Massachusetts Institute Of Technology
Artificial Intelligence Laboratory

AI Memo 383 December 1976 Logo Memo 30

Overview of a Linguistic Theory of Design

Mark L. Miller and Ira P. Goldstein

SPADE is a theory of the design of computer programs in terms of complementary planning and debugging processes. An overview of the authors' recent research on this theory is provided. SPADE borrows tools from computational linguistics -- grammars, augmented transition networks (ATN's), chart-based parsers -- to formalize planning and debugging. The theory has been applied to parsing protocols of programming episodes, constructing a grammar-based editor in which programs are written in a structured fashion, and designing an automatic programming system based on the ATN formalism.

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, in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643, and in part by the Division for Study and Research in Education, Massachusetts Institute of Technology.

Table of Contents

1. Introduction

1.1. Objectives and Methodology

1.2. A Linguistic Analogy

2. A Linguistic Theory of Planning

2.1. A Taxonomy of Planning Concepts

2.2. A Grammar of Plans

3. A Linguistic Theory of Debugging

3.1. A Taxonomy of Bugs

3.2. Diagnosis and Repair

4. Normative Aspects

4.1. SPADE -- A Grammar Based Editor

4.2. RAID -- A Debugging Assistant for SPADE

4.3. SHERLOCK -- An AI-CAI Tutor

5. Synthetic Aspects

5.1. PATN -- An Augmented Transition Network for Planning

5.2. DAPR -- A Model of Debugging

6. Analytic Aspects

6.1. Protocol Analysis as Parsing

6.2. PAZATN -- A Parser for Elementary Programming Protocols

7. Conclusions

8. References

1. Introduction

Many problem solving tasks, such as computer programming, may be characterized as the design of artifacts. This paper provides an overview of the authors' recent research on *SPADE* (*S*tructured *P*lanning *a*nd *De*bugging), a theory of this design process. Our purpose here is to provide a coherent overall framework. Each topic introduced is covered in greater detail elsewhere [Goldstein & Miller 1976a,b; Miller & Goldstein 1976b,c,d].

Figure 1 illustrates our perspective on the construction of information processing theories of cognition. We view this enterprise as involving normative, synthetic, and analytic aspects. We see it as representing a new paradigm, based upon a marriage of methods and goals from several traditional disciplines, including artificial intelligence, psychology, pedagogy, and computer science.

1.1. Objectives and Methodology: Our own research project may be viewed as an instantiation of this general paradigm, with sub-projects addressing all three aspects (figure 2). As shown by the central circle in the diagram, we seek to construct a computational theory of the design process. We wish to test the utility and validity of this theory, SPADE, in a variety of contexts. This leads to specific goals and methods, represented by the three outlying circles in the diagram, which span the synthetic, analytic, and normative aspects and applications of the theory.

1. The synthetic (AI) goal is to explore computational theories of problem solving and learning. The method is to construct programs that embody these theories. This concern is reflected in our work on PATN [Goldstein & Miller 1976b], a problem solving program which will plan and debug simple blocks world and graphics programs. The support for an AI theory is determined primarily by the competence and efficiency of the associated computer program in performing a prescribed set of tasks.

FIGURE 1 - INFORMATION PROCESSING THEORIES OF COGNITION

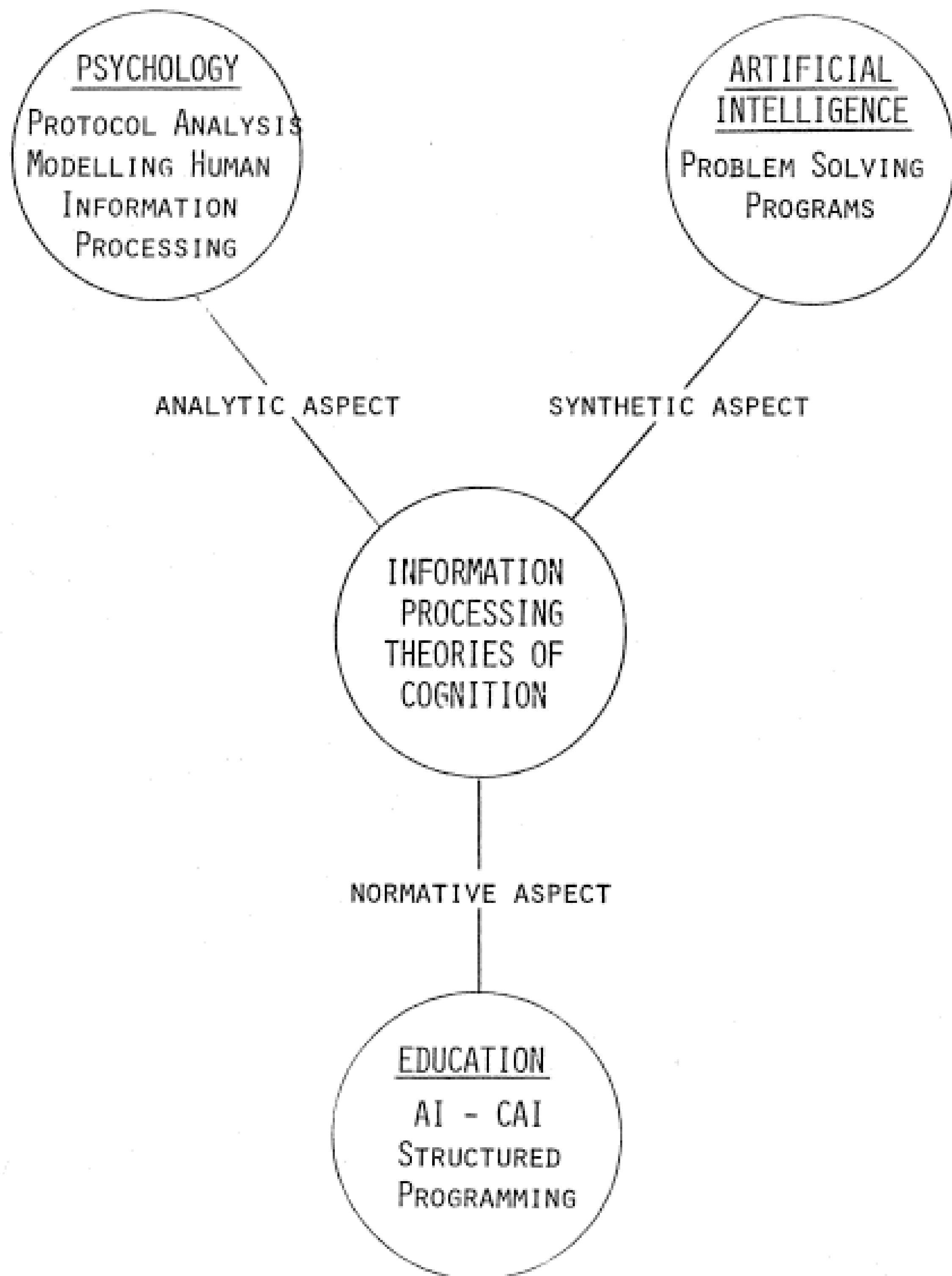
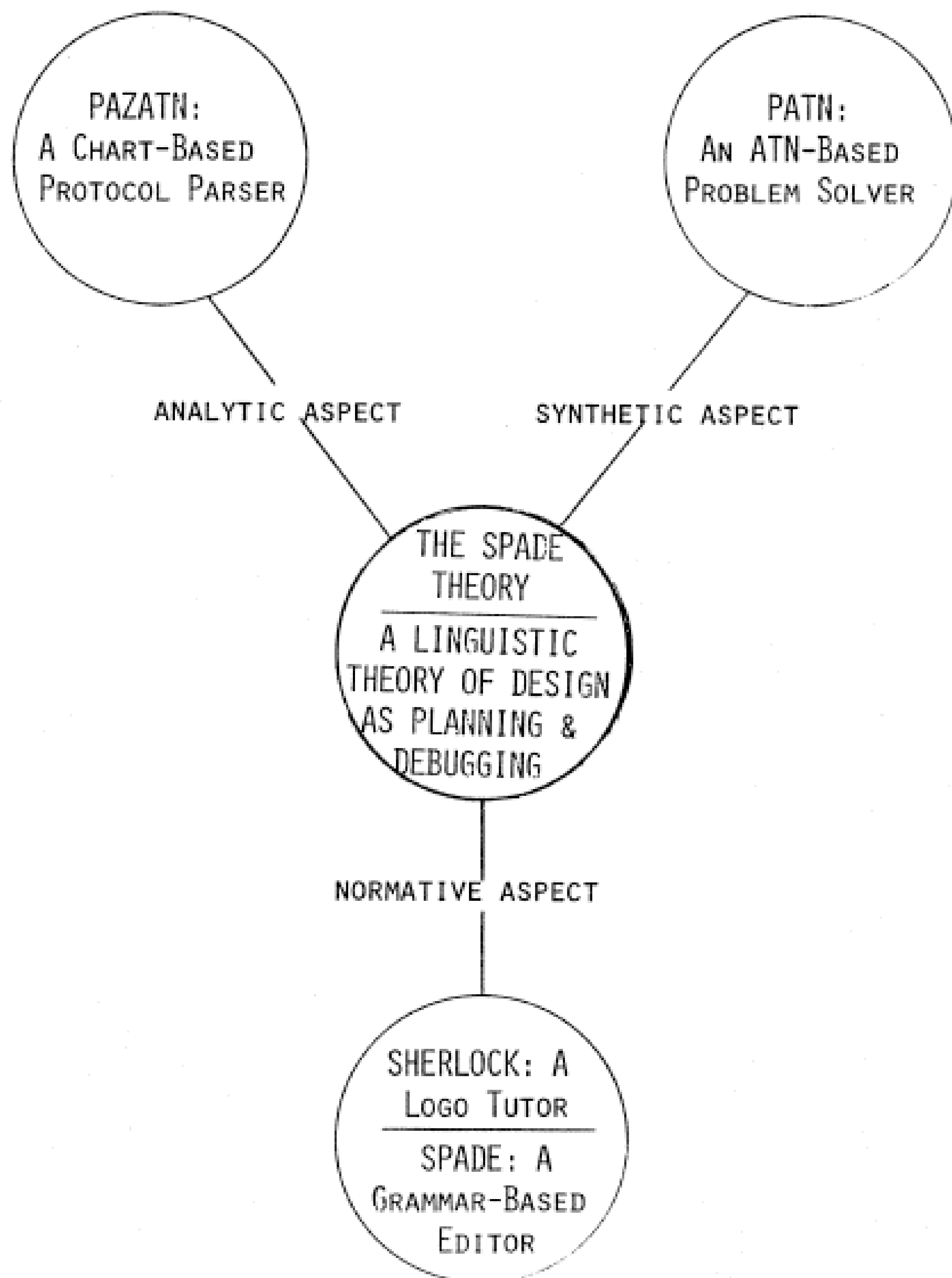


FIGURE 2 - A LINGUISTIC THEORY OF DESIGN AS PLANNING & DEBUGGING



2. The analytic (psychological) goal is to account for the knowledge states and learning strategies of individuals. Our method is to analyze protocols of subjects interacting with precisely controlled computer environments as they solve some problem [Miller & Goldstein 1976b]. We seek to model the subject's current knowledge, not only about the particular domain, but also about planning and debugging strategies. PAZATN [Miller & Goldstein 1976d], a system to analyze elementary programming protocols and reveal the use of various plans and debugging techniques, incorporates this concern. A theory of design embedded in an automatic protocol analyzer is supported to the extent to which it can describe and predict the subject's responses: both the final solution and observable details of the design process by which that solution is found.

3. The normative (educational) goal is to prescribe design methodology for both students (such as beginning programmers) and expert human problem solvers (such as professional programmers). This is partly a pedagogical concern: we wish to experiment with the SPADE theory as the basis for a curriculum about problem solving. At the same time, it shares the structured programming movement's concern to improve the quality and reliability of software. The former concern is explored through the design for SHERLOCK [Goldstein & Miller 1976a], an hypothetical computer tutor which embodies our vision of flexible, sensitive uses of computers to teach problem solving and enhance education. The latter concern is explored via the *SPADE editor* [Miller & Goldstein 1976c], a grammar-based environment to assist beginning programmers in acquiring, and professional programmers in adhering to, a top-down, structured design discipline. These systems, like PATN and PAZATN, though potentially valuable as applications programs, are mainly intended as experimental tools for testing the SPADE theory. The experimental methodology is to systematically vary the operation of the learning or programming environment. The claims of the theory are supported to the extent to which the system as a whole (as well as its various components) aid(s)

the user in solving harder problems more quickly.

Combining these methods and goals into a single research program has powerful synergistic effects. We have realized this in our particular projects through the development of a unifying linguistic theory of design.

1.2. A Linguistic Analogy: In developing a formalism for representing problem solving techniques, we have been guided by a novel perspective: an analogy to computational linguistics. We have found this analogy to be fruitful for several reasons.

1. Computational linguistics, though intended to illuminate the nature of language *per se*, has produced a set of concepts and algorithms for characterizing and explaining complex computational processes which are both perspicuous and rich in power. Problem solving, as a complex process, benefits from the application of these tools.

2. Computational linguistics decomposes computations into syntactic, semantic, and pragmatic components. This decomposition clarifies the explanation of complex processes, when viewed in the following manner: syntax formalizes the range of possible decisions; semantics the problem description, and pragmatics the relationship between the two.

3. Computational linguistics has undergone an evolution of procedural formalisms, beginning with finite state automata, later employing recursive transition networks (context free grammars), next moving on to augmented transition networks, and culminating in the current set of theories involving frames, etc. Following this evolutionary sequence in language theories illuminates their complexity. Each phase captured some properties of language, but was incomplete and required generalization to more powerful and elaborate formalisms. Moreover, the interrelationships among many of these formalisms have been thoroughly delineated.

From this evolutionary perspective, one need not necessarily view a given stage of theorizing as wrong. Sometimes an earlier theory *is* wrong, but in other cases the earlier approach can be valuable as an abstraction in its own right, which illuminates some dimension of the phenomena, even though it is inadequate as a complete theory. We are following a sequence parallel to that exhibited by computational linguistics in our own study of problem solving.

In this evolutionary development of SPADE, our theory of the design process, two sub-tasks have been addressed. First, we have analyzed certain intricacies of planning and debugging, such as are encountered in the design of programs which must take into account interactions in achieving dependent subgoals. The second sub-task has been to seek a representational framework in which to elucidate these subtleties, and in which to structure a wide variety of planning techniques. Our approach has been to begin with simple but clear formalisms, studying their virtues and limitations. Our plan is to continue to investigate a series of progressively more powerful and elaborate representations, after reaching a solid understanding as to where the extra power is needed, and why.

To date, we have explored context free planning grammars, and their generalization to ATN's; we have transferred the insight gained from studying planning to the development of a model of debugging; we have examined the metaphor of protocol analysis as parsing, and studied the use of a chart parser as a means to discovering these analyses.

2. A Linguistic Theory of Planning

The center circle of figure 2 provides the setting for the discussion in this section and the next. Then, having introduced some basic notions of the SPADE theory of design, we will be in a position to move to the peripheral aspects (the outer circles) in sections four, five and six.

2.1. A Taxonomy of Planning Concepts: The basis for SPADE is a taxonomy of frequently observed planning concepts (figure 3). We arrived at this taxonomy partly by introspection, partly by examining problem solving protocols [Miller & Goldstein 1976b], and partly by studying the literature on problem solving [Polya 1957, 1962, 1965, 1967, 1968; Newell & Simon 1972; Sussman 1975; Sacerdoti 1975]. We regard the taxonomy as neither complete nor unique. Part of the research program is the classification of additional techniques and the evaluation of alternative organizational schemes.

There are three major classes of plans in the taxonomy: identification, decomposition, and reformulation. Identification means recognizing a problem as previously solved. Decomposition refers to strategies for dividing a problem into simpler sub-problems. Reformulation plans alter the problem description, seeking a representation which is more amenable to identification or decomposition. The figure suggests how these classes of plans are further subdivided in the SPADE theory.

2.2. A Grammar of Plans: Planning, according to the theory, is a process in which the problem solver selects the appropriate plan type, and then carries out the subgoals defined by that plan applied to the current problem. From this viewpoint, the planning taxonomy represents a decision tree of alternative plans. The decision process can be modeled by a context free grammar (figure 4).

Consider the top level rule of this grammar:

P1: SOLVE -> PLAN + [DEBUG].

The nonterminal symbol SOLVE is analogous to the nonterminal SENTENCE in a grammar for language. In our notation, P1 means that planning is first used to generate a plan, with subsequent debugging then being required to complete the solution. Since the plan may be entirely correct, DEBUG is in brackets, indicating that it is an optional constituent.

The disjunctive rule, P2, represents the choice -- in our taxonomy -- between the three mutually exclusive categories of plans: identification,

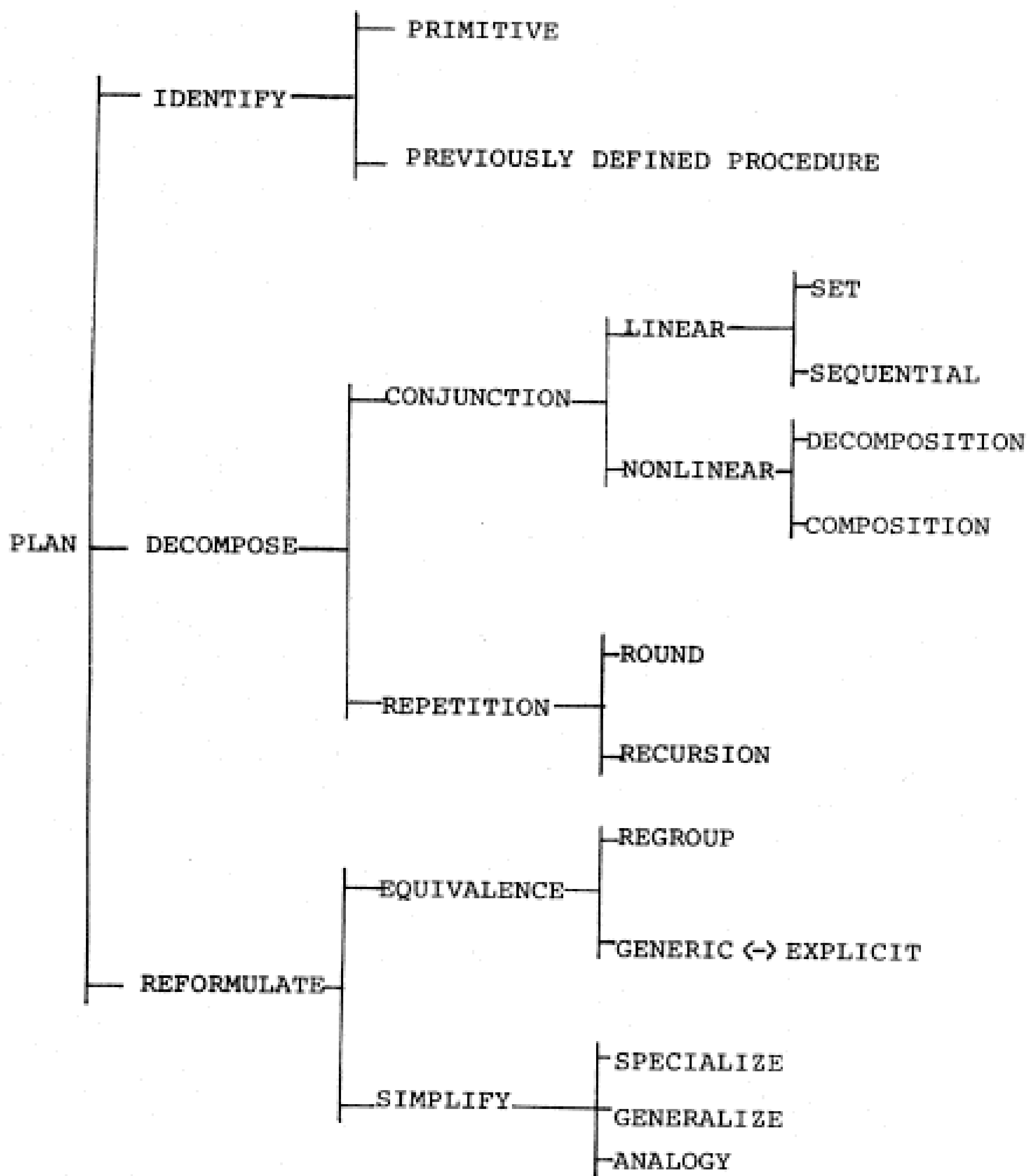


FIGURE 3
TAXONOMY OF PLANNING CONCEPTS

Figure 4. G3: A Grammar[1] of Plans

```
P1:  SOLVE        -> PLAN + [DEBUG]
P2:  PLAN         -> IDENTIFY | DECOMPOSE | REFORMULATE
P3:  IDENTIFY     -> PRIMITIVE | DEFINED
P4:  DEFINED      -> "use code" & "get file"
P5:  DECOMPOSE    -> CONJUNCTION | REPETITION
P6:  CONJUNCTION  -> LINEAR | NONLINEAR
P7:  LINEAR       -> SET | SEQ
P8:  SEQ          -> [SETUP] + <MAINSTEP + [INTERFACE]>* + [CLEANUP]
P9:  SET          -> <SOLVE>*
P10: SETUP        -> SOLVE
P11: MAINSTEP     -> SOLVE
P12: INTERFACE    -> SOLVE
P13: CLEANUP      -> SOLVE
P14: REPETITION   -> ROUND | RECURSION
P15: ROUND        -> ITER-PLAN | TAIL-RECUR
P16: ITER-PLAN    -> "repeat step" + SEQ
P17: TAIL-RECUR   -> "stop step" + SEQ + "recursion step"
```

[1]The rules of the grammar are written using the following syntax:

disjunction: "a | b" is read as, "a or b";

ordered conjunction: "a + b" is read as, "a and b", where the order is significant;

unordered conjunction: "a & b" is read as, "a and b", where the order is insignificant;

optionality: "[a]" is read as, "a is optional";

iteration: "<a>*" is read as, "a repeated 1 or more times";

lexical category: a lower case English phrase enclosed in quotation marks (e.g., "repeat step") describes a lexical item which is not further expanded in the grammar.

decomposition, and reformulation.

P2: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE

The vertical bars indicate that a choice is required. Other rules are interpreted similarly.

The SPADE theory is not restricted to any particular domain. However, to provide concrete examples, most of our papers use problems from elementary Logo graphics programming [Papert 1971a,b; 1973]. Figure 5 illustrates the grammar rules for primitives in this domain. Figure 6 shows our favorite example -- a typical goal undertaken by beginners in Logo programming -- a "wishingwell" picture. Figure 7 illustrates a solution to the wishingwell problem with its hierarchical annotation according to our planning grammar.

The grammar of plans represents a useful abstraction of the decision process involved in selecting plans from the taxonomy. We illustrate this point in the next section by analyzing debugging in terms of the grammar. Later in the paper we show how the theory may be extended to include, not only the syntax of plans, but their semantics and pragmatics as well.

3. A Linguistic Theory of Debugging

Often problem solvers must decide on a plan in the face of imperfect knowledge and limited resources. Even carefully reasoned judgments made under these circumstances may turn out to be mistaken: debugging is then required. Given a grammatical theory of planning, debugging can be analyzed as the localization and repair of errors in applying grammar rules during planning. The linguistic analogy unifies planning and debugging by tracing the origin of bugs to various types of erroneous planning choices.

Figure 5 Grammar Rules for Logo Primitives

L1.	PRIMITIVE	->	VECTOR | ROTATION | PENSTATE
L2.	VECTOR	->	FORWARD|BACK + "number"
L3.	ROTATION	->	LEFT|RIGHT + "number"
L4.	PENSTATE	->	PENUP | PENDOWN

FIGURE 6 WISHINGWELL PICTURE

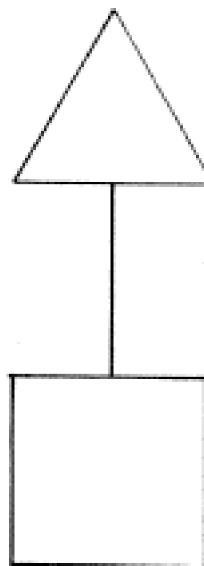
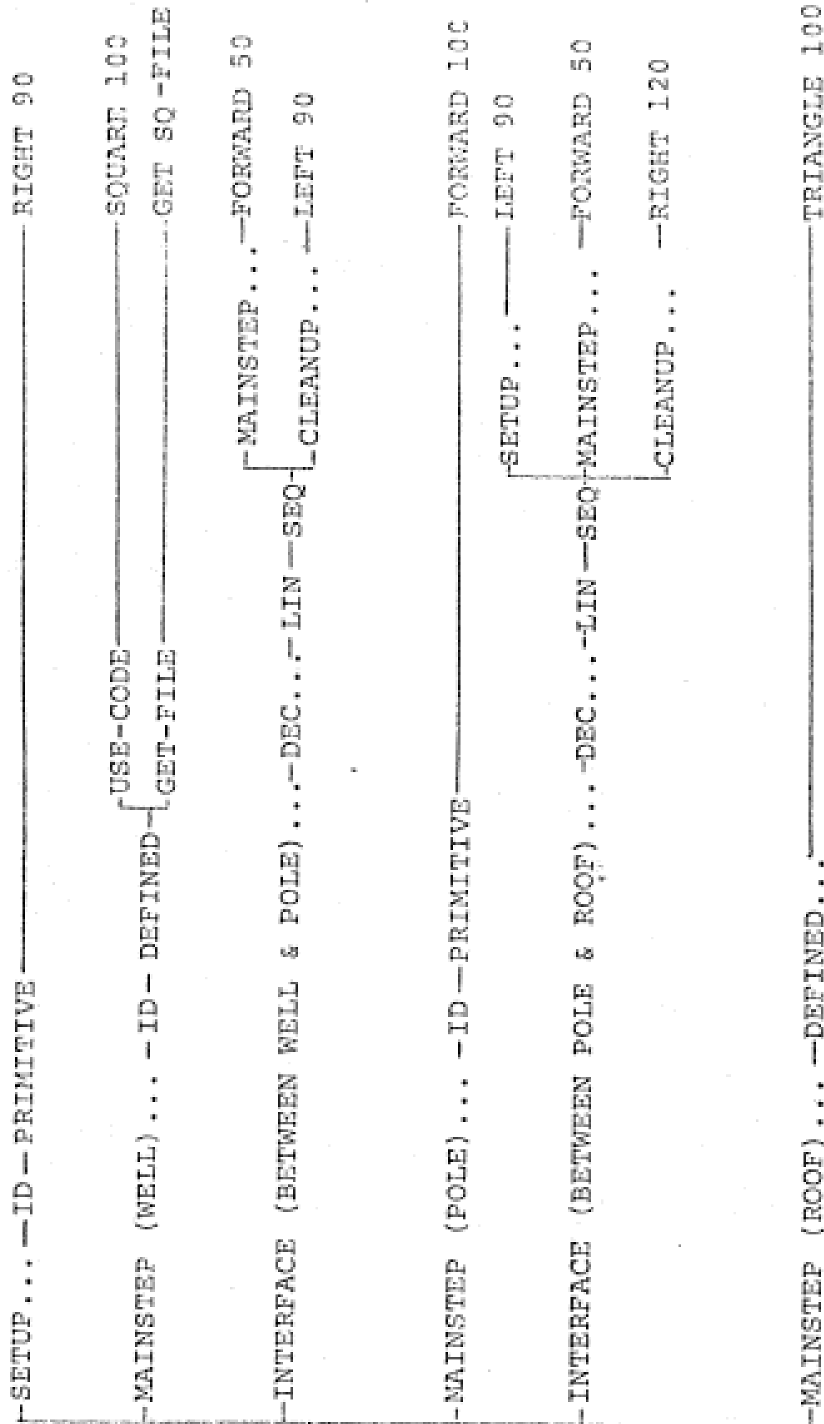


FIGURE 7

ABBREVIATED HIERARCHICAL DERIVATION TREE FOR WISHINGWELL



3.1. A Taxonomy of Bugs: Since our planning rules were constructed from operators for conjunction, disjunction, and optionality, three basic classes of errors arise:

1. syntactic bugs, in which the basic grammar is violated, such as when a required *conjunct* is missing;

2. semantic bugs, in a semantic constraint arising from the particular problem is violated, such as when a syntactically *optional* constituent, needed because of the semantics of the particular problem, is missing;

3. pragmatic bugs, in which an inappropriate selection from a set of mutually exclusive *disjuncts* is made.

These bug types are illustrated in figure 8. Although these classes are adequate to characterize many examples which arise in elementary programming, additional categories must be defined to make this taxonomy of bugs complete.

3.2. Diagnosis and Repair: An important aspect of our research is the analysis of techniques for diagnosis and repair of planning bugs. These techniques can be classified according to which representation of a problem they access: the problem specification (or model), the solution (or code), the plan derivation, or the process state. Techniques for plan diagnosis can be further categorized according to the type of planning bug hypothesized: syntactic, semantic, or pragmatic. (Further details of the debugging theory are presented in later sections.)

In the next three sections we examine several experimental applications programs which we have designed and intend to implement. The presentation is organized according the aspects of the investigation represented by the outer circles of figure 2. We must emphasize two points: first, that this division by aspects is a crude first approximation, because of the considerable overlap implied by a unified approach; second, that while the programs which we have

FIGURE 8a - SYNTACTICALLY INCORRECT PLAN

A NECESSARY CONJUNCT IS MISSING

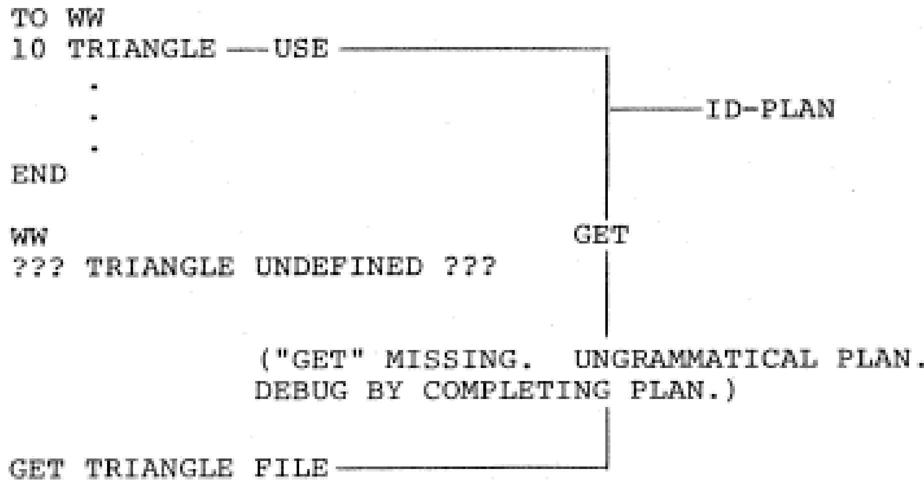


FIGURE 8b - SEMANTICALLY INCORRECT PLAN

AN OPTIONAL CONJUNCT IS MISSING

FOR EXAMPLE: "WW" MISSING INITIAL SETUP, AND INTERFACE FOR POLE.

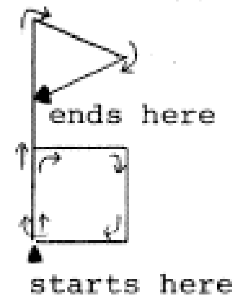


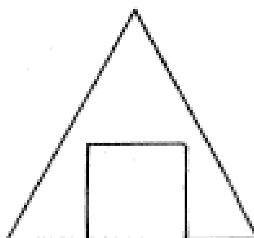
FIGURE 8c - PRAGMATICALLY INCORRECT PLAN

AN INCORRECT DISJUNCT HAS BEEN SELECTED

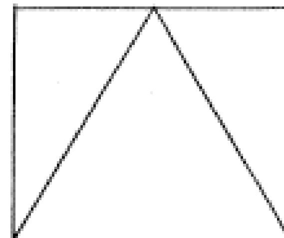
```
TO SQUARE-INSIDE-TRIANGLE
10 SQUARE
20 TRIANGLE
END
```

LINEAR PLAN -- SQUARE AND TRIANGLE DESIGNED INDEPENDENTLY.

INTENDED PICTURE:



ACTUAL PICTURE:



designed potentially have practical applicability, we regard them primarily as experimental tools, which will serve to test the validity of the underlying SPADE theory. We turn first to the normative aspects, describing systems designed to encourage and teach articulate top-down structured design.

4. Normative Aspects

How can we judge whether a particular grammar of plans captures, at some level of abstraction, the set of planning decisions which ought to be considered in solving problems for a domain? One methodology is to incorporate the context free grammar into a program editing environment, to augment and direct the capabilities of a human user. The critical question then becomes determining the extent to which such a support system aids or hinders the user. This is the rationale for SPADE, an editor that incorporates our planning grammar.

4.1. SPADE -- A Grammar Based Editor: *SPADE* [Miller & Goldstein 1976c] is an acronym for *Structured Planning and Debugging Editor*. We chose this name to emphasize the link between our research and the structured programming movement. Dahl, Dijkstra, and Hoare [1972] call for a style of programming which reflects coherently structured problem solving. But a detailed formalization of what this style entails is lacking. Our efforts in this direction, therefore, naturally supplement the work of Dijkstra and others.

Suppose a problem solver is defining a Logo program for drawing the wishingwell shown earlier. In SPADE, this is accomplished by applying the planning grammar in generative mode. For example:

```
1a. What is the name of your top level procedure?
1b. >WW

2a. Rule for WW is: SOLVE -> PLAN + [DEBUG].
        Rule for WW-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE.
        What now?
```

2b. >DECOMPOSE.

In this way, SPADE will try to encourage users to articulate their design decisions in top-down order. At the same time, the system should offer the user the freedom to escape from this strict discipline if an alternative solution order seems preferable. Here the user interrupts the top-down prompting, suspending one subgoal to pursue another:

6a. Do you wish to include the optional constituent CLEANUP?
6b. >LATER

7a. Your pending goals are POLE, ... and WELL. What now?
7b. >POLE

The implementation of SPADE involves assigning an interpretive procedure to each grammatical operator, to perform bookkeeping of the user's goals.

As a simple example of the experimental utility of SPADE, the relationship between level of expertise and application of top-down planning can be ascertained from the frequency of using the LATER capability. More subtle experiments along these lines will examine the use of LATER in various planning contexts.

4.2. The RAID Debugging Assistant: SPADE keeps a thorough record of the user's planning decisions, called the *plan derivation*. Because of this, it is possible to design a debugging assistant (RAID) for the SPADE environment which embodies what we believe to be a fairly deep notion of debugging. As the user is prompted through the decision tree represented by the planning grammar, those choices which might lead to syntactic, semantic, or pragmatic bugs can be internally flagged by the system. In subsequent debugging episodes, RAID can alert the user to the relevant bug warnings, thereby aiding in diagnosis. Moreover, the SPADE monitor can then be reinvoked to guide repair of the bug, by resuming planning in the context of the culpable decision (figure 9).

Figure 9 . RAID Advice for Syntactically Incomplete Planning Bugs

1a. Which subgoal would you like to pursue?

1b. >POLE

2a. Rule for POLE-1 is: PLAN -> IDENTIFY | DECOMPOSE | REFORMULATE.
What now?

2b. >IDENTIFY

3a. Rule for POLE-2 is: IDENTIFY -> PRIMITIVE | DEFINED.
What now?

3b. >LATER.

The LATER command causes further processing of this subgoal to be postponed. RAID, the debugging component, creates a caveat each time a LATER command occurs. The caveat records the fact that this part of the plan is syntactically incomplete at this point.

4a. POLE-2 postponed. Your pending goal list contains the remaining subgoal WELL. Which goal would like to pursue?

4b. >WELL

.
.
.

5a. We have now completed the WELL subgoal. What now?

5b. >RUN WISHINGWELL

The caveat recorded earlier is a "demon," triggered by premature execution of the procedure being designed. If this occurs, the function of the caveat is to remind the user of the syntactic incorrectness of the program.

6a. Running WISHINGWELL at this time results in the bug manifestation:
UNDEFINED PROCEDURE: POLE.
This is because WISHINGWELL is syntactically incomplete: it contains the pending subgoal POLE-2. The recommended repair technique is to complete the solution of the subgoal POLE-2. Would you like to work on POLE-2?

6b. >Yes.

7a. You were solving POLE via an IDENTIFICATION plan. You postponed the decision (POLE-2) as to which answer library to use.
The rule for POLE-2 is: IDENTIFY -> PRIMITIVE | DEFINED.
What now?

.
.
.

4.3. SHERLOCK -- A Computerized Tutor Computational theories of problem solving have important ramifications for education. Such theories can serve as the normative justification for a given curriculum, reveal critical knowledge for the student excluded from the traditional syllabus, provide a basis for qualitative -- not merely quantitative -- evaluation, and be embodied in intelligent tutoring programs to serve as personal learning environments for the student. This position is exemplified by SHERLOCK [Goldstein & Miller 1976a]. SHERLOCK is an hypothetical computerized tutor which represents a long term research direction rather than a detailed program design. SHERLOCK embodies our vision of flexible, sensitive uses of computers to teach problem solving and enhance education. The other programs we have designed, each significant undertakings in their own right, can also be thought of as necessary components for SHERLOCK.

SHERLOCK's goal is to imbue a *working knowledge* of the SPADE theory in novice programmers. This does *not* necessarily mean that fifth graders must use terminology such as *context free grammar*. It *does* mean that they should experience, actively manipulate, and explicitly discuss problem solving situations which highlight the planning and debugging concepts which the theory attempts to formalize. Let us illustrate what this might mean via a brief dialogue.

SHERLOCK: Hello Debbie.
What problem are you working on today?

Debbie: >I am going to draw a wishingwell.

Initially in a less intrusive, *backward looking* mode, SHERLOCK might remain silent as Debbie typed in her code for a wishingwell. Unlike the highly structured SPADE, SHERLOCK will not take an extremely active role in prompting the student. (We plan to experiment with the relative virtues of these two tutorial styles.) However, SHERLOCK might intervene when difficulties were

encountered by the student.

Debbie:
>forward 100
>right 90
>forward 500
>Oh no! Erase that last forward.

SHERLOCK:
Ok. "Forward 500" has been erased.
Do you wish to group the other
lines together into a procedure?

Here, a simple rule of programming style -- the use of subprocedures -- is being emphasized.

Many complex issues are raised by the design of such tutoring programs. Our purpose in introducing SHERLOCK has merely been to illustrate one potential pedagogical application for a computational theory of design. The next section turns our attention to the synthetic aspects of our enterprise, by introducing an AI problem solver called PATN.

5. Synthetic Aspects

While context free grammars can represent a useful abstraction of planning decisions, they have limitations which prevent them from providing a complete theory of design. To address this, we have designed PATN, an AI problem solver. PATN, like SPADE, starts from our taxonomy of plans. But PATN takes the linguistic analogy one step further. An augmented transition network (ATN, [Woods 1970]) is used, to capture not only the syntax of plans, but also their semantics and pragmatics.

5.1. PATN -- An Augmented Transition Network for Planning: Figure 10 provides a global view of PATN [Goldstein & Miller 1976b]. Here the decision to pursue an identification plan versus a decomposition, for example, is modeled by an arc transition. Semantics are added, by defining a set of registers to record the problem *description*, proposed *solution*, planning *advice*, and debugging *caveats*. Pragmatic information is also incorporated, by associating conditions and actions with various arcs. For instance, an identification plan cannot proceed if the problem description cannot be found in the answer library. PATN elaborates our notion of a plan, by associating semantic variables (snapshots of the ATN registers) with each node of the plan derivation. One application of PATN is as a module of SPADE, providing an enhanced set of features to aid the user in communicating plans. Our implementation plan for PATN is to provide SPADE with a mode of operation in which a progressively larger percentage of planning choices are made without consulting the user.

5.2. DAPR -- A Model of Debugging: PATN can make mistakes. That is, PATN will sometimes introduce what we term *rational bugs* into its plans, due to making arc transitions with imperfect knowledge of subtleties and interactions in the task domain. Naturally, PATN will come equipped with a corresponding debugging module (DAPR). Whereas RAID is designed to assist human programmers in finding a variety of bugs (primarily by plan diagnosis), DAPR is specifically designed to analyze PATN's bugs, employing three diagnostic techniques: model, process, and plan diagnosis. Model diagnosis is the basic technique. It amounts to comparing the effects of executing a plan to a formal description of its goals, to determine if, and in what fashion, the plan has failed. Another DAPR technique, based on Sussman's HACKER [1975], is examining the state of the process at the time of the error manifestation. Plan diagnosis is a DAPR first. It is accomplished by examining the *caveats* variable associated with various nodes of PATN's plan derivation.

DAPR could be used to provide additional guidance to RAID. This possibility illustrates the synergism possible when normative, synthetic, and analytic facets of a cognitive theory are studied in an integrated fashion. In

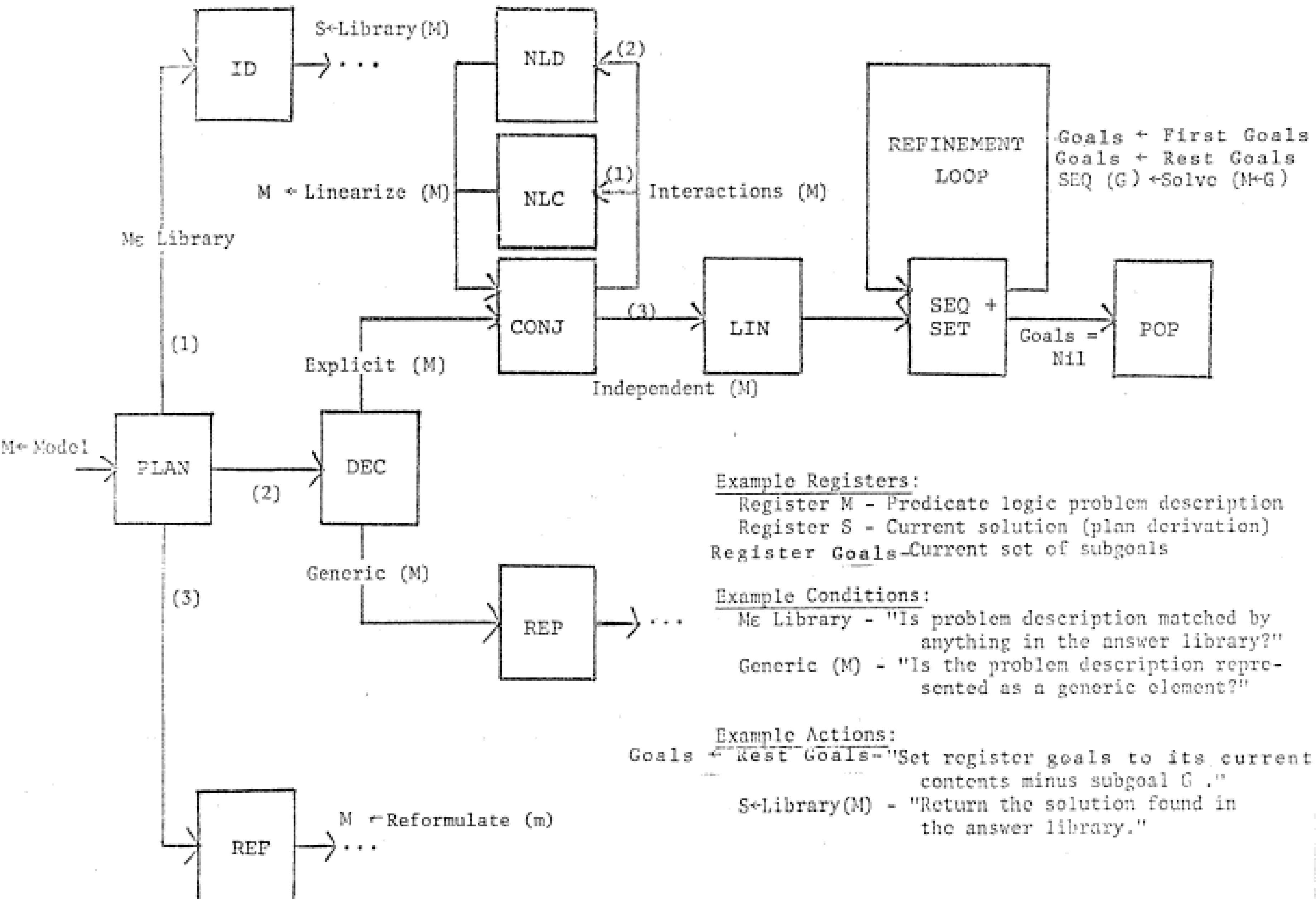


FIGURE 10- A GLOBAL VIEW OF PATN

the next section we pursue the analytic facet of our investigation of the design process, introducing PAZATN, an automatic protocol analyzer.

6. Analytic Aspects

As soon as one has an heuristically adequate theory of design, it is natural to ask, "Can the theory provide an account of how *people* solve problems?". The traditional (e.g., [Newell 1966]) experimental technique for answering this question is the analysis of protocols collected during problem solving sessions. While this generally implies transcriptions of *think aloud* verbalizations, a useful simplification is to examine the sequence of keystrokes from a console session in a computerized data gathering context.

6.1. Protocol Analysis as Parsing: The analysis task in such a setting is to interpret user type-ins during a console session in terms of a theoretical model of the planning and debugging processes. Our linguistic analogy is helpful here, wherein we define protocol analysis as the construction of an hierarchical description of the protocol in terms of our problem solving grammar. [Miller & Goldstein 1976b] provides a detailed example of such analysis being performed by hand. In that paper, we examine a high school student's Logo protocol in detail, summarizing the sorts of insights obtained when protocol analysis is viewed as parsing in this sense.

Just as a context free grammar is incomplete as a theory of planning, likewise a parse is only a partial analysis of a protocol. The theory of annotation developed in the PATN work leads us from describing only the *structure* to more complete analyses of protocols: an interpretation of a protocol is the selection of a particular PATN plan derivation. Hence analysis should consist of linking protocol events into the data structures of PATN and of advanced versions of SPADE.

6.2. PAZATN -- A Parser for Elementary Programming Protocols: Manual protocol analysis is unacceptably tedious and informal. Hence [Miller & Goldstein 1976d] introduces the design for an automatic protocol parser, PAZATN. PAZATN will analyze protocols by matching them against possible solutions which PATN generates. PAZATN will operate by causing PATN to deviate from its preferred approach in response to bottom-up evidence (figure 11). Also, "buggy" versions of synthetic plans (including *irrational* bugs which would not be introduced by PATN) can probably be recognized.

PAZATN's design is a generalization and elaboration of the *coroutine search* plan-finding procedure described for Mycroft [Goldstein 1975]. Looking to computational linguistics for guidance, PAZATN has been extended to take advantage of powerful search strategies developed in research on speech understanding [Lesser et al. 1975; Paxton & Robinson 1975], as well as sophisticated data structures developed in work on syntactic analysis [Kay 1973; Kaplan 1973].

PAZATN will be constructed by supplementing PATN with a number of additional modules and data structures. Figure 12 provides a more detailed block diagram. One data structure, the PLANCHART, keeps track of the set of plausible subgoals which have been proposed by PATN. Another, the DATACHART, records the state of partially completed interpretations. A preprocessor module will be used to suppress uninteresting syntactic details and to perform preliminary segmentation. The preprocessor employs an event classifier to determine the syntactic class of each event. Corresponding to each syntactic category, PAZATN will be supplied with an event specialist which embodies the requisite domain knowledge for assisting an event interpreter in associating an event of that type with some synthetic subgoal. Since a purely top down or bottom up strategy would be too inefficient, a scheduler module is necessary to direct the analyzer through a *best first coroutine search*.

Just as PATN will be implemented by extending SPADE to the extreme of never requesting guidance from the user, PAZATN will be implemented by extending

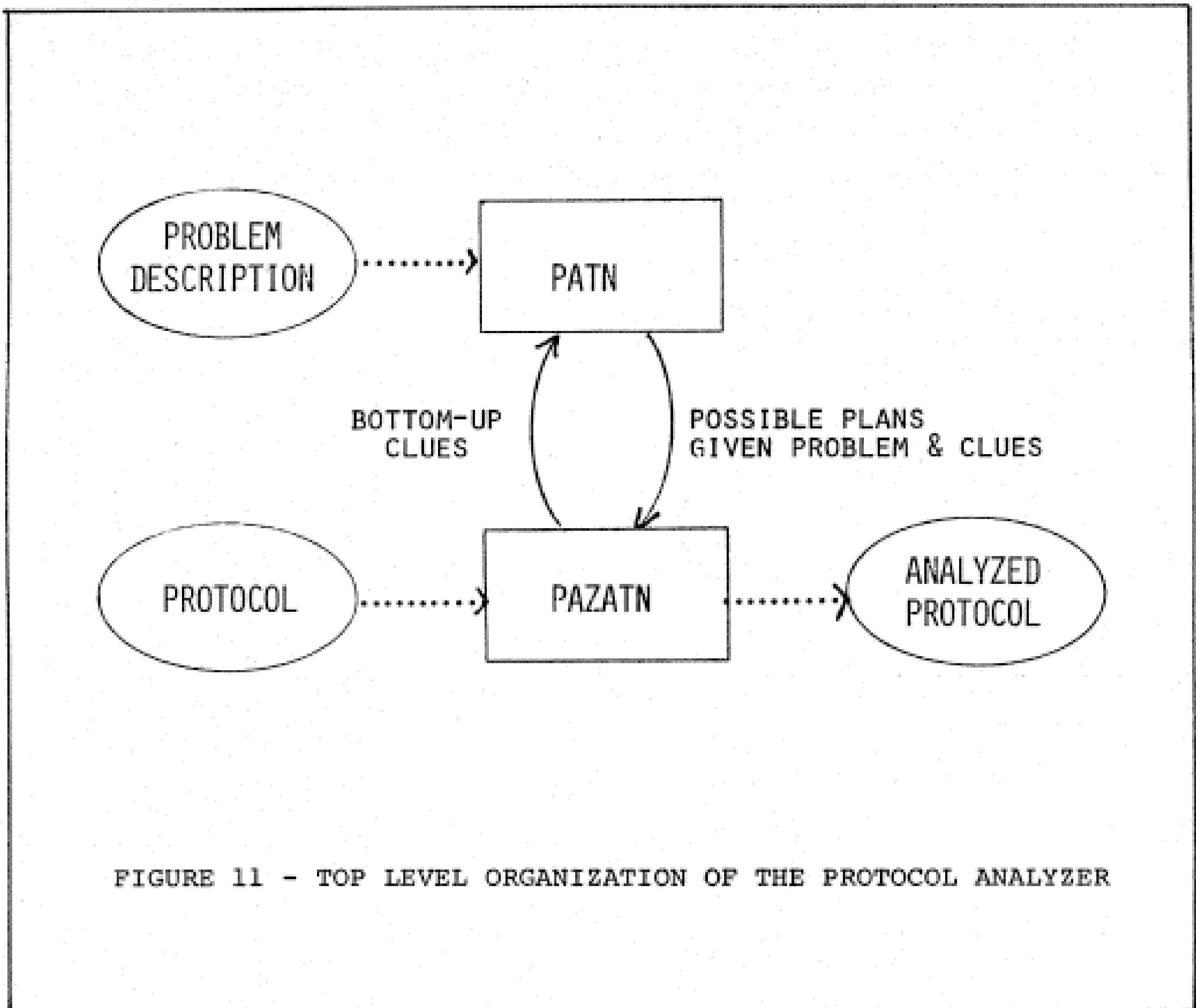


FIGURE 11 - TOP LEVEL ORGANIZATION OF THE PROTOCOL ANALYZER

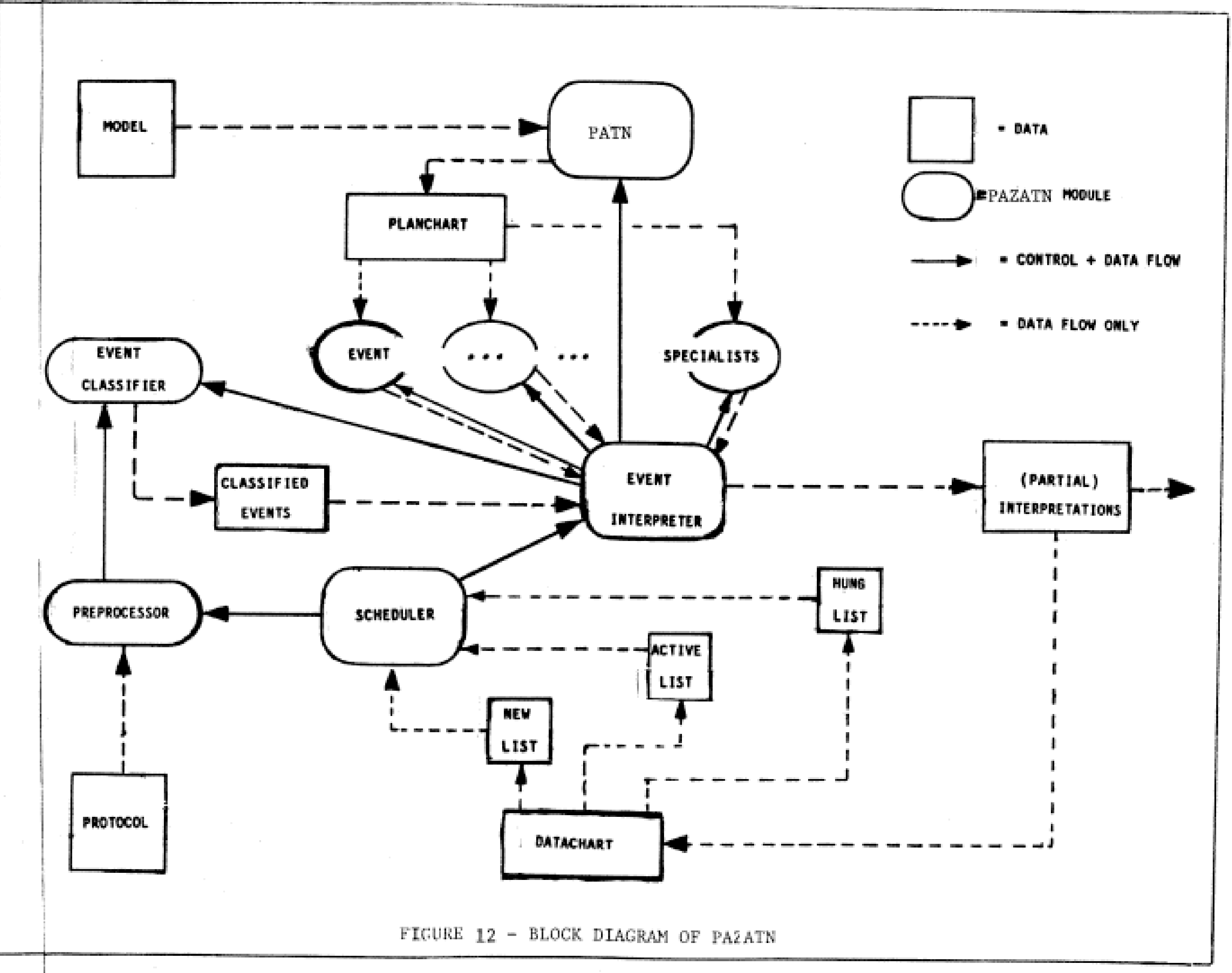


FIGURE 12 - BLOCK DIAGRAM OF PAZATN

SPADE to the opposite extreme. That is, in the pure PAZATN situation, the system must infer the user's plan entirely from code level events, with no explicit articulation of the intermediate levels of the plan. PAZATN will be useful in increasing SPADE's flexibility in handling ambiguous events, and in alleviating what might seem to some users to be an excessive allocation of time and effort to the planning phase. Moreover, systematic experimentation with PAZATN will provide evidence regarding whether PATN can serve as the basis for models of human problem solving.

7. Conclusions

We have studied problem solving for tasks which may be characterized as the design of artifacts: the outlines for SPADE, a computational theory of design, have been presented. The normative, synthetic, and analytic aspects of its role in the overall research enterprise have been illustrated by introducing several experimental applications programs. The exploitation of concepts and algorithms from computational linguistics is a recurring theme: grammars, ATN's, derivation trees, search strategies from speech understanding, chart-based parsers. We believe that our unified approach to the objectives of several fields, utilizing methods from each, represents a new paradigm which can provide benefits to all of them.

Much remains to be done. While far greater detail is available in our other papers, not every detail of the SPADE theory has been specified. Although almost all of the programs have been designed, even hand-simulated, none have been implemented. Neither the utility, the validity, nor the generality of the theory has been demonstrated. If the individual research projects succeed, a new clarity will have been brought to the study of problem solving. If, perchance, they should fail, then the reasons for the failures should nevertheless provide useful insights.

8. References

Dahl, Ole-Johan, Edsger Dijkstra and C.A.R. Hoare, 1972. *Structured Programming*. London, Academic Press.

Goldstein, Ira P., 1975. "Understanding Simple Picture Programs." *Artificial Intelligence*, Volume 6, Number 3.

Goldstein, Ira P., and Mark L. Miller, December 1976a. *AI Based Personal Learning Environments: Directions For Long Term Research*. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 384 (Logo Memo 31).

Goldstein, Ira P., and Mark L. Miller, December 1976b. *Structured Planning and Debugging: A Linguistic Theory of Design*. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 387 (Logo Memo 34).

Kaplan, Ronald M., 1973. "A General Syntactic Processor." in Randall Rustin (ed.), *Natural Language Processing*, Courant Computer Science Symposium 8 (December 20-21, 1971), New York, Algorithmics Press, pp. 193-241.

Kay, Martin, 1973. "The MIND System." in Randall Rustin (ed.), *Natural Language Processing*, Courant Computer Science Symposium 8 (December 20-21, 1971), New York, Algorithmics Press, pp. 155-188.

Lesser, V.R., R.D. Fennell, L.D. Erman and D.R. Reddy, February 1975. "Organization of the Hearsay II Speech Understanding System." in *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Volume Assp-23, Number 1, pp. 11-24.

Miller, Mark L., and Ira P. Goldstein, December 1976b. *Parsing Protocols Using Problem Solving Grammars*. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 385 (Logo Memo 32).

Miller, Mark L., and Ira P. Goldstein, December 1976c. *SPADE: A Grammar Based Editor For Planning and Debugging Programs.* Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 386 (Logo Memo 33).

Miller, Mark L., and Ira P. Goldstein, December 1976d. *PAZATN: A Linguistic Approach To Automatic Analysis of Elementary Programming Protocols.* Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 388 (Logo Memo 35).

Newell, Allen, June 1966. *On the Analysis of Human Problem Solving Protocols.* Carnegie Institute of Technology, Preprint of paper presented at the International Symposium on Mathematical and Computational Methods in the Social Sciences, Rome 1966.

Newell, Allen, and H. Simon, 1972. *Human Problem Solving.* Englewood Cliffs, New Jersey, Prentice-Hall.

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

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

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

Paxton, William and Ann Robinson, 1975. "System Integration and Control in a Speech Understanding System." in *American Journal of Computational Linguistics*, Volume 5, pp. 5-18.

Polya, George, 1957. *How to Solve It*. New York, Doubleday Anchor Books.

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

Polya, George, 1965. *Mathematical Discovery* (Volume 2). New York, John Wiley and Sons.

Polya, George, 1967. *Mathematics and Plausible Reasoning* (Volume 1). New Jersey, Princeton University Press.

Polya, George, 1968. *Mathematics and Plausible Reasoning* (Volume 2). New Jersey, Princeton University Press.

Sacerdoti, Earl, September 1975. "The Nonlinear Nature of Plans." in *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, pp. 206-218.

Sussman, Gerald Jay, 1975. *A Computational Model of Skill Acquisition*. New York, American Elsevier.

Woods, William A., October 1970. "Transition Network Grammars for Natural Language Analysis." *Communications of the ACM*, Volume 13, Number 10, pp. 591-606.