Problem Selection in Software Design

by

Steven L. Levin

November 1976

Technical Report #93

This work was partially supported by NSF Grant GJ-36414 and the Department of Information and Computer Science of the University of California, Irvine.

Table of Contents

1.Ø INTRODUCTION

1.1 Motivation

Today, more than ever, the problems of creating computer software are increasing. Software systems are complex, expensive, and have long life cycles. Many years after their design and implementation, costs continue for further system maintenance and modification. Studies by Boehm (1973) have assigned costs to the various stages of software development. These studies reveal that the design step accounts for 4Ø percent of software costs on larger systems.

There have been two major types of research to reduce design costs. The first type develops prescriptions on how to do design. Examples of such work are structured design (Constantine, 1974), structured analysis and design technique (Ross and Schoman, 1976), and design representations (Peters and Tripp, 1976). The second type studies designers at work to understand the cognitive processes of design. Examples of this research are Brooks's model of programmer's coding behavior (1975) and Eastman's study of architectural design (1968). For a survey of design models see (Levin, 1975).

Both forms of research can contribute to better designs and lessened costs. In particular, behavioral studies of

what goes on during design can help identify weak points in how designers work, which prescriptive techniques can either correct or supplement.

1.2 Overview

This paper reports the results of research into the cognitive processes of software design. (*) Design is viewed as a complex activity involving three fundamental processes: selecting problems to work on, gathering needed information for their solution, and generating solutions. A detailed model of the problem selection process is presented.

Problem selection is the process by which designers choose problems on which to work. The order of problems and subsequent design decisions can have an important influence on the completed design (Naur and Randell, 1969; Goos, 1973). Each problem and decision introduces contraints on later problems. Problems considered prematurely during design can produce undesirable features in the finished design. For example, designing the data structure for a program before tackling the problems that define the requirements for the structure can unnecessarily constrain the design.

- - - - - - - - - -

(*) This paper forms a major part of the author's doctoral dissertation.

Introduction

Behavioral processes have been studied in two ways. The most common approach is to study some behavior using large groups of subjects where potentially independent variables are controlled and manipulated. Results are judged by the statistical significance in the controlled experiments. These techniques are best applied when the variables are independent and linear.

The program design process is difficult to study in this manner. Design behavior is very complex and ill suited to controlled experimentation. First, there are too many independent variables for controlled experiments. Second, design variables are highly interactive. Third, large variances in design ability limit subject to subject comparisons.

Protocol analysis (Newell and Simon, 1972) is an experimental methodology for studying complex cognitive behavior. A protocol is a transcription of what is said by a subject who "thinks out loud" while performing a task. By studying a protocol we can infer what information a subject is using, how it is manipulated, and the cognitive processes involved in the task.

Much of the information processing theory of human problem solving (Newell and Simon, 1972) was developed using protocol analysis. Their book, Human Problem Solving,

contains a detailed description of protocol analysis and a defense of its use for studying complex cognitive processes.

The next section presents a model for problem selection in software design. Section 3 describes how the model is represented as a computer program. Section 4 contains an example of how the model operates. Section 5 analyzes the model's performance and Section 6 describes the methodology, verification, and implications of the model.

2.Ø PROBLEM SELECTION MODEL

This section presents a model for the problem selection process in software design.

2.1 Methodology

A model was formulated by studying software designers as they worked. Designers were given the requirements for a program and asked to "think aloud" as they worked out the design. (*) The transcription of what the designer has said is called a protocol. Protocols are the principal source of data for this study. A typical protocol excerpt (from Protocol B2) oppears below.

S146: The question is
S147: how are we going to build the index itself?
A148: [Writes "3) build index"]
S149: Ok

- - - - - - - - - -

(*) The instructions for task 1 are in Appendix A.

S150: Question, term file is apparently not sorted
S151: [Writes "term file not sorted"]
S152: I also notice that a term is one to five words.

Nine protocols, from three designers working on three problems were collected. Three of the protocols were selected for developing the model; the other six were set aside to be used later in testing the model.

Using three of the nine protocols we formulated a model for the problem selection process. The model is based on the author's observations and assertions on how designers use information in problem selection. The model was then expressed as a computer program. We then moved back and forth between our theory, data (the three protocols) and program until we felt that the selection behavior in the protocols was adequately reproduced by the program. How well the model explained problem selection was tested by running the program for the six protocols which had been set aside originally.

2.2 Organization

The model consists of three interacting processes: information collection, problem selection, and solution generation. Only the problem selection process is modeled in detail. The other two processes are represented by information which is inferred from the protocols.

Ideally, a model would directly accept a transcribed protocol and reproduce the desired selection behavior. However, understanding general discourse by computer remains an unsolved problem, and so the meaning of what is being said must be extracted for input to the computer program. (*) The process of inferring information from a protocol is called "encoding" and was done by the author. Encoding reduces sentences like "the system will require an error-handling module" to

Constraint: (system requires error-handling)

for input to the program.

The information-collection and solution-generation processes are represented by information encoded directly from the protocol. In addition, whenever the designer selects a new problem on which to work, the occurrence of a selection, but not what selection was made, is encoded. The principal inputs to the program are a sequence of encodings. Output occurs on selection cues. The model determines, given the current information state, which problem will be chosen by the designer. Figure 1.1 depicts the major steps in modeling a design task.

- - - - - - - - - -

(*) The terms model and program are often synonymous in some contexts here. The model is represented in operation by a program. It makes more sense to talk of "inputs to" and "running of" the program as opposed to the model.

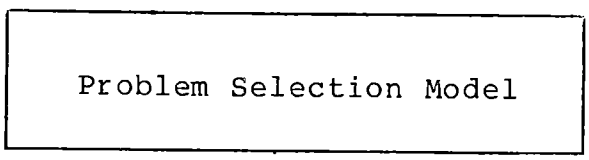
1. Collect a protocol

```
S1: All right.
S2: I guess the first thing I would do
S3: when posed with this problem would be
S4: to attempt to define the class then the
 subclass
S5: that I was going to attack including the ah,
S6: type of editor.
S7: Since you cannot build a text editor to
 include
S8: all of the functions that are available.
                .
                .
                .
```

2. Encode the protocol

```
(STRATEGY (define class-of-editors)
          (define type-of editor))
(SELECTION)
(CONSTRAINT
   (DESCRIPTION number-of functions
 constrained))
(CONSTRAINT
   (DESCRIPTION functions constrained-by
 approachto file))
                .
                .
                .
```

3. Encodings are input to program/model



4. Model reproduces selection behavior of protocol.

Figure 1.1: Major steps in the modeling process

2.3 Design Processes

Earlier we noted that design consisted of three major processes. The information-collection and solution-generation actions of designers have not been modeled. Sections 2.3.1 and 2.3.3 describe design behavior and the information used to represent these processes in the model. Section 2.3.2 presents our assertions concerning selection and their organization into a model of problem selection behavior.

2.3.1 Information Collection

Information collection is the process in which constraint and strategy information is gathered by the designer. Information is obtained by retrieval from the designer's memory or by other mechanisms like deduction and strategy formation.

The term constraint is used to describe any piece of information which may affect the outcome of a design decision. Almost any piece of information a designer introduces in design may be viewed as a constraint. Some typical constraint-like statements in the protocols are:

"So that this will be independent of the choice of a particular implementation language or run-time environment." Protocol A2, lines 22:24 (*)

"Ahmmm, data considerations as we said. Disk files will be packed. And this will be ASCII and includes line numbers." Protocol B1, lines 501:504

Strategies are plans for achieving the solution to a problem. A strategy describes a sequence of activities (problems) which when worked on may achieve a problem solution.

We do not consider the issues of how relevant information is selected for retrieval or how strategies are formed. Both questions are important but form major research questions in themselves. The fact that problem solvers retrieve information from permanent memory is documented by Reitman (1965) and Newell and Simon (1972) and is one of the tenets of information processing theory.

The existence of strategy information is readily observed in protocols of design. For example:

"I guess I could think about how to do the program structure first and then think about data structure or whatever." Protocol B2, lines 79:81

- - - - - - - - - - -

(*) Protocols are identified with a subject/task name. There are three subjects: A, B and C. There are three design tasks: 1, 2 and 3. Thus, subject A on task two corresponds to Protocol A2.

"The first, the next level of design would be to take the functions we have specified and try to put some parameters around them as to how they might be divided up into various areas to concentrate on." Protocol C1, lines 172:178

Each strategy element is a distinct problem which may be explored further by the designer. Strategies may be local (e.g. for the "build index" module, first define the initialization routine and then the module to make new entries) or global (e.g. you must first define the class of editor and then its commands).

2.3.2 Problem Selection

Problem selection occurs whenever it is necessary for the designer to select a new problem to work on. Selection occurs when 1) the problem currently under consideration is solved, 2) the designer explicitly decides to work on another problem, or 3) when the designer "gives up" because no more information can be obtained about the current problem.

Our model makes three assertions concerning this selection process.

1. Local constraints play an important role in problem selection and account for a significant percentage of new problem selections during design. (A local constraint is one that has been introduced only within

the scope of the most recently selected problem.)

2. As strategy and constraint information ages in working storage, the probability that it will be used as a problem source decreases. (*)

3. The required presence of strategies in working storage and prior use of local constraints limits the use of strategies as problem sources.

Assertion 1 states that the selection process is partially driven by the most recently obtained information. Excluding knowledge which represents decisions and plans, the bulk of this information involves constraints about the problem. Constraints are associated with new problems through the designer's store of knowledge. A finding of this research is that a large percentage of problem selection is based on a designer's use of local constraint information and problem associations using such information.

This type of problem selection behavior is characterized by 1) sequences in which a designer moves from one problem to another in a chain-like fashion, and 2) sequences where the problems are not elements of a strategy

- - - - - - - - - -

(*) Working storage (WS) is a small capacity, rapid access memory. It is similar but not equivalent to the conventional information processing theory definition of short term memory.

and do not appear as previously unfinished problems. Recent constraint information is the source for these new problems.

Evidence of this behavior is explicit in designer's work. For example, in Protocol C1:

S56: So some of the edit capabilities
S57: we would want.
S58: So first we would want to be able to look at
S59: what we have put in there.
S60: So we need a function to print or display,
S61: kind of display the file.

In lines 58 through 59 the designer stated the constraint that the text be viewable. The constraint leads to the decision to include a printing command and immediately thereafter to more detailed consideration of how to display text.

Problem selection using local contraints is limited by the recency with which the constraint information was obtained by the designer (assertion 2). In addition, how a constraint is used also depends on how fixed the constraint is for the designer. The more fixed a constraint is (e.g. integers require 36 bits), the less likely it is that the constraint will be the basis for exploring another problem to define that constraint further. Constraints which are vague or loosely defined (e.g. the general approach to text editors is governed by the type of file system) are more likely to cause a designer to select some problem (e.g.

defining the file system) which would provide more information about the constraint. Through this process the designer acquires more information and decision criteria for the problem at hand.

The use of strategies is constrained in assertion 3. First, a strategy must be present in WS to be accessible by the model. Second, strategies become a source of problems only after local constraints have been considered.

Combining both parts of the assertion produces a behavior we have termed strategy interference. This occurs when a strategy that is being followed is interrupted and not resumed. The strategy cannot be continued if it is not present in working storage.

When the constraints for the immediate problem are either highly fixed or yield no problem associations, and when no strategies are available, then the model selects the most recent unsolved problem. If no unsolved problems remain, then the model selects a problem related to constraints evoked in previous problems.

The general notion is that when the designer is not deriving problems by following a strategy, problems are generated within the context of a small body of information. Most of this information consists of constraints and constraint/problem relationships. The order in which

constraints are used in making problem associations depends on their relative recency and level of definition.

2.3.3 Solution Generation

Solution generation describes decision-making by the designer. Sometimes solution generation is invoked immediately upon the selection of a problem. This usually occurs when a solution for the problem is known to the designer. At such times the designer may state a sequence of related design decisions.

At other times the solution process is invoked by the application of one large constraint to a relatively under-defined problem. For example, the problem of designing a text editor is loosely constrained but the decision that it should be like SOS is sufficient to invoke the solution process for describing the entire repertoire of commands and formats of that text editor. (*)

The solution process is characterized by 1) no backup and 2) no statement of alternatives. It is very similar to reading off the recipe for a solution.

The solution process may result in an isolated design decision or a whole sequence of related decisions. Sequences of decisions occur from the retrieval and use of

- - - - - - - - - -

(*) SOS is a text editor on DEC System-10.

solution plans or from using a large body of non-procedural information for the solution under the appropriate constraints.

2.4 Summary of Model

The human information processing theory of Newell and Simon provides the basic framework for this model. Since our use of short- and long-term memory differs slightly from the typical information processing system (IPS) definitions, we have called these memory structures working and permanent storage. A comparison of STM to WS and LTM to PS appears in Section 3.1.

We have described the information the model uses in reproducing problem selection in software design. Constraints, decisions, and strategies are obtained directly from what the designer has said. A set of constraint/problem relationships are inferred for each designer and task. This information constitutes the only input to the model.

The model is expressed as a computer program which when run with the inputs encoded from a protocol reproduces the problem selections made by the designer as observed in that protocol. The model accurately reproduces 70 percent of the problem selections in the nine protocols we collected.

Section 6 contains a detailed analysis of the model's performance.

3.0 PROGRAM REPRESENTATION OF THE PROBLEM SELECTION MODEL

In the previous section we presented a theory of design composed of three major processes. This section describes how the problem selection component of that theory has been represented as a computer program.

The program models observed human behavior to the extent that:

1. The program selects (i.e. chooses to work on next) the same problems as the subjects do.

2. The knowledge state (as represented by the contents of WS and PS) of the program at any one time is the same as that of the subjects (as evidenced by the protocols), and it changes as the design progresses.

Problem selection is modeled using four selection agents. Each agent is a specialist which represents how particular information is used by designers. Most of the design knowledge accessed by the agents is information exhibited by the designers in their protocols. The only other information available to the program is the set of constraint/problem relations which are asserted (by the

experimenter) to be part of the designer's knowledge base.

3.1 More on the Information Processing Framework

The information processing theory of behavior specifies that individuals have access to two memories. The memories are usually referred to as short-term memory (STM) and long-term (LTM) memory, and each has its own characteristics.

Short-term memory has a small capacity and rapid access. The size of the memory has been quoted as from five to twenty "chunks" (Miller, 1956; Simon, 1974; Brooks, 1975). Chunks, or symbols as they are sometimes referred to, may represent structures of arbitrary size and complexity located in the LTM. Access time for the STM is estimated at a tenth of a second. Information in an STM, like data in a cache, is transient due to its high sensitivity to the type of processing taking place.

While the STM has fast access and low capacity, the LTM has comparatively slow access and large, if not potentially infinite capacity (Newell and Simon, 1972). Retrieval times from human LTM order about one fifth to one second with write times of five to ten seconds. Even though information is never lost from the LTM, it may become inaccessible to the designer for lack of the appropriate key under which the information was stored.

Our model does not use these conventional definitions of STM and LTM. To avoid any misinterpretation we have used the terms working storage (WS) and permanent storage (PS) to describe storage (memory) structures similar to but distinct from STM and LTM. WS is used to represent the assertions of information processing theory concerning how much information individuals can readily access at any point in time. The remainder of this section discusses the differences and similarities between WS and STM.

WS and STM are similar in size, access times, and addressing. Both structures have small capacity and fast access times. Additionally, WS and STM are both viewed as fixed size queues. Items entering the queue most recently are those that leave the queue last.

The major differences between WS and STM are in content. STM as defined by Newell and Simon contains chunks which are pointers to information in LTM. The problem selection model uses a WS that contains copies of information found in permanent storage instead of pointers to that information. Information items in WS are not chunks, but are facts (e.g., constraints, problems, strategies). Finally, nothing that falls off the end of WS is lost. Information leaving WS always passes into PS. Forgetting is the absence of information in WS.

The program uses a WS of 11 positions. This number was arrived at experimentally in the definitional stages of the model and is the size of WS at which the program's output has the closest correspondence to the protocols.

3.2 Design Knowledge

When the program runs it has two sources of information. The first is knowledge encoded from the protocol. This includes constraints, strategies, problem statements, and problem selections. The second is constraint/problem information whose availability is asserted. These are the only inputs to the program.

The general form of this information is:

(information-type (desc . value) ... (desc . value))

where the information type may be CONSTRAINT, DECISION, STRATEGY, PROBLEM, or SELECTION. If it is SELECTION then there are no descriptor/value pairs. An example of each information type would be:

```
(STRATEGY
   (DESCRIPTION (define class-of-editors)
                (decide (type-of editor))))

(CONSTRAINT
   (DESCRIPTION pdp-10 has-only sequential files))

(DECISION data-structure will-be linked-lists)

(PROBLEM define editor commands)
```

When this knowledge is represented internally, other descriptor/value pairs are added. Each type of information has a different set of applicable descriptors. For strategies, the descriptors STRATEGY-PTR and STATE are added. The first indicates which element in the strategy was used last and; the latter, whether or not the strategy is currently in use (i.e., the values may be UNUSED, ACTIVE or COMPLETED). For more detail on the knowledge representations used in the program see (Levin, 1976).

3.3 Problem Selection Agents

The problem selection agents represent the basic assertions of our model. Each agent has access to the designer's knowledge state as represented by the contents of working and permanent storage. The model specifies how the selection agents utilize that knowledge in reproducing human selection behavior in software design.

There are four selection agents; each deals with a different type of information. Agents exist for utilizing:

1. local constraints
2. strategies
3. unsolved problems
4. non-local constraints

Agents are used in the order listed above. Problem selection is attempted first using constraints, followed by strategies, unsolved problems, and non-local constraints. The ordering emphasizes the importance of constraint information in problem selection. Agents may succeed or fail in making a selection. When one agent fails, the next agent is tried. When an agent succeeds, the model prints which problem the agent has chosen. This section describes the operation of each selection agent.

3.3.1 Local Constraints Agent

The associative nature of problem selection is represented by the local constraints agent. Local constraints are those that have only been evoked since the last choice of problem. We have found that the number of local constraints almost never exceeds five and that at all times the full set of local constraints about a problem is available in WS.

WS is searched for the most recent local constraint. If there are no local constraints the agent fails. If a local constraint is found, then it is evaluated to determine its usefulness as a new subproblem source. The evaluation uses the function OPERATE. OPERATE evaluates the constraint using a usefulness measure (assigned by the encoder) and the relative recency with which the constraint has been evoked

by the designer. OPERATE computes the sum of the definitional value for the constraint and its position in WS (representing the constraint's recency).

Constraints which satisfy the OPERATE function are then matched against the designer's list of of constraint/problem relations in PS. If a match is made, that problem is selected; otherwise the local constraints agent continues the search in WS for other plausible local constraints. The operation of the local constraints agent is summarized by the metacode representation given in Figure 3.1. In this and other program segments WS is represented as a vector. The most recent element of WS would be WS(1) and the least recent, WS(n).

```
procedure local-constraints-agent;
  begin
    problem-flag:=true;
    i:=1;

    ! Search through WS for a local constraint;

    while problem-flag and i<=length(ws) do

        ! If the constraint is local and has a problem
          associated with it then select it;

      if ws(i) is a local constraint
        and operate(ws(i))
        and assoc-con(ws(i))
          then
            begin select(assoc-con(ws(i)));
                  problem-flag:=false
            end
          else i:=i+1
end
```

Figure 3.1: The Local Constraints Agent

The function ASSOC-CON simulates the process of finding a problem related to the given constraint. ASSOC-CON attempts to match the given constraint with the constraint/problem pairs in PS. ASSOC-CON is a table-lookup procedure using pattern matching. For example: given the constraint

(functions constrained-by (approach-to file))

in Protocol A1 the function ASSOC-CON returns

(determine (type-of operating-system))

resulting from the match with the constraint/problem relationship (files (type-of operating-system)).

3.3.2 Strategies Agent

A strategy is a sequence of problem actions. The strategy agent recognizes the statement of such knowledge by the designer and then supervises the selection of problems from the list of strategy elements. The agent determines the availability of a strategy by searching the WS. If no strategy is present then the strategy agent fails and the next selection agent is activated.

The first strategy encountered in the WS is activated if its STATE value is UNUSED. The strategy is made ACTIVE (by setting the value of the descriptor STATE to ACTIVE) and

the first problem element of the strategy is selected as the next problem. When an element of the strategy is chosen, then the STRATEGY-PTR is incremented by one. Attempts by the agent to used a completed strategy cannot succeed because the STRATEGY-PTR cannot be incremented beyond the number of elements in the strategy. When such an attempt occurs the state of the strategy is changed from ACTIVE to COMPLETED.

To be usable by the designer, the strategy must be accessible in WS. A strategy enters WS when it is first stated by the designer; it is subsequently returned to the front of WS (rehearsed) whenever an element of the strategy is chosen as the next problem. If strategies are not rehearsed then they are prematurely lost by the model. Strategies can leave the WS and thus be "lost" by either their completion or failure to be rehearsed. The latter case arises when the designer pursues a sequence of subproblems which introduces new problems and constraints in WS such that the original strategy is pushed out. Figure 3.2 is a program description of the strategies agent.

```
procedure strategies-agent;
  begin
    strategy-flag:=true;
    i:=1;
    while strategy-flag and i<=length(ws) do

      ! Search through WS for a strategy;

      if ws(i) is not a strategy or is a completed
        strategy then i:=i+1 else

          ! Test if strategy has been used yet;

          if strategy-ptr is unused then
            begin
              strategy-ptr:=1;
              select first element of strategy;
              strategy-flag:=false
            end else
              if strategy-ptr = length(strategy)
                then
                  begin
                    set strategy to completed;
                    i:=i+1
                  end
                else

                  ! Select next element of the strategy;

                  begin
                    strategy-ptr:=strategy-ptr + 1;
                    select strategy
                     element(strategy-ptr);
                    strategy-flag:=false
                  end
  end
```

Figure 3.2: Strategies Agent

3.3.3 Unsolved Problems Agent

This agent searches WS and then PS for an unsolved problem. WS is searched for the most recent problem whose state value is less than three. (This state value was found

to produce the best program selection of unsolved problems.) If such a problem exists, it is selected. Figure 3.3 is a metacode description of this agent.

```
procedure unsolved-problems-agent;
  begin
    problem-flag:=true;
    i:=i+1;

    ! Search for an unsolved problem;

    while problem-flag and i<=length(ws) do

      ! if problem is unsolved, select it;

      if ws(i) is a problem and its state is < 3
        then
          begin
            select ws(i) as the next problem;
            problem-flag:=false
          end
        else i:=i+1
  end
```

Figure 3.3: Unsolved Problems Agent

When there are no unsolved problems in WS, PS is searched using the criteria outlined above.

WS and PS are represented as simple lists. The actual organization of knowledge in human memory could be very different from that used in the program. Knowledge might be organized by the difficulty of the problem or with the constraints associated with that problem. Without any evidence to the contrary, we have chosen one convenient possibility.

3.3.4 Non-local Constraints Agent

non-local constraints are those that have appeared in the context of some previous problem but not within the scope of the most recently selected subproblem. Use of this agent represents a form of information backtracking in that the designer's efforts to solve some previous sequence of subproblems have failed. It is now necessary to go back and gather more information through development of problems which were previously unexplored. These problems might be characterized as those that are increasingly tangential to the original problem sequence. The designer discovers these "new" problems by returning to previously evoked constraints.

This agent is similar to that for unsolved problems because appropriate constraints are first searched for in WS and then in PS. Since the treatment of constraint information is the same for both memories, one explanation will suffice for both cases.

WS is searched for an active non-local constraint. Remember, the local constraint agent has already considered active local constraints. A constraint is active if the problem for which it was generated has not been solved. If such a constraint is found it is evaluated using the OPERATE function which was described in Section 4.4.2. Applicable

constraints are then matched with memory to produce an associated problem, if one exists. The agent fails when there are no local constraints, when none of the constraints meet the criteria imposed by the OPERATE function, or when no associated problems can be found. Figure 3.4 is a metacode description of this agent.

```
procedure non-local-constraints-agent;
  begin
    constraint-flag:=true;
    i:=0;

    ! Search for a constraint in WS;

    while i<=length(ws) and constraint-flag do

      ! If the constraint is local, active and has an
        associated problem, then select it;

      if ws(i) is non-local
        and active
        and operate(ws(i))
        and assoc-con(ws(i))
          then begin select(assoc-con(ws(i)));
                     constraint-flag:=false
               end
          else i:=i+1
  end
```

Figure 3.4: Non-local Constraints Agent

3.4 Program Organization

The encoded information of a protocol is the driver for the problem selection program. Encoded information is sequentially read (from a file) by the program's highest level procedure. This procedure acts as a dispatcher by

invoking a processor corresponding to whatever type of information has been input.

When a problem selection occurs in the protocol, a cue (SELECTION) in the program input causes the selection processor to be invoked. This process is composed of the four selection agents which we have previously described. Selecting a new problem changes local constraints to non-local. A new unsolved problem then enters WS.

The program (see Figure 3.5 for the general organization) continues execution until the input file is exhausted.

Each information processor updates the program's representation of the designer's current knowledge state (i.e., the contents of WS and PS). When the decision processor is invoked, it updates appropriate problem solution states in WS and PS. Constraints which arose because of a solved problem are marked as INACTIVE.

4.Ø DEMONSTRATION OF THE MODEL

The verbal protocols used in testing the model vary in length from a minimum of 36 minutes to a maximum of 78 minutes. Protocol A2, the shortest protocol, has 83 pieces of encoded information and produces 12Ø3 lines of detailed trace information. Due to space limitations we will present

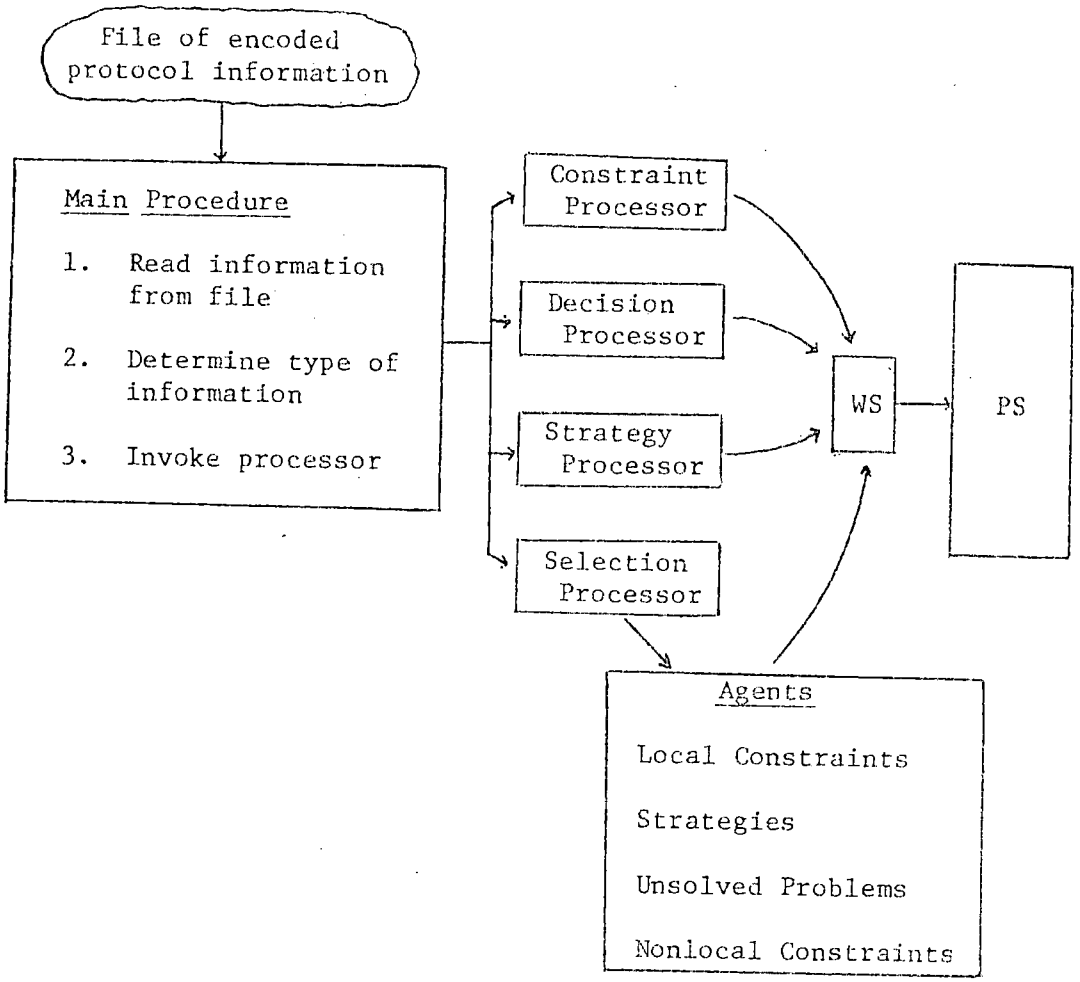


Figure 3.5: General Organization

segments of several protocols which illustrate the general nature and properties of the model.

Before proceeding with the examples of the model's operation we describe the collection and characteristics of our database.

4.1 The Database

4.1.1 The Experimental Setting

Each protocol was collected in a continuous session which commenced when the observer (the author) provided the subject with a set of written task instructions and ended when the subject determined that the design task was completed. The actions of the subject were videotaped to provide a visual as well as audio record of the subject's action. Transcription of the tapes was done by the author. The protocol transcripts included the verbalizations of the subject and any other action that took place during the session. These actions included writing, diagramming, pointing done by the subject to written materials, as well as interventions by the observer to ask the subject to speak or to answer questions.

The protocols were collected with each subject seated at a desk in an office similar to those in which the subjects normally worked. Subjects were provided with paper and used their own writing instruments. A subject would be given written general and specific task instructions and asked to begin.

The general instructions did not prohibit the subject from asking questions of the observer. When the question concerned specifications, the subject was told to make any necessary assumption. For questions about the amount of detail necessary, subjects were reminded that the design would be implemented by another person.

For the nine protocols there were 11 instances where the observer intervened to ask the subject to speak or enunciate more clearly. This averages only 1.22 interventions per protocol. There were five instances where the subject asked a question of the observer. Two of these occurred in Protocol C1 and one each in Protocols A1, A2 and B2. These interactions did not seem to affect the subject's performance.

4.1.2 The Subjects

The data used for verifying the model was provided by using protocols for three subjects, each of whom performed three design tasks. All of the subjects were graduate students in computer science. Subjects differed significantly in design experience and general expertise, i.e., skill at design. The following assessments of backgrounds and skill were derived from conversations with the subjects and with faculty who were familiar with each subject's work.

Subject A was a fourth year graduate student who was considered to have exceptionally high ability in design and programming. This subject was very involved with both small- and large-scale programming projects dealing with applications and systems work.

Subject B was a second year graduate student who possessed average skills in both design and programming. The subject was known to have programmed a variety of application programs but whose experience was limited to academic areas.

Subject C was a first year graduate student who had several years experience working for a large computer corporation in the role of an applications analyst. The subject was experienced in the programming of applications in COBOL but not familiar with other types of applications outside of the business domain. The subject had very little experience with time-sharing systems.

4.1.3 The Protocol Tasks

The data base for testing the selection model consisted of nine protocols, three subjects on three different tasks. Table 4.1 supplies the statistics regarding the length (in time and lines), the number of information items encoded and the ratio of lines to encoded information for each protocol.

Protocol	Time (minutes)	Length (lines)	Encodings	Lines/ Encodings
A1	55	1152	131	8.79
A2	36	352	83	4.24
A3	41	528	75	7.04
B1	78	988	136	7.26
B2	62	1022	98	10.42
B3	44	541	71	7.61
C1	50	512	89	5.75
C2	40	396	67	5.91
C3	51	707	89	7.94

Table 4.1: Protocol Characteristics

The variance in the ratio of protocol lines to information encoding is due to differences in the amount of writing activity and the verbosity of the subjects (some subjects are much more elaborate in describing why they make particular decisions).

The three tasks were:

1. Design a text editor for the PDP-10.
2. Design a book-indexing program.
3. Design an order-entry system.

Two of the tasks were designs of on-line systems. The third design was to operate in a batch environment. Design decisions in the tasks covered command interfaces, system structure, coding, data structures, and system performance. These tasks were chosen because they could be completed in less than two hours (at least for the level of design required of the participants) and because each subject was

familiar with at least one of the three task domains. That is, each subject had some experience or knowledge that could be brought to bear on all of the tasks. The task instructions are presented in Appendix A.

4.2 General Operation of the Model

In demonstrating the model's operation we have tried to maintain a close correspondence between the descriptive trace produced by the program and the notation used in this chapter. At times we have taken the liberty to "unravel the onion skin" format that characterizes the way the information has been represented in the program's LISP implementation.

The initial state of the model has WS "empty" and PS containing the constraint/problem pairs described in the next section. WS is empty in the sense that whatever information occupies its eleven positions is not relevant to the design task which has not yet commenced. After WS becomes full, new information which is entered at the front of WS pushes older information out of WS onto a list that represents PS. Strategies that leave WS are presumed forgotten by the designer in this model.

4.3 Constraint/Problem Associations

The program is run with the encoded protocol information and a list of constraint/problem associations. Such associative pairs are asserted to be a part of each designers knowledge. This list of items does not represent an assertion about the representation of such information in human memory.

None of the associations we supply are particularly surprising and it is certainly reasonable that the subject would have such knowledge. It is one of the findings of this research that these associations coupled with the other mechanisms of the model can explain a large percentage of the problem selection that takes place in protocols. The association pairs in Table 4.2 are those used in the section of protocol that we will describe next.

Constraint Object	Related Problem
files	type of operating system
text representation	data structures
data	type of files
speed	performance
i/o time	data formatting
modification times	data structure modification times

Table 4.2: Association Pairs for Protocol A1

4.4 Subject A on the Text Editing Task

Protocol A1 is the text editor design and was the first task given to subject A. It has been chosen for a detailed presentation because in the first twenty minutes of the protocol, most of the the model's operation is illustrated. This protocol is typical of the length of time, 55 minutes, and the number of selection events, 25, as given by the average times over all protocols. Its 1040 lines are also comparable to those of the other subjects on this task.

Part of the data (the transcribed protocol and encoding) and program output for Protocol A1 is presented in the appendices. The complete data for Protocol A1 is available in (Levin, 1976).

4.4.1 Operation of the Model for Part of Protocol A1

The subject begins immediately by stating a strategy for what to do when posed with this type of problem. This occurs in lines 2 through 6 of the protocol as:

S2: I guess the first thing I would do
S3: when posed with this problem would be
S4: to attempt to define the class then the subclass
S5: that I was going to attack and then,
S6: type of editor.

The input (i.e., the encoded information) to the program for these lines is:

```
(STRATEGY (define class-of-editors)
          (define type-of-editor))
```

This input is processed by the strategy module which adds the strategy to WS. At this point the WS appears as:

```
1.    strategy
        desc: (define class-of-editor)
                (define type-of-editor)
        strategy-ptr: Ø
        state: inactive
2.    empty
3.    empty
        .
        .
        .
11.   empty
```

Figure 4.1: Working Storage After Line 6

The value for the descriptor STRATEGY-PTR indicates the most recently selected element of the strategy. Initially this value is always zero. Similarly, the initial state of the strategy is INACTIVE. The strategy has been stated by the subject but not yet acted upon. The program then goes on to accept the next input.

The next input is:

(SELECTION)

This is a cue to the program that it should invoke the selection components of the model to predict what will be the next problem on which the subject will work. The program uses the four strategy agents in turn, attempting to find a problem using local constraints first, strategies second, unsolved problems third, and non-local constraints fourth.

In this case, WS contains no local constraints, so the local constraint agent fails. The strategy agent is then tried. WS is searched sequentially for a strategy whose STATE is either INACTIVE or ACTIVE. If the STATE is INACTIVE then by definition the strategy is unused as yet. The strategy agent selects the first element of the strategy to be the next problem and updates the STATE value to ACTIVE to indicate the current use of this strategy. STRATEGY-PTR is incremented by one. Before the actual selection of the next problem the strategy element in WS is rehearsed. (*)

Even though the strategy agent succeeds in finding a problem, the model's actions are not complete. The program queries the operator's console to check if this is the correct problem. The program does make wrong selections. If a correct selection has been made, the program is instructed to proceed. If the program is wrong, it must be corrected. Later on we will examine a case where the model makes the wrong selection and must be corrected.

Since a new problem has been selected all constraints whose TYPE value was previously LOCAL are changed to non-local. The new problem is added to WS which now appears as:

- - - - - - - - - -

(*) Rehearsal of an item in WS causes that item to move to the front of the WS list. For strategies, any time an element of a strategy is selected, the entire strategy is rehearsed.

```
1.    problem
        desc: (define class-of-editor)
        source: strategy
        state: Ø
2.    strategy
        desc: (define class-of-editor) (define
                type-of-editor)
        strategy-ptr: 1
        state: active
3.    empty
4.    empty
        .
        .
        .
11.   empty
```

Figure 4.2: Working Storage After Line 6

Notice that the new created problem has a STATE value of zero. This indicates that the problem is currently unsolved.

The next two inputs are constraint statements that occur in lines 7 through 9 and 1Ø through 11.

```
S7: Since you cannot build a text editor to include
S8: all of the functions that are available,
S9: in text editors
S1Ø: in that some of then have to do with basic approach
S11: to the file that one takes.
```

These lines in the protocol are encoded and entered as:

(CONSTRAINT number-of-functions constrained)

(CONSTRAINT functions constrained-by approach-to file)

Both inputs are handled by the constraint processor which results in their addition to WS. The current appearance of WS is given in Figure 4.3.

```
1.    constraint
        desc: (functions contrained-by approach-to
                file)
        type: local
        state: active
        source: (define class-of-editor)
2.    constraint
        desc: (number-of-functions constrained)
        type: local
        state: active
        source (define class-of-editor)
3.    problem - define class of editor
4.    strategy - general editor strategy
5.    empty
6.    empty
        .
        .
        .
11.   empty
```

Figure 4.3: Working Storage After Line 11

Both constraints are LOCAL and are associated with the problem of defining the class of editor. This information is retained so that when the problem of defining the class of editor is solved these constraints can be identified and changed to INACTIVE. Notice that we have not listed all of the features for the rest of the items in WS. For the remainder of this chapter we will only include details that are relevant to the model's operation at the current point.

The next input is a selection cue and it corresponds to line 12 of the protocol. As with every activation of the selection model, the local constraints agent is tried first. In this case the agent finds a constraint in WS that is both LOCAL and ACTIVE, e.g., (functions constrained-by approach-to file). The program then searches WS for an

association that pairs that constraint to a problem. A match is found with the association item (file typeof operating-system). As before, previously LOCAL constraints are now changed to non-local and the new problem is added to WS. All of the action we have described has occurred on the basis of the first 12 lines of the protocol.

The problem the subject is working on is identified in lines 12 through 16.

S12: Ahmmm. Okay
S13: One basic decision is that the sort of operating
S14: system that it is going to be interfacing to.
S15: Ah. There is at least four avaiable for
S16: the PDP-10 that I know of.

Lines 17 through 32 contain four constraints which are the next inputs received by the program. In this section of the protocol the subject is discussing the different types of operating systems and the files available under them on the PDP-10. The four constraints are added to WS.

Following the statement of the four constraints is an input which indicates a decision on the part of the subject. The decision is from lines 32 through 33 of the protocol and is interpreted as a decision to use virtual addressing or assume that virtual addressing (and memory operations) will be available in the operating environment for which the text editor is being designed.

S32: would be to use the virtual memory capabilities
S33: of the most of the operating systems

These lines are encoded as input in the form:

(DECISION use virtual-memory typeof
operating-system)

This is the first time that a decision input has been received and so we will describe the subsequent actions of the program in detail.

The decision statement is added to the WS and then any problems in WS or PS solved by the decision have their STATE values updated. The value five is assigned to the STATE descriptor and it represents at least a temporary, if not final solution to that problem. For each problem that is "solved" by the decision, then any constraints that were generated as LOCAL to that problem are now marked as INACTIVE. Figure 4.4 shows WS after the updating caused by the decision to assume a virtual memory operating system.

```
1.  decision
      desc: (use vm typeof operating-system)
2.  constraint
      desc: (vm use is efficient)
      state: inactive
      type: local
      source: (determine typeof operating-system)
3.  constraint
      desc: (most PDP-10s have vm)
      type: local
      state: inactive
      source: (determine typeof operating-system)
4.  constraint
      desc:  (PDP-10 hasno structured files)
      type: local
```

```
          state: inactive
          source: (determine typeof operating-system)
5.    constraint
          desc: (PDP-1Ø hasonly sequential files)
          type: local
          state: inactive
          source: (determine typeof operating-system)
6.    problem
          desc: (determine typeof operating-system)
          state: 5
          source: constraint
7.    constraint
          desc: (functions constrainedby approachto file)
8.    constraint
          desc: (numberof functions constrained)
9.    problem
          desc: (define class-of-editor)
1Ø.   strategy
          desc: (define class-of-editor)
                    (define typeof-editor)
11.   problem
          desc: (design text-editor)
```

Figure 4.4: Working Storage After Line 33

The four constraints that were generated LOCAL to the problem of determining the type of operating system are now marked as INACTIVE. In addition, the STATE of the solved problem has been changed. Notice also that as future items are added to WS, then older information will be pushed into PS.

The subject continues with the next input which is derived from lines 35 and 36 where the decision is made to bring all of the text in core and work with it there. As before, both WS and PS are searched for problems that would be solved, but in this case there are none.

An example of a problem statement then occurs in lines 42 through 44 where the subject says:

S42: not paying attention to i/o
S43: except sequentially to bring the
S44: file in and to write it out at the end.

These lines are encoded as an expression by the subject indicating an awareness of some problem yet to be solved, that of managing file input and output. The item (PROBLEM-STMT management of file-io) is added to WS. This action does not represent problem selection but rather problem recognition.

Immediately following the problem statement another selection cycle occurs. The problem predicted is to (determine typeof editor). How is this choice made?

The local constraints agent is tried and fails because there are no constraints in WS that are both LOCAL and ACTIVE. The constraint (functions constrainedby approachto file) that fired off the previous problem is not re-used because it has been inactivated by the decision regarding the type of operating system. A problem is selected because the strategy agent finds an active strategy in WS with some unused strategy elements remaining. The next element of that strategy, (determine typeof-editor), is selected and the problem added to WS.

At this point in the protocol the subject proceeds to make a number of statements concerning various possibilities for the type of editor. These are followed by a number of decisions, the most important of which is to build a TECO-like editor. (*) These decisions take place in lines 69 through 73.

S69: My own preference is toward a TECO like
S70: editor with perhaps
S71: extensions for a few of the common facilities
S72: that I use.
S73: The you know, like macro facilities

Processing of these decisions results in changing the solution state for the problem of determining the type of editor.

Rather unexpectedly in lines 75 through 76 of the protocol the subject says:

S75: Let's see,
S76: the first thing is file i/o

We have coded this as an instance of problem selection. What does the model predict?

Both the local constraints agent and the strategy agent fail, the former because there are no local constraints and the latter because the only existing strategy has been finished. In fact, it is when the strategy agent examines the strategy during this selection cycle that it is marked

- - - - - - - - - -

(*) TECO is another text editor available on the PDP-10.

as COMPLETED. Now the unsolved-problems agent is invoked.

Starting with the most recent information in WS the unsolved-problems agent searches sequentially for an unsolved problem (i.e., a problem whose STATE value is 3 or less). If such a problem is found then it is the next predicted problem. Of course, the current problem that is being worked on is unsolved and must occur before any other unsolved problems. The unsolved-problems agent is programmed to ignore this problem to prevent the model from cycling in this situation. This provision caused the model to be in error four times when the subject was actually repeating the same problem.

4.4.2 A Summary of this Example

In this section we have shown how the model's information processing components and selection agents follow the same selection behavior as that found in the protocol. The description in Section 4.4.1 of Protocol A1 represents approximately 80 lines and 19 minutes of that protocol. In it, three of the four selection agents are used. The use of the non-local constraints agent is described in the next section.

The model's use of several different types of information encoded from the protocol has been described and

how that information is processed by modules of the program illustrated. These inputs have represented strategies, constraints, and decisions.

Among the features of the program that have not been discussed are: what is done when the program makes the wrong prediction and how does the program model the interaction of strategies and constraint-driven behavior? These questions are considered separately in the next two sections.

4.5 Incorrect Predictions

The protocol excerpt in the previous section contained four problem selection events for which the model correctly predicted all four. However, the model does make wrong predictions, and at those times must be corrected to continue replicating the protocol. If the program is not corrected, it will continue selecting problems. However, now the program's knowledge state differs from the designers and the program's output can no longer be compared with the protocol.

Incorrect predictions are caused by errors in a protocol's encoding and from selection behavior that cannot be explained by the model. The former case is illustrated by the model's operation for the next few minutes of Protocol A1.

After making the correct prediction that the subject would begin work on the problem of file input and output, the designer elaborated a few more constraints and then switched to a new problem. At this point, line 93, the model predicted the next problem would be "(determine typeof operating system)" when actually it was a continuation of the problem concerning the type of editor.

The error is in our original coding of the lines 69 through 73. If those lines are recoded to be some form of constraint on the type of editor then the program will make the correct selection for this case. We assumed that a decision had been reached concerning the type of editor. Consequently, "determining the type of editor" is no longer an unsolved problem and the program must find the next unsolved problem (if one exists) or attempt to find a problem via some active, but non-local constraint.

Because the model operates from a fixed sequence of inputs it does not make sense to continue running the program without first correcting selection errors. The next set of inputs is bound to the original protocol in which some other problem was actually chosen. Allowing the program to run uncorrected creates the situation where the sequences of constraints, decisions, and strategies are inappropriate to the problems chosen.

When the model makes a prediction, the operator's console is queried to find out whether or not this was the correct prediction. On occasions when the prediction is wrong, the operator can enter the correct problem and then alter the model's current information context to correspond to the choice of that problem. In effect, whenever a wrong choice is made, the model is manually reset and put on the right track. The statistics on correct and incorrect predictions are presented in Section 5.1.

Another source of error also related to coding, arises from the linguistic limitations of the program. The program finds matching decisions, problem associations and other pieces of information by matching patterns. The decision "(use teco-like typeof editor)" would not match the problem "(determine type-of editor)" or "(determine type of editor)" because of the differences between typeof, type-of or type of. Issues in protocol encoding are discussed further in (Levin, 1976).

4.6 Illustrating Strategy Interference

The section of protocol we examine next illustrates a phenomenon of strategy and subproblem interference which we have found to occur in the design behavior of the three subjects.

It is an assertion of the model that strategies are only accessible when they are in WS and that problems generated through associations with local constraints take precedence over the use of strategies in controlling the selection process. When the model operates with those assertions, it accounts for the behavior that occurs when a designer who is seemingly carrying out steps in a strategy becomes engrossed in a series of subproblems and does not continue with the rest of the strategy after finishing with the subproblems.

The next protocol segment shows that the coupling of those assertions results in active strategies being forced out of WS. The consequence is that the remaining unfinished elements of the strategy are no longer accessible.

4.6.1 Operation of the Model for Protocol B2 Lines 79:147

In this protocol the subject was asked to design a program to create book indices. Like other protocols from this subject, it is of slightly longer duration and length than protocols from the other two subjects.

At lines 79 through 81 in the protocol the subject explicitly outlines the next two steps of the design

S79: I guess I could think about how to do the program
S8Ø: structure first and then think about data structure
S81: or whatever

This is formulated as the two element strategy:

```
(STRATEGY (determine program structure)
      (determine data-structures))
```

The subject begins carrying out this strategy in the next line. This is indicated by:

S82: Lets take a look at some of the basic program
S83: stuff

The model reproduces this behavior by correctly predicting the next problem is that of program structuring.

In the next 63 lines of protocol the subject states three decisions, five constraints, and two problem statements. These new information items, in addition to the already present current problem, results in the previous strategy being pushed out of WS. When the next selection cycle takes place the model makes the correct prediction. The next problem is not that of data structures but arises from the recent problem statement of what modules should constitute the design. Work on the new problem begins at line 146 with the comment:

S146: The question is,
S147: how are we going to build the index itself?

The exact effects of strategy interference are difficult to assess because it occurs only 8 times over the nine protocols. Strategy interference occurs at least once for each subject but does not occur in all nine protocols.

5.Ø ANALYSIS OF THE MODEL

In the previous two sections we have described the model's organization and shown how its output corresponds to the behavior in the protocols. In this section we will evaluate how well the model explains the protocols and show evidence for the assertions made by the model.

5.1 How Well Does the Program Explain the Protocols?

Evaluating a cognitive model is a difficult task. Programs of this type emphasize the detailed reproduction of behavior rather than the prediction of new behavior. Thus, the criteria for judging a psychological model is how well the model's output matches the behavioral data that was observed. An evaluation is an objective measure of how well the program explains the subject's behavior.

Evaluating a model involves two steps. First, we must establish a mapping between the subject's behavior and the program's output. Second, a measure must exist for the mapping. There are several problems inherent in creating the mapping and its measure.

Both the program and the subject are complex objects. While the program is complex, it can be examined and described at almost any level that we wish. However, descriptions of the subject must be made from the limited

amount of information that is observable in the protocol. Ideally, we would like to construct a complete mapping at all levels of detail, but the data to do this is unavailable. Comparisions must be made with a partial mapping using the information observable in the subject's protocol.

What if a total mapping could be made? Moran (1973) has said, "It is practically impossible to assign a measure of completeness because of the great amount of interdependence among all parts of the systems. Most measures are additive and assume independence among the parts being counted. In other words, a comparison between complex systems can not be reduced to a simple statistic." (*) With these issues in mind, we will describe the measure we have used and its application to the data.

5.1.1 Measuring the Correspondence of Major Events in the Protocol with the Model

Some of the features that might be used in comparing behavior in the protocols with the model are: (a) the timing of different actions in the protocol, (b) the wordings used to express actions, and (c) the correspondence of actions in

---

(*) Thomas B. Moran, "The Symbolic Imagery Hypothesis: A Production System Model," Doctoral Dissertation, Department of Computer Science, Carnegie-Mellon University, (December 1973), p. 122.

the protocols and the program. Actually, only the last measure is feasible.

Timing must be excluded as a measure for three reasons. First, we can not deduce from the data how much human processing time is devoted to decisions, selections, or information gathering. Second, even if we could determine the subject's processing times, we have not modeled all of the processes involved with design, only that of problem selection. Third, relative processing times of the program for the different conceptual agents are a function of the program's knowledge representations. We have not claimed that these representations are the same as those used by humans and therefore can not impart any significance to such timings.

Reproducing the actual verbalizations of the subject is not possible because the program only models selection actions (which are generally nonverbal) and because the model does not have a linguistic component. It should be increasingly obvious that any mapping we may produce will correspond to a comparatively high level of design activity.

The third mapping was between the programs actions and those in the protocols. The model is supplied with inputs that represent when decisions, constraints and selections take place. One useful measure of this mapping is the

frequency with which the model makes correct problem selections.

5.1.2 Measuring Correct Predictions

Any measure of the number of correct predictions made by the model depends on the criteria used in determining instances of problem selection. Those measures are only meaningful when we have corrrectly identified and encoded the principal selection behavior in the protocol. (*)

The reliability of the coding rules was tested by having an independent judge encode all of Protocol B2. Training the judge to do encoding required approximately 30 minutes. During this time the experimenter explained the written coding rules and went over a previously encoded protocol with the judge. Additional training time involved studying examples of protocols encoded by the experimenter and reading a set of encoding rules and examples (Levin, 1976). These times do not include the years of experience in designing programs which enabled the judge to understand the content and actions represented by the verbal protocols. The judge was a professional computer scientist with a doctorate in computer science and previous training in psychology. The judge had previous experience in analyzing

- - - - - - - - - - -

(*) See Section 5.3 for an evaluation of the encoding process.

verbal protocols.

In the coding of selection behavior, there was an overall match of 91 percent for the identified selection points. The remaining 9 percent involved protocol segments which the experimenter had previously identified as being especially difficult to interpret. Only 5 percent of the selections correctly matched were classed as difficult to identify. Validating the encoding using an unbiased encoder provides some assurance that the model's output does not result strictly from biases of the principal encoder (who is also the model builder).

The model's performance can be compared with behavior (as observed in the protocol) by measuring the percentage of correct problem selections predicted by the model. The percentage is a ratio of the number of correct predictions with the number of selections that take place. Table 5.1 reports the number of selection events, the number of correctly predicted selections, and the resulting percentage for the nine protocols.

Protocol	Number of Selections	Correct Selection	Percentage Correct
A1	25	17	68.Ø
A2	15	9	6Ø.Ø
A3	15	11	73.3
B1	45	3Ø	66.6
B2	24	14	63.6
B3	16	11	68.7
C1	18	14	77.7
C2	14	1Ø	71.4
C3	16	12	75.Ø

Table 5.1: Selection Frequencies

The average percentage of correct selections over all nine protocols is 69.3. How good is this performance?

One way to assess this result is to consider the percentage of correct predictions if random selection were used. Consider that at each point where the model makes a prediction there exists a space of possible problems from which one is chosen. The space is formed from problems generated by other possible constraints, strategies, and unsolved problems. If the model were to select from this problem space at random, then what would be the percentage of correct selections given that only one problem in that space is the correct choice?

If there were only two problem choices each time the model made a prediction, then random selection would account for a 5Ø percent success rate. This extreme case illustrates that under the appropriate conditions a

respectable result can be produced from a strictly random decision process.

To show that the model performs significantly better than it would if random choice were used, a measure of random selection was computed for each protocol. This measure was computed by summing the probability of a correct selection for each instance, and dividing that sum by the number of selection instances. The data and computation of this measure for Protocol C1 is given in Table 5.2.

The count of possible problems is the total of unsolved problems and the number of problems potentially arising from available strategies and constraints (local and non-local).

	Number of Possible Problems	Probability of Correct Selection
Selection 1	1	1.0
Selection 2	3	.333
Selection 3	5	.199
Selection 4	7	.142
Selection 5	5	.199
Selection 6	7	.142
Selection 7	6	.166
Selection 8	11	.090
Selection 9	11	.090
Selection 10	8	.125
Selection 11	9	.111
Selection 12	10	.099
Selection 13	17	.058
Selection 14	17	.058
Selection 15	17	.058
Selection 16	17	.058
Selection 17	17	.058
Selection 18	17	.058
		3.045

Average Probability of a Correct Random Selection = 3.045/18 = .1698 = 17%

Table 5.2: Random Selection in Protocol C1

If random selection were used in all nine protocols then the prediction rate is 12.72 percent. The actual percentage of correct selections made in the model is 69.3 which shows that the model's output is significantly better than would be accounted for by random choice. The selection percentages for each protocol is given in Table 5.3.

Protocol	Random Selection	Selection Using the Model
A1	9.Ø	68.Ø
A2	12.5	6Ø.Ø
A3	14.5	73.3
B1	4.6	66.Ø
B2	17.5	63.6
B3	13.7	68.7
C1	16.9	77.7
C2	11.3	71.4
C3	14.5	75.Ø

Average using random selction = 114.5/9 = 12.72

Average using the model = 623.7/9 = 69.3

Table 5.3: Selection Percentages

5.2 Support for the Assertions Made by the Model

The argument that the model both matches observed data and is a good predictor is insufficient for the model's acceptance. A necessary condition for a valid model is that the mechanisms of the model are consistent with other psychological data and findings. The mechanisms in the program are equivalent to three assertions that are made by the model. Evidence for the plausibility of the assertions is presented in this section.

The model makes three assertions concerning the selection process:

1. Local constraints play an important role in problem

selection and account for a significant percentage of the new problem selections that occur during design.

2. As strategy and constraint information ages in working storage, the probability that it will be used as a problem source decreases.

3. The required presence of strategies in working storage and prior use of local constraints, limits the use of strategies as a problem source.

5.2.1 The Assertion about Local Constraints

The model asserts that local constraints are a significant source of new problems during design. One way of investigating this assertion is to ask what sources other than local constraints could be used for problem selection?

One possibility might be that each designer has a large body of stored knowledge about different problems. Problem selection could involve explicitly choosing from that set. Selection could occur at random or using some set of rules. Random selection appears unlikely because of obvious patterns that designers appear to follow. More structured selection, such as choosing the most constrained problem, is not observed in the data.

Another alternative source of new problems is from strategies. Analysis of the protocols, independent of the model's output, shows strategies are used in that manner. (*) However, after we account for problems generated by strategies and those through "creativity," a great many problems remain. Thus, local constraints appear to be a plausible source for many of these problems.

From the model's output we can determine how different information is used in problem selection. Local constraints are used for 23.4 percent of all problem selections. When only new problem selections are considered, local constraints account for 33.8 percent of the selections. Strategies account for 29.2 percent of the selections (see Table 5.4). This data supports the assertion.

- - - - - - - - - - -

(*) See the examples in Section 4.4.

Protocol	Local Constraints	Strategies	Unsolved Problems	Incorrect Predictions
A1	32.Ø (8)	2Ø.Ø (5)	2Ø.Ø (5)	28.Ø (7)
A2	33.3 (5)	13.3 (2)	13.3 (2)	4Ø.Ø (6)
A3	46.6 (7)	2Ø.Ø (3)	6.6 (1)	26.6 (4)
B1	13.3 (6)	35.5 (16)	17.7 (8)	33.3 (15)
B2	12.5 (3)	2Ø.8 (5)	25.Ø (6)	41.6 (1Ø)
B3	31.2 (5)	25.Ø (4)	12.5 (2)	31.2 (5)
C1	22.2 (4)	5Ø.Ø (9)	5.5 (1)	22.2 (4)
C2	14.2 (2)	42.8 (6)	14.2 (2)	28.6 (4)
C3	25.Ø (4)	33.3 (5)	6.Ø (3)	25.Ø (4)

The figures are the percentages of selections accounted for by each agent. In parentheses are the raw scores. (*)

Table 5.4: Agent Usage

The use of strategies is also consistent with our position on their role in design. We would not expect the subjects to have a large number of strategies covering the different problems in these tasks. First, the subjects are not so familiar with the tasks that they are likely to have prestored plans. Second, there is almost no evidence to

- - - - - - - - - -

(*) There is a definite correlation between subjects and the usage frequency of different selection agents. Subject A shows a greater use of local constraints in generating problems than the other two subjects. Subjects B and C place greater emphasis on strategies. We believe these differences are idiosyncratic and represent the different knowledge and experience individual subjects use in designing. Further discussion of these differences is continued in Section 6.

indicate the development of plans during the different tasks.

5.2.2 The Assertion Concerning Recency

Even with the assertion about local constraints there remains, at any one point, a number of constraints that are possible sources for new problems. The recency assertion states that the constraints used by the local constraints agent are those that have entered WS most recently. The assertion is implemented in the model by a list which simulates a WS where new information is always at the head of the list.

WS can be examined to determine which constraint will fire off associated problems. We can use the position of a constraint in the WS as a measure of its recency. With this information we can determine for each problem chosen through local constraints how recently the activating constraint entered WS. Was it the most recent piece of constraint information? Was it the oldest? Over the nine protocols, the most recent constraint accounts for 74.4 percent of these selections. The next to most recent constraint accounts for another 18.6 percent.

The formulation of constraints and how they match and cause new problems to be created is also consistent with the

representation and actions of production system models of behavior. Production systems that model cognitive behavior contain a series of rules that are matched against the contents of STM (Newell and Simon, 1972; Davis and King, 1975). When the invoking conditions are satisfied then a particular rule will be used. The application of the rule is equivalent in most cases to some aspect of human behavior. Most production system models of behavior only allow rules to be invoked only by information that is present in STM.

Although this model has not been implemented as a production system, it shares features with those models that reflect assertions concerning what information is available in human information processing. If we redefined the actions of the local constraints agent and others as production rules, then the model's performance would not be changed. As in production systems, all the information that brings different agents (rules) into use is based on the presence of that information in WS.

Thus, the assertion of usage based on recency is supported by the performance of the program, the constraints that it uses, and similar models of human behavior.

5.2.3 The Assertion about Strategies

We have asserted that the availability of strategies as a problem source is limited by their accessibility from WS. We have also asserted that other problem selection criteria may divert the designer into subproblems whose solution may eventually block the resumption of some strategy's execution.

In the model these assertions take the form of a limited size WS in which strategies must be present if they are to be used. Before a strategy is invoked, problems arising from local constraints are considered. Is this a reasonable assertion?

In experiments of problem solving conducted by Donald Norman there is evidence that people under stress will exhibit a phenomenon Norman has termed "perceptual narrowing." Perceptual narrowing (Lindsay and Norman, 1972) occurs when the problem solver is distracted from solving the whole problem and focuses or becomes immersed in solving a subproblem. A result of this behavior is that while a good solution may result for the subproblem, it may cause the subject to fail for the general problem. An example cited by Norman is that of getting out of a building which is on fire. A general solution plan might be (1) go down the stairs and (2) out the door. However, if the problem

solver concentrates on just going down the stairs then he might never execute the critical step of actually leaving the building. The difficulty is that if you keep going down the stairs you could continue to go into the basement. Fire laws actually prevent this by requiring buildings designed so that it is necessary to exit the stair well at ground level before continuing to the below ground levels.

We believe that an analogous process takes place in problem solving in design. The narrowing of attention is on recent problems and their implications. This behavior is common and in programs is usually described as "goal-directed" or depth-first problem solving.

Combining goal-directed design with a structural framework that limits working memory to a small size produces the effect of strategy interference that was described in Section 4.6. At that time we presented an example where an active strategy was lost from WS by the introduction of new constraints, decisions, and other problem information.

5.3 Evaluating the Encoding Process

The reliability of the coding rules was tested by having an independent judge encode all of Protocol B2. Overall there was a match of 83 percent between the judge's

encodings and those done by the experimenter. The match rate rises to 91 percent when only the identification of selection points is considered.

The 17 percent of mismatches distributes into two groups. The first group accounts for 10 percent of the 17 percent of mismatches and are codings where either the judge or experimenter encoded some piece of information that the other had not. The remaining 7 percent were codings where the judge and experimenter had coded the same segment of protocol differently.

Both the judge and experimenter agreed that the greatest difficulty was in differentiating between statements of constraints and decisions. Differences in the coding of these two items accounts for 64 percent of the second group of coding variances.

6.0 CONCLUSIONS

6.1 Review

This paper has presented a general model of design composed of three processes: information collection, problem selection, and solution generation. Section 2 described these processes, gave evidence of their existence, and presented a model for problem selection in software design.

Conclusions

The selection model was formulated as a computer program (see Section 3) that reproduced human problem selection behavior of designers at work.

Protocols were collected for three designers on three design tasks. The characteristics of the designers and the tasks were described in Section 4.1. The design tasks were of moderate complexity and required from 30 minutes to 90 minutes to complete.

As designers worked they were asked to verbalize their thoughts. The transcriptions of their verbalizations were later encoded and used as input to the model. The encodings were a form of deep structure representation of statements by the designers expressing constraints, strategies, problems, and decisions that they were making. Section 4 demonstrated the model's ability to reproduce sequences of problem selections observed in protocols. Section 5 evaluated the model's performance in reproducing selection behavior and presented empirical support for the main assertions of the model.

We have learned that problem selection behavior in software design is primarily a function of constraint and strategy knowledge and the structural limitations of human information processing systems. Our research has presented a model of how such knowledge interacts with those

processing mechanisms. We have verified the model by implementing it as a computer program and comparing the performance of the program with behavior observed in verbal protocols.

We have learned about the role of different types of knowledge in design. We have increased our understanding of how behavior is influenced by the structure of our own information processing systems. The implications of these findings are discussed in Sections 6.4 and 6.5.

This is exploratory research. Only a handful of computer-based cognitive models have been built. Our model is the first we know of that has specifically studied psychological aspects of the design of computer programs and it has only explored a part of a much larger process.

6.2 Further Research

One area of further research is the study of design processes not modeled in this research. In particular, how do designers gather information during design? Each designer possesses an immense amount of information. How does the designer determine what information is relevant? How much information is enough? How is it organized?

Our research indicates that these questions may be related to how a designer's information is structured and

"parameters" of the designer's decision-making processes. Identifying relevant information might be a function of how the designer's knowledge is organized. The decision-making parameters, such as how many alternatives to consider before making decisions, what effort to expend in finding alternatives, and how to collect information may change depending on the designer or the problem.

We have observed that designers write code while designing. Sometimes it is very high level metacode. At other times the code may be extremely detailed and expressed in a particular implementation language. We have just begun to consider how such code-writing is related to our model.

This behavior appears similar to the code-writing processes described by Brooks (1975). The question remains whether Brooks's model can be generalized to the design problems we have studied.

6.3 Limitations of the Model

The model lacks a deductive component and thus has no understanding of what is taking place during the design. There is no evaluation of decisions, constraints, or the importance of one problem over another.

When the model chooses incorrectly, it is often because it lacks such a capability. A typical instance involves a

designer's constraining the solution of a data structure to alternatives involving linked structures. It appears obvious that this would prevent the designer from going back and looking at other features of the alternative sequential structures. Nevertheless, it is possible for the model to do just that because the program can not deduce the effect of that constraint on the space of likely problems.

The program does not "understand". The program uses a simple pattern-matching scheme for processing protocol encodings instead of a natural language processor. Pattern matching was chosen to make the model applicable to several designers and widely differing task domains.

Including language understanding would have required limiting the domain to at most one task and probably to one subject because of the resources required for encoding knowledge about the domain to "understand" what was being said. In general, such a understanding component would be a general language understander which was beyond the scope of this research.

The idea of a goal is missing from the model. Designers tell us that they work much of the time with some overriding purpose in mind. The goal may be to minimize or maximize some constraint. The goal may be to finish one part of the design before another. These goals do not

always appear as strategies. They are parts of a more general mechanism, in the form of a constraint, influencing the behavior of the designer.

A goal mechanism could be added to the model by extending the interpretation used for constraints. Part of the extension would require adding a scope and importance description to each constraint. The goal problem is ultimately related to the absence of a deductive component.

6.4 Implications for Teaching Design

Two kinds of knowledge are used by the model in reproducing problem selection behavior: constraints and strategies. During design, strategies provide the breadth-wise coverage of design problems. They dictate which problems to follow across some level of the design. Strategies also serve as general plans. Constraints used in design supply the depth-first element of selection behavior. Constraints act as the connectors between levels in design and are used by designers to follow a problem through several directions and levels.

When we look at the nine protocols, there is a distinct pattern of information usage amongst the designers. Two of the designers use strategies much more frequently than the third. These designers seem to deal with the tasks more

generally. They systematically look at one level after another of the problem. The third designer moves quickly from one problem to another through many levels of the design. Most of the transitions can be explained by the large body of constraint information which the designer uses. Intuitively, we feel greater progress with the third designer than with the two designers using their general but weaker strategies.

We find that when the designer is able to evoke constraints, the problem is quickly bounded. Constraints combined with a body of relational information allows the designer to explore the problem in several dimensions in constrast to the systematic approach typified through the usage of strategies. This implies that a designer's education should provide a substantial body of interrelated design knowledge. For example, knowledge of different ways to structure data within a computer should not be isolated from the issues of space, time, and implementation demands. The bringing together of such knowledge is critical to the designer's ability to explore design problems better.

6.5 Implications for Design Methods

The model we have presented is not specific to any one design method. It does not attempt to decompose design into a series of steps or stages. We did not observe subjects

designing in that manner.

We have seen subjects attempting to follow some general design approaches. Several times, subjects would say that they were approaching the problem in a top-down fashion. Top-down design involves specifying the design hierarchically with the highest level details first and proceeding downward, systematically, level by level, to the finest level of detail. Subjects regularly deviated from their top-down approaches.

Strategies were the main method of expressing top-down problem approaches. As we have already shown these strategies can be interfered with by limits on the size of working storage and the appearance of constraint information that invokes related problems.

When the designer uses a "systematic" strategy, the information collection associated with working on the problems of that strategy is not systematic. Information is often gathered through seemingly loose associations. These associations also form the basis for some of the problem selection behavior that we have already discussed.

The implication we derive for design methods is that they should place greater emphasis on helping the designer gather and concentrate information. In addition, design methods should improve the planning a designer uses in

choosing a sequence of problems.

ACKNOWLEDGEMENTS

I would like to acknowledge my thesis advisor, Peter Freeman, for his contributions to this work and for his constant guidance during my graduate studies. I also thank Ruven Brooks, Rob Kling, and Fred Tonge for their valuable assistance in this research. Jim Meehan's careful reading and numerous editorial comments are gratefully acknowledged.

REFERENCES

Boehm, Barry W. "The High Cost of Software." in Proceedings of a Symposium on the High Cost of Software, Naval Postgraduate School, Monterey, California, September 17-19, 1973.

Brooks, Ruven. "A Model of Human Cognitive Behavior in Writing Code for Computer Programs." unpublished PhD thesis, Carnegie-Mellon University, Department of Computer Science, May 1975.

Constantine, L. L., Myers, G. J. and Stevens, W. P. "Structured Design." IBM Systems Journal, Vol. 13, No. 2, pp. 115-139, May 1974.

Davis, Randall and Jonathan King. "An Overview of Production Systems." Stanford Artificial Intelligence Laboratory Memo AIM-271, Computer Science Department Report No. STAN-CS-75-524, Computer Science Department, Stanford University, October 1975.

Eastman, Charles M. "Explorations in the Cognitive Processes of Design." Carnegie-Mellon University, ARPA Report DDC No. 671-158, 1968.

Eastman, Charles M. "Design Augmentation." Computer Science Research Review, Carnegie-Mellon University, 1972-1973.

Goos, G. "Hierarchies." Advanced Course on Software Engineering. Ed. F. L. Bauer, Springer-Verlag, 1973.

Levin, Steven L. "A Short Survey of Models of the Design Process." University of California, Irvine, Department of Information and Computer Science, TR #71, October 1975.

Levin, Steven L. "A Model of the Problem Selection Process in the Design of Computer Programs." unpublished PhD dissertation, University of California, Irvine, Department of Information and Computer Science, September 1976.

Levin, Steven L. "Problem Selection in Software Design: The Data for Protocol A1." University of California, Irvine, Department of Information and Computer Science, TR #94, November 1976.

Lindsay, Peter H. and Donald A. Norman. Human Information

Processing: An Introduction to Psychology. New York: Academic Press, 1972.

Miller, G. A. "The Magical Number Seven, Plus or Minus Two: Some Limits On Our Capacity for Processing Information." Pyschological Review, Vol. 63, March 1956, pp. 81-97.

Moran, Thomas P. "The Symbolic Imagery Hypothesis: A Production System Model." unpublished PhD thesis, Carnegie-Mellon University, Department of Computer Science, December 1973.

Naur, P. and B. Randell. Eds. Software Engineering. report of NATO Science Committee, Garmisch, Germany, 7th to 11th October, 1968, published January, 1969.

Newell, Allen and Herbert A. Simon. Human Problem Solving. Prentice-Hall, 1972.

Peters, Lawrence J. and Leonard L. Tripp, "Design Representation Schemes." MRI Symposium on Computer Software Engineering, April 20, 1976.

Reitman, W. R. Cognition and Thought. Wiley, 1965.

Ross, D. T. and K. E. Schoman, "Structured Analysis for Requirements Definitions." Proceedings of the 2nd International Conference on Software Engineering, San Francisco, October 13-15, 1976.

Simon, Herbert A. "How Big Is a Chunk." Science, Vol. 183, February 8, 1974, pp. 482-488.

APPENDIX A: INSTRUCTIONS

A.1 General Instructions

In this session I would like you to begin work on the design task described on the following page. You are to do the specified design such that when you are done your design could be given to another individual for implementation.

The attached page contains the specifications for the required design. Please speak aloud what you are doing and thinking while you work. If you are silent for more than a few seconds I will prompt you by saying "speak."

Thank you for your cooperation.

A.2 Design Task A

In order to utilize many programs and facilities of timeshared computers users must be able to create and modify files of information which are stored on disk. The programs used for this task are commonly known as text editors. Your task is to design such a text editor for the PDP-10

A.3 Design Task B

Purpose

Produce a page-keyed index.

Typical Usage

The main input file contains the source text for a book with pagination indicators. The output will be an index for the book.

Inputs

(a) A source file; read from the standard input unit.

(b) Marker characters 'x' and 'y' (may be the same); read from standard input unit.

(c) The name of a file containing a list of terms to be indexed; read from standard input unit.

The system should accept as input orders from customers that describe a part and quantity desired. The system should fill as many of these orders as possible. For each order the system should print a summary of that order, if it can be filled, or an error message if it cannot. The process should continue for all of the orders.

The system should fill partial orders and at the same time generate a request reordering the exhausted part to maintain some inventory level. At such times the customer should get a summary of how much of the order has been filled and its cost. In addition, the system should generate an order request to the appropriate vendor for a new supply of that part. Order requests should also be generated when an order depletes the supply of some part below a minimum level.

Environment

The company has its own batch processing computer system. Hardware includes tape and disk drives. The software consists of the usual language processors and support facilities.

APPENDIX B: SEGMENT OF PROTOCOL A1

The numbers on the left hand side of the page are the line numbers referred to in the discussions of the protocols in the text. Lines preceded by a "S" are transcriptions of audio information. Lines preceded by an "A" are actions obtained from the visual portions of the tapes.

S1: ALL RIGHT.
S2: I GUESS THE FIRST THING I WOULD DO
S3: WHEN POSED WITH THIS PROBLEM WOULD BE
S4: TO ATTEMPT TO DEFINE THE CLASS THEN THE SUBCLASS
S5: THAT I WAS GOING TO ATTACK INCLUDING THE AH,
S6: TYPE OF EDITOR.
S7: SINCE YOU CANNOT BUILD A TEXT EDITOR TO INCLUDE
S8: ALL OF THE FUNCTIONS THAT ARE AVAILABLE.
S9: IN TEXT EDITORS.
S10: IN THAT SOME OF THEM HAVE TO DO WITH BASIC APPROACH
S11: TO THE FILE THAT ONE TAKES.
S12: AHMM. OKAY.
S13: ONE BASIC DECISION IS THAT THE SORT OF OPERATING
S14: SYSTEM THAT IT IS GOING TO BE INTERFACING TO.
S15: AH. THERE IS AT LEAST FOUR AVAILABLE FOR THE
S16: PDP-10 THAT I KNOW OF.
S17: AH. MOST OF THOSE FOUR
S18: ALL USE SEQUENTIAL FILES FOR TEXT.
S19: THERE AREN'T ANY STRUCTURED FILES
S20: SO THAT A STRUCTURED EDITOR
S21: ISN'T AH TOO IMPORTANT AH
S22: AFTER THE STYLE OF EDIT ON THE SEVEN.
S23: ONE MIGHT HOWEVER DECIDE TO GO TO
S24: A STRUCTURED FILE IF AH
S25: ONE EXPECTED THE TEXT EDITOR TO RUN ON A VERY
S26: SMALL PDP-10 SYSTEM.
S27: HOWEVER MOST TENS BEING SOLD THESE DAYS PROBABLY
S28: HAVE THE VIRTUAL MEMORY CAPABILITY
S29: AND CERTAINLY MOST ALL IN THE FUTURE WILL.
S30: SO, THAT
S31: MUCH MORE EFFICIENT APPROACH PROBABLY
S32: WOULD BE TO USE THE VIRTUAL MEMORY CAPABILITIES
S33: OF THE MOST OF THE OPERATING SYSTEMS OF THE
S34: 10 TO MANAGE THE TEMPORARY FILE
S35: AND SIMPLY BRING ALL THE TEXT INVOLVED
S36: INTO CORE AND WORK ON IT DIRECTLY.
S37: AND AS LONG THE MANAGEMENT FACILITIES YOU USE DON'T
S38: TAKE TOO MUCH PAGING OVERHEAD THEN THERE SHOULDN'T
S39: BE ALOT OF PROBLEMS WITH
A40: [WRITES "FILE I/O"]
S41: AH USING EVERYTHING INCORE AND
S42: NOT PAYING ATTENTION TO AH I/O

S43: EXCEPT SEQUENTIALLY TO BRING THE
S44: FILE IN AND TO WRITE IT OUT AT THE END.
S45: AHMM. LET'S SEE.
A46: [WRITES "TYPE OF EDITOR"]
S47: I GUESS THE SECOND THING TO DO
S48: IS TO DECIDE THE AH
S49: TYPE OF EDITOR IT IS TO BE.
S50: WHETHER IT IS JUST A SIMPLE ONE
S51: AH OR AH SAY LINE ORIENTED OR
S52: SAY CHARACTER ORIENTED LIKE TECO.
A53: [WRITES "LINE OR CHARACTER"]
S54: AH OR MORE COMPLEX ONE INVOLVING SPELLING
S55: CORRECTION AND AH COMPLEX STRING MANIPULATION.
A56: [WRITES "SIMPLE OR ELABORATE"]
S57: AH.
S58: TECO IN ITSELF HAS A LOT OF FACILITIES BEYOND
S59: WHAT TYPICAL TEXT EDITOR HAS BUT
S60: IT LACKS A LOT OF THE INTERACTIVE SORT OF
S61: THINGS THAT EDITORS LIKE QED AND SOS HAVE.
S62: MY OWN INCLINATION I GUESS IS TO FOLLOW
S63: MY OWN PREFERENCE
S64: FAILING
S65: IF I DONT HAVE AH USER INSISTENCE ON ONE
S66: KIND OF TEXT EDITOR OR ANOTHER IS SIMPLY TO
S67: CREATE ONE THAT I LIKE.
S68: AH, AND
S69: MY OWN PREFERENCE IS TOWARD A TECO-LIKE
S70: EDITOR WITH PERHAPS
S71: EXTENSIONS FOR A FEW OF THE COMMON FACILITIES
S72: THAT I USE.
S73: THE YOU KNOW LIKE MACRO FACILITIES.
S74: AH.
S75: LET'S SEE.
S76: THE FIRST THING IS AH FILE I/O.
S77: AHMM. AND VIRTUAL MEMORY.
S78: ON SOME SYSTEMS TO GET TO AH
S79: HANDLE MEMORY MANAGEMENT PARTICULARLY ON AH
S80: KA10 BASED SYSTEMS AND AH ONES WHERE
S81: MEMORY IS EXTENDED GRADUALLY AS OPPOSED TO
S82: ON TENEX WHERE YOU CAN
S83: GRAB MEMORY AT ANY TIME YOU HAVE
S84: TO HAVE A LITTLE BIT OF MANAGEMENT FACILITY
S85: TO EXTEND MEMORY PERIODICALLY AS REQUIRED.
S86: SO THAT SIMPLY INITIALIZING THE WHOLE
S87: PAGE SPACE WOULD INVOLVE ALOT OF
S88: OVERHEAD FOR SMALL FILES
S89: WHICH PROBABLY IS THE LARGEST
S90: PERCENTAGE OF CASES.
A91: [WRITES "VIRTUAL MEMORY MANAGEMENT"]
A92: [WRITES "VERSUS INTERNAL SOFTWARE FILE STRUCTURE"]
S93: AND THE SECOND ONE IS THE AH STYLE OF
S94: EDITOR.

S95: LINE OR CHARACTER ORIENTED.
A96: [WRITES "LINE OR CHARACTER"]
S97: AND, AH
S98: SIMPLE OR ELABORATE.
A99: [WRITES "SIMPLE OR ELABORATE"]
S100: AHMM.
S101: THE THIRD CASE WOULD BE WHETHER IT IS
S102: PROGRAMMING LANGUAGE OR INTERACTIVE
A103: [WRITES "PROGRAMMING LANGUAGE OR INTERACTIVE"]
A104: [WRITES "STRUCTURED FOR PROGRAMMING LANGUAGE"]
S105: SORT OF LANGUAGE.
S106: LESS IMPORTANT ARE THE PARTICULAR DETAILS OF
S107: THE IMPLEMENTATION,
S108: THE INTERNALS IN SO MUCH AS
S109: THE EXTERNAL FORM MAY DICTATE ALOT OF THE DETAILS.
S110: MY OWN CHOICE IS FOR A CHARACTER EDITOR WITH
S111: FAILRY ELABORATE FACILITIES AND THE
S112: STYLE OF A PROGRAMMING LANGUAGE.
S113: THE,
S114: IN ORDER TO IMPLEMENT IT I THINK I WOULD
A115: [WRITES "IMPLEMENTATION SCHEDULE"]
S116: TAKE AH
S117: AH SERIES OF STEPS AND AH
S118: BUILD FIRST THE BASIC EDITOR SO AS TO
A119: [WRITES "BASIC FACILITIES"]

The entire protocol is 1152 lines.

APPENDIX C: ENCODED INFORMATION FOR PROTOCOL A1

This appendix contains a segment of the encoded protocol and constraint/problem pairs for Protocol A1.

C.1 Encodings for Protocol A1

```
(STRATEGY (DEFINE CLASS-OF-EDITORS)
          (DECIDE (TYPE-OF EDITOR)))
(SELECTION)
(CONSTRAINT (DESCRIPTION NUMBER-OF-FUNCTIONS CONSTRAINED))
(CONSTRAINT
 (DESCRIPTION FUNCTIONS CONSTRAINED-BY (APPROACH-TO FILE)))
(SELECTION)
(CONSTRAINT (DESCRIPTION PDP-1Ø
                         HAS-ONLY
                         (SEQUENTIAL FILES))
            (DEFVALUE . 1Ø))
(CONSTRAINT (DESCRIPTION PDP-1Ø HAS-NO (STRUCTURED FILES))
            (DEFVALUE . 1Ø))
(CONSTRAINT (DESCRIPTION (MOST PDP-1Ø)
                         HAVE
                         (VIRTUAL MEMORY))
            (DEFVALUE . 1Ø))
(CONSTRAINT (DESCRIPTION VM USE-OF (IS EFFICIENT))
            (DEFVALUE . 6))
(DECISION USE VM (TYPE-OF OPERATING-SYSTEM))
(PROBLEM-STMT MANAGEMENT OF FILE-IO)
(SELECTION)
(CONSTRAINT (DESCRIPTION DESIGNER
                         PREFERENCE
                         (TECO-LIKE EDITORS))
            (DEFVALUE . 1Ø))
(DECISION BUILD TECO-LIKE (TYPE-OF EDITOR))
(SELECTION)
(CONSTRAINT (DESCRIPTION USE (AVAILABLE MEMORY-MGT)))
(CONSTRAINT
 (DESCRIPTION PAGE-SIZE INITIALIZATION PROHIBITIVE))
(CONSTRAINT
 (DESCRIPTION (MOST EDITING) PERFORMED-ON (SMALL FILES)))
(SELECTION)
(CONSTRAINT
 (DESCRIPTION (IMPLEMENTATION-DETAILS) UNIMPORTANT))
(DECISION BUILD
          (CHARACTER ELABORATE PROGRAMMING-LANGUAGE)
          (TYPE-OF EDITOR))
(PROBLEM-STMT FUNCTIONS
```

```
                CONSISTS-OF
                (SEARCH ADDRESSING
                        INSERTION
                        DELETION
                        CHANGING
                        LISTING
                        MACROS
                        CONDITIONALS))
(SELECTION)
(CONSTRAINT
 (DESCRIPTION SEARCHING MOST-IMPORTANT OPERATION))
(DECISION INCLUDE PATTERN-MATCHING)
(CONSTRAINT
 (DESCRIPTION GENERAL-SEARCH-ALGS ARE NON-LINEAR))
(DECISION ELIMINATE GENERAL-ALGS FOR SEARCHING)
(SELECTION)
(CONSTRAINT
 (DESCRIPTION INPUT FROM (SOURCE (OR USER-FILE))))
(CONSTRAINT (DESCRIPTION INPUT DEPENDS-ON SOURCE))
(CONSTRAINT (DESCRIPTION (INPUT FILE) IS SEQUENTIAL))
(CONSTRAINT (DESCRIPTION (INPUT USER) IS RANDOM))
(CONSTRAINT
 (DESCRIPTION (MOST MODIFICATIONS) ARE (INTERNAL-TO TEXT)))
(CONSTRAINT
 (DESCRIPTION INPUT-TYPES INFLUENCE TEXT-REPRESENTATION))
(SELECTION)
(CONSTRAINT
 (DESCRIPTION TREE-STRUCTURES REQUIRE BALANCING))
(CONSTRAINT
 (DESCRIPTION (LARGE-FILE IN-CORE) USE LINKED-STRUCTURE))
```

There is a total of 151 encoded statements for Protocol A1.

C.2 Relational Information for Protocol A1

```
(FILE TYPE-OF OPERATING-SYSTEM)
(TEXT-REPRESENTATION DATA-STRUCTURES)
(DATA TYPEOF FILE)
(SPEED PERFORMANCE)
(GOOD-SPEED PERFORMANCE)
(IO-TIME DATA-FORMATS)
(MODIFICATION-TIMES RELATIVE
                    DATA-STRUCTURE
                    MODIFICATION-TIMES)
(CHARACTER-MOVEMENT TEXT-MOVEMENT-TIME)
(FAST-INDEXING SEARCHING-TIMES)
```

APPENDIX D: PROGRAM OUTPUT FOR PROTOCOL A1

```
*(SIM)

SYSINIT: Enter source for information: *A1

Extension= INI

Print external information trace? Y
Print detailed internal trace? Y
Where should the trace be printed? TTY:
Enter external input source: (A1 . EXT)
File for memory inputs is: (A1 . MEM)


Start of simulation
----- -- ----------

External input: STRATEGY
  Elems:
       ((DEFINE CLASS-OF-EDITORS) (DECIDE (TYPE-OF EDITOR)))

External input: SELECTION

    ACTION in LOCAL-CONSTRAINTS-AGENT
        Attempt to find problem based on local constraints.

    ACTION in STRATEGY-AGENT
        Attempt to find problem via the strategy agent.

    ACTION in SELECT
        Next problem selected.  Add to WS.

        Prob:  (DEFINE CLASS-OF-EDITORS)

****** Selection No. 1 ******

Problem to be selected is: (DEFINE CLASS-OF-EDITORS)
CORRECT SELECTION? (Y,N OR B): Y

External input: CONSTRAINT
  Desc:  (NUMBER-OF-FUNCTIONS CONSTRAINED)

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: CONSTRAINT
  Desc:  (FUNCTIONS CONSTRAINED-BY (APPROACH-TO FILE))

    ACTION in CONSTRAINT-PROCESSOR
```

```
        Add constraint to WS.

External input: SELECTION

    ACTION in LOCAL-CONSTRAINTS-AGENT
        Attempt to find problem based on local constraints.

    ACTION in FIND-CONSTRAINT-PROB-GEN
        Problem found by local constraints agent.

    ACTION in SELECT
        Next problem selected.  Add to WS.

        Prob:  (DETERMINE TYPE-OF OPERATING-SYSTEM)

****** Selection No. 2 ******

Problem to be selected is: (DETERMINE TYPE-OF
 OPERATING-SYSTEM)
CORRECT SELECTION? (Y,N OR B): Y

External input: CONSTRAINT
  Desc:  (PDP-1Ø HAS-ONLY (SEQUENTIAL FILES))

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: CONSTRAINT
  Desc:  (PDP-1Ø HAS-NO (STRUCTURED FILES))

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: CONSTRAINT
  Desc:  ((MOST PDP-1Ø) HAVE (VIRTUAL MEMORY))

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: CONSTRAINT
  Desc:  (VM USE-OF (IS EFFICIENT))

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: DECISION
  Desc:  (USE VM (TYPE-OF OPERATING-SYSTEM))

    ACTION in SEARCH-SOLVED-PROB
        Attempt to find problem in WS solved by decision.

    ACTION in SEARCH-SOLVED-PROB
```

Problem found in WS. Change solution state to SOLVED.

Prob: (DETERMINE TYPE-OF OPERATING-SYSTEM)

ACTION in MARK-INACTIVE-CONSTRAINTS
The following constraint in WS is now INACTIVE because of a recent decision.

Desc: (VM USE-OF (IS EFFICIENT))

ACTION in MARK-INACTIVE-CONSTRAINTS
The following constraint in WS is now INACTIVE because of a recent decision.

Desc: ((MOST PDP-10) HAVE (VIRTUAL MEMORY))

ACTION in MARK-INACTIVE-CONSTRAINTS
The following constraint in WS is now INACTIVE because of a recent decision.

Desc: (PDP-10 HAS-NO (STRUCTURED FILES))

ACTION in MARK-INACTIVE-CONSTRAINTS
The following constraint in WS is now INACTIVE because of a recent decision.

Desc: (PDP-10 HAS-ONLY (SEQUENTIAL FILES))

ACTION in SEARCH-SOLVED-PROB
Attempt to find problem in PS solved by decision.

ACTION in DECISION-PROCESSOR
Decision does not cause a currently unsolved problem to be solved.

External input: PROBLEM-STMT
Prob: (MANAGEMENT OF FILE-IO)

External input: SELECTION

ACTION in LOCAL-CONSTRAINTS-AGENT
Attempt to find problem based on local constraints.

ACTION in STRATEGY-AGENT
Attempt to find problem via the strategy agent.

ACTION in SELECT
Next problem selected. Add to WS.

Prob: (DECIDE (TYPE-OF EDITOR))

****** Selection No. 3 ******

Problem to be selected is: (DECIDE (TYPE-OF EDITOR))
CORRECT SELECTION? (Y,N OR B): Y

External input: CONSTRAINT
  Desc: (DESIGNER PREFERENCE (TECO-LIKE EDITORS))

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: DECISION
  Desc: (BUILD TECO-LIKE (TYPE-OF EDITOR))

    ACTION in SEARCH-SOLVED-PROB
        Attempt to find problem in WS solved by decision.

    ACTION in SEARCH-SOLVED-PROB
        Problem found in WS. Change solution
        state to SOLVED.

        Prob: (DECIDE (TYPE-OF EDITOR))

    ACTION in MARK-INACTIVE-CONSTRAINTS
        The following constraint in WS is now INACTIVE
        because of a recent decision.

        Desc: (DESIGNER PREFERENCE (TECO-LIKE EDITORS))

    ACTION in SEARCH-SOLVED-PROB
        Attempt to find problem in PS solved by decision.

    ACTION in DECISION-PROCESSOR
        Decision does not cause a currently unsolved
        problem to be solved.

External input: SELECTION

    ACTION in LOCAL-CONSTRAINTS-AGENT
        Attempt to find problem based on local constraints.

    ACTION in STRATEGY-AGENT
        Attempt to find problem via the strategy agent.

    ACTION in SELECT-UNSOLVED-PROBLEM
        Found unsolved problem in WS.

    ACTION in SELECT
        Next problem selected. Add to WS.

        Prob: (MANAGEMENT OF FILE-IO)

```
****** Selection No. 4 ******

Problem to be selected is: (MANAGEMENT OF FILE-IO)
CORRECT SELECTION? (Y,N OR B): Y

External input: CONSTRAINT
  Desc:  (USE (AVAILABLE MEMORY-MGT))

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: CONSTRAINT
  Desc:  (PAGE-SIZE INITIALIZATION PROHIBITIVE)

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: CONSTRAINT
  Desc:  ((MOST EDITING) PERFORMED-ON (SMALL FILES))

    ACTION in CONSTRAINT-PROCESSOR
        Add constraint to WS.

External input: SELECTION

    ACTION in LOCAL-CONSTRAINTS-AGENT
        Attempt to find problem based on local constraints.

    ACTION in STRATEGY-AGENT
        Attempt to find problem via the strategy agent.

    ACTION in SELECT-UNSOLVED-PROBLEM
        Found unsolved problem in PS.

    ACTION in SELECT
        Next problem selected.  Add to WS.

        Prob:  (DEFINE CLASS-OF-EDITORS)

****** Selection No. 5 ******

Problem to be selected is: (DEFINE CLASS-OF-EDITORS)
CORRECT SELECTION? (Y,N OR B): N
Enter correct problem: (DETERMINE TYPE-OF EDITOR)

----> Simulation break

>GO

External input: CONSTRAINT
  Desc:  ((IMPLEMENTATION-DETAILS) UNIMPORTANT)
```

ACTION in CONSTRAINT-PROCESSOR
Add constraint to WS.

External input: DECISION
Desc: (BUILD (CHARACTER ELABORATE PROGRAMMING-LANGUAGE) (TYPE-OF EDITOR))

ACTION in SEARCH-SOLVED-PROB
Attempt to find problem in WS solved by decision.

ACTION in SEARCH-SOLVED-PROB
Problem found in WS. Change solution state to SOLVED.

Prob: (DETERMINE TYPE-OF EDITOR)

ACTION in MARK-INACTIVE-CONSTRAINTS
The following constraint in WS is now INACTIVE because of a recent decision.

Desc: ((IMPLEMENTATION-DETAILS) UNIMPORTANT)

ACTION in SEARCH-SOLVED-PROB
Problem found in WS. Change solution state to SOLVED.

Prob: (DECIDE (TYPE-OF EDITOR))

ACTION in MARK-INACTIVE-CONSTRAINTS
The following constraint in WS is now INACTIVE because of a recent decision.

Desc: (DESIGNER PREFERENCE (TECO-LIKE EDITORS))

ACTION in SEARCH-SOLVED-PROB
Attempt to find problem in PS solved by decision.

ACTION in DECISION-PROCESSOR
Decision does not cause a currently unsolved problem to be solved.

External input: PROBLEM-STMT
Prob: (FUNCTIONS CONSISTS-OF (SEARCH ADDRESSING INSERTION DELETION CHANGING LISTING MACROS CONDITIONALS))

External input: STRATEGY
Elems:
((DETERMINE FUNCTIONS SEARCH)
(DETERMINE FUNCTIONS ADDRESSING)
(DETERMINE FUNCTIONS INSERTION)
(DETERMINE FUNCTIONS DELETION)
(DETERMINE FUNCTIONS CHANGING)

```
        (DETERMINE FUNCTIONS LISTING)
        (DETERMINE FUNCTIONS MACROS)
        (DETERMINE FUNCTIONS CONDITIONALS))

External input: SELECTION

    ACTION in LOCAL-CONSTRAINTS-AGENT
        Attempt to find problem based on local constraints.

    ACTION in STRATEGY-AGENT
        Attempt to find problem via the strategy agent.

    ACTION in SELECT
        Next problem selected.  Add to WS.

        Prob:  (DETERMINE FUNCTIONS SEARCH)

****** Selection No. 6 ******

Problem to be selected is: (DETERMINE FUNCTIONS SEARCH)
CORRECT SELECTION? (Y,N OR B): Y

There are 36 pages of program output for Protocol A1.
```