (AT MAN SFO)
PROC24 ATTEMPTS_TO_ACHIEVE_ (AT MAN SFO)
PROC23 ATTEMPTS-TO-ACHIEVE- (AT MAN SFO);
   BEG IN
   IF ¬CLEAR(HOME SFO) THEN
      IF ¬HASUMBRELLA(MAN) THEN
         PROC23(MAN SFO)
      ELSE PROC24(MAN SFO)
   ELSE
      BEGIN
      IF ¬WALKABLE(HOME SFO) THEN
         PROC25 'MAN SFO)
      ELSE
         BEGIN
         WALK(MAN HOME SFO;
         END
      END
   END

PROC12(MAN NYC)
ROB(MAN);
COMMENT
INPUT-ASSERTION:
AT(MAN HOME)^CLEAR(HOME NYC)^WALKABLE(HOME NYC)
```

Figure 10b

```
OUTPUT-ASSERTION:
AT MAN NYC);
COMMENT
PROC30 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC)
PROC27 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC)
PROC26 ATTEMPTS_TO_ACHIEVE_ (AT MAN NYC);
   BEGIN
   IF ¬CLEAR(HOME NYC) THEN
      IF ¬HASUMBRELLA(MAN) THEN
         PROC26(MAN NYC)
      ELSE PROC27(MAN NYC)
   ELSE
      BEGIN
      IF ¬WALKABLE(HOME NYC) THEN
         PROC30(MAN NYC)
      ELSE
         BEGIN
         WALK(MAN HOME NYC)
         END
      END
   END
```

Figure 10c

5.9 CORRECTNESS The format algorithm of Section 2.3 can be extended to include the case when G' is undetermined in I' by formalizing a simplified version of the system methods described above. We shall mention some of the pertinent details here.

The extension requires formalizing the subproblem stack and the methods of choosing contingency goals. Also, it is necessary to add clauses for assembling conditional statements into the answer A according to rule R5. Thus contingency goals must be "marked" and the appropriate undetermined subgoals associated with them, so that when a contingency goal is achieved during the generation of the trunk program, the related conditionals are assembled into A. The computation of the state I(n) must be modified when G(n) is the contingency goal for G(i) by removing G(i) if it is not in the output assertion of the program segment generated between achieving G(i) and G(n). We do not justify the system method of computing input assertions, and instead assume that in the formal algorithm the state at any node in the subgoal tree is the input assertion for the following segment of the generated program.

To extend the correctness proof of Section 2.5, we must extend the induction step to include the cases when (a) G(n+1) is undetermined in I(n), and (b) G(n+I) is achieved from I(n) and is -the contingency goal for G(i), say, where $i<n+1$. The induction hypothesis must be modified to take account of any undetermined goals (assumed true in the trunk program) whose contingency goals have G(n) within their scope. Thus, typically, the hypothesis would be $I\{A(i)\}I(i)$ and $I(i)\wedge G(i)\{A(i,n)\}I(n)$, where G(i) is undetermined in I(i) and has a contingency goal more global than G(n), and A(i,n) denotes the program segment generated between achieving G(i) and G(n).

Case (a); G(n+1) is achieved by assumption in generating the trunk program, $I(n+1)=I(n)\wedge G(n+1)$ and A(n+1,n+1) is empty

Case (b): let B be the contingency branch. The previous proof implies that $I(n+1)\supset G(n+1)$. We also have that A(n+1) = A(i);IF C(i) THEN A(i,n+I) ELSE B.

(1) $I\{A(i)\}I(i)$, hypothesis,
(2) $I(i) \wedge G(i)\{A(i,n+1)\}I(n+1)$ hypothesis
(3) $I(i) \wedge \neg G(i)\{B\}I'(n+1)$ assumption,
(4) $I'(n+1)\supset I(n+1)$ rejoin condition,
(5) $I(i)\{$IF G(i) THEN A(i,n+1) ELSE $B\}I(n+1)$ R5,2, R1,3,4
(6) $I\{A(n+1)\}I(n+1)$ and $I(n+1) \supset G(n+1)$ R2,1,5.

The proof of $I\{A(m)\}G$ follows by noting that all contingency goals must have been achieved when the final goal G is achieved.

6. GENERATION OF ITERATIVE STATEMENTS

An iterative rule allows the program generator to construct a WHILE loop provided it can construct a loop body to satisfy the' premisses of the rule. Ultimately such rules should require the user merely to specify an invariant in order to have the system write a correct iterative program. At the moment, the user needs to furnish some additional relevant facts. The algorithms used in the system to implement iterative rules of the form S2 (Section 2) and to assemble while loops are described briefly and an example given.

6.1 PREMISSES FOR CONSTRUCTING A LOOP: An iterative rule is defined by the assertions P(basis), Q(loop invariant), R(iteration step goal), G(rule goal), L(control test) and S(output assertion). All the free variables in R and L must be among the free variables in Q. In order to use the rule, to achieve I{?}G say, the formal algorithm requires that all of the following subgoals be achieved or be true:

(i) Construct A such that L(F)||- I{A}P
(ii) L(F)|- I{A}Q
(iii) Construct B such that L(F)||-Q∧L{B}R
(iv) L(F) |- Q∧L{B}(∃Z)Q(Z)∨(¬(∃Z)Q(Z)∧ ¬L)
(v) Construct C such that L(F) ||- Q∧L{B;C}Q∨-L

Note that {ii) and (iv) are restricted to first order rules (consequence, invariance, and the frame axioms). The input state for (iii) is Q∧L. In addition, an iterative rule must satisfy the following minimal consistency requirements within the frame F:

(vi) ¬(S U F ⊃ L) and S U F ⊃ G.

The conclusion of the rule is: I{A;WHILE L DO BEGIN B;C END}G.

Iterative frame rules are instances of the iteration rule [Hoare 1969]:

$$\frac{Q\wedge L\{A\}Q,\ Q\wedge\neg L\supset G}{Q\{\text{WHILE L DO A}\}G}$$

It is possible to derive a weak form of the rule:

$$\frac{Q\wedge L\{A\}Q\vee\neg L,\ \neg L\supset G}{Q\{\text{WHILE L DO A}\}G}$$

The weak form allows the invariant to fail on exit from the loop. We have found the weak form convenient to use in many examples.

The present implementation sets up clauses (i) - (iv) as a THAND of subgoals to be

achieved. More specifically, suppose an iterative rule is invoked to solve the problem I{?}G. Let V be the list of variables in Q. The system does the following:

(1) A program segment p(P) is generated such that I{p(P)}I' and I'UF |- P (note that p(P) may be empty).

(2) An inst ance Qλ of the loop invariant must be true in the state I', i.e. $\lambda = \{<v_1 \leftarrow s_1>,...,<v_n \leftarrow s_n>\}$ is constructed such that I'UF ⊃ Qλ.

(3) A program segment p(R) is generated such that Q ∧ L{p(R)}I" and I"UF ⊃ R.

(4) It is checked that I"UF ⊃ Qβ∨¬Lβ for some substitution β and a set of conditional assignment statements C is constructed such that I"{C}Q ∨ ¬L.

Thus, at the moment, clause (iv) ensures that C need contain only conditional assignments. In the future we would want to relax this restriction. It is assumed that the user's definition of the rule satisfies (vi). The user may omit S or L; in the latter case ¬G is used as the control test.

6.2 ASSEMBLY OF WHILE LOOPS: After the premisses have been achieved, a loop is assembled as follows:

(1) Let Y and W be two distinct lists of variables in one-to-one correspondence with V. For each $<v_i \leftarrow s, > \in \lambda$ construct an initial assignment statement "$y_i \leftarrow s_i$". Let "Y ← S" denote "$y_1 \leftarrow s_1 ; y_2 \leftarrow s_2 ; \ldots ; y_n \leftarrow s_n ;$".

(2) The WHILE loop is then assembled in the form:

```
p(P);
Y ← s;
WHILE L(Y) DO
   BEGIN
   p(R(Y));
   IF Q(W) THEN Y ← W;
   END
```

where Q(W) is an expression containing calls to Boolean procedures indicated (syntactically) by the presence of the special W-variables (Section 2, Rule RO). Q(W) is constructed from Q(V) by replacing V-variables by corresponding W-variables; p(R(Y)) is obtained in a similar way from p(R(V)). Since the variable lists are disjoint, none of the Y-variables occurs in Q(W).

There are many heuristics in the system to reduce the number of program variables, i.e. y's and w's generated, to select the relevant portion of Q to be used in conditional assignment statements, to generate simple assignment statements (whose right hand sides are functional terms composed from functions in the frame} instead of conditional

assignments, and to eliminate unnecessary assignment statements in the assembled program. These may all be classified as optimizations, some of which are done as the WHILE loop is assembled and others during a later optimization phase.

6.3 UPDATING THE STATE: After the while statement has been generated, the system updates the state. If an explicit output assertion S is given then the rule of invariance is applied in the same manner as with the postcondition of a primitive procedure. In the absence of an output assertion, a special update procedure runs the loop interpretively on the state until the goal G becomes true. The resultant state is used in further planning. This latter method is useful when the global effects of the loop computation are so extensive, or even unpredictable, that an explicit specification of S is difficult. It may result in excessive update computation, particularly when loops are nested.

6.4 CORRECTNESS: We sketch how the basic correctness proof of the formal algorithm (section 2.5) may be extended to the case where iterative rules are used to develop nodes in the successful subgoal tree. This requires that we supply the argument for this extra case in the induction step of that proof.

Let node G(n+l) be developed using an iterative rule, and assume first that this is the only iterative rule used. To simplify the notation, we shall assume that the matching substitution between the rule goal G and G(n+1) is the identity, i.e. $G = G(n+1) \wedge G'$.

It is convenient to view G(n+1) as being the root node of a THAND subtree (see e.g. figure 3, Section 2.3). The immediate subgoals of G(n+1) are (i) to (iv) above (6.1). Suppose that the last node to be achieved in the main tree is G(n), the associated state and program being I(n) and A(n) respectively. The induction hypothesis is $I\{A(n)\}I(n)$.

Let us abbreviate "IF Q(W) THEN Y←W" by C. In the successful subgoal tree, the subgoals of G(n+l) are all achieved so that we have

1. $I(n)\{p(P)\}I(n)'$ where $I(n)' \cup F \supset P$ and $I(n)' \cup F \supset Q\lambda$ (subgoals (i) and (ii)).

2. $Q\lambda\{Y \leftarrow S\}Q(Y)$ by the assignment axiom, R0.

3. $Q(Y) \wedge L(Y)\{p(R)\}I(n)''$ where $I(n)'' \cup F \supset R(Y)$ (see comment below), and $I(n)'' \cup F \supset (\exists Z)Q(Z) \vee (\neg(\exists Z)Q(Z) \wedge \neg L(Y))$ (subgoals (iii) and (iv)).

4. $(\exists Z)Q(Z)\{C\}Q(Y)$ by R0,

5. $\neg(\exists Z)Q(Z) \wedge \neg L(Y)\{C\}\neg L(Y)$ by R0,

6. $(\exists Z)Q(Z) \vee (\neg(\exists Z)Q(Z) \wedge \neg L(Y))\{C\}Q(Y) \vee \neg L(Y)$ by OR-lemma, 4,5,

7. $I(n)''\{C\}Q(Y) \vee \neg L(Y)$ by consequence R1,

8. $Q(Y) \wedge L(Y)\{p(R);C\}Q(Y) \vee \neg L(Y)$ by composition R2,3,7,

9. Q(Y){WHILE L DO p(R);C}G by iteration, 8,

10. I(n){p(P);Y←S;WHILE L DO p(R);C}I(n+1) by R2,R3,1,2,9

where I(n+1) = Inv(S,I(n))

Finally, A(n+1) = A(n); p(P); Y←S; WHILE L DO p(R);C; so that I{A(n+1)}I(n+1). Since SUF ⊃ G is assumed true and G = G(n+1)∧G', it follows that I(n+1)UF⊃G(n+1).

COMMENT; Step 3 above is justified by a second induction, L(F)||-Q(Y)∧L(Y){p(R)}R(Y), namely that programs constructed without using iterative rules are correct. This follows from the proof for the simplified case (Section 2.5), since the variables in the goal, R(Y) are required to occur in the initial state, Q(Y) ∧ L(Y).

The extension of the proof for more than one iterative rule is similar.

6.5 AN EXAMPLE: As an example of "while" loop generation consider the task of generating a program to compute the value of n factorial for some positive integer n where multiplication is not a primitive operation but is done by repeated addition. The Frame for this problem is shown in figure 11. Also used is the primitive procedure for assignment used in the example in Section 3. To achieve the goal "FACT(X0,N)" the system applies the iterative rule TFACT. The premises are achieved according to Section 6.1 which results in an application of another iterative rule TPROD. The premises of TPROD are achieved, the "inner" loop assembled and optimized and state is updated with respect to the output assertion. The assembled while loop is appended to the iteration step program for TFACT. The "outer" loop is then assembled and optimized and the state further updated reflecting the total state transformation of an execution of the nested loop program.

The output program after optimization with statements labeled according to their source of generaton in the algorithm is shown in figure 12. Note that successive values of the loop variables (called "UPDATE ASSIGNMENTS") are obtained by simple assignment statements rather than by conditional assignment as described in the algorithm. This is the result of applying system heuristics which are able to use the arithmetic operations PLUS and ADD1 which are primitive functions in the frame, to replace the conditional assignments.

RELATIONS	DEFINITION	FLUENT	PARTIAL	UNIQUENESS
VFACT(X,Y)	"The value of Y factorial is X"	TRUE	FALSE	VFACT(*,*)
C(X,Y)	"The contents of variable X is Y"	TRUE	FALSE	C(X, *)
FACT(X,Y)	"The variable X contains Y factorial"	TRUE	FALSE	FACT(X,*)
VPRODUCT(X,Y,Z)	"X is equal to the product of Y and Z"	TRUE	FALSE	FALSE
INTEGER(X)	"X is an integer"	FALSE	FALSE	FALSE
ISVAR(X)	"X is a variable"	FALSE	FALSE	FALSE
NEWVAR(X)	"X is a new local variable"	TRUE	FALSE	FALSE
=(X,Y)	"X equals Y"	TRUE	FALSE	FALSE

****

AXIOM	ANTECEDENT	CONSEQUENCE
TAFACT	{=(V9,1)∧=(V1Ø,1)} ∨ VFACT((DIV V9 V1Ø),(SUB1 V1Ø));	VFACT(V9,V1Ø);
TAPROD	{=(V5,Ø)∧=(V6,Ø)} ∨ VPRODUCT((MINUS V5,V3),(SUB1 V6),V3);	VPRODUCT(V5,V6,V3);

****

SIMPLIFICATION RULES

(ADD1(SUB1 x)) → x
(SUB1(ADD1 X)) → X
(MINUS(PLUS X Y)Y) → X
(DIV(PROD x Y)Y) → x

FUNCTION OUTPUT SYNTAX

(ADD1 X) = (X + 1)
(SUB1 X) = (X - 1)
(PLUS x Y) = (x + Y)

Figure 11a

ITERATIVE RULES

RULE NAME	TFACT	TPROD
BASIS CONDITION	NEWVAR(V7)∧INTEGER(V4) ∧VFACT(V5,V6)∧C(V3,V5) ∧C(V7,V6);	NEWVAR(V4)∧C(V4,∅) ∧C(V1,∅);
INVARIANT	C(V7,V1∅)∧C(V3,V9) ∧VFACT(V9,V1∅);	C(V4,V6)∧C(V1,V5) ∧VPRODUCT(V5,V6,V3);
ITERATION STEP	C(V7,(ADD1 V1∅))∧ PRODUCT(V3,V4,(ADD1 V1∅));	C(V4, (ADD1 V6)) C(V1,(PLUS V5,V3));
GOAL	FACT(V3,V4);	PRODUCT(V1,V2,V3);
TEST	¬=(V1∅,V4);	¬=(V6,V2);
OUTPUT ASSERTION	C(V3, (FAC V4));	C(V1,(PROD V2,V3));

****

Figure 11b

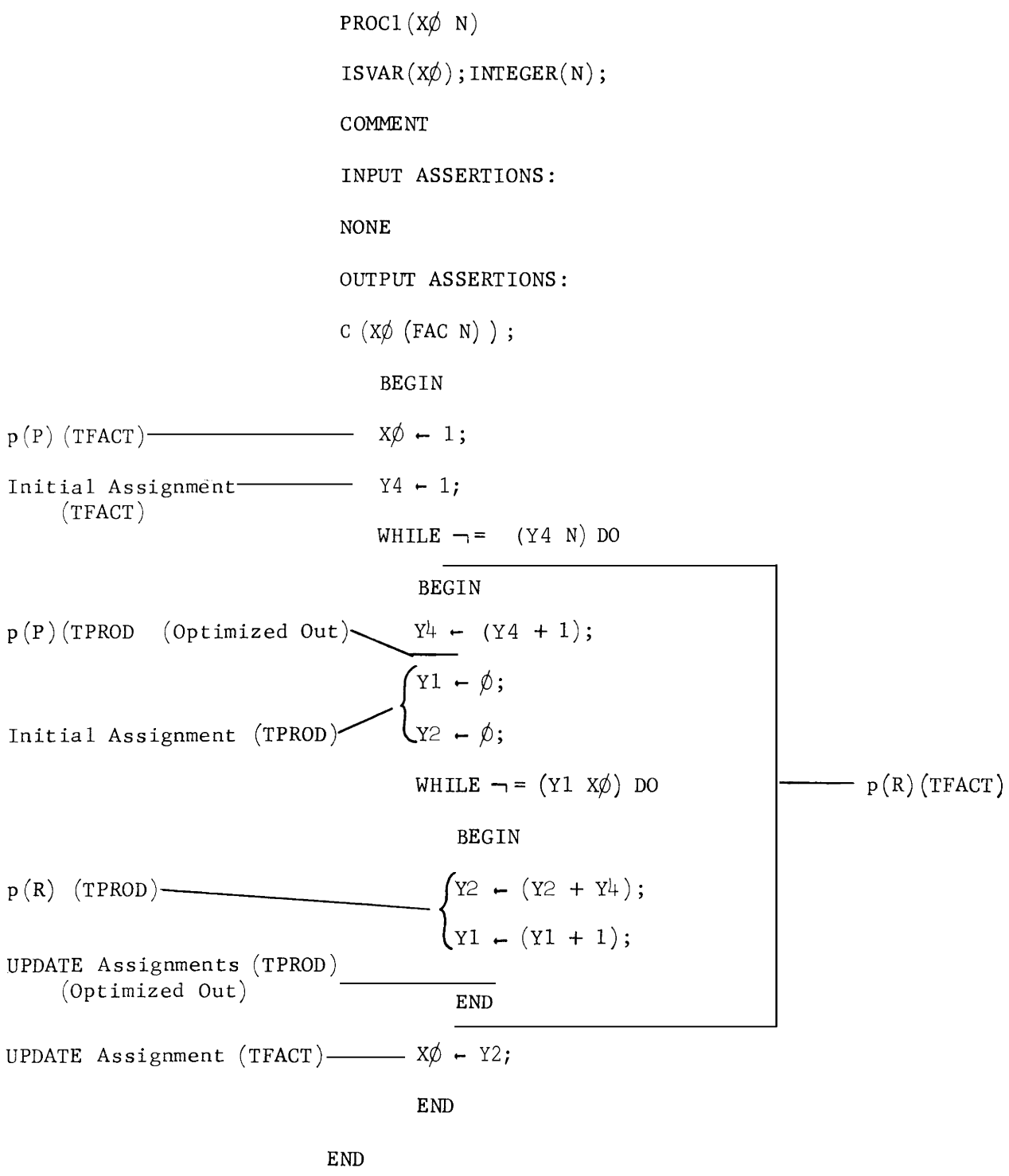

PROC1(XØ N)
ISVAR(XØ);INTEGER(N);
COMMENT
INPUT ASSERTIONS:
NONE
OUTPUT ASSERTIONS:
C (XØ (FAC N)) ;
BEGIN
p(P) (TFACT)
XØ ← 1;
Initial Assignment (TFACT)
Y4 ← 1;
WHILE ¬= (Y4 N) DO
BEGIN
p(P)(TPROD (Optimized Out)
Y4 ← (Y4 + 1);
Y1 ← Ø;
Initial Assignment (TPROD)
Y2 ← Ø;
WHILE ¬= (Y1 XØ) DO
p(R)(TFACT)
BEGIN
p(R) (TPROD)
Y2 ← (Y2 + Y4);
Y1 ← (Y1 + 1);
UPDATE Assignments (TPROD)
(Optimized Out)
END
UPDATE Assignment (TFACT)
XØ ← Y2;
END
END

Figure 12.

7. PROGRAMMING AIDS

The complexity of programs that can be generated using the system is increased by some simple facilities described in this section. The capabilities discussed here are incremental extension of a current program, use of a program library, and expansion of assumpt i ons.

The system enables a user to plan incremental extensions of a program simply by saving each completed program segment A and its output state 0 in a stack. The user may then pose a new goal G and solve the problem O{B}G. The composition A;B will then be output. He may choose to start from any previously saved state and associated program segment,

7.1 PROGRAM LIBRARY When a program A has been generated to solve P{A}Q, the user may request that it be "generalized" and filed in the program library where it may be accessed by the subgoaler (similar use of a library in robot planning is reported in [Fikes,Hart, and Nilsson 1972]).

Generalization is a process which constructs a procedure declaration for the library as follows. Let I and 0 be the input-output assertions computed for A during its construct ion. We assume P⊃I, O=Q∧O', and I{A}O. The non-fluent conjuncts of I are taken as the type declarations, their variables being the parameters of the new procedure. These actual parameters are replaced throughout I{A}O by new formal parameter variables. An entry of the form:

((<procname> <goal> <effects> <type conditions> <state condition>)<body>)

is made in the library, where <procname> is a name and parameter list, <goal> is Q, <effects> is O', <body> is A, and it is assumed that

<type conditions> ∧ <state condition>{<procname>}<goal> ∧ <effects>

Library procedures are used during program generation by matching on the <goal> then establishing the <type conditions> and <state conditions> as subgoals in that order, If the conditions are satisfied then the instantiated <body> is included in the program. There is no attempt to organize the library for efficient selection; the system merely tries all library procedures before any frame rule.

As an example of program assembly using the library consider the task of building a tower to reach an object, i.e. achieve "HAS(M,B)". Use will be made of a library program to find and put on shoes which achieves WEARING(M,SHOES), previously generated using the same Frame. The generated program is then extended interactively by posing a new goal, AT(M,P).

A robotics Frame for this problem is shown in Figure 13, and the generated programs in Figure 14.

RELATIONS	DEFINITION	FLUENT	PART IAL	UNIQUENESS
ROBOT X)	"X is a robot"	FALSE	FALSE	FALSE
BOX X)	"X is a box"	FALSE	FALSE	FALSE
AT(X,Y)	"X is at location Y"	TRUE	FALSE	AT(X,*)
ON X,Y)	"X is on Y"	TRUE	FALSE	ON'X, *)
HAS- X,Y)	"X has possession of Y"	TRUE	FALSE	FALSE
STACKED(X,Y,Z)	"X isstacked on Y at location Z"	TRUE	FALSE	FALSE
TSSTACK X,Y)	"X is in a stack at location Y"	TRUE	FALSE	INSTACK(X,*)
STACKHEIGHT X,Y)	"the stack height at location Y is X"	TRUE	FALSE	STACKHEIGHT(*,Y)
HEIGHT X,Y)	"X is positioned at a height of Y"	TRUE	FALSE	HEIGHT (X,*)
TOP X,Y)	"X is the top object in stack at Y"	TRUE	FALSE	TOP(*,Y)
HIENUF(X,Y,Z)	"X is as high as Y at Z"	TRUE	FALSE	FALSE
HOLDING(X,Y,Z)	"X is holblng Y at location Z"	TRUE	FALSE	HOLDING(X,*,Z)
CHAIR X)	"X is a chair"	FALSE	FALSE	FALSE
CLOTHES(X)	"X is an article of clothing"	FALSE	FALSE	FALSE
UNDER X,Y)	"X is under Y"	TRUE	TRUE	FALSE
WEARING X,Y)	"X is wearing clothing Y"	TRUE	FALSE	FALSE
FOUND X,Y)	"X found Y"	TRUE	FALSE	FALSE
=(X,Y)	"X is equal to Y"	FALSE	FALSE	FALSE
ABOVER(X,Y,Z)	"object X is above robot Y at Z"	TRUE	FALSE	FALSE
ABOVE X,Y,Z)	"object X is above object Y at Z"	TRUE	FALSE	FALSE
BOTTOMBOX X , Y)	"X is the bottom box at Y"	TRUE	FALSE	FALSE
BOTTOMBOXU X,Y,Z	"X is the bottom box at Z under Y''	TRUE	FA LSE	FALSE
BELOWR X ,Y , Z)	"object X is below robot Y at Z"	TRUE	FALSE	FALSE
BELOW X,Y,Z)	"object X is below object Y at Z"	TRUE	FALSE	FALSE
SUPPLY X)	"the supply is at location X"	FALSE	FALSE	FALSE
NEXTBOX X,Y)	"X is the next box after Y"	TRUE	FALSE	FALSE

Figure 13a

PRIMITIVE PROCEDURE	PRE-CONDITIONS	POST-CONDITIONS
travel (R1,L1,L2) "R1 travels from L1 to L2"	ROBOT(R1)∧AT(R1,L1)∧HEIGHT(R1,0);	AT(R1,L2);
move(R1,O1,L1,L2) "R1 moves O1 from L to L2"	ROBOT(R1)∧BOX(O1)∧AT(O1,L1)∧¬INSTACK(O1,L1)∧ CLOTHES(O3)∧WEARING(R1,O3)∧AT(R1,L1);	AT(O1,L2)∧AT(R1,L2);
stack(R1,O2,O1,L1) "R1 stacks O2 on O1 at L1"	ROBOT(R1)∧BOX(O1)∧BOX(O2)∧≠(O1,O2)∧AT(O1,L1)∧ AT(O2,L1)∧AT(R1,L1)∧HOLDING(R1,O2,L1)∧ HEIGHT(R1,0)∧ON(R1,O1,L1)∧¬STACKED(O3,O1,L1) ∧STACKHEIGHT(H1,L1);	STACKED(O2,O1,L1)∧ STACKHEIGHT((EVN(ADD1 H1)),L1) ∧TOP(O1,L1);
climb(R1,O1,L1) "R1 climbs O1 at L1"	ROBOT R1)∧ABOVER(O1,R1,L1)∧AT(R1,L1)∧ ¬INSTACK(O1,L1)∨ {STACKED(O1,O2,L1)∧ON(R1,O2,L1)}∧ REQUEST(HEIGHT(R1,H1));	ON(R1,O1,L1)∧ HEIGHT(R1,(EVN(ADD1 H1)));
unclimb(R1,O2,L1) "R1 unclimbs O2 at L1"	ROBOT(R1)∧BELOWR(O1,R1,L1)∧AT(R1,L1)∧ REQUEST(HEIGHT(R1,H1))∧ REQUEST(STACKED(O2,O1,L1))∧ ON(R1,O2,L1);	ON(R1,O1,L1)∧ HEIGHT(R1,(EVN(SUB1 H1)));
stepoff(R1,O1,L1) "R1 steps off O1 at L1"	=(H1,0)∧HEIGHT(R1,1)∧ON(R1,O1,L1);	HEIGHT(R1,H1)∧ ¬ON(R1,O1,L1);
reach(R1,O1,L1) "R1 reaches O1 at L1"	ROBOT(R1)∧AT(O1,L1)∧HIENUF(R1,O1,L1);	HAS(R1,O1);
lift(R1,O1,L1) "R1 lifts O1 at L1"	ROBOT(R1)∧BOX(O1)∧AT(O1,L1)∧AT(R1,L1)∧ ¬INSTACK(O1,L1);	HOLDING(R1,O1,L1);
find(R1,O1,L1) "R1 finds O1 at L1"	ROBOT(R1)∧CHAIR(O2)∧AT(O2,L1)∧AT(R1,L1)∧ UNDER(O1,O2);	FOUND(R1,O1);
put_on R1,O1) "R1 puts on O1"	ROBOT(R1)∧CLOTHES(O1)∧FOUND(R1,O1);	WEARING(R1,O1);

****

AXIOM	ANTECEDENT	CONSEQUENCE
TABOVER	¬ON R1,O2,L1)∨{ON(R1,O3,L1)∧ABOVE(O1,O3,L1)};	ABOVER(O1,R1,L1);
TABOVE	=(O1,O3)∨{STACKED(O2,O3,L1)∧ABOVE(O1,O2,L1)};	ABOVE(O1,O3,L1);
TBELWR	ON(R1,O2,L1)∧BELOW(O1,O2,L1);	BELOWR(O1,R1,L1);
TBELOW	=(O1,O3)∨{STACKED(O3,O2,L1)∧BELOW(O1,O2,L1)};	BELOW(O1,O3,L1);
TBOT	TOP(O3,L1)∧BOTTOMBOXU(O1,O3,L1);	BOTTOMBOX(O1,L1);
TBOTU	STACKED(O3,O4,L1)∧STACKED(O4,O2,L1)∨ STACKED(O3,O1,L1)∧¬STACKED(O4,O2,L1)∨ BOTTOMBOXU(O1,O4,L1);	BOTTOMBOXU(O1,O3,L1);
TNEXT	SUPPLY(L1)∧AT(O4,L1);	NEXTBOX(O4,O3);
TINSTACK	TOP(O2,L1)∧BELOW(O1,O2,L1);	INSTACK(O1,L1);

****

DEFINITION

THITE HEIGHT(O1,H1)∧STACKHEIGHT(H1,L1)∧TOP(O2,L1)∧ON(R1,O2,L1) ≡ HIENUF(R1,O1,L1)

Figure 13b

ITERATIVE RULE	BASIS CONDITION	INVARIANT	ITERATION STEP	GOAL	TEST	OUTPUT ASSERTION
TUP	REQUEST(HEIGHT(R1,H2)) ∧GZ(H2)∨ {BOTTOMBOX(O3,L1) ∧ON(R1,O3,L1)};	ON R1,O1,L1)∧ . STACKED(O2,O1,L1) ∨TOP(O1,L1);	ON(R1,O2,L1);	HEIGHT(R1,H1);	--	--
TDOWN	GZ(H1)∧ REQUEST(HEIGHT(R1,H2)) ∧GT(H2,H1);	ON(R1,O1,L1)∧ STACKED(O1,O2,L1) ∨BOTTOMBOX(O1,L1);	ON(R1,O2,L1);	HEIGHT(R1,H1);	--	--
TSTA	STACKED O2,O1,L1) ∧ON(R1,O2,L1);	TOP(O3,L1)∧ STACKHEIGHT (H2,L1)∧ NEXTBOX(O4,O3);	HOLDING(R1,O4,L1) ∧HEIGHT(R1,H2) ∧STACKED(O4,O3,L1);	STACKHEIGHT (H1,L1);	--	--

INITIAL STATE

ROBOT M)∧BOX(B2)∧BOX(B5)∧BOX B3)∧BOX(B6)∧BOX(B4)∧BOX(B7)∧AT(M,P)∧AT(B,U)∧AT(B2,SLOC)∧AT(B5,SLOC)∧AT(B3,SLOC)∧ AT B ,SLOC)∧AT(B4,SLOC)∧AT(B7,SLOC)∧SUPPLY(SLOC)∧STACKHEIGHT(Ø,U)∧HEIGHT(M,Ø)∧HEIGHT(B,4)∧CLOTHES(SHOES)∧ CHAIR(CHAIR1)∧CHAIR CHAIR2)∧AT SHOES,CORNER)∧AT(CHAIR1,CORNER)∧AT(CHAIR2,CORNER);

ADVICE

RECURSIVE RULES: CLIMB,TABOVE,TBELOW,TBOTU

PAIRWISE INEQUALITIES: travel(R1,*,*),move(R1,O1,*,*) STACK(R1,*,*,L1)

Figure 13c

```
PROC1 (M SHOES)
ROBOT M);CHAIR(CHAIR2);CLOTHES(SHOES);
COMMENT
INPUT ASSERTION:
HEIGHT(M Ø )∧AT(M P)∧AT(CHAIR2 coRNER)
OUTPUT-ASSERTION:
AT M CORNER)∧FOUND(M SHOES)AWEARING(M SHOES);
COMMENT
PROC2 ATTEMPTS-TO-ACHIEVE- (FOUND M SHOES);
   BEGIN
   TRAVEL(M P CORNER);
   IF ¬UNDER(SHOES CHAIR2) THEN
      PROC2(M SHOES)
   ELSE
      BEGIN
      FIND(M SHOES CORNER)
      END
   PUT_ON(M SHOES)
   END
```

```
. PROC3 (M B)
ROBOT(M);BOX(B7);CLOTHES(SHOES);CHAIR(CHAIR2);BOX(B4);SUPPLY(SLOC);BOX(B6);BOX(B3);
COMMENT
INPUT_ASSERTION:
AT (M P)∧AT(B7 SLOC)∧HEIGHT(M Ø )∧AT(CHAIR2 CORNER)∧AT(B4 SLOC)
∧HEIGHT(B 4)∧STACKHEIGHT(Ø U)∧AT(B6 SLOC)∧AT(B3 SLOC)
OUTPUT-ASSERTION:
AT(M P)∧AT(B7 U)∧AT(B4 U)∧STACKED(B4 B? U)∧AT(B6 U)
∧STACKED(B6 B4 U)∧STACKHEIGHT(4 U)∧HAS(M B)∧HEIGHT(M Ø)
∧FOUND(M,SHOES)∧WEARING(M,SHOES);∧AT(B3 U)∧STACKED(B3 B6 U);
   BEGIN
   TRAVEL(M P)CORNER);
   IF¬ UNDER(SHOES CHAIR2) THEN
      PROC?(M SHOES)
   ELSE
      BEGIN
      FIND(M SHOES CORNER)
      END
   PUT_ON(M SHOES);
   TRAVEL(M CORNER SLOC);
   MOVE(M B SLOC U);
   TRAVEL(M U SLOC);
   MOVE(M B4 SLOC u);
   LIFT(M B4 U);
   CLIMB(M B7 U);
   STACK(M B4 B7 U);
   CLIMB(M B4 U);
   Y3 ← 2;
   Y 4 ← B4;
   IF NEXTBOX(W4 Y4) THEN
      Z4 ← w4;
   WHILE ¬STACKHEIGHT(4 U) DO
      BEGIN
      Z3 ← ADD1(Y3);
      Y1 ← Y4;
      IF STACKED(Y1 W1 U) THEN
         Z1 ← W1;
      WHILE ¬HEIGHT(M 1)DO
         BEGIN
         UNCLIMB(M Y1 U);
         Y1 ← Z1;
         IF STACKED(Y1 W1 U) THEN
            Z1 ← W1;
         END
      STEPOFF(M B7 U);
      TRAVEL(M U SLOC);
      MOVE(M Z4 SLOC U);
```

Assembled from library →

Figure 14a

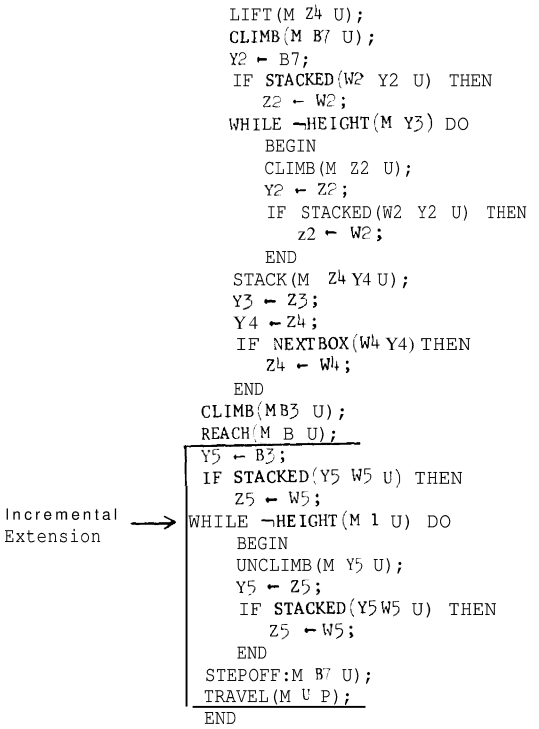

LIFT(M Z4 U);
CLIMB(M B7 U);
Y2 ← B7;
IF STACKED(W2 Y2 U) THEN
Z2 ← W2;
WHILE ¬HEIGHT(M Y3) DO
BEGIN
CLIMB(M Z2 U);
Y2 ← Z2;
IF STACKED(W2 Y2 U) THEN
z2 ← W2;
END
STACK(M Z4 Y4 U);
Y3 ← Z3;
Y4 ← Z4;
IF NEXTBOX(W4 Y4) THEN
Z4 ← W4;
END
CLIMB(MB3 U);
REACH(M B U);
Y5 ← B3;
IF STACKED(Y5 W5 U) THEN
Z5 ← W5;
Incremental
Extension
WHILE ¬HEIGHT(M 1 U) DO
BEGIN
UNCLIMB(M Y5 U);
Y5 ← Z5;
IF STACKED(Y5 W5 U) THEN
Z5 ← W5;
END
STEPOFF:M B7 U);
TRAVEL(M U P);
END

Figure 14b

7.2 EXPANSION OF ASSUMPTIONS: A basic capability for structuring programs is provided by interactively allowing the user at any level in program generation to define a primitive procedure, P{p}Q, as an assumption. The program generator will then use p as usual except at each point of call to p in the program the current state I' and current goal G will be saved. The triple <p,I',G> is placed in a stack of **subtasks** for later expansion.

When a program containing assumed primitive procedures has been generated, the user is given the list of assumptions his program depends on and allowed to selectively expand them in terms of lower level procedures. For the **subtask** <p,I',G>, the state is initialized to I', the frame may be changed, G is given as the goal,and a body for the procedure p is generated.

Consider the example given in Section 6 of computing the value of n factorial where multiplication is not a primitive operation. The initial frame is the same except that in place of an iterative rule for multiplication, there is an assumed primitive procedure

ISVAR(V1){times(V1,V2,V3)}PRODUCT(V1,V2,V3),

where PRODUCT(V1,V2,V3)≡C(V1,(PROD V2,V3)).

The program generated using this frame is given in Figure 15. To expand the **non-**primitive procedure **"times(V1,V2,V3)"** the full frame including the iterative product rule is given and the sub-program generated is shown in Figure **16.**

In the current implementation it is assumed that the expanded,sub-programs will have no side effects, However this assumption could be removed **by a** mechanism similar to checking rejoin conditions for contingency programs (Section 5.4).

To develop a useful structured programming system interaction appears essential **along** with further study about how humans do (or should do) programming.

```
PROC1( XØ N)
ISVAR(XØ);INTEGER(N);
COMMENT
INPUT ASSERTION:
NONE
OUTPUT ASSERTION:
C(XØ (FAC N));
COMMENT
THIS PROGRAM RELIES ON THE FOLLOWING ASSUMPTIONS:
(TIMES)
   BEGIN
   XØ ← 1;
   Y1 ← 1;
   WHILE ¬ >(Y1 N) DO
      BEGIN
      Y1 ← Y1+1;
      TIMES(XØ XØ Y1)
      END
   END
```

Figure 15

```
TIMES(XØ Y1 Z1)
ISVAR(XØ);
COMMENT
INPUT ASSERTION:
NONE
OUTPUT ASSERTION:
C(XØ (PROD Y1 Z1));
   BEGIN
   XØ ← 0;
   Y0 ← Ø;
   WHILE ¬ =(Y0 Y1) DO
      BEGIN
      Y0 ← Y0+1;
      XØ ← XØ+Z1;
      END
   END
```

Figure 16

REFERENCES

Balzer, R. 1972. "Automatic Programming", Information Sciences Institute, Univ. Southern California, Technical Memorandum, September 1972.

Buchanan, J.R. 1974. Ph.D. Thesis, Stanford University, 1974.

Fikes, R.E.; Hart, P.E.; Nilsson, N.J. 1972. "Learning and Executing Generalized Robot Plans", Artificial Intelligence 3, 251-288.

Hewitt, C. 1971. "Description and Theoretical Analysis of Planner" Ph.D. Thesis, M.I.T., 1971.

Hoare, C.A.R. 1969. An axiomatic basis for computer programming, Comm. ACM, 12,10, October 1969, 576-580, 583.

Hoare, C.A.R.; and Wirth, N. 1972. An axiomatic definition of the programming language Pascal, Bericht e der Fachgruppe Computer-Wissenschaft en 6, E.T.H., Zurich, November 1972.

Igarashi, S.; London, R.L.; Luckham, DC. 1973. "Automatic Program Verification I: A Logical Basis and Implementation", Stanford AIM 200, May 1973. (to appear in Act a Informatica)

Luckham, DC.; Buchanan, J.R. 1974. "Automatic Generation of Programs Containing Conditional St at ements", Proceedings A.I.S.B. Summer Conference, Sussex, England, July 1974.

McCarthy, J.; and Hayes, P. 1969. "Some Philosophical Problems from the Standpoint of Artificial Intelligence" Machine Intelligence 4, pp. 463-502, Edinburgh University Press.

Nilsson, N., "Problem Solving Methods in Artificial Intelligence", McGraw-Hill, 197 1.

Sussman, J.; Winograd, T. 1972. "Micro Planner Reference Manual", M.I.T. Project MAC Report 1972.

APPENDIX 1 - AN INTERACTIVE SESSION

A sample interactive session is here presented to illustrate the system's **use in frame** definition and program generation. Statements typed by the user will always be prompted by "*". The top level system function is "SUBGOAL" which is called in the manner given below to accept a frame definition from the terminal. Comments to aid the reader's understanding of the dialogue will be enclosed in quotes,

*(SUBGOAL)
"The system now enters an interactive mode for Frame definition."
**** SEMANTIC FRAME DEFINITION ****

RULE TYPE* AXIOM
RULE NAME* AONTOP
IS THIS AN ASSUMPTION?* NIL
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(X1) A ON(X1 ,X2) A ¬STACKED(X3,X2);
POSTCONDITIONS:
* ONTOP(X 1);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* STANDON(R1,Z1)
IS THIS AN ASSUMPTION?* NIL
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(R1) ∧ ¬ON(R1,W1) ∧ BOX(Z1) ∧ CLOTHES(O1) A WEARING(R1,O1)
A AT(Z1,Y1) A AT(R1,Y1);
POSTCONDITIONS:
* ON(R1,Z1);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* DRESS(R1,O1)
IS THIS AN ASSUMPTION?* T
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(R1) A CLOTHES(O1);
POSTCONDITIONS:
* WEARING(R1,O1);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* TRAVEL(R1,L1,L2)
IS THIS AN ASSUMPTION?* NIL
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* (R 1 ,*,*)

PRECONDITIONS:
* ROBOT(R1) ∧ AT(R1,L1) ∧ ¬ ON(R1,O2,L1);
POSTCONDITIONS:
* AT(R1,L2);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* STEPUP(X1,Y1,Z1)
IS THIS AN ASSUMPTION?* NIL
IS THE RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* (R1,*,*)
PRECONDITIONS:
* BOX(Z1) ∧ ROBOT(X1) ∧ STACKED(Z1,Y1) ∧ ON(X1,Y1);
POSTCONDITIONS:
* ON(X1,Z1);

RULE TYPE* ITERATIVE
RULE NAME* ITONTOP
IS THIS RULE DIRECTLY RECURSIVE?* NIL
BASIS CONDITION:
* ROBOT(X1) ∧ ON(X1,X2);
INVARIANT:
* ON(X1,X3) ∧ STACKED(X4,X3);
ITERATION STEP CONDITION;
* ON(X1,X4);
CONTROL TEST* NIL
OUTPUT ASSERTION* NIL
GOAL* ONTOP(X1);

RULE TYPE* NIL

INITIAL STATE:
* AT(M,CORNER) ∧ AT(B1,L) ∧ STACKED(B3,B2) ∧ STACKED(B2,B1)
∧ BOX(B3) ∧ BOX(B2) ∧ BOX(B4) ∧ STACKED(B4,B3) ∧ BOX(B1)
∧ ROBOT(M) ∧ CLOTHES(SHOES);

SEMANTIC PROPERTIES OF RELATIONS:

IS ROBOT(R1) A FUNCTION OF THE STATE?* NIL
IS ROBOT(R1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS AT(R1,L1) A FUNCTION OF THE STATE?* T
IS AT(R1,L1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* (R1,*)

IS STACKED(X4,X3) A FUNCTION OF THE STATE?* T
IS STACKED(X4,X3) PARTIAL?* NIL

```
ARGUMENT UNIQUENESS PROPERTIES* (X4,*)

IS BOX(Z 1) A FUNCTION OF THE STATE?* NIL
IS BOX(Z 1) PARTIAL?+ NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS ONTOP(X1) A FUNCTION OF THE STATE?* T
IS ONTOP( X 1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS CLOTHES(O1) A FUNCTION OF THE STATE?* NIL
IS CLOTHES(O1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS WEARING(R1,O 1) A FUNCTION OF THE STATE?* T
IS WEARING(R1,O 1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS ON(X 1 ,Z 1) A FUNCTION OF THE STATE?* T
IS ON(X 1 ,Z 1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* (XI,*)

FILENAME* DSK:PCLI
TRACE MODE?* T
PERFORMANCE STATISTICS?* T
LOOKAHEAD?* NIL
ALGEBRAIC SIMPLIFICATION?* NIL

SUBGOALING SYSTEM GENERATED!!!
"A subgoaling system corresponding to the Frame has now been generated
and the system may now receive a goal to achieve."

SUBMIT GOAL* ONTOP(M)
DO YOU WANT THE PROGRAM LIBRARY?* NIL
DO YOU HAVE ANY ADVICE?* T
*** ENTERING ADVICE SYSEM ***
1* TRY STANDON BEFORE STEPUP
 2* NIL "Exit advice system and begin program generation."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---ITONTOP

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M X2))STANDON

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M X2))(STANDON(WEARING M SHOES))DRESS
```

((DRESS M SHOES))
"Current program segment generated is displayed in this form."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M X2))(STANDON(AT M L))TRAVEL

((DRESS M SHOES)(TRAVEL M CORNER L))
((DRESS M SHOES)(TRAVEL M CORNER L)(STANDON M B1)
"This constitutes the basis program for the iterative rule."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M B2))STANDON

STANDON IS FAILING!!!
---(¬ON M W 1) WAS THE LOSER
"STANDON is only applicable for climbing from ground level."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ON M B2))STEPUP

((STEP M B1 B2))
"This is part of the loop body."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---(ITONTOP(ONTOP M))AONTOP
"The system now interpretively updates the state until the goal is true, then the while loop is assembled."

DO YOU WANT TO OPTIMIZE THE PROGRAM?* NIL
IS THIS PLAN USEFUL ENOUGH TO GENERALIZE?* T
IS THIS A PROCEDURE WITHOUT SIDE EFFECTS?* NIL

THE GOAL (ONTOP M) IS ATTAINABLE BY THE FOLLOWING PROGRAM:
"The desired program has been generated and will now be displayed,"

```
PROC1(M)
ROBOT(M);CLOTHES(SHOES);(BOX(B1);BOX(B2);
COMMENT
INPUT ASSERTIONS:
AT(M CORNER) ∧ AT(B1 L) ∧ STACKED(B2 B1)
OUTPUT ASSERTIONS:
WEARING(M SHOES) ∧ AT(M L) ∧ ONTOP(M);
COMMENT
THIS PROGRAM RELIES ON THE FOLLOWING ASSUMPTIONS:
(DRESS);
   BEGIN
   DRESS(M SHOES);
```

```
   TRAVEL(M CORNER L);
   STANDON(M B1);
   Y1 ← B1;
   IF STACKED(W1 Y1) THEN
      Z1 ← W1;
   WHILE ¬ ONTOP(M) DO
      BEGIN
      STEPUP(M Y1 Z1);
      Y1 ← Z1
      IF STACKED(W1 Y 1) THEN
         Z1 ← W1;
      END
   END

DO YOU WANT TO DO STRUCTURED PROGRAM DEVELOPMENT?* T

TRYING---((DRESS M SHOES)(WEARING M SHOES)(STAT1.AST))
"This task triple consists of procedure name, goal and state."

DO YOU HAVE ANY ADVICE?* T

***ENTERING ADVICE SYSTEM***
 1* ADD PUT-ON

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* PUT-ON(R1,O1)
IS THIS AN ASSUMPTION?* NIL
IS RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(R1) A CLOTHES(O1) A FOUND(R1,O1);
POSTCONDITIONS:
*WEARING(R1,O1);

RULE TYPE* PRIMITIVE PROCEDURE
RULE NAME* FIND(R1,O1,L1)
IS THIS AN ASSUMPTION?* NIL
IS RULE DIRECTLY RECURSIVE?* NIL
INEQUALITIES IN ARGUMENT POSITIONS* NIL
PRECONDITIONS:
* ROBOT(R1) A CHAIR(O2) A AT(O2,L1) A AT(R1,L1) A UNDER(O1,O2);
POSTCONDITIONS:
* FOUND(R1,O1);

RULE TYPE* NIL

INITIAL STATE:
```

```
* CHAIR(CHAIR1) A CHAIR(CHAIR2) A AT(CHAIR1,CORNER)
A AT( CHAIR2,CORNER);

SEMANTIC PROPERTIES OF RELATIONS:

IS FOUND(R 1 ,0 1) A FUNCTION OF THE STATE?* T
IS FOUND{ R 1 ,0 1) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES+ NIL

IS CHAIR(02) A FUNCTION OF THE STATE?* NIL
IS CHAIR(02) PARTIAL?* NIL
ARGUMENT UNIQUENESS PROPERTIES* NIL

IS UNDER(01,02) A FUNCTION OF THE STATE?* T
IS UNDER(01,02) PARTIAL?* T
ARGUMENT UNIQUENESS PROPERTIES* NIL

ALGEBRAIC SIMPLIFICATION?* NIL

SUBGOALING SYSTEM GENERATED!!!
"The Frame addit ion has now been translated,"

 2* DELETE DRESS
 3* NIL
"Exit Advice system."

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH:
---( PUT-ON( FOUND M SHOES))FIND

((FIND M SHOES CORNER))
((IF(¬UNDER SHOES CHAIR1) THEN (PROC2 M SHOES}
ELSE((FIND M SHOES CORNER)))(PUT-ON M SHOES))
"The conditional statement is generated since it is not known where
the shoes are."

DO YOU WANT TO OPTIMIZE THE PROGRAM?* NIL
IS THIS PROGRAM USEFUL ENOUGH TO GENERALIZE?* T
IS THIS PROCEDURE WITHOUT SIDE EFFECTS?* NIL

THE GOAL (WEARING M SHOES) IS ATTAINABLE BY THE FOLLOWING PROGRAM:
"This procedure is the structured expansion of the non-primitive
procedure DRESS called in PROC1."

DRESS(M SHOES)
ROBOT( M);CLOTHES( SHOES);CHAIR(CHAIR 1);
COMMENT
INPUT ASSERTIONS:
```

```
AT(M CORNER) ∧ AT(CHAIR1 CORNER)
OUTPUT  ASSERTIONS:
WEARING(M SHOES) A FOUND(M SHOES) A WEARING(M SHOES);
COMMENT
PROC2 ATTEMPTS TO ACHIEVE FOUND(M SHOES);
   BEGIN
   IF ¬UNDER(SHOES CHAIR1) THEN
      PROC2(M SHOES)
   ELSE
      BEGIN
      FIND( M SHOES CORNER);
      END
   PUT-ON(M SHOES)
   END

DO YOU WANT TO DO CONTINGENCY PLANNING?* T
WHAT IS YOUR PREFERENCE?
----IF NONE TYPE NIL* NIL

TRYING---(PROC2 (FOUND M SHOES)(STAT2.CST))
"The contingency task triple consists of procedure name, goal and state,"

DO YOU HAVE ANY ADVICE?* NIL

RULES ENTERED AND GOALS PENDING IN CURRENT SUBGOAL TREE PATH;
---FIND
((FIND M SHOES CORNER))
DO YOU WANT TO OPTIMIZE THIS PROGRAM?* NIL
IS THIS PROGRAM USEFUL ENOUGH TO GENERALIZE?* T
IS THIS PROCEDURE WITHOUT SIDE EFFECTS?* NIL

THE GOAL FOUND(M SHOES) IS ATTAINABLE BY THE FOLLOWING PROGRAM:

PROC2(M SHOES)
ROBOT(M);CHAIR(CHAIR2);
COMMENT
INPUT ASSERTIONS:
AT(CHAIR2 CORNER) A AT(M CORNER)
OUTPUT ASSERTIONS;
FOUND(M SHOES);
COMMENT
PROC3 ATTEMPTS TO ACHIEVE FOUND(M SHOES);
   BEGIN
   IF ¬UNDER(SHOES CHAIR2) THEN
      PROC3(M SHOES)
   ELSE
      BEGIN
```

```
      FIND(M SHOES CORNER);
      END
   END
DO YOU WANT TO DO CONTINGENCY PLANNING?* NIL
DO YOU WANT TO CONTINUE FROM THE CURRENT STATE?* NIL
```