Modeling Software Design Within a Problem-Space Architecture

Beth Adelson
Tufts University

Introduction

In this paper[1] we describe research on modeling software design skills within the Soar problem-solving architecture (Laird, Newell & Rosenbloom, 1987). We focus on an analysis of software designers designing an electronic mail system. The research addresses the issues of: 1. Guiding mental simulations of a design-in-progress using learned schemas. 2. The interaction of general knowledge of design with domain knowledge about the system being designed. 3. Progressive deepening of problem representations during problem-solving (Newell & Simon, 1972; deGroot, 1965).

Below we describe Soar, the theory which underlies the theoretical perspective of the research described here. We then discuss our protocol data on software design. This is followed by a description of the Soar-based system suggested by the data.

Soar: A General Architecture for Cognition

Ultimately, the Soar architecture is intended to embody a *unified theory of cognition*; capable of accounting for a range of cognitive problems or "tasks". Additionally, it is expected to be able to do so by relying on the mechanisms of recursive sub-goaling and chunking.

Currently, Soar can solve a wide range of standard AI problems. It can solve most of the "toy" problems such as eight puzzle and tower of Hanoi (Laird & Newell, 1983) which require goal-oriented action, without requiring knowledge about the problem domain. It can also solve knowledge-intensive, expert system tasks, such as those solved by R1 and Neomycin. A variety of search strategies have been implemented as well as weak method problem-solving strategies such as generate and test and means-ends analysis. Soar also exhibits learning learning with practice, transfer across tasks and generalization.

In terms of accounting for tasks that are central for a theory of cognition, research is now being conducted to look at learning by analogy and reasoning with mental models (Golding,1988; Polk, 1988; Steier & Kant, 1985; Steier & Newell, 1988). The research presented here is intended to be part of this effort to extend the range of Soar's performance to complex cognitive tasks.

The Nature of Problem-Solving in Soar

In Soar problem-solving is characterized as movement through successive *states of knowledge* in a *problem space* in order to achieve a *goal* (Newell, Shaw, & Simon, 1960; Newell & Simon, 1972; Card, Moran & Newell, 1980; Newell, 1980). The problem-solver starts out in an initial state which contains an incomplete representation of the problem solution and a *description* of what would constitute a sufficient solution. The description of the solution could be, for example, the desired *behavior* for the electronic mail system designed here, whereas the solution *itself* would be a *pseudocode* specification of the *mechanism* producing the behavior.

The problem solver's relevant knowledge is then brought to bear and the initial representation of the problem is transformed in a way that brings it closer to the goal state representation; the problem solution. Relevant knowledge may consist of specific information about the problem domain as well as general problem-solving strategies.

[1] We are grateful to David Steier for his continuing generous help. This work was supported by grants from the Design, Manufacturing and Engineering Program and the Knowledge and Data Base Systems Program at NSF.

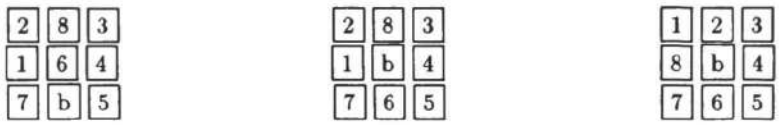

Figure 1: Starting State (left), Second State (center) and Goal State (right) for the Eight Puzzle. (The "b" represents the blank cell.)

Elements of the Architecture

To model problem-solving as it is framed above we need to be able to provide accounts of: the representation of the current problem solution at varying stages of completion; the representation of whatever is known about the desired problem solution; and knowledge about how to assess and transform the partial solution with regard to the desired solution.

The above are realized using the following elements of the Soar architecture:

- Production Memory (PM). This encodes the long-term knowledge that is needed during problem-solving. This can include factual knowledge about the problem being solved, strategic knowledge about how to proceed in problems like the current one and operational knowledge about specific problem-solving moves to be made in a given situation. It is the use of this operational knowledge that transforms the problem solution from an inital version into a goal state version. The productions that contain this type of operational knowledge place operators into working memory. When these operators are applied they transform the solution-in-progress (Section).
- Working Memory (WM). This holds the representations of the current and desired problem solution. WM also holds long term knowledge that has been identified as relevant.
- The Decision Cycle. This brings the appropriate knowledge in production memory to bear given the state of things in working memory. The difference between the starting and goal states is reduced through the decision cycle. The decision cycle is made up of two phases:
  1. An *elaboration* phase that causes already known information in production memory to be added to working memory. Information in production memory is added to working memory if it is relevant to what presently is in working memory. Elaboration is acheived by a matching process. The antecedents of all productions in PM are matched against the contents of WM; all productions that do match "fire" causing the objects described in the productions' consequents to be placed in WM.
  2. A *decision* phase that makes problem-solving decisions based on the information in working memory. The decision process begins once the elaboration process has added all that it currently can to working memory.

Using the Architecture: An Example of Problem-Solving in Soar

Below we present a description of Soar solving the eight puzzle (Figure 1). The example illustrates how the elements of the architecture function in order to move the problem-solver through the problem space towards the goal state. The eight puzzle is chosen here *not* as a representative cognitive task, but because its simplicity allows us to focus on the Soar architecture.

Soar begins by moving the 6 down into its desired spot. This occurs because of the elaboration and decision processes acting in turn. First, during elaboration, productions fire and **down**, **right** and **left** are placed in WM and marked as acceptable candidate operators. Additionally, a means-ends analysis production fires

marking **down** as best, because it moves the 6 into its desired place. Next, during the decision cycle 6 is chosen as the best move. This allows the move to actually be made during the following cycles.

Next, while trying to make the second move, **down**, **right** and **left** are all placed in WM and all marked as acceptable candidate operators. This causes the decision process to reach a "tie impasse". The information in production memory about this problem space is incomplete; it cannot resolve the tie. However, here is where the notion of sub-goals and problem spaces comes into play.

Sub-goaling to resolve impasses

When an impasse is reached the Soar architecture sets up a sub-goal to resolve the impasse. Here we see a tie impasse, however, no-change, conflict and rejection impasses are also possible.

For a tie impasse in the eight puzzle the sub-goal is to "Select" an operator from the set of possible ones. This is achieved by moving into the *selection problem space*. A further sub-goal results in which the candidate operators are actually tried out and the state that will result from each one is evaluated. For this example, **down** is found to be best because it moves the 6 into place whereas **left** and **right** move the 5 and 7 out of their desired spots.

Three points are important here: 1. The detection of the impasse and the setting up of the appropriate type of sub-goal is *not* done by the task specific eight puzzle productions; it is done by the architecture. 2. Once a sub-goal is established it is pursued and resolved in the *same* way as a higher level goal. A problem space is selected; a current and goal state are defined; and operators are then applied to the current state in order to transform it into the goal state. 3. This sub-goaling can occur to an arbitrary depth. These three points lead to some of the appeal that Soar has as a theory. By being able to detect impasses and set up appropriate sub-goals, the architecture, the part of Soar which is specified in advance and remains constant across tasks, does a good deal of the problem-solving. Additionally, the ability to solve problems in this uniform way (by recursive sub-goaling) allows Soar to provide a parsimonious account of complex problem solving.

Turning to software design we will see that organizing the problem-solving into problem spaces continues to be useful. We will also look at the way in which the problem spaces are related and how information from one problem space can further problem-solving in another.

Modeling Software Design within Soar

Method

Below we present our data on software design.
Subjects. Three expert software designers served as subjects.
Procedure. We presented each of the designers with the following design task to work on.

> Design an electronic mail system around the following commands:
> `READ, REPLY, SEND, DELETE, SAVE, EDIT,` and `LIST-HEADERS`.
> The goal is to get to the level of *pseudocode* that could be used by professional programmers to produce a running program. The mail system will run on a very large, fast machine so hardware considerations are not an issue.

Analysis of the Protocol Data

Generally protocol data can be seen as a series of episodes, with each episode reflecting the single, current focus of the subject's attention.

Pairs of episodes
One striking aspect of the protocol discussed here is that the episodes formed related pairs. The first episode

Episode 1. View the system as a set of functions.
Episode 2. Simulate the behavior of the system's functions. The commands prepare, send receive and store must be included in specifying the mailer's functionality. Discover that error recovery must be handled gracefully throughout the system.

Episode 3. View the system as a set of data objects.
Episode 4. Elaborate the features of the data objects . In the mailer messages are the data objects. Messages have destinations of senders and receivers. Additionally, messages are grouped together in stores. The stores can have various functions. For example the mail system needs a store for messages that the user has received but not yet read, as well as a store for messages that have been read but not yet saved or deleted.

Episode 5. View the system as a set of concurrent functions.
Episode 6. Simulate the behavior of a system in which there are concurrent senders and receivers. Discover that the design needs to specify when users should be notified that new mail has arrived (as it arrives, only at log on, etc). Also discover that, since mail is both being sent and received, more than one type of processing must be handled and therefore a "dispatch demon" is needed to handle the flow of messages.

Episode 7. View the system as a state machine for the states of a user.
Episode 8. Simulate the behavior of the system as a scenario in which the user logs on and issues a sequence of mail commands. He is notified that he has mail, he lists the headers, he reads a message, he makes some disposition of the message and then is able to begin again (listing headers, etc.). The designer discovers that the post-conditions of the commands need to be enumerated both to refine the command definitions and to understand the potential interactions between commands. For example, if READ includes an implicit and immediate DELETE it will prevent the user from being able to save or forward the message.

Episode 9. View the system as a state machine for the states of messages.
Episode 10. Simulate the behavior of the system in terms of the states of the messages. A message is created, sent, received, read and disposed of. These actions described at the level of files and locations within files are sufficient to generate pseudocode.

Table 1: Description of Episodes 1 through 10.

in a pair appears to take place in a general design space and the second episode appears to take place in a space containing knowledge about mail systems. The first two episodes from S1 illustrate this phenomenon.

> Episode 1: S: "....I'm going to start working here, functions of an electronic mail."
> {writes 'Functions' and 'Data' in two separate columns}
> Episode 2: "We must be able to: Prepare, Send, Receive...
> {writes prepare, send, receive under the heading 'Functions'}
> ...the system must be able to store them,
> the system must be able to handle abnormalities throughout it."

In episode 1 S1 decides to view the system as a set of functions. In episode 2 he goes on to enumerate what those functions would be.

In Table 1 we present a summary of episode pairs for the first 10 episodes of S1's protocol. The first episode in each pair establishes the goal of viewing the sytem from a particular *perspective*. The second episode instantiates the view as a simulation.

Schemas

The design process seems driven by an experience-based schema for two reasons: First, successive episodes do not appear to arise from the context that immediately precedes them; in episodes 3 and 4 the system is viewed as a set of data objects and then in episodes 5 and 6 as a set of concurrent processes. Second, the particular views chosen, such as dealing with concurrency issues, would be ones to develop given these designers' experience with communications systems.

The structure of the schema is also interesting; taken in order the five views comprise a set that would be effective in uncovering most of the aspects of the system that need refinement. The first view looks at the commands of the mailer, the second view at the messages themselves. Once the commands have been specified it becomes possible to look at the interactions produced when they are used in sequence. This is

uncovered by the fourth view which looks at the system as a state machine from the user's perspective. It also becomes possible to look at their concurrent functioning (view 3). The fifth view looks at the interaction of commands and messages; it therefore, is dependent upon having specified the commands and messages in the first and second views. Both the pairing and the ordering of the episodes is explained by the use of an underlying schema.

Interaction of Domain and General Knowledge

In designing a large software system the designers employ three types of knowledge; general knowledge of *design*, knowledge for representing *systems as pieces of pseudocode* and domain specific knowledge of how a mail system *behaves*. From our perspective, these bodies of knowledge can be seen as three *problem spaces*; a *design* space, a *pseudocode* space and a *mail* space. In simulating views of the system's behavior there has to be a mapping between the designer's high-level problem space for design, in which the schema resides and the *domain* space where knowledge about the *behavior* of mailers resides; it is in this domain space that the view is instantiated, run, and evaluated. For example, in episode 7 the designer chooses to view the system as a state machine in which the user goes through a sequence of state transitions. This gives rise to episode 8, in which the designer instantiates this state machine by constructing a simulation in which the user logs on, is notified that he has mail, lists the headers, reads and saves a message and then begins again. This means that the user's state transitions need to be put into correspondence with the issuing of commands such as **READ** or **SAVE**. Additonally, data objects must be understood to correspond with messages.

In the domain space there needs to be enough knowledge about the behavior of the mail commands to propose and simulate candidate versions of them. The candidate versions then have to be evaluated by comparing their behavior to some representation of the ideal behavior. The candidates can then be modified in accord with the results of the evaluation. The ability to modify a representation of a command in pseudocode space based on the results of a simulation in mail space implies a mapping between mail space and pseudocode space.

Progressive deepening

Progressive deepening is the retracing of steps along a previously taken problem-solving path (Newell & Simon, 1972; deGroot, 1965). The retracing is done because the first trip down the path was not sufficient; and new, relevant information has been acquired.

In episodes 2 through 10 (Table 1) we see the progressive deepening of the mailer's commands. In episode 2 the commands are just listed as a set of functions to be specified. In episode 8 the designer simulates these same commands using each one's output as input to the next and discovering that the side-effects of each have to be elaborated. For example, he decides that a user should be able to list all the message headers without being committed to then reading them. In episode 10 the designer finally simulates the commands at a level sufficient to generate pseudocode. Here he uses language that refers to reading from and writing to files.

Simulation *and* progressive deepening arise naturally within a Soar architecture. There are two reasons why simulation occurs within Soar. The first reason has to do with seeing complex problem solving as occuring in a *set* of related problem spaces. Problem-solvers have different *types* of representations of the problem solution in different problem spaces. Additionally, the information contained in one type of representation may contribute to the development of another. In design, simulations arise because detailed information about the behavior of a mail command, obtained from a simulation in *mail* space, can help in developing the representation of the command in *pseudocode* space.

The second reason for simulation has to do with comparing current and goal states. In designing the mail system, the goal state is described in terms of the desired *behavior* of the mail system. However, the problem solution is a pseudocode description of the mailer. In order to compare this pseudocode representation to the goal the pseudocode must be simulated.

The simulations done by the designers need to go through progressive deepening; at the beginning of the design session the designers' representation of the mailer is in terms of the high level *behavior* of the system.

Design Space Operators:
The operators causing differing views of the system to be taken.

1. Design the system as a set of functions.
2. Design the system as a set of data objects.
3. Design the system as sets of concurrent functions.
4. Design a state machine of the states a user goes through.
   (Resulting in a focus of attention on interactions between functions.)
5. Design a state machine of the states a message goes through.
   (Resulting in a focus of attention on interactions between function and data.)

Pseudocode & Mail Space Operators:
The operators to generate, run and evaluate candidate versions of the functions' mechanisms in pseudocode space and the functions' behavior in mail space.

1. Prepare.
2. Send.
3. Receive.
4. List-Headers.
5. Read.
6. Store.
7. Delete.

Table 2: Operators in the Design and Mail Spaces

This is the representation that would be likely for a person who had used, but not actually designed such a system. The representation of the behavior needs to be refined to a degree that allows that behavior to be expressed as pseudocode. But in a task that has a complex solution there are many aspects to the refinement. The use of repeated simulations, from different perspectives, allows the designer to attend to different aspects of the refinement in a systematic way. This allows the designer to bring his understanding to the required degree of specificity without overloading working memory.

Sketch of the Mail Designer-Soar System

In the context of Soar, problem solving is characterized as movement through successive states of knowledge in order to achieve a goal. The states of knowledge contain representations of the problem at various points in the problem-solving process. Additionally, the different aspects of the problem are regarded as different problem spaces in which different, appropriate, kinds of knowledge are brought to bear.

In order to model the design process within the Soar framework we need to provide accounts of: 1. The initial and goal state representations which form the system's input and output: In the initial state there is a description of the desired behavior of the mailer. The goal state would be a pseudocode representation of the mailer's commands. 2. The problem spaces with their appropriate operators (Table 2): The problem solving consists of trying first to apply existing knowledge relevant to a *pseudocode* problem space in order to represent the design as pieces of pseudocode. If existing knowledge is not sufficient to directly represent the design in terms of pseudocode the behavior of the mailer is simulated in *mail* space. The simulations are repeated, from varying perspectives and in increasing detail until the designer understands the system's behavior at a level that allows it to be expressed as pseudocode. The set of perspectives used in the simulations in pseudocode and mail space are generated by the strategic knowledge in the *design* problem space. Table 2 lists the operators that apply in the "design", the "pseudocode" and the "mail" problem

spaces.

Conclusions

We have described modeling software design within a Soar framework. Using this framework we are able to provide accounts for:

1. The role of schemas in bringing general knowledge to bear on knowledge about a domain: In the data presented the designer uses a high-level schema in order to create an ordered set of pairs of episodes in which a variety of inter-dependent aspects of the mailer are considered and refined.

2. The role of simulation: Simulation supports the design process in two ways. It allows the comparison of current and goal states when the current state is represented as a mechanism and the goal state is represented as behavior. Additionally, simulation supports the development of a representation of the mechanism of the system being designed when the system's behaviors are simulated at a level of detail that allows the behaviors to be expressed as mechanisms.

3. The role of progressive deepening: Because a set of simulations from a variety of perspectives are needed to complete the design we find that the same set of commands is simulated repeatedly at increasing levels of refinement.

We are optimistic that a Soar framework will continue to support detailed accounts of the mechanisms that underlie cognitve problem-solving skills.

REFERENCES

Adelson, B. and Soloway, E. The role of domain expertise in software design. *IEEE:TSE*, November, 1985.

Adelson, B. and Soloway, E. A model of software design. *International Journal of Intelligent Systems*, Fall, 1986.

Card, S., Moran, T. and Newell, A. Computer text editing. *Cognitive Psychology, 12, 1.* (1980) 32-74.

de Groot, A.D. Thought and Choice in Chess. Paris: Mouton & Cie. 1965.

Golding, A. Learning to Pronounce names by taking advice. Thesis proposal, Stanford U. 1988.

Kant, E. and Newell, A. Problem solving techniques for the design of algorithms. *Information Processing and Management*. 1984. pp. 97-118.

Laird, J. Newell A. and Rosenbloom, P. Soar: An architecture for General Intelligence. CMU CS Tech Report. 1986;

Newell A., Shaw, J. and Simon, H. A. Peport on a general problem-solving program for a computer. Proceedings of the International Conference on Information Processing, UNESCO, Paris, 1960.

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

Polk, T. Fourth Annual Soar Workshop. University of Michigan, Ann Arbor. January, 1988.

Steier, D. M. Proceedings of the 1987 IJCAI Conference.

Steier, D. M. Proceedings of the 1988 AAAI Conference.

Steier, D. M. and Kant, E. *IEEE:TSE*, November, 1985.