Structured Planning and Debugging

A Linguistic Approach to Problem Solving

by

Ira P. Goldstein and Mark L. Miller

Abstract

A structured approach to planning and debugging is obtained by using an Augmented Transition Network (ATN) to model the problem solving process. This proves to be a perspicuous representation for planning concepts including techniques of identification, decomposition and reformulation. It also provides an elegant theory of debugging, in which bugs are identified as errors in transitions between states in the ATN. Examples from the Blocks World and elementary graphics programming problems are used to illustrate the theory.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. It was supported in part by the National Science Foundation under grant C40708X and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

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 the National Science Foundation or the United States Government.

Working Papers are informal papers intended for internal use.
## Contents

1. Introduction 3
   Rigor 3
   Clarity 4
   Power 4
   Prediction 5
   Outline 6

2. Structured Planning 7
   Problem Identification 9
   The Plan Node of the ATN 12
   Problem Decomposition 14
   Decomposition by Conjunction 15
   The ATN Subgraph for Conjunction 18
   Composition by Sequential Refinement 22
   The SIMPLE ATN 27
   Decomposition by Repetition 27
   Problem Reformulation 29

3. Searching for the Plan 32
   Lookahead 32
   Least Commitment 33
   Differential Diagnosis 39
   Lemma Libraries 39

4. Limitations and Extensions of Structured Planning 40

5. Structured Debugging 43
   Model Diagnosis 44
   Process Diagnosis 44
   Plan Diagnosis 45
   Debugging in Context 47

6. A SIMPLE Description of HACKER 49
   Bugs Arising from Incomplete Plans 50
   Bugs Arising from Incorrect Conjunction Plans 51
   Bugs Arising from Incorrect Disjunction Plans 53
   HACKER Conclusions 55

7. Conclusions 58
   Protocol Analysis 58
   Computer Aided Instruction 60
   The Science of Heuristics 62

8. References 63
1. Introduction

Though it is difficult to prescribe any Thing in these Sorts of Cases, and every Person's own Genius ought to be his Guide in these Operations; yet I will endeavor to show the Way to Learners.

Newton, Universal Arithmetick translated by Ralphson, 1769, p. 198 from [Polya, 1965, p. 89].

The structured programming movement [Dahl, 1972] has focused the concern of computer scientists on the process of creating programs. Work in Artificial Intelligence (AI) has developed a complementary theory of debugging [Sussman, 1973; Goldstein 1974]. But, except for work on Procedural Nets [Sacerdoti, 1975] discussed in section 3, a comprehensive approach has not yet been attempted. This is a preliminary report on the development of a structured theory of planning and debugging which we believe to be a step towards an integrated problem solving theory. In developing this theory, we have been governed by four criteria: rigor, clarity, power, and prediction.

Rigor

We strive for rigor through the use of concepts from computational linguistics to structure the planning and debugging processes. Specifically, we utilize an Augmented Transition Network (ATN) for expressing planning concepts, registers for specifying context, and constraints on the transition arcs for representing pragmatic knowledge used in the choice of alternative planning methods. The ATN defines a problem solving program called SIMPLE which is sufficiently precise to be hand-simulated, but has not yet been implemented. SIMPLE is an acronym for Structured Imperative Planner, LinguisticallyEncoded.

An Augmented Transition Network [Woods, 1970; Woods, Kaplan & Webber 1972] is a finite state transition graph with labelled states and arcs, augmented by recursion and a finite number of registers. Associated with each arc may be an arbitrary condition on following the arc, and an
arbitrary action to be executed if the arc is followed. Typically the conditions are restricted to Boolean predicates over the contents of the registers; while the actions are restricted to structure building and modifying the contents of the registers. Woods definition was an elaboration and formalization of earlier work by Bobrow and Fraser [1969], and by Thorne, Bratley and Dewar [1968]. Woods attributes some aspects of the ideas to Kuno [1965] and Conway [1963].

The possibility of rational errors makes debugging an important part of a structured planning theory. Rational errors are mistakes in planning that arise from the use of reasonable heuristics. In terms of Structured Planning, diagnosis is the identification of incorrect or incomplete transitions made between ATN states during the planning process. Repair involves re-planning, guided by advice from the diagnosis.

Clarity

Rigor can be achieved, but clarity sacrificed, through the use of clumsy or obscure procedural representations. Expressing a theory in terms of a program is not necessarily perspicuous. However, Structured Planning achieves considerable clarity through the use of concepts from computational linguistics such as grammars, ATNs, and parse trees. It is not surprising, perhaps, that constructs for representing linguistic theories should apply to problem solving and programming behavior. Protocols of rational problem solving can be viewed as structured communication, with the problem solver expressing a plan, listening to the environment's responses and restructuring his or her solutions accordingly.

Power

AI has had success in the construction of expert special purpose problem solving programs, for example, DENDRAL [Buchanan, Sutherland & Feigenbaum, 1969], MACSYMA [Moses, 1974], and EL [Sussman & Stallman, 1975]. But, there has been less progress in the creation of equally powerful general problem solvers, with reasoning expertise relevant to a wide number of applications.
HACKER [Sussman, 1973] and NOAH [Sacerdoti, 1975] are forerunners in this direction, both being attempts at general planning and debugging systems. We believe that Structured Planning is a generalization of both of these systems and constitutes a powerful and widely applicable paradigm. We support this claim by applying SIMPLE to two benchmark AI domains: the Blocks World and the Logo Turtle World.

Blocks World problem solvers include SHRDLU [Winograd 1972], BUILD [Fahlman 1974], HACKER and NOAH. Hence, applying SIMPLE to the Blocks World provides a common set of problems for comparison. The virtues of the Logo graphics world [Papert 1971a, 1971b, 1973] are: (a) graphics is an environment in which multiple problem descriptions are possible ranging from Euclidean geometry to Cartesian geometry, (b) the possible programs range over a wide spectrum of complexity, and (c) there is extensive documentation on human performance in this area [G. Goldstein, 1973; Okumura, 1973].

Prediction

Structured Planning can model aspects of human problem solving. In this respect, our work is germane to what Allen Newell has called "theoretical psychology" [Schank & Colby, 1973, p. 25]. In [Goldstein & Miller, 1976], a context free grammar was used to represent a taxonomy of planning and debugging concepts. This grammar was then applied (by hand) to parsing the problem solving behavior of a student programmer. The SIMPLE ATN generalizes this context free grammar in order to incorporate semantic and pragmatic planning knowledge, thereby defining a deterministic problem solving process.

We do not discuss protocol analysis in this paper. The interested reader should see [Miller & Goldstein, 1976] in which we explore the question of Structured Planning's ability to supply explanations and predictions of human problem solving behavior by considering the design of an an automated protocol analysis program based on the ATN developed here.
Outline

Planning is discussed in sections 2-4. Section 2 constructs the SIMPLE ATN from a taxonomy of planning techniques. Section 3 discusses subtleties in planning when viewed as a search process. Section 4 considers limitations and extensions of our approach.

A description of basic bug types described in terms of errors in the planning process is undertaken in section 5. Section 6 presents a re-description of the blocks world debugging program HACKER from the Structured Planning standpoint. Section 7 concludes with possible applications to protocol analysis, structured programming, and computer aided instruction.
2. Structured Planning

For a fortnight I had been attempting to prove that there could not be any function analogous to what I have since called Fuchsian functions. I was at that time very ignorant. Every day I sat down at my table and spent an hour or two trying a great number of combinations, and I arrived at no result. One night I took some black coffee, contrary to my custom, and was unable to sleep. A host of ideas kept surging in my head; I could almost feel them jostling one another, until two of them coalesced, so to speak, to form a stable combination. When morning came, I had established the existence of one class of Fuchsian functions...


Is there a well organized collection of planning concepts, or does human problem solving depend upon an exhaustive consideration of all combinations, as Poincare suggests? The many insightful analyses of problem solving provided in [Polya, 1957; 1962; 1965; 1967] support the assertion that planning knowledge is highly structured. A taxonomy diagramming part of this structure is given in figure 1. This taxonomy is incomplete, but it clearly illustrates Structured Planning's goal of organizing problem solving knowledge in a hierarchial fashion. In this paper, we limit ourselves to a detailed consideration of only a portion of this taxonomy. We shall discuss Identification and Decomposition by Conjunction. The taxonomy of figure 1 is more extensive than this in order to give a feel for the context in which our discussion takes place. Specifically, our goal will be to define the procedural content of the subtrees of the taxonomy concerned with Identification and Decomposition by Conjunction by embedding them in an ATN. A brief discussion of Repetition and Reformulation is given at the end of this section.

Planning begins with a choice between three methods -- identification, decomposition and reformulation. The first identifies the problem with known solutions. The second divides the problem into sub-problems that are (hopefully) easier to solve. The third reformulates the problem to an alternative whose solution is equivalent to, or at least a stepping stone towards, the solution of the current problem.
Figure 1
Taxonomy of Planning Concepts
Problem Identification

From desire ariseth the thought of some means we have seen produce the like of that which we aim at; and from the thought of that, the thought of means to that mean; and so continually, till we come to some beginning within our own power.


The power of problem identification arises from an extensible library of known solutions, indexed by their problem descriptions. This Answer Library is initialized with primitives provided by the current domain of interest, described by their effects. But with each successful problem solving episode, the library grows in power and breadth through the addition of new solutions.

To develop problem identification rigorously, a precise description of primitives and problems is required. A traditional method is used: problems are presented as predicate calculus models of the desired objects to be achieved, their properties and their relationships. Predicate calculus is the problem description language of mathematics as well as a variety of AI programs, most notably the STRIPS series of problem solvers [Fikes, 1971; 1972]. More powerful descriptive languages based upon such concepts as frames [Minsky, 1975; Winograd, 1975; Goldstein, 1975] may be more appropriate, but we have not yet explored this issue. For our purposes in this article, the problem descriptions are simply a conjunction of properties and relations about some set of objects. As such, they are common to most descriptive schemes including the predicate calculus, frames, and semantic nets [Quillian, 1968; Winston, 1975; Woods, 1975].

To illustrate the use of the predicate calculus for indexing the Answer Library, consider an example from the Logo turtle world. The turtle is a graphics cursor on a display that is moved primarily by two commands: FORWARD and RIGHT. The former moves the turtle display in the direction of its current heading; the latter rotates the turtle around its own axis.

In the Answer Library, the primitive FORWARD is described by a Post model that indicates
Structured Planning

Its effects, i.e. what it can be used for, and a Pre model that states its prerequisites.

Pre Model of Forward X
(EXISTS TURTLE)
(EXISTS DISPLAY)
(WITHIN (NEWPOSITION (FORWARD :X)) :SCREEN)

Post Model of Forward X
(EXISTS VECTOR V))
(= (LENGTH V) X)
(= (HEADING V) (HEADING TURTLE))
(= (VISIBILITY V) (PEN TURTLE))

This is analogous to the definition of operators in STRIPS. Identification uses the Post Model to match the problem with known solutions. The Pre Model establishes necessary sub-goals, for a given solution. For a more detailed discussion of the link between turtle primitives and model descriptions, see [Goldstein, 1974, chapter 6].

Figure 2 illustrates a typical picture that a Logo student attempts to accomplish by manipulating the turtle. This kind of project is commonly undertaken by fifth graders after about five hours of experience with the computer [G. Goldstein, 1973, p. 23]. A predicate calculus model for the Wishingwell picture is:

FIGURE 2
WISHINGWELL PICTURE

\[ \text{\textbf{FIGURE 2}} \]
\[ \text{\textbf{WISHINGWELL PICTURE}} \]
A glossary of primitive predicates for describing elementary Logo pictures is not given here [See Goldstein, 1974, Appendix B], but the basic form of these models should be self-evident. Currently, SIMPLE is given the problem in the form of a model, and is not asked to generate the model from graphic input. The particular choice of model is not absolutely critical, as SIMPLE is prepared to reformulate it if necessary. An interesting problem for future research is to construct a program that attempts to induce the model from a sketch.

For the Blocks World, the basic instruction to the one-armed robot is (PUTON X Y). The Pre Model is:

(CLEARTOP A) ;A has a cleartop, in order to be picked up.

(ON A ?OLD POSITION)

(SPACE-FOR A B) ;The top of B has room for A.

The Post Model asserts:

(NOT (ON A ?OLD POSITION))

(ON A B)

(NOT (CLEARTOP B)).

For problem solvers like HACKER, BUILD and NOAH, the problem model is simply a conjunction of ON relationships. For example, (AND (ON A B) (ON B C)) is the model for a
tower of three blocks.

**The Plan Node of the ATN**

Method consists entirely in properly ordering and arranging the things to which we should pay attention.


Let us now consider how this first part of the planning process is represented in the SIMPLE ATN. The nodes of the taxonomy will be the states of the ATN. This results in a nondeterministic transition graph. To direct computation, the ATN allows arc ordering and arc predicates. Arc ordering specifies that the arcs exiting from each node are attempted in a certain order. The default ordering from a given node is clockwise, beginning at the entrance point of the incoming arc. This ordering embodies judgments about the relative simplicity and probability of success of alternative methods.

With this ordering, the ATN can support a backtracking algorithm. Transitions are attempted depth first, with alternatives stored on a backup list. Failure occurs if some plan leads to a subgoal that cannot be solved. Ultimately, all plans of which SIMPLE is capable will be tried in this mode. Of course, an exhaustive backtracking search is neither a practical automatic planning technique nor an accurate model of human planning. To decrease aimless searching, predicates are added to the transition arcs to provide further constraints. Also, a lookahead ability can be incorporated. This is discussed in section 3.

The arc predicates make decisions on the legality of a transition, based on the contents of a small number of registers. The registers are used to describe the semantics of the program being constructed, including the sub-problem being worked on (Model), and the proposed solution (Solution). To actually produce the solution or modify the problem model in accordance with a reformulation or nonlinearity, the ATN also allows the registers to be set.

Figure 3 illustrates the initial part of the SIMPLE ATN. The arcs are labelled by small letters
FIGURE 3
THE PLAN NODE OF THE SIMPLE ATN
Structured Planning to facilitate discussion. Arc a begins the planning process by setting \( M \) to the formal model description of the problem. For example, to draw figure 2, \( M \) would be set to the wishingwell model. Arcs b, c and d are the possible transitions from the planning state. They are numbered, resulting in IDENTIFICATION being attempted before DECOMPOSITION or REFORMULATION. This reflects the heuristic judgment that it is preferable to check if the answer is already known before attempting to decompose a problem into subgoals or reformulate the problem description.

Arc b from PLAN to IDENTIFY has an arc constraint. Identification is pursued only if the model \( M \) is in the Answer Library. If it is, then SIMPLE executes arc e. Here the Solution variable is set with the answer found in the library. The POP causes SIMPLE to return with the answer. There are no arc predicates on c or d because DECOMPOSE and REFORMULATE are prepared to analyze any model.

**Problem Decomposition**

Divide each problem that you examine into as many parts as you can and as you need to solve them more easily.

Descartes, Œuvres, vol. VI, p. 18; Discours de la Methode, Part II.

This rule of Descartes is of little use as long as the art of dividing ... remains unexplained .... By dividing his problem into unsuitable parts. the unexperienced problem-solver may increase his difficulty.


From [Polya, 1965, p. 129].

Structured Planning addresses Leibnitz's criticism by developing more precisely the nature of decomposition techniques. The taxonomy identifies two important methods: the first is appropriate for a goal described as a *conjunction*. The second involves constructing a subgoal that, by means of *repetition*, can serve as the solution to the entire problem.

The decision to pursue CONJUNCTION versus REPETITION is based on the form of the model. For our purposes in this essay, a model is restricted to be *explicit* or *generic*. The former has an
explicit list of parts. Wishingwell is an example of such a model. The latter uses quantification to describe the overall model in terms of a generic part. EQUITRI1 and EQUITRI2 given below are two equivalent models for an equilateral triangle. The first is explicit while the second is generic.

MODEL EQUITRI1
1 PARTS S1 S2 S3 R1 R2 R3
2 LINE S1; LINE S2; LINE S3
3 ANGLE R1, ANGLE R2, ANGLE R3
4 S1 = S2 = S3
5 R1 = R2 = R3
6 CONNECTED S1 S2
7 CONNECTED S2 S3
8 CONNECTED S3 S1
END

MODEL EQUITRI2
1 PARTS (S 3) (R 3)
2 FOR-EACH I, LINE S(I)
3 FOR-EACH I, ANGLE R(I)
4 FOR-EACH I, I=1,3, S(I) = S(I+1, MOD 3)
5 FOR-EACH I, R(I) = 120
6 FOR-EACH I, CONNECTED S(I) S(I+1, MOD 3)
END

In the general case, models can be arbitrary logical expressions with mixed existential and universal quantification. The elementary Blocks and Logo Worlds we are considering do not require this complexity. A direction for future research is to extend the ATN to handle more complex problem descriptions.

Examine the DECOMPOSE node of figure 4. The transition to CONJUNCTION is made only if the problem is described by a model with explicit parts such as EQUITRI1 or WISHINGWELL. If the model is constructed from a generic description as in EQUITRI2, then REPETITION is selected. Thus, in terms of the arc predicates, the alternatives of the DECOMPOSE node are mutually exclusive. It is possible that a REPETITION plan, for example, might eventually be produced for a problem initially described by an explicit model. However, this can occur only through an intermediate reformulation in which EQUIVALENCE converts the original model to generic form. This allows a recursive call to the ATN in which DECOMPOSITION will choose REPETITION.

**Decomposition by Conjunction**

For conjunctive plans, the issue is whether the conjuncts are independent, or, alternatively, whether notice must be taken of interactions. For example, a linear plan for the wishingwell of figure 2 would solve for the three sub-pictures -- ROOF, POLE and WELL -- as separate
FIGURE 4
THE DECOMPOSE NODE OF THE SIMPLE ATN
subprocedures, each constructed independent of context. As a result, the ROOF and WELL would not be started and stopped from the middle of a side to facilitate connection with the POLE. Such an optimization requires knowledge of context. A nonlinear plan, on the other hand, could take account of the interaction between the POLE and other parts and suitably modify the subprocedures for the parts.

Let us be more precise in our classification of nonlinearities. The goal is to construct a procedure to accomplish a conjunction. Nonlinearities in decomposition add constraints to the design of the subprocedures. Nonlinearities in composition -- i.e., in putting the parts back together -- add constraints to the design of the superprocedure.

For the wishingwell example, adding the constraint to the design of the subprocedures for the ROOF and WELL that they start in the middle of a side is an example of a nonlinear decomposition. Another example for the Logo world that involves more than optimization occurs for problems which specify that one object X is to be INSIDE another Y. Y must be larger than X, if the required topological relation is to hold. This means that a linear decomposition that ignores the INSIDE relation and draws Y to be some default size can fail. The correct approach is to add a SIZE property to the descriptions of both X and Y.

A nonlinear composition adds constraints to the design of the superprocedure. For the Blocks World, the most common form of this nonlinearity is the existence of a partial ordering on the sequence in which the subgoals can be achieved. This arises from the use of some temporary resource, such as space, by one subgoal that is eventually used in a conflicting way by another. An example discussed by Sussman [1973] and Sacerdoti [1975] is the construction of a tower of three blocks, i.e. (AND (ON A B) (ON B C)). The tower must be built from the bottom up if the subgoals are not to conflict. The constraint (BEFORE(ON BC) (ON AB)) must be added to the design of the superprocedure.
The same kind of nonlinearity can arise in a Logo animation. To create a "snapshot" of some picture which can be displayed anywhere on the screen, the picture must first be drawn and "photographed". This process, called "snapping", involves first drawing the picture, snapping it, and then erasing it. Now the erasure is of an entire screen region. Hence, no other shapes should be present. If another shape is present, then it will be destroyed. Thus, a constraint must be asserted that requires that the snapping subgoal be achieved before any subgoals that draw a permanent shape in the critical screen region.

Nonlinear decomposition and composition constraints are not mutually exclusive. A given problem can exhibit both kinds of interactions. In the next section, we take account of this by including a cycle in the ATN that progressively linearizes each interaction detected in the model.

The ATN Subgraph for Conjunction

Figure 5 shows the subgraph of the SIMPLE ATN for conjunction. Arc b from CONJUNCTION to LINEAR decomposes the model into submodels that will be solved for independently by recursive calls to SIMPLE. The set \( \{ M_i \} \) consists of a conjunction of the properties describing each object in the model. Accomplishing these \( M_i \) corresponds to creating objects that satisfy these submodels. We call this subset of \( M \) the mainsteps. The second set consists of conjunctions \( M_{ij} \) of the relations between each pair of objects. These are the interface steps: their solution brings the objects \( X_i \) into the proper relationship with each other. We call this division into mainsteps and interface steps a linear decomposition and define it formally as:

\[
\text{if } M = \bigwedge_k P_k(X_i) \land R_i(X_i,X_j), \text{ then } M_i = \bigwedge_k P_k(X_i) \text{ and } M_{ij} = \bigwedge_i R_i(X_i,X_j).
\]

A linear decomposition of the wishingwell is:

\[
M1: \text{TRIANGLE ROOF; HORIZONTAL (BOTTOM (SIDE ROOF))}
\]

\[
M2: \text{LINE POLE;}
\]
\[ M_i + \sum_k^P_k(X_i) \]
\[ M_{i,j} + \sum_k^R_k(X_iX_j) \]

**CONJUNCTION**

**NONLINEAR DECOMPOSITION**

\[ M + M + P_e(X_i) + P_m(X_j) \]

**NONLINEAR COMPOSITION**

**ADVICE + ADVICE + CONSTRAINT**

---

**FIGURE 5**

THE CONJUNCTION NODE OF THE SIMPLE ATN
A linear decomposition is valid if the following solution process by sequential refinement possible:

1) organize the mainsteps into a sequential procedure, choosing an ordering that satisfies any linearization advice.

2) solve for the mainsteps independently.

3) solve for the interfaces in the order in which they occur in the procedure.

Implicit interactions can make this linear decomposition invalid.

The ATN attempts to linearize the model by checking for known types of interactions. The nonlinear decomposition node adds properties to the descriptions of individual subgoals that take account of interactions. The nonlinear composition node sets an advice variable that is accessed by the SEQ operator (arc b of figure 6) in constructing the superprocedure.

The linearization cycle consists of arcs c,f and arcs d.g. &NLD is a conjunction of predicates, each of which checks the model for a particular relation or pattern of relations that have nonlinear consequences for the decomposition. If any of these predicates detect their kind of interaction, properties are added to the description of individual objects that explicitly account for the dependency. The objective is that with these additional properties an independent treatment of the objects involved will be successful.

For example, as we discussed above, INSIDE is a relation in the turtle world that has
consequences for the properties of the objects involved. Thus, NLD-INSIDE checks for the existence of (INSIDE X Y) in the model. If found, SIZE properties describing x and y are added to the properties of these objects. The result is that an independent solution for x and y will not prevent the INSIDE relation from being accomplished.

NLC checks for patterns in the model that have consequences for the eventual composition of the subgoals. If such properties are detected, then explicit relations are added to take account of the interactions. An example is NLC-ANIMATION that checks for a Logo animation that creates snapshots and shows them. If detected, (BEFORE SNAP DISPLAY) is added to ADVICE. Similarly, for the blocks world, NLC-TOWER adds (BEFORE (ON B C) (ON A B)) to ADVICE.

The NLC and NLD constraints arise from two sources. The first is that they may be supplied by the creator of the SIMPLE ATN. Alternatively, following Sussman [1973], SIMPLE can be designed to summarize bugs by classifying the nature of the nonlinearity and adding it to the NLC and NLD constraints. In these terms, the acquisition of skill is, at least partly, the growth of more elaborate recognition routines for implicit interactions. Sussman called this process the compilation of critics. The theoretical progress of the Structured Planning research is to make clear that these critics are simply additional arc constraints in the planning transition graph. They are not different in kind from any other planning constraints.

To summarize, processing implicit dependencies is handled by the ATN through a linearization loop. If the problem is identified as involving some kind of nonlinearity, then the model or advice registers are modified to make the interaction explicit. Processing then returns to the CONJUNCTION node. Further processing of interactions occurs, until no more are detected. Control then passes to the LINEAR node for actual decomposition. If an undiscovered interaction still exists, subsequent debugging will be necessary. This is discussed extensively in sections 5 and 6.
Composition by Sequential Refinement

Once the nonlinearity loop is complete, SIMPLE is ready to solve the individual subgoals and compose a complete solution. In this section, we shall discuss a composition technique we term Sequential Refinement. A generalization of this approach, Net Refinement, based on the procedural net representation for programs, is discussed in section 3.

Figure 6 illustrates the ATN graph for the Sequential Refinement cycle. The basic process is cycling through the subgoals identified by the Linear Decomposition and solving for each by recursive application of the ATN. Arc b enters the Sequential Refinement loop. The solution register S is set to a sequential superprocedure for the mainsteps Mi and interface steps Mij identified by the decomposition. The SEQ operator on the arc chooses an order for this superprocedure that is consistent with any ADVICE recorded by the linearization loop. SEQ can also bring various criteria to bear on the organization of the superprocedure, such as choosing an order that mirrors chains of predicates in the model such as X connected to Y connected to Z. This often simplifies interfacing.

As an example, for the wishingwell problem, given the Mi and Mij specified above, a plausible sequence would be:

```
TO WW
10 ROOF
20 POLE
30 WELL
END
```

The ORDER operator on arc b of figure 6 chooses the sequence in which the sub-problems are solved. This may not be, indeed, probably is not, identical to the order of occurrence of the sub-problems in S. A criterion for the order of solution, for example, is to solve for the mainsteps before the interfaces. Another criterion is to order the mainsteps with respect to their complexity. Lookahead (section 3) can estimate this. For the wishingwell, it makes sense to solve for the POLE
FIGURE 6
SEQUENTIAL REFINEMENT
first since Lookahead can identify this as a primitive. Criteria for ordering the relations can exist as well, although the default ordering is usually simply their order of occurrence in the procedure.

Arc c is a cycle that recursively solves for the subgoals in the order selected by ORDER. The solution to each subgoal is attached to S at the subgoal's node. The solved subgoal is then deleted from \( G \). When all subgoals have been solved, the cycle is exited via arc d. The ATN pops, returning the solution.

For the Logo world, the initial Answer Library contains both primitives and their associated models as well as schemata for accomplishing particular model relations. Thus, if the subproblem is to achieve \((\text{ABOVE } X \ Y)\), where X and Y are mainsteps that have already been solved, then the answer library will contain procedural knowledge that designs an interface relative to the adjacent mainsteps that satisfies the relation. The nature of these imperative definitions is discussed in [Goldstein, 1974, Appendix D]. We do not give details here.

For the wishingwell, the mainsteps for the ROOF, POLE and WELL would be solved first. Then, if the default order for relations is pursued, first the interface between ROOF and POLE and then between POLE and WELL would be constructed. Figure 7 shows the analysis generated by the ATN and illustrates the solution produced.

The hierarchical commentary is a trace left by the ATN of the states passed through in generating the program. In this parse tree, each node has a copy of the values assigned to the registers at the time the node was generated. This serves as a description of the purposes of the code in the form of the MODEL assignments, ADVICE for future modifications and CAVEATS regarding possible bugs. Caveats are generated by the planner when making possibly erroneous heuristic decisions and are discussed in section 5 on debugging. The parse tree for the mainsteps of the wishingwell procedure is illustrated in figure 8.
\textit{STRUCTURED PLANNING}

\textbf{Plan (M → WW)}

\textbf{DECOMPOSE}

\textbf{CONJUNCTION}

\textbf{LINEAR}

\textbf{SEQ*}

\textbf{POP(S)}

\textbf{WHERE S IS:}

\textbf{TO WW}

\begin{align*}
10 & \text{ M}_1 + \text{TRI} + \text{ROOF} \\
20 & \text{ M}_{1,2} + \text{BELOW, CONNECTED} \\
30 & \text{ M}_2 + \text{LINE} + \text{POLE} \\
40 & \text{ M}_{2,3} + \text{BELOW, CONNECTED} \\
50 & \text{ M}_3 + \text{SQ} + \text{WELL}
\end{align*}

\textbf{FIGURE 7}

\textbf{SOLVING THE WISHINGWELL PROBLEM}
FIGURE 8
HIERARCHICAL ANNOTATION
The SIMPLE ATN

At this point, we have described the basic planning strategies for Conjunction and Identification, and embodied them in an ATN. The conversion of the taxonomy to procedural form proceeded in five steps. The first was to introduce registers to carry the semantics of the problem solving process. The second was to order the decisions at each node, thereby defining a backtracking problem solver. The third was to supplement this ordering with predicates on the transitions, making the choices more sensitive to the problem context. The fourth was to reconsider the nature of nonlinearity, and, as a result, introduce a Linearization Cycle. Finally, the design of SIMPLE was completed by adding a Sequential Refinement loop that recursively solves for subgoals. The end result is an Augmented Transition Network. Figure 9 provides a global view of the ATN, showing the connections between the various planning states.

Decomposition by Repetition

We conclude this section with a brief discussion of other planning techniques mentioned in the taxonomy but not discussed. Repetition plans embody the problem solving method of structuring the solution in terms of either the same goal applied to simpler arguments (recursive plans) or another simpler goal repeated some number of times (round plans). The former technique is more powerful than the latter in the sense that every round plan can be accomplished by means of a recursion, while every recursion cannot be accomplished by iteration [Hewitt, 1970]. But we have included explicit mention of Round Plans because they are triggered by a different problem formulation than Recursive Plans. In the former case, the problem \( P \) is described as \( n \) repetitions of problem \( Q \), where \( Q \cap P \), while in the latter \( P \) is described in terms of repeated occurrences of problem \( Q \cup P \).

Round plans are the natural planning technique for generic models. We intend to handle this in the ATN via an arc operator ROUND that formulates a submodel for the generic part and advice
FIGURE 9
THE SIMPLE PLANNING ATN
for the composition requesting an iterative control structure. Having decomposed the problem in this fashion, control is passed to the Sequential Refinement Loop. Figure 10 illustrates this subgraph.

![Diagram](image)

**FIGURE 10**
**ROUND PLANS**

EQUITRI2 was an example of a generic model. The ROUND operator would isolate subgoals for accomplishing a SIDE and a ROTATION. The repetition advice would be for three iterations. The result would be a program of the following form:

```
TO TRI
10 REPEAT 3 20, 30
20 FORWARD 100
30 RIGHT 120
END
```

**Problem Reformulation**

When a problem arises, we should be able to see soon whether it will be profitable to examine some other problems first, and which others, and in which order.


Finally, there are the problem reformulation techniques. (See [Polya, 1965, Ch. 9] for a relevant discussion of Problems within Problems.) The importance of these methods can be understood by recognizing that all of the problem solving strategies mentioned above are triggered by pattern matching against the description of the problem. The reformulation techniques,
however, are prepared to generate an altered problem description. In the SIMPLE Planner, these reformulation techniques are applied if solution by identification or decomposition fails. Their action is to reformulate the problem description, and then to pass the new description back to the Planner.

The taxonomy cites two reformulation techniques. The first, attempts to find an equivalent problem that will be easier to solve, and whose solution will satisfy the original task. The second searches for a simplification that can be used as a stepping stone to solving the original problem.

The difficulty here is recognizing which reformulation will aid the solution progress. Currently, for equivalency, SIMPLE is capable of reformulations that move between predicate calculus descriptions given in terms of multiple objects to equivalent descriptions in terms of a single generic object, thus transitioning from a Conjunctive decomposition to a Repetition decomposition, or vice versa. An example is moving between the EQUITRI1 and EQUITRI2 triangle models. Another reformulation technique involves regrouping the parts. Figure 11 illustrates a regrouping of the wishingwell. The virtue of regrouping is that it can produce a model whose parts are in the Answer Library.

For Simplification plans, we have analyzed elementary techniques based on generalization, specialization, and analogy. Specialization involves instantiating variables in a model by specific constants, restricting their range, or adding additional model predicates to constrain the problem. Generalization would include the opposite processes. Analogy often amounts to first generalizing and then specializing. Thus, for the Logo world, if the original model is for a triangle with sides of a certain size, generalization would produce a model for a triangle of arbitrary size. Analogy might then respecialize to a square, perhaps, or a triangle of another size. The virtue of these reformulation techniques is reaching a problem description whose solution is known. We envision that each technique has associated with it an inverse mapping on the solution found so that that
FIGURE 11
REFORMULATING THE WISHINGWELL IN TERMS OF A TREE

A solution can be mapped back to a solution for the original problem.
3. Searching for the Plan

If you see several plans, none of them too sure, if there are several roads diverging from the point where you are, explore a bit of each road before you venture too far along any one -- any one could lead you to a dead end.


There exist both explicit and implicit choice points in the Planner. By this we mean non-exclusive alternatives to accomplishing the current goal. An example of an explicit choice is the decision between decomposition and reformulation for a given problem. Implicit choices occur in identifying past solutions (there may be more than one); creating super-procedures (there may be more than one reasonable sequence); and, in general, whenever knowledge on the transition arcs sets registers or makes decisions. We have discussed ordering and predicates associated with the transition arcs to direct explicit choices in the planning ATN. For each implicit decision, a similar local approach is taken. The decision process locally determines the order of the alternatives, pursues the first, and pushes the remainder onto a failure stack. Thus the overall planning process is a depth first search.

In this section, we briefly consider additional techniques germane to resolving planning decisions. We do not develop them in detail, however, in this paper. Possible improvements to make the planning process more directed and less susceptible to blind search are: (a) lookahead, (b) least commitment, (c) differential diagnosis, and (d) lemma libraries.

**Lookahead**

In general, lookahead consists of a limited search of available alternatives, with associated plausibility criteria for judging the probable success of a given non-terminal state. An elementary but useful form of lookahead can be accomplished by the SIMPLE planner by pushing the analysis forward some fixed number of recursive levels, looking to see if a solution arises from an Identification. Thus, a decomposition that can solve most of its subgoals in terms of the answer
library is preferred to a decomposition that must recursively apply decomposition techniques to its subgoals. In effect, such lookahead attempts to select those plans that accomplish the goal with a minimum number of recursive calls to the problem solver.

For example, consider again the Wishingwell scenario. Suppose the answer library contains not a TRIANGLE program but a TREE procedure. Lookahead prevents the planner from blindly pursuing the decomposition in terms of ROOF, POLE, and WELL over a reformulation that describes the wishingwell as a TREE and a SQUARE (figure 10). This is accomplished by lookahead indicating that the reformulation produces a problem description whose decomposition can be partly solved by means of the answer library, while the standard decomposition results in two subgoals (the ROOF and the WELL) that require further analysis.

Lookahead can be implemented in the usual fashion. A static plausibility function assigns a plausibility of one to problems that can be solved via identification, and zero to problems that require decomposition or reformulation. Lookahead pushes the analysis through a fixed level of recursion, and then computes the plausibility as the sum of the plausibilities of the subgoals appearing as the tips of the problem tree divided by the number of these subgoals. The division serves the purpose of resolving the following situation: given two situations in which the same number of subgoals are known, the problem tree with fewer unsolved subgoals is preferred.

An improved algorithm for lookahead might give weight to those plans that involved identifications involving the more complex subgoals. This could be approximated by the heuristic of measuring the complexity of the model in terms of such syntactic criteria as the number of predicates involved.

Least Commitment

This is the problem solving technique of avoiding premature decisions. It is elegantly developed by Sacerdoti [1975] in the form of procedural nets. Sacerdoti observes that some bugs in
planning can arise from premature commitment to a particular sequence, when the problem does not in fact require such a determination. His solution is to represent the program, not in the usual sequential format, but as a net.

Figure 12 illustrates a net for building a tower from three blocks. Sacerdoti’s planning proceeds by successively expanding goals, committing the system to a sequence only when a conflict in ordering arises. In levels 1, 2 and 3, no order has been chosen for the sequence of accomplishing (ON A B) and (ON B C). It is not until level 3, after criticism, that the problem solver commits itself to an order for placing the blocks.

This technique can be incorporated in SIMPLE by replacing the Sequential Refinement loop with a Net Refinement cycle. Figure 13 illustrates this replacement. Instead of SEQ organizing the subgoals into a typical procedure, NET organizes the subgoals into a network.

We propose a generalized form of Sacerdoti’s approach. We represent in the net not only the main subgoals as alternative branches unless ordering is required, but also the relations between these goals. Figure 14 shows the general form of a Structured Planning net.

The planner solves for subgoals, following the Procedural Net technique of expanding nodes. But eventually the planner also solves for relations. When all subgoals, both to construct individual objects and to satisfy their relations are satisfied, then the result is an executable net, i.e. any branching left in the net can be executed in arbitrary order. Specifically, figure 13 describes the following process. The operator NET on arc b sets the solution variable S, not to a superprocedure, but to a net of the form given in figure 14. \( G \) is the set of subgoals ordered in the same fashion as was described for Sequential Refinement.

Arc c recursively calls the planner to solve for a subgoal. If the subgoal is a mainstep, it is simply spliced into the net as a refinement. But if the subgoal is a set of relations, then the solution may involve establishing a specific interface. If so, a sequence is enforced on the
FIGURE 12
SUCCESSIVE REFINEMENT OF A PROCEDURAL NET FOR BUILDING A TOWER
FROM [SACERDOTI, 1975, 211-212]
FIGURE 13
NET REFINEMENT

S- CRITICIZE(S)

{G} +{G}- G_k

NODE (G_k) +PLAN(M-G_k)

c

{G_k} = φ
e

POP S

S- NET{Mi,Mij}

{G_k} +ORDER {Mi,Mij}
b

a

LINEAR

CRITICIZE

searching for the plan
FIGURE 14
GENERALIZED PROCEDURAL NET

FIGURE 15
SOLVING FOR RELATIONS
mainsteps adjacent to this interface. The result would be that, in figure 15, A is transformed to figure B. If there are no relations between two mainsteps that require interfacing, then no ordering will be imposed and the net will preserve its branching. The result is executable under the interpretation that parallel branches can be executed in any order. If there are a great many relations, then the net will ultimately reach its most constrained form -- a sequence.

Following Sacerdoti, Arc d criticizes the network for interactions that have become apparent after expansion. A typical example is noticing that the prerequisites of one subgoal are clobbering a brother goal. For the blocks world, this might involve observing by means of a table of multiple effects that the prerequisites of one goal are clearing a block that was placed there by another. We shall not go into detail regarding these critics. The interested reader should see Sacerdoti [1975]. However, it is worth noting that if the original linearization was completely successful, criticism will find no hidden interactions. But it is a useful heuristic check on the decomposition to include this criticism process.

There are subtleties in handling relations between non-adjacent mainsteps. For such cases, a relation such as $R(X_1, X_3)$ might have to be replaced by an equivalent description in terms of objects accomplished by adjacent mainsteps, say $(\text{AND } R_1(X_1, X_2) \quad R_2(X_2, X_3))$. We shall not discuss this further here. Our purpose here is only to indicate the direction research would take to link our ATN representation for planning to Sacerdoti's procedural net representation for programs.

SIMPLE represents an extension of NOAH, Sacerdoti's program for refining procedural nets, in that NOAH’s only planning technique is successive goal expansion. This is equivalent to SIMPLE’s Decomposition by Conjunction. But SIMPLE also represents a variety of other planning strategies, including the major categories of Identification and Reformulation. NOAH improves the representation of the procedures produced (by using nets), but does not emphasize SIMPLE’S
Searching for the Plan

central concern of how this goal structure is arrived at. Hence NOAH makes an important
collection, for the fashion in which it captures this principle of least commitment, but it is not a
total theory of program composition.

Differential Diagnosis

Critics are an example of differential diagnosis. A critic analyzes the problem description,
and advises SIMPLE as to which transitions are permissible and which are prohibited. A Block's
World example is HACKER's critic, attached to SIMPLE's Conjunction node, that diagnoses an (AND
(ON X Y) (ON Y Z)) problem as involving non-linear relations between the subgoals.

Lemma Libraries

If a sub-problem is successfully solved, it is added to the Answer Library, even if the overall
approach fails. This allows SIMPLE to avoid repeated attempts to solve the same subgoals. This
corresponds to the well-formed substring table utilized by Woods (1972) in applying ATN's to
natural language parsing.
4. Limitations and Extensions of Structured Planning

My mind was struck by a flash of lightning in which its desire was fulfilled.
Dante, Paradiso, Canto XXXIII in [Polya, 1965, p. 54]

In concluding our discussion of the planning ATN, it is appropriate to examine the limitations of this approach. Needless to say, there are many aspects of human problem solving and its flashes of lightning that we have not touched upon. What follows is some of the specific limitations that we perceive in the SIMPLE theory and possible extensions to remedy them.

In the previous section, we discussed how the generation algorithm running on the ATN could be improved. These improvements could obtain better performance within the boundaries implied by the knowledge present in the network. They do not address those limitations implied by the particular subset of planning knowledge present, i.e. the basic taxonomy.

If we realize that our problem descriptions are composed of logical operators, then we see that the network currently contains techniques for solving for conjunctions and universal quantification over a finite domain (repetition). The network, however, does not contain strategies for handling disjunction, negation, and existential quantification. These clearly could be incorporated using the ATN formalism, but we not address the first two in this paper. The last, disjunction, is briefly discussed in section 6.

With techniques for all of the logical operators, the planner would still not be complete. Even if a problem is described as a conjunction, the planner may not find the constructive solution necessary to accomplish the conjuncts. Interactions can exist that make it impossible, or the particular technique for resolving a certain interaction may be unknown. Nevertheless, we believe that the logistic framework for describing problems at least gives a super-structure on which to build more elaborate planning techniques. The success of this super-structure can be evaluated by the extent to which future research allows the collection of decomposition techniques to be
extended within the ATN framework.

Another limitation of the ATN is in the *ad hoc* nature of the reformulation techniques. Theoretically, a general theorem prover could recursively enumerate all equivalent models. But, such a strategy is computationally too costly to be useful. Instead, we enumerated a small number of heuristics. Future research might attempt to find a happy medium between general deductive strategies and specific procedural heuristics. Such an accommodation is suggested by recent the work on theorem proving [Kowalski, 1979; Moore, 1975].

In this paper, we have emphasized a hierarchical approach to planning. Such a philosophy is incomplete in that it does not take account of possible heterarchy [Minsky & Papert, 1974]. By this we mean that in some planning situations, a person clearly takes advantage of bottom up evidence to guide an ordinarily top-down analysis. Heterarchy refers to the fact that information and decisions do not inevitably flow in a single direction. A robot that trips over a bag of money on its way to rob a bank should not kick the money aside and continue with the caper. Figure 16 illustrates a Heterarchical Refinement loop in which goals can be reordered after each recursive solution of a subgoal. Eventually this complexity must be addressed. However, our research plan is first to construct and experiment with a clearly structured top-down planner, in order to understand its virtues and limitations.

In the remainder of this paper, we consider the rational bugs that can arise in SIMPLE'S planning and how they can be diagnosed and repaired.
FIGURE 16
HETERARCHICAL REFINEMENT
Let us focus on one particular component of [general heuristic knowledge]: the art and techniques of ... debugging. The school experience is dominated by the normative attitude implied by "right answer vs. wrong answer". The mathematician's experience of mathematics is dominated by the purposeful-constructive attitude implied by the struggle to "make it work". He abandons an idea not because it happened to go wrong, but because he has understood that it is unfixable. Dwelling on what went wrong becomes a source of power rather than a piece of masochism (as it would appear to most fifth graders in traditional math classes).


We agree with Papert in his assessment that debugging is an essential part of problem solving. A powerful debugging system frees the planner from the necessity of always producing entirely correct plans. More precisely, bugs arise from heuristic choices made in constructing the plan. From the structured planning standpoint, such heuristics are embedded in the default ordering of transition arcs. In the absence of specific arc constraints, the planner will prefer linear to non-linear plans, round repetitions to recursion. Such heuristics can lead to bugs. But they also provide crucial advantages to the planner: (a) these heuristics allow the planner to attempt new problem types with which it has had no experience, (b) they are often successful, and (c) in those cases where an error arises, the nature of the difficulty can serve as a specific diagnostic regarding the locus of the incorrect decision and the alternative choice required. Indeed, should subsequent experience lead to bugs, the debugger can abstract the problem description, embed it in a critic at the point in the planning ATN where the incorrect choice was made, and thereby prevent future occurrences of the same error. We shall call the class of bugs that arise from reasonable heuristic judgments made in planning rational bugs. In this section, we show how this class of difficulties can be perspicuously explained with reference to the SIMPLE Planning theory. We introduce strategies for Structured Debugging, i.e. techniques for diagnosis and repair of rational bugs based on identifying incorrect or incomplete plans.
The critical problem in debugging rational errors is diagnosing the underlying cause of the bug. The bug is manifest if the program produced by a plan fails to satisfy the model which the plan was intended to accomplish. Following [Goldstein 1974], we shall call the set of unsatisfied predicates *violations*. Diagnosis consists of deciding in what fashion the plan is incomplete or incorrect.

We shall discuss three Structured Debugging diagnostic techniques: model, process, and plan diagnosis. A diagnostic technique we shall not discuss that is useful in analyzing human code, but not especially appropriate for programs written by machine, is code-diagnosis. This amounts to having a list of rational form criteria, and examining the code to find if any are violated [Goldstein, 1974, pp. 137-138]. The SIMPLE planner does not make this kind of mistake.

**Model Diagnosis**

This is the basic diagnostic technique, in that it involves the determination of whether the program has succeeded or failed in accomplishing its intended model. In logistic terms, it amounts to a verification in which the model predicates are applied to the structures -- pictures or block arrangements -- produced by the program.

The violated model predicates can suggest whether the underlying bug is an incomplete plan by determining whether any code is intended to accomplish the predicate or its prerequisites. If the plan is incomplete, then debugging can be accomplished by invoking the planner to supply the code. Alternatively, if the plan indicates that code was created to accomplish the predicate, then further diagnosis is necessary. Perhaps there are unexpected interactions. Process diagnosis is the next stage.

**Process Diagnosis**

An examination of the state of the process at the time that a bug is manifest is often helpful in diagnosing unexpected interactions or unsuccessful performance. This is the diagnostic
technique used by HACKER. Conflicts between goals are diagnosed as non-linearities and reflect the underlying bug of having applied an incorrect plan. The essential ingredient is observing that one goal has violated a model predicate describing the intended effects of a prior step. The HACKER bugs of Prerequisite Clobbers Brother Goal, Strategy Clobbers Brother, and Prerequisite Conflicts with Brother are all of this type.

Sussman develops elaborate process state patterns for classifying kinds of interactions which we shall not repeat here. The essential ingredient is observing that a model predicate is being undone within a scope during which it is expected to be true. A predecessor of this diagnosis technique can be found in the PLANEX capability of the STRIPS problem solver [Fikes 1972]. In executing a plan, PLANEX checked for model predicates being accidentally undone. The generalization of HACKER is to check for situations in which they are intentionally undone, i.e. the plan itself is flawed.

Process diagnosis may fail because the subgoal interaction is too complex for the debugger to recognize. Plan diagnosis is available to aid in isolating the culpable code.

**Plan Diagnosis**

Plan Diagnosis is based on the fact that the planner has knowledge of various heuristic decisions it has made which may prove unsuccessful. For example, SIMPLE knows that it may have decomposed a model linearly without any actual proof that no interactions existed. Obviously such a simplification may prove incorrect. Such choices are recorded during the planning phase as possible sources of bugs.

For example, consider the task of drawing a face on the basis of the following model.
In the absence of specific critics (i.e., before SIMPLE has learned NLD-inside), the planner will design the eyes and the head independently. But if the head and eyes are all circles of the same default size, then satisfying the relation of the eyes being inside the head will be impossible. A linear plan that solves for the main steps independently of the relations will lead to a bug. The relation INSIDE has consequences for the properties, especially the size, of its arguments.

Debugging this difficulty is aided by plan diagnosis which notices the linear treatment of the subgoals for eyes and head in the plan. In the absence of other guidance, this signals a potential bug. A sufficiently detailed examination, in this case of the implications of relations like INSIDE, could then show a non-linearity with respect to the size property as the cause of the problem.

Naturally we do not advocate that the planner continue to make such mistakes in the future. Improvement can be obtained by recording as a critic that INSIDE implies a non-linear decomposition. Thus, in subsequent problems, if two parts are described by this relation, non-linear planning will be chosen immediately. In particular, the model will be reformulated with size properties imposed on the parts so that in terms of this description modular decomposition is possible.

Debugging caveats are also generated by the use of heuristics for reformulating the problem. For example, the planner may construct what it believes is an equivalent problem statement, but not in fact prove the equivalency. Such a heuristic approach, though usually successful, can be the cause of problems. Hence, this too is marked in the plan and noticed during debugging by plan
diagnosis.

Recall that each node of the parse tree for a plan contains semantic variables for the model, code and other concepts. A CAVEAT variable is used to record advice from the planner regarding possible bugs. This variable, associated with the equivalency node in the parse, would record that a model reformulation had been done heuristically and not proven equivalent.

The action to be taken given such advice is to call upon techniques such as careful proof of equivalency to see if the heuristic involved in fact led to an incorrect plan. Some critics may involve such costly processing of the problem model that, even though already learned from prior encounters, they were not applied in first planning for the problem. If plan diagnosis points to a possible error, these critics can still be applied.

**Debugging in Context**

Before concluding this section, it is appropriate to mention other kinds of bugs that arise in human programming, but do not fall under the heading of rational planning errors. These range from execution errors to the construction of irrational plans. Execution bugs consist of those errors due to mistypings, misspellings, incorrect programming language syntax, noise on the computer line, and other such failures to successfully execute a statement of code. They are often diagnosed by the computer, simply as a result of the code being unrecognizable. Repair is accomplished by re-executing an edited line. The plan is not affected.

Irrational plans can be precisely defined with respect to SIMPLE. They correspond to making transitions that are not allowed in the planning network or failing to make transitions that are required. An example would be pursuing a repetition plan and failing to handle the terminal cases. SIMPLE, as a theory of rational planning, does not explain these kinds of errors, and we shall not discuss them further here.

It is sometimes argued by people in the Structured Programming movement that good
discipline in coding can eliminate all bugs. The Structured Planning-Debugging taxonomy of plans and bugs sheds some light on this issue. Rational bugs are not avoidable. They correspond to heuristic planning judgments made when no better criteria are available, as occurs when programmers are solving new problems. Indeed, it is through the experience of whether their default heuristics succeed or fail on a new class of problems that individuals learn and acquire skill. On the other hand, bugs due to ungrammatical plans must surely be increased by unstructured, careless programming. It is this class of errors, and not rational bugs, that we believe the structured programming movement has in mind, in calling for better and more disciplined coding. We shall develop some possible applications of Structural Planning to improving human programming in the conclusion.

Another source of dissatisfaction with programs (that we mention for completeness but will not discuss here) arises from efficiency considerations. The SIMPLE Planning ATN is not a compiler and does not attempt to optimize the programs which are produced. Structured debugging is restricted to correcting programs that fail, in some fashion, to achieve their model. Programs that are far from optimal, but nevertheless successful in terms of their models, are correct with respect to rational bugs. However, we believe an interesting question for future research will be to explore the extent to which a hierarchical plan associated with the code can aid a compiler in optimization.

In the next section, we develop further the Structured Debugging approach to categorizing, diagnosing, and repairing rational errors by analyzing the debugging behavior of the Blocks World problem solver HACKER.
6. A SIMPLE Description of HACKER

... the current bug classifier in HACKER is an *ad hoc* program and thus, the body of knowledge (called Types of Bugs in the overview flowchart) on which it operates is difficult to separate out and display. This, of course, makes Types of Bugs also very difficult to extend. The hope is, however, Types of Bugs is essentially independent of the problem domain and need only be expanded when new problem solving methods (the Programming Techniques Library) are introduced. An important area for development of HACKER-like problem solving methods would be the systematization of the knowledge in Types of Bugs in a more modular way.

[Sussman, 1973, pp. 103-104]

The HACKER program [Sussman 1973] represents an important landmark in AI theory for its emphasis on debugging as an important constituent of learning. However, HACKER is theoretically incomplete, for it fails to integrate debugging expertise with a theory of plans. The underlying bug types in HACKER appear as a miscellany of debugging knowledge with no underlying regularity. As a result, the classification algorithm that maps manifestations to causes is *ad hoc*. Structured Debugging extends the HACKER paradigm by developing debugging knowledge in the context of a structured planning theory. From this vantage point, underlying causes in bugs are seen as specific errors in plan synthesis. The types of causes follow from the obvious failing in interpreting an ATN: making an incorrect arc transition.

In this section, we analyze HACKER from the SIMPLE standpoint. Our goals will be to demonstrate how SIMPLE provides:

(1) greater theoretical clarity, by means of a unified planning and debugging theory;

(2) greater depth and breadth, by means of natural extensions to HACKER's set of bug types and debugging techniques suggested by SIMPLE's planning structure.

There are four bug types in HACKER: Prerequisite Missing (PM), Prerequisite Clobbers Brother Goal (PCBG), Prerequisite Conflicts with Brother (PCB), and Strategy Clobbers Brother (SCB). We analyze each of them in turn.
Bugs Arising from Incomplete Plans

From the Structured Planning standpoint, the HACKER bug type — Prerequisite Missing -- is an instance of incomplete planning. This kind of difficulty can arise when the accomplishment of a model predicate depends critically on the particular environment in which the procedure is executed. Sometimes this may be recognized and recorded in caveats associated with the plan which specify that the success of the plan depends on certain initial state conditions. The issue of dependency on the initial state was discussed in [Goldstein, 1974, pp. 85-88] in which ASSUMPTION commentary was used to record known dependencies between the program and its initial environment. For the Blocks world, [Sacerdoti 1975] uses phantom nodes to represent goals which are true in the initial state that would otherwise have to be accomplished. But a rational planner may not realize (or be prepared to take the extensive time necessary to deduce) all interactions between the model and every possible initial environment. For example, he may use a plan from the Answer Library because the problem matches the Post Model and not prove that all the statements in the Pre Model are true for all environments. Hence, the plan may be incomplete with respect to a new environment. Debugging consists of modifying or extending the plan to accomplish the new set of violated predicates.

Missing prerequisites are made manifest by primitives generating complaints. In the Blocks world, for example, the robot will complain if asked to move a block to a position that some other object already occupies, or to grasp a block whose top is cluttered. Analogous complaints are generated by Logo turtle primitives. Logo will complain if the turtle is asked to move off the screen or if a turtle command is executed prior to the turtle being created by a start-display function. How can we obtain a unified approach to these complaints generated by primitives and the broader class of model violations that refer to the general case of a program failing to accomplish its goals? An answer is suggested by the use of Pre Models, discussed earlier in section
2. If we associate a Pre model with each primitive, then unsatisfied prerequisites simply become model violations. For HACKER, the Pre model for the operator, "Move block A onto block B", would contain the assertions:

(CLEARTOP A) ; A has a cleartop, in order to be picked up.

(SPACE-FOR A B) ; The top of B has room for A.

The inclusion of unsatisfied prerequisite manifestations in the class of model violations, and the identification of Prerequisite Missing bugs with Incomplete Plans allows a unified use of diagnostic and repair techniques. Each model predicate, whether part of the operator model or the problem model, has procedural knowledge associated with it that aids in isolating the bug locus, proposing repairs, and thereby completing the plan.

**Bugs Arising from Incorrect Conjunction Plans**

PCBG and PCB both arise from a linear plan being applied to a non-linear problem. PCBG is the underlying problem in attempting to build towers, in HACKER terms (MAKE (AND (ON X Y) (ON Y Z))), from the top-down, i.e. it is a bug that arises in the situation where the planner ignores that one conjunct must precede another. From the Structured Planning standpoint, this corresponds to the planner making the wrong choice at the Conjunction node. SIMPLE chooses a linear plan, unless the Nonlinear Composition and Decomposition predicates recognize an interaction.

In these terms, it is clear how debugging is to be accomplished. The debugger simply re-applies the planner to the problem with the advice that a Linear plan is prohibited. Indeed, the pattern-matching of the state of the halted process against abstract bug patterns allows the debugger to recall the planner with the specific advice that an ordering between the conjuncts is required. For our example, this relation is that (ON X Y) must precede (ON Y Z).

PCB arises in the following problem: HACKER is asked to find space for both blocks A and B
GOAL: (AND ON A C) (ON B C))

**FIGURE 17**
**PREREQUISITE CONFLICTS WITH BROTHER**

on base block C, i.e. to accomplish figure 17-B. In attempting this problem linearly, HACKER first places A on the center of C (figure 17C), with no consideration of the brother goal of placing B on C. When the time comes to place B on C, there is insufficient room and block A must be pushed left (figure 17D). This results in a double move manifestation. HACKER's debugging strategy is to construct a plan that takes simultaneous account of the two pre-requisites: (PLACE-FOR A C) and (PLACE-FOR B C).

The Structured Planning approach to this HACKER problem is to have the debugging episode produce a nonlinear decomposition critic that recognizes that multiple SPACE-FOR predicates, i.e. (SPACE-FOR X Z), (SPACE-FOR Y Z), require the addition of location properties on the
placement of blocks X and Y. We are not claiming that this allows SIMPLE to do more than HACKER in fixing this difficulty. Rather we only claim that the planning taxonomy makes it easier to understand the issues involved. The classification of conjunctive non-linearities into nonlinearities in the solutions to the conjuncts and in the composition makes both PCBG and PCB understandable and, indeed, to be expected, given the default preference for Linear plans.

**Bugs Arising from Incorrect Disjunction Plans**

The bug Strategy Clobbers Brother arises when two strategies are attempting to accomplish a given goal, and they conflict with each other. The particular blocks world example discussed in HACKER involves the findspace strategy "remove block from surface" conflicting with its brother "compact by pushing to the left" (Figure 18). Removal can undo a prior compacting. HACKER notices the conflict and debugs by imposing an ordering on these strategies. Removing must be accomplished before compacting.

From the Structured Planning standpoint, this type of bug can be understood as arising from an incorrect disjunctive plan. The disjunction is in the set of alternative methods for accomplishing the FINDSPACE goal. Figure 19 illustrates a planning taxonomy for the decomposition of disjunctions.

```
|-- LINEAR
   |-- EXCLUSIVE
      |-- NONLINEAR

DECOMPOSE -- OR --
   |-- LINEAR
      |-- ADDITIVE
         |-- NONLINEAR
```

*Figure 19 -- A Planning Taxonomy for Disjunction*
GOAL: \((\text{AND} (\text{ON } \text{C } \text{A}) (\text{ON } \text{D } \text{A}))\)

Compacting: Blocks pushed to leftmost position

Removing: Blocks not required on A are removed

Top of A is compact

Conflict: Top of A no longer compact

Compacting then removing leads to conflict. The removing strategy has undone the compacting.

FIGURE 18
STRATEGY CLOBBERS BROTHER
The first major decision involves resolving whether the disjuncts are mutually exclusive or additive. Exclusive disjunction refers to a set of options in which only one can be chosen. Exclusive Disjuncts cannot "partially" succeed. Crossing the Atlantic by steamer or plane are mutually exclusive travel strategies. One does not travel half way by plane and then switch to ship.

Additive disjuncts can partially succeed and indeed may behave cooperatively. Strategies for finding space are of this kind. However, having decided that the disjuncts are cooperative still leaves open the question of whether they are linear, i.e., independent, or alternatively, whether there are possible interactions. We envision implementing this taxonomy in a similar fashion to conjunction with linearization cycles.

Relative to this taxonomy, Strategy Clobbers Brother is an incorrect plan bug, in which the planner has chosen to treat the subgoals as independent additive disjuncts, when in fact they are dependent and subject to a non-linear relation in terms of their order of application.

**HACKER Conclusions**

We have discussed the fashion in which faults in plans -- either incorrect plans or incomplete plans -- provides a unifying framework in which to understand the miscellany of HACKER bug types. We shall conclude by mentioning other dimensions along which SIMPLE allows a broader view of program planning and debugging than is present in HACKER.

HACKER contains an implicit theory of planning, consisting of a miscellany of programming techniques. A program is written through successive macro expansion using these techniques. SIMPLE is an improvement over HACKER in this dimension, by bringing more organization to planning. Rather than a "bag of tricks" [Sussman 1973, p. 57], SIMPLE organizes programming knowledge as decomposition operators that convert various logistic descriptions -- AND, OR, FOR-EVERY -- into procedural form. From this standpoint, HACKER's program writing capability is a
sub-graph of the planning ATN, consisting of the Identify and Decompose portions and excluding problem reformulation.

Secondly, HACKER is critically dependent on the annotations associated with the programs it writes. But there is no clear theory of annotation present. Structured Planning develops the idea of annotation by considering not just the program, but the entire planning and debugging process, as a comprehensible dialogue between programmer and computer. Following a linguistic philosophy, annotation is the hierarchical parse tree that the plan generates, augmented by the semantic variables attached to nodes in the parse which specify such contextual information as the model, caveats, code (statements of code inserted into program definitions and not subsequently deleted by edits), picture, etc. Commentary follows from the structure of the grammar and semantics of SIMPLE’s ATN’s.

Thirdly, by having a comprehensive set of planning constructs, it is possible to clearly see other kinds of bugs. For example, just as the wrong choice between linear and non-linear conjunction plans leads to bugs, so too does the wrong choice between any set of mutually exclusive plans. Thus, we would expect a similar class of bugs to arise in deciding between round (simple tail recursion) and full recursion repetition plans; and, indeed, in human problem solving, this confusion is often displayed. Another class of bugs arises not from one conjunct completely clobbering another, but from partial interference. The possibility of this is seen clearly when it is recognized that goals are described by models which can be more complex than the patterns discussed for HACKER. An example of this in the Blocks World is the following: building a green tower and a green table. There may be no interference between the choice of color, but there may be interference in the choice of blocks, as occurs if there are only a limited number of blocks available.

A fourth generalization which we have discussed above is that HACKER’s critics can be
characterized as transition constraints on the arcs. From this broader viewpoint, one sees clearly the possibility of positive and negative critics that argue for or against particular plans. Indeed, given the situation of having to choose a transition arc at a given node in the planning network, a critic is simply some selection function on the arcs.

A fifth generalization which also was discussed briefly above was that unsatisfied prerequisite manifestations can be considered instances of the more general class of model violations. All that is needed is to include Operator Models as well as Problem Models. This is not an added burden, since these operator models are necessary anyway as part of the Primitive Library used by the Identification planning technique.

In conclusion, we must stress that we agree with the overall HACKER philosophy that problem solving consists of both planning and debugging. Our objection is that HACKER treats these two complementary activities in an isolated fashion. HACKER does not pay sufficient attention to the theory of description for problems, for operators and for plans. It is our belief that Structured Planning remedies this.
7. Conclusions

The proper study of those who are concerned with the artificial is the way in which that adaptation of means to environments is brought about -- and central to that is the process of design itself. The professional schools will reassume their professional responsibilities just to the degree that they can discover a science of design, a body of intellectually tough, analytic, partly formalizable, partly empirical teachable doctrine about the design process.

[Simon, The Sciences of the Artificial, p. 58]

Our overall goals were to achieve rigor, clarity, power, and prediction in describing the problem solving process, using concepts from computational linguistics. To these ends, a structured theory of planning was developed by representing the program writing process as an Augmented Transition Network. Structured Debugging was characterized as the repair of incorrect or incomplete plans, which naturally arise in the course of rational but heuristic planning. In the remainder of this section, we describe various extensions of this paradigm to protocol analysis, structured programing and computer aided instruction.

Protocol Analysis

In [Goldstein & Miller, 1976], we applied an earlier version of the planning grammar to the task of parsing protocols. The recognition process was done by hand. Continuing our strategy of applying concepts from computational linguistics to problem solving, we plan to experiment with the application of various algorithms for natural language comprehension to the task of automated protocol analysis.

A critical question that will arise is whether SIMPLE is a good spanning model for elementary human problem solving. By this we mean: if SIMPLE is put in a mode wherein it generates all possible solutions to a given problem (primarily through successive problem reformulations), will the set of programs produced include most of the successful solutions generated by people? More specifically, is the protocol analysis task profitably approached from the standpoint of identifying
which of SIMPLE's possible plans for a given problem is being used by a student? In effect, can SIMPLE be used as a recognition grammar on protocols?

For example, we do not yet know whether SIMPLE is sufficiently powerful to include all of the plans typically pursued by students in elementary Logo programming tasks. If so, it represents an important step forward in information processing psychology [Newell & Simon, 1972]. Preliminary analysis of many Logo protocols indicates that it is. But extensive experimentation is needed before a definite answer is available. Fortunately, we are in a good position to attack this set of psychological questions because the Logo project has collected extensive data on student performance [G. Goldstein, 1973; Okumura, 1973]. [Miller & Goldstein, 1976] presents a preliminary design for a SIMPLE-based automated protocol analyzer.

In applying SIMPLE to protocol analysis, we envision approaching the problem of modelling the individual by inducing from analyzed protocols a modified version of the SIMPLE ATN for each student. The success of these models will be judged by the extent to which they successfully predict subsequent behavior. Again, experimentation is needed to determine whether this approach is viable.

The parsing problem is complicated in analyzing human protocols by the possibility of irrational planning errors and execution errors, in addition to the rational planning bugs discussed earlier. Because of the increased uncertainty introduced by possible mistakes in executing a statement of code or constructing an ungrammatical plan, we envision taking advantage of the powerful search programs created for parsing speech utterances [Allen, 1975; Woods et al, 1975; Paxton & Robinson, 1975; Lesser, 1975], in which uncertainty in the auditory interpretation similarly complicates the parsing process.

**Structured Programming**

Another application of the Structured Planning theory is to the design of improved
environments for programming. We envision the use of SIMPLE to support an editor in which changes to a program can be specified in terms of its plan. The virtue of expressing oneself in this fashion are (a) that it can lead to increased clarity in programming by drawing the programmer's attention to the nature of the plan being applied, and (b) that it gives the system much more leverage in aiding in the diagnosis and repair of bugs.

For example, one might instruct the editor to change a particular subgoal from being accomplished by means of an Identification to a plan based on Decomposition by Conjunction. The reason might be that the original sub-routine fetched from the Library had unacceptable side effects. The SIMPLE-based editor could then lead the programmer through a sequence of top-down planning decisions that would eventually realize the new plan.

The SIMPLE ATN is a top-down structured programmer. The SIMPLE editor can aid the programmer in exactly this process. The benefits of such an editor over current Structured Programming environments lies in the larger taxonomy of planning concepts available to SIMPLE. It is our belief that Dijkstra pointed in the right direction, in calling for a structured approach to programming [Dahi 1972]. We believe SIMPLE provides a next step in detailing exactly what rational planning involves.

In future research, we plan to construct a SIMPLE-based editor and experiment with its performance as a programming tool. The criteria by which it may be judged are the extent to which programmers find it useful, and the decrease, if any, in program writing and debugging time.

Computer Aided Instruction

In designing CAI programs, three problems are critical: (a) inducing a model of the student, (b) having a model of the expert, and (c) generating a tutorial plan for guiding the student toward expert competence. SIMPLE may aid in the resolution of these three problems in the design of CAI
systems for tutoring programming and problem solving.

We have already discussed how SIMPLE may provide an important modelling tool. Implicit in SIMPLE is also a theory of learning. From the SIMPLE standpoint, learning is the acquisition of new grammatical rules, new semantic variables, and new pragmatic constraints for deciding between alternatives. Hence, a SIMPLE-based tutor could compare the grammar induced for the student with the full SIMPLE grammar and choose a difference as the issue to be taught. Alternatively, the tutor can parse a given protocol, compare it with how SIMPLE would have solved the problem, and utilize the differences as the specific issues to be discussed with the student in analyzing his or her performance on the problem. In this fashion, we are extending the Issues and Examples paradigm developed by Burton and Brown [1976] for an elementary arithmetic world to the more complex environment of programming and problem solving.

Of course, there are many other subtleties in designing intelligent computer tutors not touched upon here. For example, in what sequence should the knowledge be taught, how intrusive should the tutor be, how can the tutor’s behavior be made clear to the student so that its actions are not mystifying, and can sufficiently powerful natural language capability be provided so that the student can interact comfortably with the tutor. However, we believe that SIMPLE will ultimately aid in creating better computer tutors, since the most fundamental issue is the extent to which the tutor understands the domain. Without this capability, tutoring can be little more than superficial.

It is also worth observing that automatic protocol analysis and student modelling, even without any CAI, can be equally valuable to a human teacher. The parsed protocol and student model allows the teacher to more easily notice when the student is relying on a limited lexicon of planning strategies, and whether the strategies that are known are organized in a successful fashion. This kind of detailed description of the reasoning process offers the possibility of escaping from the tyranny of standardized tests whose outcome is an uninformative numerical
The Science of Heuristics

Polya has called *heuristics* the study of the "means and methods of problem solving" [Polya, 1962, p. vi]. His various books [Polya 1957, 1962, 1965, 1967] offer insight into the nature of problem solving, discussing skills and abilities far in advance of even the most intelligent AI programs. But heuristics, as Polya develops it, is not a science. There are no formal representations for problem solving concepts: no rigorous means for experimenting with alternative theories. The use of the computer to implement and experiment with such theories makes the study of heuristics a science. SIMPLE represents a small contribution to this enterprise by experimenting with a particular procedural representation -- the Augmented Transition Network.

The most common criticism of even the most insightful analyses of problem solving is "but how can I realize that a particular problem solving strategy is appropriate?" An example of the gap that can exist between a discussion of thinking and specific, useful guidelines was found in the description by the great mathematician Poincare of his own mathematical problem solving given at the beginning of section 2. Surely we can do better than advising a student to drink coffee before going to sleep.

Attempting to structure the skills of various fields, whether they be mathematics or carpentry -- in a form that provides useful and precise guidelines to the students -- is the most fundamental task of education. Research in computer science, computational linguistics, and artificial intelligence is helping to find representations of active knowledge that are perspicuous, powerful, and precise. Ultimately, SIMPLE's most important contribution is as an experiment in this vein, exploring whether a particular computational approach is useful as a representation of problem solving skill. As such, it is a vital part of the investigation of the design process that Simon calls for in the quote given at the beginning of this section.
8. References

[Bobrow and Fraser 69]

[Buchanan et al 1969]

[Burton & Brown 1976]

[Conway 1969]

[Dahl et al 1972]

[Fahlman 1974]

[Fikes 1972]

[Fikes & Nilsson 1971]

[Fikes et al 1972]

[G. Goldstein 1973]


Kowalski, Robert, *Predicate Logic as a Programming Language*, University of Edinburgh, Department of Computational Logic, School of Artificial Intelligence, Memo 70, 1973.


References

[Minsky & Papert 1974]

[Moore 1974]

[Moses 1974]

[Newell & Simon]

[Okumura 1973]

[Papert 1971a]
Papert, Seymour A., Teaching Children to be Mathematicians Versus Teaching About Mathematics, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 249, 1971.

[Papert 1971b]
Papert, Seymour A., Teaching Children Thinking, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 247 (LOGO Memo 2), 1971.

[Papert 1973]
Papert, Seymour A., Uses of Technology to Enhance Education, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 298 (LOGO Memo 8), June 1973.

[Paxton & Robinson 1975]

[Polya 1957]

[Polya 1962]
Polya, George, Mathematical Discovery (Volume 1), New York, John Wiley and Sons, 1962.
References

[Polya 1965]

[Polya 1967]

[Quillian 1968]

[Rapport 1963]

[Sacerdoti 1975]

[Schank & Colby 1973]

[Sussman 1973]

[Sussman & Stallman 1975]

[Thorne et al. 1968]

[Winograd 1972]

[Winston 1975]

[Woods 1970]
[Woods 1975]

[Woods et al 1975]

[Woods et al 1972]

[Woods et al 1974]