Computer Science

A methodology and software environment for testing process model's sequential predictions with protocols

Frank E. Ritter

21 December 1992
CMU-CS-93-101

Ritter, F. E. (1993). TBPA: A methodology and software environment for testing process models' sequential predictions with protocols (Technical Report No. CMU-CS-93-101): School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Carnegie Mellon

A methodology and software environment for testing process model's sequential predictions with protocols

Frank E. Ritter
21 December 1992
CMU-CS-93-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to the Carnegie Mellon University Department of Psychology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Psychology in the AI and Psychology program

Copyright © 1992 Frank E. Ritter. All rights reserved.

This research was partially sponsored by a training grant from the Air Force Office of Scientific Research, Bolling AFB, DC; in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597; in part by the School of Computer Science, Carnegie Mellon University; and in part by Digital Equipment Corporation through an equipment grant.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DEC or the U.S. Government.

The document was printed in 11.25 pt Times Roman. It was composed with the Scribe (v. 7) document preparation system. All the tables were laid out with the Dismal spreadsheet developed as part of this work. All graphs except where noted were prepared using the S interactive, graphical statistics package.

Keywords: Spreadsheets; programmer workbench; program editors; tracing; display algorithms; training, help, and documentation; simulation support systems; expert system tools and techniques; model development; model validation and analysis; cognitive simulation; relations among models; protocol analysis; Soar.

Abstract

Getting the most out of information processing models requires that testing and refining them be straightforward. This requires that (a) large amounts of data be easily compared with the model's performance, (b) descriptions of how and where the model mismatches are readily available and easy to interpret, and (c) the models themselves can be refined in a straightforward way. Current methods for testing the sequential predictions of process models provides none of these. It is a difficult, time consuming, boring task, requiring the full attention of a skilled analyst. Despite the importance and difficulty of testing process models against protocol data, and in contrast to the rich methodology for analyzing samples of numerical data, there is no explicit methodology or set of tools for automatically or semi-automatically doing this task.

This thesis specifies a methodology for testing process models sequential predictions through comparison with verbal and non-verbal protocols. Each of its steps are delineated, and the requirements to perform these steps developed. An environment required by and based on the needs in this specification is built to support these needs and move towards automating them. These needs are primarily to judge the model's predictions by using them to interpret and align the data with respect to the model, to understand the comparison in terms of the model's strengths and weaknesses, and to then modify the model to improve its performance. Although not limited to symbolic models, the focus of this work has been models in the Soar architecture.

SPA-mode, a spreadsheet-type tool developed for this environment supports interpreting the model's predictions with the data. Its tabular display also supports simple visual analyses of the fit. Several graphic displays are developed as ways to summarize the model's performance. One shows which model actions are supported, and the others show the relative processing rate between subject and model. Both can provide suggestions for improving the model.

Process models in Soar exist implicitly in their production rules. By making the structure of these models explicit and allowing the user to directly manipulate the appropriate theoretical objects, the Developmental Soar Interface provides an improved ability to understand and manipulate process models built within Soar, the ability to use their theoretical components to summarize their support, and to use them in further analyses.

The complete Soar/MT environment is demonstrated and developed through use on the Browser-Soar model and its data set developed by Peck and John (1992). The analyses were produced far more rapidly than those used by the original developers, and extend further. Additionally, the verbal sequentiality assumption of Erikson & Simon's (1984) verbal protocol theory was tested, and found to hold. The sequentiality assumption is then extended to apply to motor actions as well. Sequentiality, however, does not appear to hold between modalities for this data set.

Acknowledgements

I first heard Allen Newell speak at the William James Lectures at Harvard in the Spring of 1987. I was impressed with his style, his intellectual honesty, and the direction he was headed. It was a pleasure to work with and learn from him the past five years. Herb Simon gave these lectures too, but I was too young and not yet in Boston to hear him. Together they shaped the departments of Psychology and Computer Science as themselves, and made them excellent places to get work done. Because the Soar project is interdisciplinary, I had the pleasure of working with both departments. Anne Fay called this environment "Cognitive Heaven", and it is.

My interests and Jill Larkin's interests slowly merged until she became my co-advisor. Upon Allen's death she seamlessly took over. With that I was blessed with a wonderful advisor again. I always will remember some unsolicited help on writing she offered my first year. I hope it made her job easier over the last few months and two hundred pages.

Lynne Reder was the co-advisor on my masters degree and project on meta-memory. She was very patient in helping transform me into a psychologist. John Anderson provided sage advise upon request.

Virginia "Gin" Peck and Bonnie John graciously provided Browser-Soar and their data to me, and took the time to explain them. Gin was the most adventuresome user of the DSI and Soar/MT, and was the first to find many bugs. Bonnie provided the diagrams included here as noted.

The Air Force Office of Scientific Research provided a generous fellowship that allowed me to concentrate on my studies. I thank them. I also must thank the U. of Nottingham for encouraging me and then waiting for me.

The guys who went out for wings, to eat hot food and talk science answered a lot of my questions. Erik Altmann, Uli Bodenhousen, Fernand Gobet, Joe Mertz, Akira Miyake, and Chris Schunn. The R crowd my first year helped me settle into being a graduate student, R. Mary Hegarty, R. Jon King, R. Clare McDonald, R. Leigh Nystrom and R. Norm Vinson. R. Paul Reber was added at a later date.

The role of the Internet must also be noted here (which Allen had a hand in too). It has contributed over ten colleagues, many that I have never met, and in most cases have not even talked to. But we correspond, build and share software. I must thank Doug Bates, David Fox, Ed Kademan, Chris McConnell, Brad Myers, Ed Pervin, Andrew Mickish and the Garnet project, Olin Shivers, David Smith, Richard Stallman, Johan Vromans, and Chris Ward. In particular, the GNU software developers and its user community have been very helpful.

I also thank the Soar group in general, and in particular Erik Altmann, our British visitors Richard Young and Andrew Howes, and the users of preliminary versions of the environment who gave me feedback and responded to my survey. In every case where a Soar model is mentioned, the authors have provided me with the source code and usually some assistance in using and understanding it. In addition to the previous authors of interfaces for Soar, several people in the Soar group have provided detailed comments on previous drafts of chapters and at demos. Currently I need to thank Arie Covrigaru, Jill Larkin, Allen Newell, the NL-Soar group, and Paul Rosenbloom. Mike Hucka has traded Emacs code and hints on numerous occasions. He rewrote my initial GNU-Emacs package for Soar enough to be a co-author. Tom McGinnis has capably helped maintain the DSI for the last year.

Finally, I thank my Colleen, for waiting for me. Now I can come out and "play softball" (Groening, 1987).

Table of Contents

I Introduction to TBPA 1
1. Testing process models through protocol analysis 3
1.1 The need for routinely testing process models' sequential predictions 4
1.1.1 The potential benefits of routinely testing process models' sequential predictions 4
1.1.2 The difficulty of testing sequential predictions 5
1.2 The steps of testing process models' sequential predictions with protocol data 6
1.3 Developing a methodology for routinely testing process models' sequential predictions 7
1.3.1 A detailed specification of what is necessary for routine testing of process models with protocol data 7
1.3.2 An environment to support the needs of routine testing of process models 8
1.3.2.1 A tool supporting the interpretation and alignment of the data with respect to the model's predictions 8
1.3.2.2 A measurement system for telling where a model needs improvement 8
1.3.2.3 An interface for tracing, understanding, and modifying models 8
1.3.3 Documentation of the utility of the environment and methodology 9
1.3.4 Testing and extending the sequentiality assumption of verbal protocol generation 9
2. Testing process models with protocol data: Review of past work 11
2.1 The possible relationships between process models and protocols 11
2.2 Review of creating and testing models with protocol data 15
2.2.1 Exploratory analysis leading to process models 15
2.2.2 General testing of process models 18
2.2.3 Trace based protocol analysis 19
2.2.4 Summary of important data features 21
2.3 Tools related to process model testing 23
2.3.1 Tools for building models from protocols 23
2.3.1.1 Declarative knowledge coding tools 23
2.3.1.2 Exploratory protocol analysis tools 23
2.3.2 Model testing tools 25
2.3.2.1 Strategy classification tools based on process models 25
2.3.2.2 Model tracing modules within intelligent tutoring systems 25
2.3.2.3 Tools for aligning the sequential predictions with data 26
2.3.3 Tools for building and understanding models 28
2.3.3.1 Process model induction tools 28
2.3.3.2 Tools for understanding and building symbolic cognitive models 28
2.3.3.3 Knowledge acquisition tools 30
2.3.4 Summary of useful tool features 31
2.4 Measures of model to data comparison 33
2.4.1 Using criteria to develop a set of measurements 35
2.4.2 Description of measurement inputs 37
2.4.3 Non-numeric descriptive measures 41
2.4.4 Simple numeric measures 43
2.4.5 Measures of component utility 46
2.4.6 Inferential measures 47
2.4.7 A unified view: Criterion based model evaluation 49
2.4.8 Summary of measures 50
2.5 Previous models of process model testing 52
2.6 Summary of lessons for process model testing methodology and tools 55

Appendix to Chapter 2: Review of the Card model alignment algorithm 57
3. Requirements for testing process models using trace based protocol analysis 59
3.1 Definition of trace based protocol analysis (TBPA) 59
3.1.1 The inputs to TBPA 59
3.1.1.1 A 0th order functional model 59
3.1.1.2 Transcribed protocol data 60
3.1.2 The TBPA loop and its requirements 60
3.1.2.1 Step 1: Run the model to create predictions 61
3.1.2.2 Step 2: Use the predictions to interpret the data 64
3.1.2.3 Step 3: Analyze the results of the comparison 65
3.1.2.4 Step 4: Revise the model to reduce the discrepancies 66
3.2 Supporting TBPA with an integrated computer environment 68
3.2.1 Why an integrated environment is needed 68
3.2.2 The environment must automate what it can 69
3.2.3 The environment must support the user for the rest 69
3.3 The role of an intelligent architecture in the testing process 70
3.3.1 Soar: The architecture used in this environment 70
3.3.2 Making functional models examinable 72
3.3.3 Using the architecture to automate the analysis 74
3.4 Summary of requirements and description of the environment's design 74
II Supporting the TBPA methodology: A description of the Soar/MT environment 79
4. A spreadsheet for comparing the model's predictions with the data 81
4.1 Displaying and editing the correspondences 82
4.2 Automatically aligning unambiguous segments 85
4.3 Interpreting ambiguous actions 87
4.4 Supporting the global requirements 88
4.4.1 Providing an integrated system 88
4.4.2 Automating what it can 88
4.4.3 Providing a uniform interface including a path to expertise 88
4.4.4 Providing general tools and a macro language 89
4.4.5 Displaying and manipulating large amounts of data 89
4.5 Summary 89
5. Visual, analytic measures of the predictions' fit to the data 91
5.1 Creating the operator support display automatically 91
5.2 Understanding the relative processing rate 93
5.2.1 A display for comparing the relative processing rate 93
5.2.2 Using the relative processing display to test the sequentiality assumption of verbal protocol production 98
5.3 Creating additional displays 99
5.3.1 S: An architecture for creating displays 99
5.3.2 S-mode: An integrated, structured editor for S 100
5.4 Supporting the global requirements 101
5.4.1 Providing an integrated system 101
5.4.2 Automating what it can 101
5.4.3 Providing a uniform interface including a path to expertise 101
5.4.4 Providing general tools and a macro language 101
5.4.5 Displaying and manipulating large amounts of data 102
5.5 Summary of measures and recommendations for use 102
6. The model manipulation tool -- the Developmental Soar Interface (DSI) 105
6.1 Providing the model's predictions in forms useful for later comparisons and 106

analysis
6.1.1 Providing predictions for comparison with the data 107
6.1.2 Aggregating the model's performance 108
6.2 Displaying the model so that it can be understood 109
6.2.1 Normative displays of the model 111
6.2.2 Descriptive displays of the model's performance 114
6.2.3 The working memory walker 116
6.2.4 A pop-up menu and dialog boxes to drive the display 117
6.3 Creating and modifying the model 119
6.3.1 Soar-mode: An integrated, structured editor for Soar 119
6.3.2 Taql-mode: An integrated, structured editor for TAQL 120
6.3.3 The Soar Command Interpreter 120
6.4 Supporting the requirements based on the whole process and its size 121
6.4.1 Providing consistent representations and functionality 121
6.4.2 Automating what it can: Keystroke savings 122
6.4.3 Providing a uniform interface including a path to expertise 122
6.4.4 Providing a set of general tools and a macro language 123
6.4.5 Displaying and manipulating large amounts of information 124
6.5 Lessons learned from the DSI 124
6.5.1 The relatively large size of the TAQL grammar 124
6.5.2 Behavior in Soar models is not just search *in* problem spaces 124
6.5.3 Soar models do not have explicit operators 127
6.6 Summary 128
III Performance demonstrations of Soar/MT and Conclusions 131
7. Performance demonstration I: Analyzing the Browser-Soar model faster and more deeply 133
7.1 Description of Browser-Soar and its data 133
7.2 Producing richer analyses more quickly 140
7.2.1 The interpretation of data with respect to the model trace done faster and tighter 140
7.2.2 Operator support displays created automatically -- as a set they highlight periodicity in behavior 141
7.3 Where the model and subject process at different rates shown clearly 144
7.3.1 Processing rate display based on decision cycles shows that the quality of fit is high 144
7.3.2 The processing rate display can be based on other measures of the model's effort 147
7.4 High level features of the Browser-Soar model made apparent 148
7.4.1 Browser-Soar as routine behavior is made directly visible 148
7.4.2 Noting Browser-Soar's large goal depth 149
7.4.3 Modifying Browser-Soar 149
7.4.4 Testing the modified Browser-Soar 150
7.5 Testing and extending the sequentiality assumptions of protocol generation theory 151
7.5.1 Are verbalizations generated sequentially? 155
7.5.2 Are mouse actions generated sequentially? 155
7.5.3 Does the sequentiality assumption hold across verbalizations and mouse actions? 155
7.6 Conclusions about Browser-Soar and the TBPA methodology 157
7.6.1 Some conclusions about Browser-Soar 157
7.6.2 Some conclusions about the methodology 158
Appendixes to Chapter 7 159

1 Alignment of the Write episode of Browser-Soar 159
2 Displays of each analytical measure for each episode of Browser-Soar 165
8. Performance demonstration II: Use of Soar/MT components by others 171
8.1 Usage of the Developmental Soar Interface to develop Soar models 171
8.2 Usage of S-mode to create functions in S 173
Appendix to Chapter 8: Survey distributed to Soar users 175
9. Contributions and steps toward the vision of routine automatic model testing 179
9.1 A methodology for testing the sequential predictions of process models 180
9.2 Each step in the methodology was supported in a software environment 181
9.2.1 Interpreting and aligning the model's predictions and the data 181
9.2.2 Analyzing the results of the testing process 182
9.2.3 Steps related to manipulating the model: Prediction generation and modification 182
9.2.4 The synergy from integration 183
9.3 Validated and extended the sequentiality assumption of protocol generation theory 183
9.4 Progress toward the vision of routine applied theoretically guided protocol analysis 184
9.5 Concluding remarks 185
References 187
I. How to obtain the software described in this thesis 201

List of Figures

Figure 1-1: Schematic of trace based protocol analysis. 7
Figure 2-2: Information streams of subject and model used in testing process theories. 12
Figure 2-3: Schematic of possible levels of comparison between model and data. (Numbers are referred to in the text.) 13
Figure 2-4: How theory level influences comparison with data. 16
Figure 2-5: Example output of TAQL space graph. 29
Figure 2-6: Example displays for comparing the model's predictions with the data. 34
Figure 2-7: Example operator application support graph, from Peck and John, 1992. 44
Figure 2-8: Example match over time display, taken from Sakoe and Chiba (1978). 45
Figure 2-9: (a) Example cumulative hit curve (right). (b) Redesigned cumulative hit curve (left). 47
Figure 2-10: Example criterion table taken from Ritter (1989) 50
Figure 11: Example alignment by the Card1 algorithm. The two strings being aligned are "DUC" and "DUDUDU". 58
Figure 3-12: Diagram showing the inputs (in bold) to trace based protocol analysis (TBPA): A computational model and transcribed and segmented protocol data. 61
Figure 3-13: Diagram of the steps in testing process models with TBPA. 62
Figure 3-14: Diagram illustrating direct trace modification as a form of pseudo-model revision. 69
Figure 3-15: Grand design for an intelligently automatic protocol analyzer. 73
Figure 3-16: Requirements for an environment for testing process models and overview of the Soar/MT environment to support these requirements. 75
Figure 4-17: 82
Example display of a model trace aligned with data (taken from the Write episode of Browser-Soar). Left-hand columns "T" (time of subject's actions) through "MDC" (matched decision cycle) are one meta-column, and columns "DC" and "Soar trace" on the right are another meta-column. The right-most simple column of the left meta-column (in this case the H column) is indicated to uses in the editor's mode line (the bottom line of the figure) as "<H]".
Figure 4-18: Types of correspondences that can be represented in Spa-mode. 84
Figure 4-19: A simple fix is applied to the original Card1 algorithm to return the edit sequences in the correct order without explicitly reversing the returned list. 86
Figure 5-20: 92
Example operator prediction support display taken from the Unit episode of Browser-Soar. The model's operators are shown on the left-hand side, indented according to their depth in the problem space hierarchy. The connected black squares represent the model's performance. Corresponding data are represented by overlapping symbols. Unmatched data are placed at the bottom of the display as if it matched the *Not matched* operator.
Figure 5-21: 94
Depiction of Sakoe and Chiba's (1978) correspondence diagram from their speech recognition task. The A axis represents the times of the subjects utterances, and

the B axis represents the times of the model's predictions. The places where they correspond are represented by the C terms. The relationship of all the correspondences is seen as a warping function between the axis.
Figure 5-22: 95
Example relative processing rate display based on decision cycles taken from the Unit episode of Browser-Soar. The straight, solid line is a least-squares regression line through all the correspondences. Its slope is the relative rate between decision cycles and seconds. The dashed lines indicate the expected range for this measure. The location and type of the correspondences are marked on the connected line.
Figure 5-23: Example relative processing rate display based on operator applications taken from the Unit episode of Browser-Soar. 97
Figure 6-24: Original and modified Soar trace. 107
Figure 6-25: PSCM level statistics for approximately 100 decision cycles of the Sched-Soar model (which is shown in Figure 6-27). 110
Figure 6-26: 112
The problem space structure of MFS-Soar (picture taken by David Steier). Learned chunks (small bricks) shown on chunk walls to right of each problem space (triangles). Lines between problem spaces labeled "OP NC" stands for operator no-change impasses in the higher space that are resolved by lower level spaces. The grey fill in the problem space on the right-hand side, *Output-Constraints*, indicates that it has recently been selected to be moved or to have its contents displayed in an examiner window.
Figure 6-27: Normative display of Sched-Soar showing the productions in each problem space as chunks on the chunk wall to the right of each problem space. 113
Figure 6-28: 115
Example descriptive display of Sched-Soar at decision cycle 27. The chunks reported as belonging to each space are not learned chunks, but are the model's own productions loaded as chunks and assigned to spaces based on the algorithm presented in Chapter 6 on the graphic display.
Figure 6-29: 117
Example display of examiner windows of Rail-Soar (Altmann, 1992). The *Switch* problem space has been opened, and the impasse goal *g115* has been opened from it. From within that examiner window (labeled "g115") the *m104* operator was opened, and then the *desired* attribute of that, *Car c32*, has been opened from within the operator examiner by clicking on it. A Soar-mode editor is on the right.
Figure 6-30: 118
The pop-up menu and dialog boxes within the SX graphic display. Moving clockwise, the pop-up menu is followed by a GNU-Emacs window, which has the Soar process buffer as one of its windows. The DSI help window is below that, partially obscured. This help window is accessible from the pop-up menu, and provides general guidance for how to get help, mostly through Soar-mode. At the bottom right is the static display menu that allows the user to create static views of a model on the problem space level. To its left is a dialog box for modifying some of the Soar parameters, and some of the graphic display's parameters. Next to that, on the bottom and left, is a dialog box for setting the Soar learning algorithm. Finally, there is a dialog box for setting the macro-cycle.
Figure 6-31: TAQL-mode templates menu. 120

Figure 6-32: Example TAQL-construct template. 121
Figure 7-33: The problem space organization of the 19 problem spaces in Browser-Soar generated with the SX graphic display. 135
Figure 7-34: The problem space organization of Browser-Soar taken from Peck and John (1992). 136
Figure 7-35: Browser-Soar during a run. 139
Figure 7-36: Browser-Soar problem space organization with productions shown by their problem space. 140
Figure 7-37: 142
Portion of the alignment of the protocol and model trace from the Axis episode. On each row: T is time of subject's actions in seconds. MOUSE ACTIONS is any mouse action. WINDOW ACTIONS are any responses from the actual cT system that the subject saw. ST is the segment type. VERBAL is any verbal utterances by the subject. # is the segment number. MTYPE is type of match, MDC is the decision cycle matched, DC is corresponding Soar decision cycle. Soar Trace holds the model's predictions.
Figure 7-38: Operator support display for the Write episode. 143
Figure 7-39: Relative processing rates display in decision cycles for the Write episode. 145
Figure 7-40: Operator applications vs. subject time display for the Write episode. 148
Figure 7-41: The nine problem spaces in the modified Browser-Soar (see Figure 7-33 for the original structure). 150
Figure 7-42: Operator support displays for the Array episode. The original Browser-Soar predictions are on the top, and the modified version on the bottom. 152
Figure 7-43: DC time based plots for the Array episode. The original Browser-Soar predictions are on the top, and the modified version on the bottom. 153
Figure 7-44: Relative processing rates displays based on operator applications for the Array episode. The original Browser-Soar predictions are on the top, and the modified version on the bottom. 154
Figure 7-45: Histogram of the lags (in decision cycles) of the verbal utterances. 156
Figure 46: The operator support displays for each of the episodes. 165
Figure 47: The relative processing rates displays based on decision cycles for each of the episodes. 167
Figure 48: The relative processing rates displays based on operator applications for each of the episodes. 169

List of Tables

Table 2-1: Examples of protocol datasets, their sizes and ways they can be used to build and test process models, with example experimental studies for comparison. 17
Table 2-2: Summary of previous uses of protocol data to test process models. 20
Table 2-3: Types of protocol analysis tools and their features. 24
Table 2-4: The five major types of measures of model fit and the criteria supporting them. 35
Table 2-5: Types of correspondences between the model's predictions and the data 38
Table 2-6: Ways to deal with mismatches 39
Table 2-7: Types of data that have been used to test process models. 51
Table 2-8: Steps in protocol analysis method (Newell, 1968). 52
Table 3-9: Requirements for the process model's trace. 62
Table 3-10: Requirements for using the model's predictions to interpret the data. 64
Table 3-11: Requirements for analyzing the comparison of the data with the model's predictions. 65
Table 3-12: Some of the model modification clues based on the types of matches in Table 2-5 and their aggregation. 67
Table 3-13: Requirements for modifying the model. 67
Table 3-14: Requirements based on integrating the steps and supporting TBPA with a computational environment. 68
Table 3-15: The features that all parts of the Soar/MT environment share as aids for ease of use and learnability. 77
Table 4-16: Requirements supported by Spa-mode 81
Table 5-17: Requirements supported by the graphic comparison displays and S-mode. 91
Table 5-18: Signature correspondence patterns indicating types of model mismatches. 96
Table 5-19: Further displays for summarizing the fit of data to model predictions 99
Table 5-20: Functionality supported by S-mode. 100
Table 6-21: Requirements supported by the Developmental Soar Interface. 105
Table 6-22: Requirements for the working memory graph examiner. 116
Table 6-23: Overview of the functionality offered by Soar-mode. 119
Table 6-24: Most important commands in the Soar Command interpreter (SCI). 121
Table 6-25: Keystroke savings for Soar-mode accelerator keys, the Soar-mode menu, the SCI, and the SX graphic display compared with the default Soar process. (All measures in keystrokes unless otherwise indicated.) 123
Table 6-26: The size of the TAQL grammars within TAQL-mode and the programming languages supplied with the underlying template-mode. 125
Table 6-27: Descriptions of Soar and Soar model's behavior as search *in* problem spaces, presented in chronological order except for the final quote (All italics in original). 125
Table 6-28: The number of operators, problem spaces, and instantiations of these per run for several Soar models. 127

Table 7-29: 137
Problem space level statistics for the "Write" episode. The top block presents the problem spaces and operators represented in the graphic display. The selection counts for each goal, problem space, state, and opeator are presented in their hierarchical calling order.
Table 7-30: Summary of raw measures for each episode and regression results. 146
Table 7-31: Problem spaces and operators removed from the Browser-Soar model simulating the effects of learning. 149
Table 7-32: Suggested changes to Browser-Soar based on analyses performed. 158
Table 8-33: Survey responses categorized by usage pattern. Each row represents a user. Totals do not include "tried" users. 172
Table 9-34: The ease of use and learnability design features met by each tool in the environment. 182

I Introduction to TBPA

Chapter 1

Testing process models through protocol analysis

"... our confidence in a theoretical explanation of phenomena will be greater the more points of contact there are between theory and empirical observations and the more detailed are the components of the theory that can be confronted directly with data. There is a great deal to be gained, therefore, in the testing of process theories if we can increase the temporal density of our data points so as to increase the number of testable predictions of the theory relative to the number of its degrees of freedom."
Simon (1979, p. 373)

Beginning with Newell, Shaw & Simon (1958; Newell & Simon, 1972) psychologists have examined human performance using information processing models that perform the task being studied. That is, these models predict the sequence of steps a human executes while performing the task. Such models intrinsically demonstrate that the information they use and the way they process it is sufficient to do the task. Both their overt behavior sequences and descriptions of their internal states provide predictions of what subjects will do, what processes and data structures they use, and the order in which they use them. In order to keep the details straight, they have almost exclusively been implemented as computer programs. I shall refer to such cognitive models as "process models", or when the context is clear, simply models.

The process models that I will generally reference in this work will be symbolic cognitive models based on problem spaces and operators (Newell, 1980b), and providing knowledge for an architecture to manipulate (Newell, 1982). In this framework, a model is the necessary knowledge to perform a task along with an architecture to interpret it. In particular, Soar (Newell, 1991) is used to ground this work in an actual architecture. The details of Soar will be deferred until they are actually needed. Other types of cognitive models can also provide process information (van Gelder, 1991), and could be tested in a similar manner — there is nothing known that precludes this, but they will not be directly addressed here.

While small in absolute numbers, process models have been quite influential. Efforts to produce and test such models have lead to the field of information processing psychology, characterized by such successes as models of general problem solving (GPS: Newell & Simon, 1972; LT: Newell, Shaw & Simon, 1958), long term memory (EPAM: Feigenbaum & Simon, 1984; HAM: Anderson & Bower, 1973), learning to program (Anderson, Conrad, & Corbett, 1989) and scientific discovery (Bacon: Langley, Bradshaw, & Simon, 1983; Kicada: Kulkarni & Simon, 1988). The utility of process models is now an established position, and has been extended to other fields (e.g., sociology, Heise, 1989). The reader is referred to other sources for a complete and full rationale of using information processing models to understand human behavior (Greeno & Simon, 1988; Simon & Newell, 1956; Simon, 1990; Neches (1982) notes that they are not a panacea and presents some useful caveats).

In order to test process models' predictions of subjects' internal and external action sequences, corresponding types and amounts of data are needed. Subjects' final responses and total reaction times do not provide enough information to test these predictions. Protocol data are needed, data made up of sequentially ordered (and preferably time stamped) measurements taken while the subject performs the task. These measurements can be motor actions, such as keyboard presses. Verbal protocols, talk-aloud verbal utterances (Ericsson & Simon, 1980; Ericsson & Simon, 1984), are particularly needed to provide a view of the contents of working memory that would otherwise not be accessible.

A few process models have been tested directly against subjects' protocols, and some have been tested against aggregated data, such as reaction times and relative rates of strategy choice. The difficulty of current testing methods has forced most process models to be presented only as sufficiency arguments for the cognitive mechanisms proposed. Their numerous and specific action sequence predictions are not tested against data (e.g., Hegarty, 1988; Klahr & Dunbar, 1988), and are, in a certain sense, ignored.

Getting more out of process models, by reclaiming their sequential actions as theoretical predictions, requires that testing and refining these predictions be more straightforward. This requires that (a) large amounts of data be easily compared with the model's predictions, (b) descriptions of matches and mismatches between the model's predictions and the data are readily available and easy to interpret, and (c) the models themselves can be refined in a reasonably straightforward way.

These three capabilities needed for routinely testing the sequential predictions of process models are not currently available. Interpreting and aligning the data with respect to the predictions must be done by hand; there are no widely used methods to analyze where the model's predictions and data mismatch; and understanding and manipulating the models is difficult because the structures of these models are often only available implicitly in the production rules or other structures in which they are implemented. Thus testing and refining process models using sequential predictions is a difficult, time consuming task that requires a skilled analyst to perform it, and that requires huge amounts of tedious bookkeeping.

This thesis describes a methodology for routine testing of process models through comparison with protocols. Based on the requirements developed through a complete description of the methodology, a computer based environment was designed and implemented: It automates and facilitates the comparison, analysis, and refinement processes. The environment's analysis tools were demonstrated and extended through use on a sample model, and through application by others to subtasks of model testing. The remainder of Chapter 1 provides an overview of the topics addressed in creating this methodology. These include (a) a discussion of the scientific need for the routine testing of process models' sequential predictions; (b) a description of a methodology for routine process model testing through trace based protocol analysis (TBPA) and the requirements for supporting this methodology with a computational environment; (c) a summary of the computer environment developed to support this methodology; and (d) a description of how this methodology and environment were tested on an example data set and model. As part of this testing process, the verbal sequentiality assumption of Ericsson and Simon's (1984) verbal protocol theory (that memory elements are reported in the order that they enter working memory) was also tested, and extended to non-verbal data.

1.1 The need for routinely testing process models' sequential predictions

Process theories need to be easily compared with data on a detailed level in order to completely test them. The more detailed predictions the model makes, the more power and potential utility the model has. Process models make detailed predictions of action sequences. Thus testing them requires detailed comparison with the data they explain, sequences of human actions. The ability to rapidly test theories is a hallmark of a strong, progressive science (Platt, 1964). The inability to do this comparison easily has held back the development of psychology models that can make predictions on the individual action level.

1.1.1 The potential benefits of routinely testing process models' sequential predictions

Making the testing of the sequential predictions of process models straightforward enough to do routinely offers several benefits. Five of the most direct benefits are:

More rapid development of process models. The application of the sequential predictions of process models, in HCI, human factors, and automated testing, to name just a few example areas, has been retarded by the lack of a fast way of testing these predictions. Making the testing routine would allow them to be applied to new areas more easily.

More detailed categorization of human performance. If process models could be compared routinely with data, the comparison could be used as part of model based achievement testing (Ohlsson, 1990), or general computerized testing (Embretson, 1992). This routine use would amortize the effort required to build an initial model, which remains difficult, over larger amounts of data, and provide a

more detailed description of human performance and achievement than is currently available.

More generalizable models. The ability to compare process models with data from multiple subjects would make the tested process models more believable because their generality would have better support. Single subject studies can be appropriate and useful when between subject differences are assumed to be small (Dukes, 1968), but we cannot use them exclusively. Critics have questioned, and rightfully so, the generality of process models when they are only validated against one subject, and the inter-subject variability has not been proven to be small (Kiearas, 1992).

The ability to examine the human cognitive architecture. Models that predict behavior down to sequences of actions are also needed for verifying the details of the human cognitive architecture (Newell, 1973; Newell, 1990). Some architectural details will hide in aggregated non-sequential means. For example, we must know when structures enter working memory if we wish to test whether verbal utterances are always reported in the order that their corresponding structures enter working memory (WM) or if they are based on structures chosen randomly. A model that predicts when information appears in working memory is one way to do this. Each utterance, not an aggregate, will have to be tested.

A step towards automatic analysis of behavior. Facilitating the testing of process models' sequential predictions to the point where it becomes a routine will require a deeper understanding of the testing process. This level of understanding is a necessary step towards the goal of automatic modeling of human behavior. Some of the steps towards making the analysis routine will provide useful lessons for automating all of process model development and testing.

1.1.2 The difficulty of testing sequential predictions

Despite the importance of testing process models against protocol data, it is not often done. The central difficulty of testing process theories appears to be the lack of a well-developed methodology and associated tools for automatically or semi-automatically performing this analysis. The extent of these difficulties, as well as descriptions of applications where it has been done, and tools related to automating it are discussed in more detail in Chapter 2. An outline of the problems is presented here.

If model creation was the only significant cost to developing process models, one would expect the model developers to amortize this cost by testing many subjects and using large amounts of data per subject. While process models are now routinely developed, they are not yet often tested (Kaplan, 1987), and when they are, the number of subjects and measures used is small. The discomfort analysts report in doing this task may be a sign of its undefined character.

The problem is the primitive state of the current methodology and technology used to support it. The method used in early studies (Newell & Simon, 1972; Ohlsson, 1980) was to align the traces by hand, usually on paper. It is still the default approach (Peck & John, 1992). Comparing any model, even informal ones, to verbal protocols, is seen as tedious and dangerously boring by its practitioners, offering plenty of opportunities for mistakes (Newell, P. Reber, Simon, personal communications; Ericsson & Simon, 1984, p. 271). The current methods are too difficult to be done routinely and become unwieldy for large data sets and large models. There have been a few small attempts to assist this process, and there are numerous related tools, but no extant software to comprehensively assist this process. The lack of an explicit methodology may have held back automating it. Having a full description of the methodology is necessary in order to provide automation or facilitation. It is also necessary to teach it to others and move a technique from its originating lab (Hall, 1992). Sociologically, testing the sequential predictions of process theories has not become widespread.

Poor measures of fit. An additional problem in testing process theories is that there are no adequate measures for characterizing the degrees of correspondence between model and data. These are necessary for comparing and improving models. Although a few measures have been put forward, as discussed in the review in Chapter 2, they all have deficiencies. Some attempt to prove that the model matches the data better than chance; others attempt to prove that the model and data do not appear to

be different. Both questions are unreasonable statistically (Grant, 1962; Gregg & Simon, 1967). In all cases the measures indicate only weakly where the model could be improved, which is the most important aspect of these measures.

Lack of shared methodology with other sciences. The problems with testing process models can be contrasted with testing linear equation models. Linear equations are used by nearly every scientific field. For linear equations there exist well-developed statistics and computer packages for describing, manipulating, and analyzing them. With a few exceptions, symbolic process models are unique to psychology, and cannot draw on other areas of science for methodology or tools. We are on our own in this area.

1.2 The steps of testing process models' sequential predictions with protocol data

Figure 1-1 diagrams the development and testing of a process model. The development process starts on the left with the first input to this process, a 0th order model — a model that can perform the task and is a reasonable candidate for accounting for the data. This initial model can be based on an analysis of the task, informal examinations of the protocols generated by a subject (or several subjects) performing the task, or a combination of these two information sources. Protocols can provide informal insights into how subjects do the task and on what they focus their attention. Verbal protocols, because of their information density and access to internal states (Ericsson & Simon, 1984) are often used for this purpose, particularly for pre-theoretical inquiries (e.g., Fisher, 1987). The modeler is not limited to verbal protocols. Eye-traces can provide similar information on the subject's attention and information input (e.g., Hegarty, 1988).

In addition to a 0th order model, the routine steps of model testing require a detailed record of the subject's behavior. Verbal protocols are often used as data because they provide a principled description of the subject's internal mental state (Ericsson & Simon 1984). Multiple data streams can also be used, such as the motor actions performed to do the task along with the concurrent verbal protocol.

With these inputs in hand the testing loop of trace based protocol analysis (TBPA) can be approached. The first step is to run the model on the task, generating a trace of its external task actions and intermediate internal states and actions. The trace of the model's actions are predictions of the subject actions and the contents of their utterances.

The second step is to compare the predictions to the data. This is done by aligning the two action sequences, interpreting the subject's actions with respect to the model's actions. Aligning and interpreting the two information streams is often a difficult task when done by hand. Even with a good model, the actions may not directly correspond. The model may perform additional task actions that the subject does not, and vice versa. The model may include items in its trace that are not visible in the subject's data, such as initializations related to being a computer program, and the subject's protocol may include non-task utterances (yawns, jokes, and so on).

The results of the low level comparisons of the predictions with the data need to be aggregated and summarized with respect to the model that generated the predictions to be understood. These analyses can be used for two primary purposes, and the final processing steps depend on the desired result from testing.

If the model is considered mutable (an open analysis, Ohlsson, 1990), the results of the comparison will be used to evaluate, validate and improve the model. For this process, one needs to know the locations where the two sequences do not correspond. These locations are used as leads to improve the model, and the loop of running the model and comparing its output with the subject's performance is repeated.

If the model is considered fixed (a closed analysis, Ohlsson, 1990), the results of comparing the

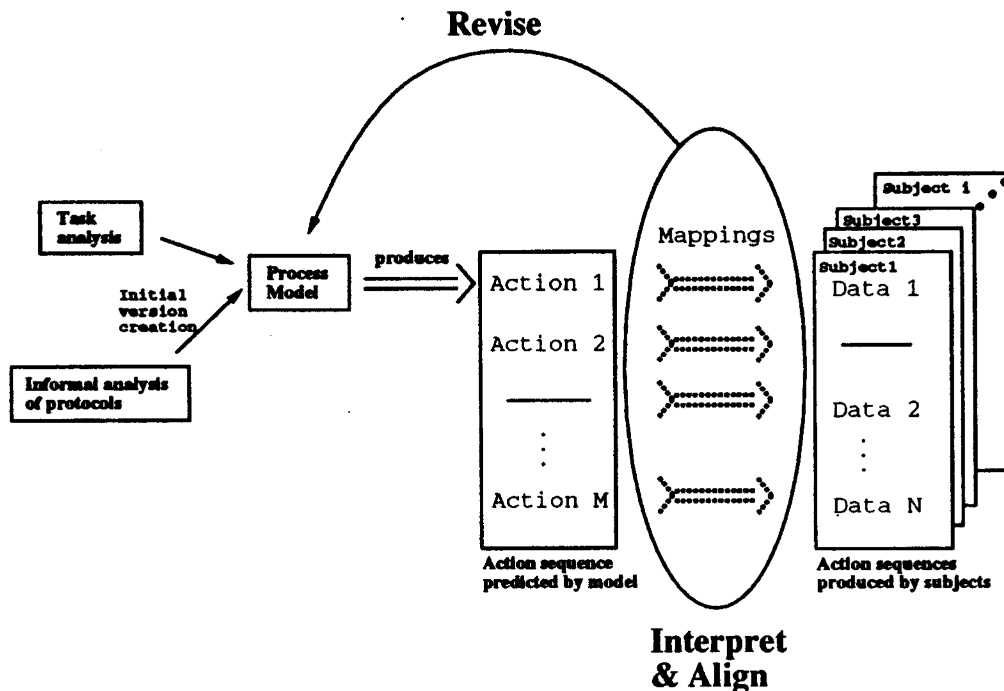


Figure 1-1: Schematic of trace based protocol analysis.

subject's and model's actions are used to evaluate how closely the subject's performance matches the model, and if the comparison is to be used to diagnose the subject's behavior, where the model fits and where it does not (Brown & Burton, 1980). If there are several similar models, the results can also be used to assign an interpretation to the subject's behavior. This thesis concentrates on the first approach, evaluating and improving the model.

1.3 Developing a methodology for routinely testing process models' sequential predictions

This thesis describes a methodology, called trace based protocol analysis (TBPA), for routinely testing process models with both verbal and non-verbal protocols. The requirements for each step in the process are noted in a detailed manner, so that a computer environment to support this methodology can be designed and created. Finally, the methodology and the environment are demonstrated and improved through application to a sample process model. This demonstration example included testing the sequentiality assumption of Ericsson and Simon's (1984) protocol theory, and extending the assumption to non-verbal protocols as well. These four main accomplishments are briefly described in this section, and also serve as an introduction to the rest of the thesis and an outline of its contents.

1.3.1 A detailed specification of what is necessary for routine testing of process models with protocol data

Chapter 3 presents a detailed description of trace based protocol analysis, a methodology to test process models using protocol data that can be made routine. Testing process models this way is not new or unique, but specifying how it is done in a detailed manner is. The major steps of this method are (1) running the model to generate a sequence of behavior, (2) aligning this with the subject's action sequence, (3) analyzing this fit to generate suggestions of where the fit could be improved, and (4) modifying the model. As each step of the methodology is defined, the requirements to support it in a

computation environment are noted. Where appropriate these requirements are illustrated by examples drawn from the review of previous analyses and tools presented in Chapter 2. Taken together, the requirements form a specification for a computer environment and analytic measures to support and help automate the testing of the sequential predictions of process models.

Creating a specification of this methodology is a necessary step for automation, but it may have two other direct effects: (a) It will remove some of the guess work in the analysis, which may make the process less difficult and more pleasant to do by hand. (b) The specification will allow it to be more easily taught.

1.3.2 An environment to support the needs of routine testing of process models

A computer environment called Soar/MT is built to support the requirements noted in the specification of TBPA. This environment is a direct step toward automatic model testing with TBPA. It has three main systems. They are described separately, but they are integrated with each other. By using the model and its predictions as common data structures, data can be passed between the separate tools, and analyses may draw from more than one system's representations and functionality. This supports more powerful analyses than can be provided by any single tool. Each system can be menu driven, and a pathway to expertise is provided through providing help and keystroke accelerators on the menus. This simple learning approach is backed up by complete manuals and on-line help.

1.3.2.1 A tool supporting the interpretation and alignment of the data with respect to the model's predictions

Spa-mode (Chapter 4) is a tool that helps the analyst do the initial interpretation and alignment. Spa-mode can automatically align simple, unambiguous portions of the model's actions with the subject's actions, and tools are provided for aligning the remainder semi-automatically. By displaying the actions and their correspondences in a tabular format, direct visual examination of the comparisons are possible. Spa-mode is based on a spreadsheet, so the spreadsheet's commands can be used to perform simple aggregations such as the number of data points matched by the model, or the average number of words matched. As an extension to the GNU-Emacs editor environment, it includes all the GNU-Emacs text editing commands, and a macro language for creating temporary analyses and for adding permanent extensions.

1.3.2.2 A measurement system for telling where a model needs improvement

The most important output of the testing process is a description of the model's predictions fit the data. So that the model's fit can be improved, the descriptions should indicate, in terms of the model, where and how the predictions do not match the data. A set of measures designed to show the correspondences and misfits with respect to the model is implemented as two families of displays in the S statistics package (Chapter 5). One set shows the subject's actions with respect to the model's actions and structures, and the other set shows the relative processing rates based on the correspondences in seconds and simulation cycles. Information about each point on these graphs is available by clicking on them.

These analyses are only a subset of a large number of similar displays. An example of modifying a display by changing the unit of model time from simulation cycles to operator applications is presented. Additional graphs and modifications will be required as additional users analyze more models, so an environment for designing and manipulating these types of displays is also provided.

1.3.2.3 An interface for tracing, understanding, and modifying models

The Developmental Soar Interface (Chapter 6) is a tool that supports the needs associated with running and manipulating models within the Soar architecture (Laird, Congdon, Altmann, & Swedlow, 1990; Laird, Newell, & Rosenbloom, 1987; Newell, 1990), starting with the very first step of producing

predictions that can be used in an automatic interpretation and alignment system. It makes the implicit structure of the Soar models available graphically for inspection by the analyst and in an internal format for use in analyses by the other tools. Integrated, structured editors provide the ability to directly edit, examine, and load the building blocks of Soar models.

1.3.3 Documentation of the utility of the environment and methodology

As a new methodology for routinely testing process models' sequential predictions, the methodology and the environment supporting it are demonstrated and developed through an example of its use. Performing a sample analysis also provides feedback to develop the methodology and environment further. As this methodology is of routine testing and not a theory of scientific abduction and initial creation of models, the applications of the environment starts with a previously created model and previously gathered data. The testing methodology and environment are first validated by duplicating existing analyses. Then, with the increased power and capabilities of the environment, some new analyses are performed that further indicate where to modify the model, and provide new descriptions of the Soar architecture.

Browser-Soar (Peck & John, 1992) and its data are often used as examples, and Soar/MT was used to duplicate and extend the tests of the Browser-Soar model against its data. Browser-Soar is a model of how a user interacts with an on-line help system. It has been tested with ten, approximately one minute episodes from a single subject. The reanalysis showed where to improve Browser-Soar, often validating problems previously known to the model's authors, but unknown to the current analyst. In addition, the nature of Browser-Soar and the data supported initial measurements of the Soar architecture's speed, finding it to be approximately as predicted at 10 decision cycles per second. The environment and its parts have been used by other users, and a survey of their evaluations is included (Chapter 8).

1.3.4 Testing and extending the sequentiality assumption of verbal protocol generation

By putting the Browser-Soar model into closer contact with the data than its developers had been able to do by hand, the sequentiality hypothesis of Ericsson and Simon's (1984) theory of verbal protocol generation was tested (Chapter 7). The sequentiality assumption specifies that structures in working memory are reported in the order that they enter. There are two ways to test this, one is to have an empirical measure of when structures enter working memory, and the other is to use a theory to predict when structures enter.

Browser-Soar was used to predict the entrance of information into working memory. As predicted by Ericsson and Simon's (1984) theory, verbal utterances were produced in the order that their underlying structures appeared in working memory. This was true across the ten episodes and 630 seconds modeled by Browser-Soar. The analysis also suggested three extensions to Ericsson and Simon's theory: (a) that verbal utterances were not often prospective, and tended to be retrospective by 1-3 s. (b) The sequentiality assumption holds for non-verbal task actions with respect to other non-verbal task actions, but (c) Sequentiality should also hold across the two modalities, with the non-verbal motor actions serving as useful reference points for measuring the lag of verbal utterances. These are not predicted by Ericsson and Simon's theory, but are consistent with it.

Chapter 2

Testing process models with protocol data: Review of past work

This chapter reviews the previous uses of protocol data for building models and testing them, the various tools that have been built to assist in this, and the measures of model fit to sequential data that have either been computed by hand or incorporated into tools. The lessons noted here will be useful guides for the methodology for testing process models developed in the next chapter.

Process model testing does not live alone, but on a continuum along with data analysis, model generation, and model refinement. These activities can be viewed as manipulating two information streams, the subject's action sequences and the model's action sequences. The various uses of protocol data can be compared with respect to the strength of the model being developed, and how it uses the two information streams. With a more detailed description of this process we can explain the apparent difficulty of using protocols in general, and for testing process models in particular.

Surveying the previous tools for manipulating and comparing protocols to a model indicates several general needs. On the data side, these include the ability to edit the protocols on the segment level, to view large numbers of segments simultaneously, and to compute aggregate measures of them. Similar tools are required of the model, but are less often provided. The structure of the model must be available, and its performance must be aggregated if it is to be compared with aggregate subject measures. The correspondence between the model's predictions and the data must be computed and viewable. It is very enlightening to be able to directly compare these aggregate measures with the model. Similar tools are required for gathering and manipulating the model's behavior, but they are less often available.

Examining the previous measures of model fit indicates several measures that should be included in any future tool. Being able to aggregate the model's behavior in order to describe it and compare it with the subjects behavior is also important. This cannot be directly incorporated into any tool, but must be indirectly supported.

2.1 The possible relationships between process models and protocols

This section creates a framework for organizing most uses of protocol data with respect to process models. The use of protocols are shown to exist on a continuum from model building to model testing. The comparison between the model's predictions and the protocol data can be described as comparing two information streams at various levels of resolution. Examining these previous uses of protocol data provides several guidelines for creating tools for testing process models with protocol data.

Figure 2-2 shows an information stream description of the sequential data structures, analysis, and possible data comparisons between process models and subject data. These operations and information streams are a superset of the data and operations for less predictive models so they provide a conceptual framework for all analyses of action sequences in terms of the information being manipulated, its transformation and losses, and aggregation processes. While only a single stream is represented in the diagram, each virtual stream can be made up of several streams. For example, the human information stream may contain separate streams for verbal protocols, eye movement protocols, and key presses.

The desired end of architecture + knowledge. When developing a process model the analyst starts in the upper left corner of Figure 2-2 with a *mind + knowledge*. We believe that the most desirable analysis is to find out what is in that box, that is, what is necessary to recreate the mind's behavior? What is the *architecture* of the mind, and what *knowledge* must it have to do a particular task? We attempt to duplicate this box by creating a process model that is also broken up into an *architecture* to interpret and apply specific *knowledge* to perform the given task (Newell, 1990; Newell, 1992). The knowledge represents the information required to perform the particular task, and the architecture

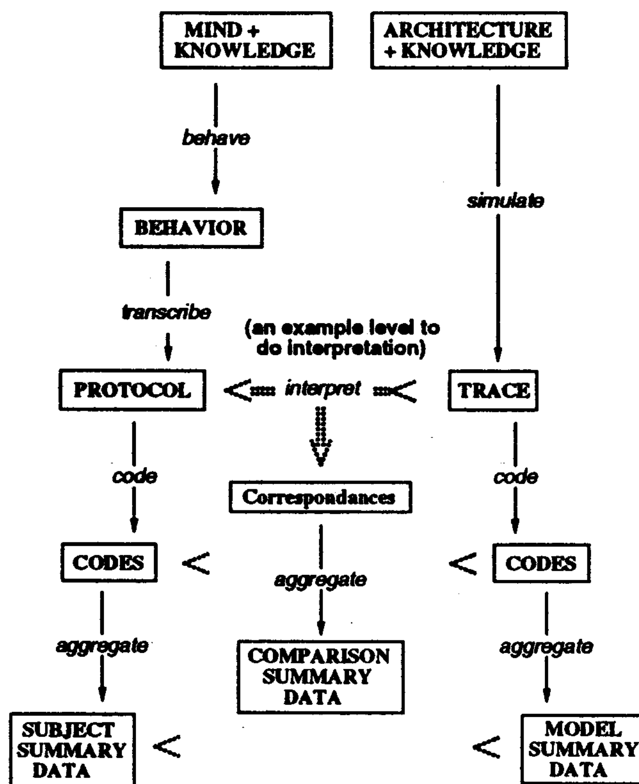


Figure 2-2: Information streams of subject and model used in testing process theories.

represents the fixed structure between tasks. Therefore, the model being tested consists of task knowledge and perhaps some general knowledge, within an architecture to apply it. Creating the model can be viewed as a type of programming task, of putting knowledge (programs) into the architecture (interpreter), except that testing the models is more difficult than testing programs because it is based on conforming to various non-obvious and indirect constraints, such as learning rates. If we choose the right architecture, we should be able to handle all situations. Modeling a situation then consists of specifying what knowledge is used in that situation.

The human information streams. The mapping between *mind+knowledge* and *architecture+knowledge* is not straightforward. We cannot open up a mind and examine its contents, we are only allowed to observe it *behave*. *Protocol* data are transcriptions of behavior that retains their sequential nature — data that remain ordered with and can be understood only with respect to other temporally contiguous actions. The data can include all types of human behavior: verbal utterances, key presses, and mouse movements. This data inherently occurs in time, and can have time stamps associated with it. This time stamp must be applied when it is *transcribed*, that is, recorded from the environment into a format that can be processed.

The sequential nature of the data can also be discarded. This results in non-sequential data that can be more easily aggregated and analyzed because it can be directly aggregated. Much of classical experimental psychology uses only behaviors' time stamps as non-sequential data, and while it is a valid subset, it is one with less information.

As a small example of how much information is lost when the time stamp is discarded, consider how much information is recorded when 64 subject actions are recorded, with these actions coming from 64 equally likely categories, and measured (time stamped) with 100 ms accuracy over a range 0 and 100 s. It takes 22 bits to record each measurement (6 for sequential position, 10 for reaction time, 6 for category). Discarding the sequential position of the measurement removes 6 of the 22 bits necessary to record each action.

Any transcription process will introduce noise and remove information. These losses may be small and systematic (e.g., most button presses can be recorded accurately to within 1 ms). The losses can also be large and disruptive. In particular, verbal protocols can lose information where the speaker did not speak distinctly, and noise (in the sense of erroneous information) can be added by misunderstandings of the transcriber. Information is also lost from the original protocol because such things as pauses, inflections, and environmental context are often not transcribed. Anything not transcribed is gone.

That is not to say that the protocol should not be transcribed, *coded* to create local summaries, or *aggregated* to create global summary statistics. In its original form the data may be unwieldy and the signal to noise ratio may be large. By condensing it one often ends up much better off; although less information remains further down the information stream, it is in a more understandable form.

The coding process can also be influenced by the level of the model and its stage of development. Models on the knowledge level (Newell, 1982) predict the knowledge applied. Lower level models may predict symbolic actions. The interpretation and comparison of the subject's actions with the model's would vary in each case. In each case the subject's behavior must be interpreted with respect to the model's predictions. Initially, when a model is being built, the codes may be derived from the data. When a model is being tested the categories are taken strictly from the model. Subject actions that cannot be interpreted this way are indicators where the model could be improved.

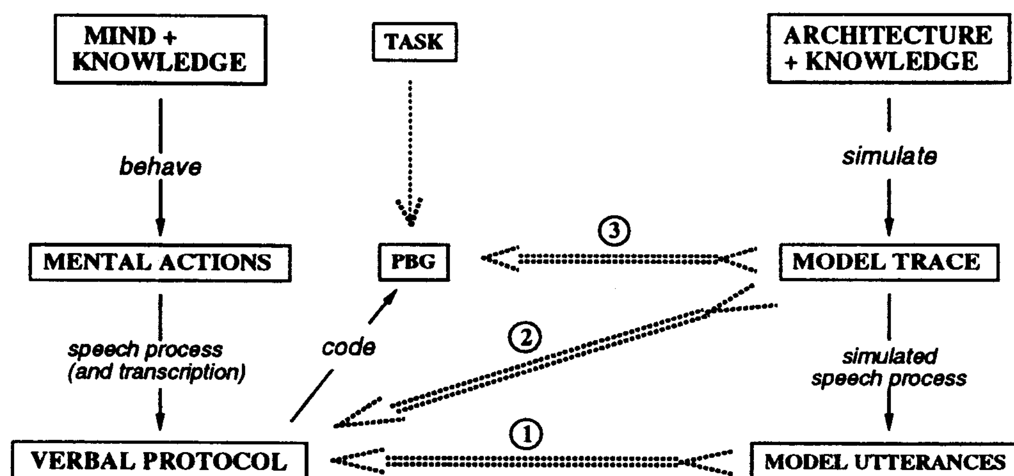


Figure 2-3: Schematic of possible levels of comparison between model and data. (Numbers are referred to in the text.)

The simulation information stream. Running the simulation *architecture+knowledge* (on the top left of Figure 2-2) produces the simulation's *behavior*. While this process should be completely accurate, the model traces never describe all the model's internal functions. The model builder often has a choice of how much information to make available in the trace. Like the *mind* information stream, the analyst

may further *code* and *aggregate* the information for subsequent analyses.

Comparing the information streams. After subject data have been collected and summarized to the desired level, and the simulation has been created based on task analysis and/or inspection of the behavior, the two information streams are ready to be compared. This will be done not only to indicate how similar they are, but also to provide information on how to improve the model. Figure 2-2 shows that there are several possible levels of comparison between the information streams as indicated by the arrows pointing in. A comparison consists of *interpreting* the protocol data with respect to the predictions of the model, to produce *correspondences*. Finally, these correspondences that can themselves be *aggregated* to produce *summary statistics* of the comparison, or displays indicating the major features of the model and how its predictions interpret the data. Although they will have a different form, the correspondences and summary statistics here serve the same role as residuals in linear regression, indicating the quality of fit locally and the direction of actions that must be taken to improve it.

The single most information-intensive approach is to compare the two information streams at the level of the *protocol* and *trace*. If the information in each information stream is coded before comparison, there is less information compared, but the comparisons may require less effort. If the model does not completely predict all the subject's actions on the protocol level, understanding the regularities that it does not match and where it needs to be improved can only come from comparing aggregate measures.

Some notes about verbal reports as data. For the most direct theoretical support, equivalent model and data constructs should be matched. If the process theories we are testing are about linguistic production, then verbal protocols are data on the most fundamental level. However, process models are usually not about linguistic production, so the verbal protocols must be interpreted in some way.

Figure 2-3 diagrams the relationship between verbal utterances and the process model's trace. If we are strictly to compare only equivalent items, we must compare the subject's overt task actions with the model's task actions, and the subject's verbal utterances with utterances generated by the model. This strict interpretation is depicted by the comparison arrow between verbal protocol box and the model utterances box (number 1). This approach has rarely been tried, but in a singular example Ohlsson (1980) showed that a simple model of utterance produced reasonable prose in a limited domain.

Fortunately, we do not have to add a speech process to our models. There is a theory of verbal protocol production (Ericsson & Simon, 1984) that under most conditions allows us to directly interpret the subject's verbal utterances as representing a subset of their mental representation of state information or operator application. This allows us to use the more direct comparison drawn between the verbal protocol box, and the model trace box (number 2), representing the comparison between the model's predictions of what could have been said, and what was said.

If the verbal utterances are difficult to decipher, we may choose to explicitly apply this theory of verbal production and create a coded representation of the subject's mental state (number 3) as a problem behavior graph, but we will take the approach in this work of using direct comparison wherever possible. It is simpler and subjects the data to one less transformation. Similarly, task actions are assumed to follow directly from their mental representation. The implementation of motor outputs is a separate process, but by definition cognitive models are not models of motor output. The effects of motor processing are usually kept small and ignored.

Predictions of actions should be matched by procedural data, and predictions of declarative mental structures should be matched by data representing declarative mental facts. Most verbal (and non-verbal) protocols in cognitive science are procedural data, a trace of an ongoing procedural task, which should be compared to the trace of a process model. When analyzing a protocol for declarative information, for example, content analysis (Stone, Dunphy, Smith, & Ogilvie with associates, 1966; Carley, 1988), matching verbal descriptions of declarative information to static declarative relationships is just the right thing to do.

We can now note the distinction between overt and verbal behavior as support for a cognitive model. Different aspects of the model match different data streams, that of verbal goals and states, and those of overt, motor actions. Both are support for the model, showing that it did predict mental or overt subject actions. The biggest difference is that the verbal utterances include an additional layer of theory specifying how they match, that of verbal protocol production.

Some analyses, typically those developing expert systems (Brueker & Wielinga, 1989; Shadbolt & Wielinga, 1990), compare procedural data to their static model. In this case, they are comparing disparate object, verbal utterances generated while performing the task to long-term knowledge structures or the static elements of a process model. Presumably this works well enough because they are using this to develop a model, which will not be tested on a fine grain level, and the rules are correct enough that they would fire in such a situation. Given a complicated enough model, the emergent properties of the model will prevent matching the procedural data to a description of a procedural model. For example, new operators could be created from existing knowledge or from new knowledge learned from the environment.

This view of protocols as data to test models suggests that the analyst is not just trying to "assign" a trace element to a subject segment, with that accounting as the sole result. It is not. The assignment of segments is a way to test a model, finding support for its components in the protocol. This approach equally includes the desire to know where the predictions fail to match the protocol so that the model may be improved.

2.2 Review of creating and testing models with protocol data

The framework shown in Figure 2-2 supports the comparison of research approaches based on which information streams they use, the processes they use to transform data, and any tools used to automatize the transformations or aggregations. Consider as a straightforward initial example under this framework, experimental psychology. It generally lives only on the left hand side information stream of human data, and as mentioned above, on a subset of that data even. Reaction choices and times are taken, often automatically coded by computer-based apparatus into categories and aggregated into means and other numeric measures by common statistical software.

As a more complete example, consider Qin and Simon's (1990) work examining the process of scientific discovery. Subjects (mind+knowledge) talked aloud (producing a protocol) while they attempted to find Kepler's laws of planetary motion based on Kepler's data. The subject's utterances were recorded on audio tape and transcribed into text (protocol). The subject data was coded and aggregated by hand into the functions examined, showing that the subjects' work appeared to be done as search in problem spaces (summary data). On the model side, they gave Bacon, a scientific discovery system, the same task. They took a trace of its operations while it solved the problem. The trace was coded and summarized by hand. The aggregations summarized Bacon's behavior as search in problem spaces (summary data). Qin and Simon then informally compared the summary of the subjects' behavior with that of Bacon, and were able to conclude that the mechanisms in Bacon are sufficient to account for the subjects' ability to perform the task, and the style in which they did it. (interpret and code). Their comparison (as briefly explained in this simplification) is easy to follow partially because it used only summary data.

2.2.1 Exploratory analysis leading to process models

Figure 2-4 depicts several of the levels of theory that are built and tested with protocol data. Levels near the top represent exploratory analyses necessary for creating models that can later make more concrete predictions of behavior. Analysis of the data creates the model. As the model becomes more predictive (middle line), it is derived less from the data at hand and attempts to interpret the data. When a model makes predictions of actions sequences (bottom line), it becomes a process model. It is no longer directly influenced by the data during analysis. Its predictions are first compared to the data, and the areas where they mismatch may suggest where to modify the model.

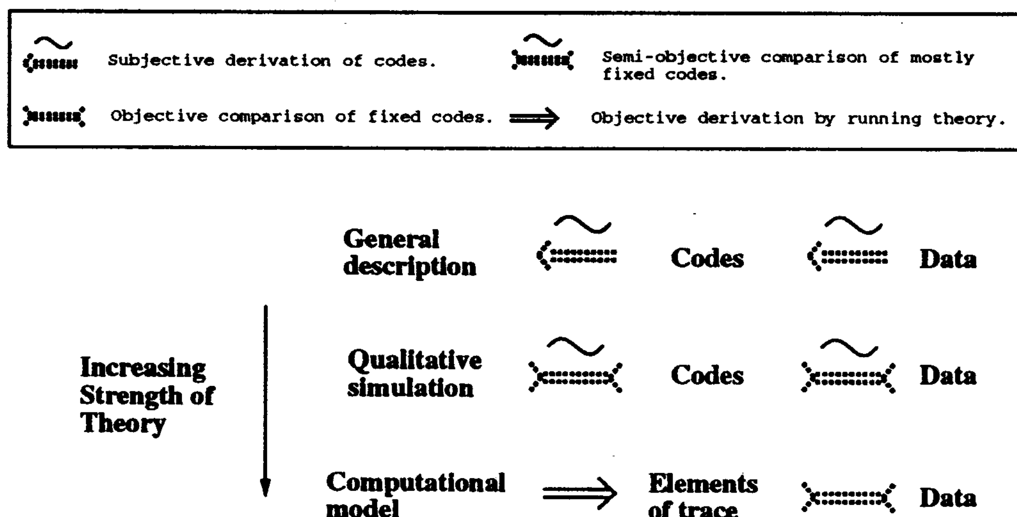


Figure 2-4: How theory level influences comparison with data.

It is worth noting how process models are developed, for many of the same subtasks will be used in refining them after testing. The first line in Figure 2-4 refers to coding that starts out without a model of the subject's processing or knowledge and leads to at least a mental simulation of the subject's structures. The coding process is data driven and is done to derive the underlying actions and the process generating them. This work can be characterized by categories that are general and that may change during the analysis. The first analyses of the data provide only aggregate measures of the human behavior. This exploratory coding works primarily with the human information stream. There is little formal or direct comparison with a model. The aggregate measures produced by the analyses create constraints and hints for creating a model that can be mentally simulated.

Table 2-1 shows several examples of each types of analyses, including two experimental psychology studies for comparison. From left to right, its columns are Citation, a citation for the study; Domain, the task domain studied; N, the number of subjects used in the analysis; Data Points, the total number of data points as defined by the experimenter or protocol segments, analyzed or compared to a model; Total time, the total subject time in seconds included in the analysis; Data types, the data types taken, in the first subcolumn S is sequential data where each action is ordered in time, N is non-sequential data where the subject makes a single response, like an answer to an arithmetic problem, O is overt task actions, I is eye data, and V is verbal data; Codes, the type of codes derived from the data for the analysis, as in Figure 2-2; How, how the codes were assigned to the data, by hand, semi-automatically with a tool, or automatically by a tool; Model compared, how the model was compared with the data or what analyses were performed; Model, the type of model tested with the data.

Examining the data in this way is necessary when a model is not available or not yet developed, when a complete understanding is not necessary, or when time is not available to derive it. For example, fast user testing of computer interfaces (Kennedy, 1989) by categorizing user's actions into errors and correct keystrokes does not require a complete model to indicate where they may need improvement. This style of work particularly exists in areas that are trying to create initial models (such as computer supported cooperative work; Olson, Olson, Storrosten, & Carter, 1992) and knowledge rich areas that are still difficult to model (for example, computer programming, Fisher, 1987, and process control, Sanderson, Verhage, & Fuld, 1989). Taken as a group, these types of studies (shown in Figure 2-1) require relatively larger data sets than are used for testing process models, and verbal data are always used, and sometimes overt task actions are also collected.

Citation	Domain	N	Data Points	Total Time	Data Types	Codes	How Coded	Model compared or Analysis Performed	Model
Typical experimental psychology studies: (examples)									
Reder & Ritter 1992 - I	Arithmetic	34	40k	146 ks	NO..	RT	auto	regression	Log&LinReg
Woltz 1989	Skill acquisit.	701	1.3M	9 Ms	NO..	RT	auto	regression	LinReg
Exploratory coding of verbal protocol: (examples)									
Fisher 1987	Programming	3	na	18 ks	SO.V	Operator	semi	cycle detection	op names
Bree 1968	Reading	12	~4k	21 ks	S..V	Strategy	hand	auto PBG f/ codes	qualitative
Olson, et al. 1992	Design w/ CSCW	6	na	32 ks	SO.V	Action	hand	counts	none
Sanderson et al. 1989	Process control	12	~2.9k	108 ks	SO.V	State	semi	transitions	state types
Process-based trace comparisons (examples):									
Dillard et al. 1982	Accounting	3	210	na	S..V	KnowLev	semi	match to codes	fixed manual
Wagner & Scurrah 1971	Chess	2	271	7 ks	S..V	State	hand	rule match to state	gen'l rules
Bhasker & Simon 1977	Chemical Eng.	1	280	na	SO.V	Operator	semi	assignment to model	fixed manual
Gascon 1976	Seriation	33	1153	na	SO..	Operator	hand	match to codes	structural
Development and general comparisons with procedural model (examples):									
Anzai & Simon 1979	Tower of Hanoi	1	224	5 ks	SO.V	State	hand	informal comparison	simulation
Karat 1982	Tower of Hanoi	192	10k	~6 ks	SO..	State	auto	linreg on time/rule	simulation
Ohllson 1980 - II	Lin. syllogism	10	1.5k	17 ks	C..V	PBG	hand	informal	qualitative
Qin & Simon 1990	Sci. Discovery	6	na	32 ks	SO.V	State	hand	aggregates comparison	simulation
Simon & Reed 1976	Mis'ry/Can'bls	100	2500	na	S..V	State	hand	relative state rates	Markov
Atwood & Polson 1976	Water jugs	229	13.5k	na	SO..	State	auto	relative state rates	sim & Markov
Hegarty 1988	Diagram use	40	4.8k	36 ks	S.I.	Vis. area	auto	informal comparison	simulation
Klahr & Dunbar 1988	Sci. Discovery	30	na	~48 ks	SO.V	State	hand	informal comparison	description
Newsted 1971	Concept Ident.	2	~1.2k	144 ks	SO.V	KnowLev	hand	aggregate	simulation
Aggregate model time stats vs. Subject states (example)									
John 1988	Typing	29	regularities			Aggregates	hand	qual & quantitative	process
Full trace based comparisons, non-sequential response data (example):									
Thibadeau et al. 1982	Reading	14	~27k	7 ks	N.I.	RT	auto	regression on time	simulation
Category coding of declarative verbal statements (examples):									
Carley 1988	Relationships	16	na	115 ks	N..V	KnowLev	semi	insert into net	sem nets
Stone, et al. 1966	(varied)	>100	>100k	360 ks	N..V	Concept	auto	content analysis	none

In the data type column: N indicates non-sequential data.
S indicates sequential data.
O indicates overt task actions protocols.
I indicates eye tracking protocols.
V indicates verbal protocols.

Table 2-1: Examples of protocol datasets, their sizes and ways they can be used to build and test process models, with example experimental studies for comparison.

As a final analysis, exploratory coding has several drawbacks compared to one done with a model in hand. By creating the codes from the data, it decreases the degrees of freedom in the analysis (Ericsson & Simon, 1984). Without a functional model in hand, the codes may be unequal sizes, and aren't defined operationally. As new ideas are explored the comparison between predictions and observations can end up being informal. Unless rigorous coding schemes are put in place for the coders, the codes may be difficult to keep in focus as they will change as more data are coded, and the model is further developed.

As the analyst forms a mental model of how the subject performs the task, the coding task changes to semi-theoretical coding (Figure 2-4, line 2). The derivation will be incomplete, but will be better motivated than those with no model in mind at all. Analysts with a mental model in mind may be informally comparing the subject's action sequence with the action sequence or its components that the mental model might produce. The codes will have more meaning, corresponding more directly to mental operations or mental state information, but are not yet completely formalized. Typically the aggregate measures taken are the number of codes in each category. Detection of behavior cycles through such measures as sequential lag analysis (Gottman & Roy, 1990) becomes more important because they are used to suggest action orderings for the model. As a step towards process models, users may also create problem behavior graphs of subjects (Bree, 1968).

When the process model is not yet implemented as a simulation, but its actions are fairly well defined, the subjects sequential behavior can still be examined by comparison with something other than a trace. Bhaskar and his colleagues (1978; Bhaskar & Simon, 1977; Dillard, Bhaskar, & Stephens, 1982) compare actions in the subject's sequential behavior to the steps in a block diagram of the process. The coded protocol can also be examined by a pattern matching program to find strategies based on their signature patterns (Gascon76, 1976), or some of the rules for behavior can be tested by hand against the data (Wagner & Scurrah, 1971). As shown in Table 2-1, these studies typically use less data than more exploratory ones.

2.2.2 General testing of process models

The last line of Figure 2-3 represents testing the sequential predictions of a process model with data. Once a process model has been created, it can be run on the studied task, generating a trace of its implicit and overt behaviors. This is the model information stream in Figure 2-2. The model's actions are no longer being derived from the data, but the model is attempting to predict the data. This is a more powerful result than the transition probabilities that sequential lag analysis provides.

There are many degrees of freedom in these models, which correspondingly require large amounts data to test them. Each production (or subprocedure specifier) is like a parameter in a regression, in that it is included to explain or account for some of the data. Therefore, proportional to the degrees of freedom in the model, large amounts of data are needed to create and test process models. This data has been found in verbal and non-verbal protocols, and sometimes from aggregate measures taken from other studies.

Process models are more difficult to build and manipulate than verbal theories because they are more detailed and make more direct predictions of the subject's action sequences. We can also take additional measures from them as they perform the task. These measures have included the time to do the task (Carpenter, Just & Shell, 1990; Dansereau, 1969) and information attended to in the environment (Hegarty, 1988).

Most often these models have been tested against the data through aggregate measures. In the terms of the diagram depicted in Figure 2-2, the model is run and its behavior is summarized into aggregate measures, which are then compared with the aggregate measures taken from several subjects. These aggregate measures have included state transitions (Atwood & Poulson, 1976; Simon & Reed, 1976), time to make state transitions or apply operators as indicated by overt task actions (John, 1988; Karat, 1968), and eye movements (Hegarty, 1988; Just & Carpenter, 1980). More often just general regularities that subjects exhibit are compared with general descriptions of the model's behavior, such

as strategy preferences (Hegarty, 1988; Klahr & Dunbar, 1988; Qin & Simon, 1990; John, 1988). These studies use more data than those that start to test process-like models, partially because the data set is also used to create the model.

2.2.3 Trace based protocol analysis

The third line of Figure 2-4 depicts directly comparing the process model's trace with the subject's actions and verbal utterances without aggregation of either information stream. This is called model-trace based protocol analysis by Ohlsson (1990), and trace based protocol analysis in this thesis.

When subject actions are assigned to match a trace segment of the model, the codes assigned have a deeper meaning; segments of data are not being grouped into general categories, but are being assigned as supporting a specific, formal prediction from an IP model about the type and order of the action sequences used to perform a task. The organization of the model determines which actions and codes are similar and can be aggregated.

Process models have been primarily tested on this level with three types of data: verbal protocols, overt task action protocols, and eye movement protocols. Verbal protocols were the first to be developed as a way to test process theories. As specifications of working memory and operator applications (Ericsson & Simon, 1984) they provide the most detailed data points to match. The top section of Table 2-2 presents a list of studies that have tested process models with verbal protocols. It is nearly a complete list, and there are probably no more than ten more models tested with verbal protocols. In sharp comparison, consider the position of another, much simpler modeling technique, multivariate analysis. In 1980 it was estimated that there was over 10,000 papers in the literature (Bentler, 1980).

As a group these studies are fairly homogeneous. They nearly all deal with problem solving on a puzzle type task and studied a small number of subjects. There are four studies that then stand out. Three do so because of their size. The original Newell & Simon (1972) work, Ohlsson's (1980) second study, and Larkin, McDermott, Simon, & Simon (1980), all used more than one subject, and examined relatively large amounts of data. Peck & John's (1992) work stands out because it is not of problem solving, but of performing what is claimed to be routine behavior, that of using an on-line help system. In all cases, the number of verbal segments that have been used to directly test the sequential predictions of a process model are relatively small.

Overt task actions have been used most often to test process models, so often that it is not possible or interesting to enumerate all of them. They have been taken for a variety of tasks and a variety of methods. The most common types that have been used are mouse and other computer input devices (see Table 2-2 for examples). The protocols are often directly taken by computer, but don't have to be. Transcribed or actual pen actions have been used to model subtraction. Young and O'Shea (1981) used marks generated on paper as byproducts of the task of subtraction to test their theory of subtraction bugs, and VanLehn and Ball (1987) used ones transcribed by an electric pen. While eye movements are fairly routinely used to develop informal models (Hansen, 1991) and process models (Reader: Just & Carpenter, 1980), they are probably the least used to test process theories directly, and are rarely as the only data source. Table 2-1 indicates that when the overt task responses can be treated non-sequentially, relatively large amounts of data can be compared with the model's predictions (e.g., Thibadeau Just, & Carpenter, 1982).

Some models have been tested with more than one type of data, and doing this appears to be useful. Young (1972) used primarily task actions (moving blocks) along with some eye movements to build and test a model of children solving seriation tasks. Peck & John (1992) used verbal utterances and task actions (mouse movements and clicks) to build and test a model of a user searching for help in an on-line browser. Newell and Simon (1972, pp. 310-327) report on Winikoff's work (Winikoff, 1967) using eye movements as an adjunct to verbal protocols.

Citation	Domain	N	Data Points	Total Time	Data Types	Coding	How Coded	Model compared or Analysis Performed	Model
Full trace based comparisons, continuous data: (believed complete)									
VanLehn 1989 - I	Miss. & Cann. 5	1	9	na	S..V	State	hand	trace	simulation
Ohllson 1980 - I	Lin. syllogism	1	56	220 s	S..V	PBG	hand	trace	simulation
Greeno 1978	Geometry	6	72	na	SO.V	KnowLev	hand	trace	simulation
Koedinger 1990	Geometry	5	98	na	SO.V	Operator	hand	trace	simulation
Luger 1981	Physics	5	102	na	SO.V	KnowLev	hand	general trace	simulation
Feldman 1959	Binary choice	1	200	~1 ks	S..V	Trace	hand	trace	simulation
VanLehn 1991	Tower of Hanoi	1	224	5 ks	SO.V	State	hand	trace	simulation
Newell 1990 p364+	Cryptarithmetic	1	238	200 s	S..V	PBG	hand	trace-search control	simulation
Newell & Simon 1972	Chess	1	242	na	S..V	PBG	hand	rule trace	hand sim.
Larkin et al. 1980	Physics	21	254	~8 ks	SO.V	KnowLev	hand	trace	simulation
Johnson, et al. 1981	Heart diagnosis	12	264	na	S..V	State	hand	state trace	simulation
Ohllson 1980 - II	Lin. syllogism	5	~400	~2 ks	S..V	PBG	hand	trace	simulation
Baylor 1971	Visual puzzle	1	423	727 s	S..V	PBG	hand	production trace	simulation
Peck & John 1992	Help menus	1	624	~600 s	SO.V	Operator	hand	operator trace	simulation
Moran 1973	Visual imaging	1	656	1.2 ks	S..V	State/Op	hand	trace	simulation
Newell & Simon 1972	Cryptarithmetic	1	1.9k	810 s	S.IV	PBG	hand	rule trace	simulation
Newell & Simon 1972	Cryptarithmetic	5	2.0k	10 ks	S..V	PBG	hand	rule trace	hand sim.
Newell & Simon 1972	Logic	10	3.7k	na	S..V	PBG	hand	rule trace	simulation
Jones & VanLehn 1992	Physics	9	4.3k	na	S..V	KnowLev	hand	trace	simulation
Full trace comparison with non-verbal: (examples)									
Newell & Simon 1972	Chess	1	16	na	SO..	Operator	hand	overt actions	simulation
John et al. 1991	Nintendo	1	73	27 s	SO..	Operator	hand	trace	simulation
Ruiz & Newell 1989	Tower of Hanoi	3	~96	na	SO..	Operator	hand	trace	simulation
Young 1973	Seriation	11	~570	~760 s	SOI.	Op & prod	hand	trace	simulation
Young & O'Shea 1981	Subtraction	33	>2k	na	SO..	Prod	hand	trace	simulation
VanLehn & Ball 1987	Subtraction	26	~3k	na	SO..	Operator	na	operator constraints	simulation
Koedinger 1990	Geometry	30	na	864 ks	SO..	Operator	auto	trace	simulation
Anderson and group	Tutoring	>300	>500k	>10 Ms	SO..	Prod	auto	trace + reset	simulation

In the data type column: N indicates non-sequential data. S indicates sequential data. O indicates overt task actions protocols. I indicates eye tracking protocols. V indicates verbal protocols.

Table 2-2: Summary of previous uses of protocol data to test process models.

2.2.4 Summary of important data features

With the studies reported in Tables 2-1 and 2-2 spread before us, we can examine what general patterns characterize the testing of process models, particularly testing with protocol data.

1. Verbal reports are not yet treated as data. The amount of protocol data used to test the model is often not directly reported, as indicated by the relatively large number of not available (na) or approximated (indicated with a ~) measures in Tables 2-1 and 2-2. Researchers appear to routinely obtain more protocol data then they use to test the model. Sometimes this is explained by subjects performing too randomly to code, but often it appears that they lacked the resources to take advantage of the data. In either case the fail to report its amount and disposition as carefully as reaction time data are treated.

2. Testing process models with verbal protocol data appears to take a lot of effort. The testing is seen as tedious and dangerously boring, offering plenty of opportunities for mistakes (Ericsson & Simon, 1984, p. 271). Generating and testing process based theories by hand takes a lot of time and is tedious. The total analysis time to subject behavior time may be greater as 3600:1 if you base the measures on reported lengths of studies (Ohlsson, 1980). The testing of a model may account for perhaps 1/4 to 1/2 of the total time to develop a process model (Ohlsson, 1992). The number of categories that are used to code that data is the number of operators or trace elements. Each of the trace elements will require a rule for coding it. If the analyst is someone experienced with simulations they may get by with fewer rules. Later steps of modifying the model require someone who is both experienced with simulations so that they can create and modify the process model, familiar with psychology so that they can understand additional behavior constraints, and patient enough to do the alignment and realignment to get the best fit between data and model.

3. When process models are tested they are not tested with a lot of data. The difficulty of testing is also supported by how often process theories are created versus how often their sequential predictions are tested. While process models are now routinely developed, they are not often tested. Testing may become semi-routine within a single study (e.g., Peck & John, 1992), but there appears to be no routine use of protocol data to test process theories. Only four of the 27 process models designed or implemented as computer programs that appeared in Cognitive Science between 1980 and 1986 compared the model's behavior with a subject's action sequence, although several more compared the aggregate performance of subjects and the model (Kaplan, 1987). If model creation was the only significant cost to developing process models, one would also expect the developers to amortize this cost by testing additional subjects, or using more data per subject. As shown in Figure 2-2, the number of subjects and measures used to test these theories is small. Often papers compare a model with the actions of just 1 subject. Sometimes models are compared with up to 5 subjects, but rarely, if ever, are the action sequences of 30 to 100 subjects compared with the model's actions. In addition to a small number of subjects, the total amount of subject actions included in the comparison is often quite low. It is not particularly uncommon to see less than 1000 s (16 minutes) of subject data, and the largest amount of subject data compared with a single model's actions is on the order of several thousand seconds. Sometimes process models are compared with large numbers of subjects over hour-long time periods, but then they are only tested against aggregate measures. Testing process models with the amount of data in the example experimental psychology subject numbers and contact times per study shown in Table 2-1 (20 subjects by 1 hour) is out of reach. Large experimental studies with 700 subjects for 3.5 hours are inconceivable. This comparison process appears so difficult to some, that only general requirements are referenced (see Kaplan (1987) for further citations), or anecdotal evidence is used without reference to subjects at all (Schank, 1982).

4. Testing process models with verbal protocol is limited to a few scientific centers. Sociologically, the methodology of trace based verbal protocol analysis has not become widespread. All of the researchers using verbal protocol to test process models (with a few exceptions) started using it while students or associates of Newell or Simon. If we include researchers that use non-verbal protocols we find that they are more widely dispersed, but still centered around Carnegie-Mellon and the University of Colorado at Boulder, and that few investigators have done more than one such study.

5. Sequential data are more difficult to use. Based on the number of task actions per task, subjects' actions can be categorized into two groups, sequential and non-sequential. Sequential data represent ongoing activity, are ordered in time, and are contingent on previous actions. For example, the actions taken to solve a cryptarithmetic problem represent an ongoing process, and the verbal protocol generated must be understood with respect to what has gone before it. Solitary data are subject actions that represent unitary responses to the task. Simple responses to subtraction problems are an example. There is a continuum. The number of segments in a protocol can be one, like in subtraction, or number four or so, as in solving physics textbook problems (Larkin, et al., 1980; Luger, 1981), or number in the hundreds, as in cryptarithmetic. As long as there is more than one of them, they are sequential. If learning is included (which is rare), then even the single responses are no longer non-sequential. Longer groups of sequential data are used less often, appear to be more difficult to model, and should be more difficult than solitary data because the model must fit for longer periods of time.

6. Discrete data are easy to interpret and helps interpret other data. Discrete data are taken to mean data points that are individually distinct, unconnected elements, taking on a finite (or countably infinite) number of values. The transformation from data to codes in Figure 2-2 is straightforward because the data already represent actions on the task level. If the data are fully discrete, then the testing can be more easily automated, and the full power of visual display and human mediated analysis is not needed. Having directly interpretable data included in the protocol helps to tie down with respect to the model trace less directly codable data, such as verbal protocols. Mouse clicks are presumably discrete; mouse movements in x,y coordinates are not. Eye movements once interpreted as regions of a diagram or as words in a text are discrete (before that they are not). Verbal protocol is not discrete, it can take on many values that map to the same representation, all of which require interpretation. Producing discrete data requires an interface (usually on a computer) that dictates the subject's interactions. This limits the types of tasks it can be collect from, and the actions subjects can use to perform the task.

7. The level of the model affects the amount of data required. The grain-size of the process model and its stage of development affects the amount of data that can be used and the size of the task that can be approached. Initial development of a model usually requires more information than is used to test it. Theories done on a larger grain-size will have a process trace that matches larger lengths of subjects protocols than more fine grained analyses. Models of actions on the knowledge level (Newell, 1982), such as Able (Larkin, et al., 1980), will match large portions of data (perhaps 20 seconds), and models of actions on the keystroke level, such as Browser-Soar (Peck & John, 1992), will match small portions of data (taking seconds or several hundred milliseconds). There are classical tradeoffs here. Higher level models account for more data. More detailed models use less data, but account for it in a more detailed way. The success of closer encoding to a model comes at the price of more comparisons for a given stretch of time. The very direct lesson then, is to model only the appropriate results, not any unnecessary details.

2.3 Tools related to process model testing

There are numerous processes and capabilities required for building and testing process models. The tools to do this can be grouped by primary capability. Examining them will highlight useful features for testing, understanding, and manipulating process theories. There are no tools for routinely testing process models, but there are often tools to perform many of the subtasks. Process theories are often started from simply examining protocol data, and that is the first tool type we examine. A shorter, and slightly different view of available tools is available in Sanderson, James, Watanabe, & Holden (1990).

2.3.1 Tools for building models from protocols

2.3.1.1 Declarative knowledge coding tools

The simplest way to code a verbal protocols or texts is to note the concepts or declarative knowledge that make up each segment. The sequential nature of the protocol is ignored, and the coding is done in a straightforward manner. While the tools for coding declarative knowledge differ fundamentally in function, they hold a direct lesson for process model testing tools. They show how successful semi-automatic and automatic tools can be. The first of these was the General Inquirer (Stone, et al., 1966). It routinely took in a 100s of thousands of data points (as words from texts), and automatically categorized them into semantic types based on a database of words it understood. It was widely and apparently easily used to do content analysis. More recently, Carley (1988) has built a less automatic, but more theory guided set of tools to code knowledge in texts into semantic networks. Segments are presented automatically to be coded, and there are tools to check to make sure the networks are built correctly and consistently. The major analyses she supports are how much knowledge is shared between subjects or within a subject across time. These tools are related to all simple text processing tools, such as automatic indexers (Waltz, 1987).

2.3.1.2 Exploratory protocol analysis tools

Tools that support the levels of exploratory analysis and qualitative simulation (lines 1 and 2 in Figure 2-4) share many of the same features and are best discussed together. The success of these simple tools establishes a baseline of manipulation requirements and shows that computer assisted coding and tabular displays of data are useful.

Simple coding tools. Simple coding tools are defined by the limit of the aggregate measures that they provide. The only aggregation method that they support is the ability to sum the number of times each code has been used. At a minimum, simple coding tools assist the coding process by (a) presenting menus of codes, (b) support treating the text as a set of segments so that the analyst can manipulate and move between segments directly, and (c) support aggregating the codes. In addition to these direct needs, enough editing of strings goes on that a full word processor ends up being generally required. There are numerous tools running on a variety of platforms that provide no more than this (e.g., VPA: Brown, 1986, Lueke, Pagerey & Brown, 1987; PAP: Poltrock & Nasr, 1989; Cref: Pitman, 1985; and others presented in Fielding & Lee, 1991). This level of tool may also assist by automatically presenting segments to the analyst to code, such as MPAS (Erikson & Simon, 1984) was explicitly designed to do.

Recently analysts have started coding the protocol on the video tape directly to represent the protocol (Kennedy, 1989; Mackay, 1989; SigChi, 1989). Video is more compelling data representation because it is more complete, but it is more unwieldy. When the protocol exists on video, more specialized tools are necessary and available. Sometimes the video based tools are just simple coding tools, and some of them include additional features making them exploratory data analysis tools.

General purpose tools have also been appropriated to the task of coding. Analysts have used spreadsheets (Excel: Peck & John, 1992; Lotus 1-2-3: Cohen, Payne, & Pastore, 1991), databases (Filemaker, Mac-Allegro and DataDesk: Larkin, personal communication, July 1992), and even multi-

column text editors (Prep: Neuwirth, Kaufer, Chandhok, & Morris, 1990). These tools offer a level of polish that the research tools don't, but they fail to be integrated into other programs, or be extendable to directly interact with specialized analyses.

Table 2-3: Types of protocol analysis tools and their features.

	General Features				Trace Alignment		Model Incorporated			
TOOL TYPE	CODING AIDS	COUNT CODES	TABULAR DISPL	MACROS /EXTEND	PLAIN	AUTO-MATIC	INTE-GRATED	PROCESS BASED	FLEX-IBLE	AUTO-MATIC
DECLARATIVE (eg, Carley 1988)	+/-	X	+/-	.	.	.	+/-	.	X	.
SIMPLE CODING (eg, VPA: Brown 1986)	+/-	X	.	.	.	.	.	.	.	.
EXPLORATORY (eg, SHAPA: James et al. 1990)	Semi	X	.	.	.	.	.	.	.	.
GENERAL PURPOSE (eg, Excel)	Semi	X	X	X	X	a	.	.	.	.
KNOWLEDGE ACQUISITION (eg, Keats: Motta et al. 1988)	+/-	.	.	X	.	.	+/-	X	X	+/-
INDUCTION TOOLS (eg, Cirrus: VanLehn & Garlick 1987)	Hand	.	.	X	.	X	X	X	X	X
PROCESS BASED (SAPA: Bhaskar & Simon 1977)	Semi	X	.	.	X	.	X	.	.	.
GENERAL PROCESS BASED (Pas-II: Waterman & Newell 1973)	Auto	.	.	X	X	.	X	X	X	.
INTELLIGENT TUTORING SYSTEMS (eg, Angle: Koedinger 1990)	Auto	X	.	.	.	X'	X	X	.	+/-
GENERAL TRACE-BASED (Trace&Transcription, John 1990)	Semi	X	X	X	X	.	.	X	X	.
Soar/PA	Semi	X	X	X	X	X"	X	X	X	.

Legend:
- . Feature not provided.
- +/- Varies between tools in this category.
- X Feature provided.
- X' but not generally available to analyst.
- X" based on keyword matches only.
- Man Done manually.
- a Could be added.

ESDA tools. Exploratory sequential data analysis tools represent a slightly higher level of functionality (e.g., Paw: Fisher, 1991; Shapa: Sanderson, 1990). They tend to support more detailed coding schemes, more closely tied to theoretical constructs, and more sophisticated aggregating measures. They typically directly support a method for detecting loops in behavior, such as sequential lag analysis (Shapa, Paw), or maximum repeating pattern (Siochi & Hix, 1991). These tools are not useful for testing process models because (a) they don't include a process model, (b) they do not generally provide tabular displays, and (c) they lack an automatic interpretation and alignment routine and the facilities to easily incorporate one.

Childes-Clan (MacWhinney, 1991; MacWhinney & Snow, 1990) is the most sophisticated of these programs, perhaps because it was developed for working with children's utterances, which have a greater variety of styles of speaking than problem solving protocols. It is probably the most widely used, with approximately 250 users (MacWhinney, personal communication, October 1992). It includes a structured editor for semi-automatic coding, a language for creating hierarchical codes, the ability to perform numerous types of counts, and to compute context sensitive measures. It comes with several hundred megabytes of transcribed and annotated protocols.

Programs to summarize the data can also exist on their own. Bree (1968) wrote an analysis program that took in coded protocols, and created a type of problem behavior graph that highlighted the cycles in the subject's behavior.

2.3.2 Model testing tools

2.3.2.1 Strategy classification tools based on process models

Tools have been developed that test process models, but they do not directly compare the protocol with a process model trace. These include tools that set model parameters based on the protocol (ASPM: Polk, 1992), that compute the order of unit tasks (SAPA: Bhaskar & Simon, 1977) or knowledge applications (KO: Dillard et al., 1982) in the protocol, or that characterize the protocol as matching one out of a set of strategies (Gascon76, 1976).

Given a model that performs the task, a list of model parameters and how they vary its performance, and behavior to model, ASPM (Polk, 1992) will find the optimal setting of parameters. Its strength is that finds an optimal setting by searching efficiently over huge combinatorial spaces. Its drawback is that it requires a lot from the model and modeler. The model must be expressed in a set form, a complete specification of parameters is required, and a way to derive the models outputs automatically for each input must be provided. Each step in a process must be treated individually, testing a process model thus consists of numerous individual tests.

SAPA (Bhaskar & Simon, 1977) starts out with a fixed flow diagram model that predicts the subjects unit tasks, their possible orderings, and the decision points that subjects will use to solve a general class of problems. In Bhaskar & Simon (1977) the model is for solving thermodynamics problems. Analyzing a protocol consists of assigning each segment to a task. When segments match, the model moves ahead and proposes the next step to match the next segment. On mismatches, the analyst is expected to reset the model appropriately. No provision is made for modifying the model under test. Segments can be presented to the analyst automatically for coding. Accounting, thermodynamics and business policy problems have also been used with SAPA (Bhaskar, 1978), and Dillard et al. (1982) develop a similar system called KO that works for declarative knowledge in accounting.

Gascon (1976) in his thesis wrote a program in Conniver and Lisp to recognize a fixed set of strategies (ten in all) for a weight seriation task. Based on codes assigned by hand to the subject's actions, his system recognizes the global strategy and assigns the individual actions to particular steps of the strategy chosen.

2.3.2.2 Model tracing modules within intelligent tutoring systems

There are at least a few examples that show that the action sequences generated by process models can be compared automatically with human behavior. Some intelligent tutoring systems (particularly Anderson and his group's tutors (Anderson, Farrell, & Sauers, 1981; Anderson, Greeno, Kline, & Neves, 1981; Reiser, Anderson, & Farrell, 1985; Singley, 1987; Singley & Anderson, 1989; but also Sleeman, Hirsh, Ellery, & Kim, 1990, and VanLehn, 1983) provide these exceptions. Subjects as students now routinely work with Anderson's tutors, and have their actions compared with a process model of geometry proof construction (Koedinger & Anderson, 1990) or lisp programming (Anderson, Farrell, & Sauers, 1981). The process models used in the tutors come from previous non-automatic

analysis (Singley & Anderson, 1989), but sometimes they are able to make use of the tutor format to refine existing theories (Koedinger & Anderson, 1990).

Closer examination reveals that this extreme level of automation requires several constraints that are not always available or acceptable: (a) The domain is not open ended, there are fixed operations and responses. (b) All subject actions are readily interpretable because they must come from a fixed set available on menus, and no verbal protocols are taken. (c) The subject model is a well developed one, so that when the match is done automatically it will be a close fit that is easier to do. (d) The nature of the tutoring task and mature models allows the analyses to be done in a closed manner — when the subject's protocol mismatches the model, the subject as a student is reset and told to start over.

2.3.2.3 Tools for aligning the sequential predictions with data

A necessary step in analyzing the predictions of a process model is to bring the predictions into correspondence with the sequential actions of the subject. The model and subject have to be initially synchronized, and then after each time they lose step with each other. This discussion assumes that both sequences are in already in hand, and that the alignment does not have to be computed dynamically.

Two general systems have been created to test unrestricted process models by aligning their predictions to the data either semi-automatically (Pas-II) or by hand (Trace&Transcription). In both cases they do not directly incorporate the model being tested, but provide a framework for comparing the model's predictions with protocol data.

Aligning the model's predictions with the subject's actions and verbal utterances would be particularly nice to automate. The alignment has mostly been done by hand; performing this automatically would remove a tedious task for the analyst. Performing the comparison automatically will require that the process is fully specified, which will also make the prediction testing more reliable. No extant cognitive modeling system or protocol analysis systems supports this, although it has been done automatically on a small scale to explicitly test small models (Card, Moran, & Newell, 1983).

PAS-I and II. Pas-I (Waterman & Newell, 1971) was a system to automatically analyze verbal protocols on cryptarithmetic. Pas-II (Waterman, 1973; Waterman & Newell, 1973) was a generalized version of Pas-I. They provide a complete model of how to code verbal protocol into problem behavior graphs. Their strengths were: (a) They made the analysis process very objective and explicit. (b) They were complete and integrated. One could use them to code verbal protocols from transcript to problem behavior graph. (c) They were both flexible. Pas-I could be modified to deal with any cryptarithmetic protocol, and Pas-II could be modified to work with any domain.

Despite their strengths, they appear to have been applied only to their initial test data. Their weaknesses may be even more enlightening than their strengths:

- They appear to have been complicated and hard to use. Pas-II, the general system, required the user to represent all the knowledge for each of the analysis steps as productions. Facilities for creating, editing, and testing these productions do not appear to have been provided.
- The number of steps may have seemed daunting. There were 21 steps implemented out of 28 steps in the complete design. Each step required it own mini-production system for coding the protocol.
- They did not directly include the process model they were attempting to match. This means that the same information for interpreting the data with respect to the model had to be represented in several locations.
- The output of the analysis was not in a directly interpretable form. Pas-I provided the

PBG as straight text that had to be reformated into a PBG.

- They did not directly test the model. While the design included comparing the subject's PBG to a trace of a process model, this appears not to have been implemented (Waterman, 1973). In the end, the test was whether the rules based on the model could parse the data with the assistance of a analyst.
- They suffered from poor displays. The intermediate representations in the processing were not directly visible, but were displayed only on command.
- They were naive about the difficulty of parsing natural language. They attempted to do a complete parse without a theory of parsing, using only rewrite rules, and without tying it to a model of the task.

Trace&Transcription. The Trace&Transcription system (John, 1990) was the first and (until Soar/MT) only system explicitly created to analyze protocols with respect to a running process model. It took as input a trace of the transcribed protocol, which could include multiple behavior streams, and a trace from a running cognitive model. In its one and only application these were verbal utterances, mouse button actions, and mouse movements of a user using an on-line help system, and a trace of the Browser-Soar model. These two information streams were semi-automatically aligned; the user would click on the two segments (data and trace) to be aligned, and Trace&Transcribe would add additional cells in one or the other columns to bring them into correspondence.

Trace&Transcription included two innovative ideas that simplify the analysis task and provide more power to the analyst. (a) Treating as a database the data, the model trace, and their comparison. The underlying database system of Trace&Transcription (Oracle) supported queries of where the model matched and mismatched, making available groups such as all unmatched mouse clicks, or all verbal utterances with the word "draw". The database approach also allowed the analyst to group columns of data together, so that they stay aligned as blank cells are inserted to align the model columns with the data columns. (b) Tabular presentation of the data. Each field (e.g., the model trace, the verbal protocol data, the mouse movement data) was represented as a separate column, as were the comparison, comment and time stamp fields. This allowed more data segments to be displayed per given screen size. It also created a visual representation of the alignment of the model trace to the protocol. Rows that are filled all the way across represented correspondences. Blank spaces in the data column while the model column was full represented the model doing more than the subject, and vice versa.

The Card1 algorithm. Card, Moran, and Newell (1983, Appendix to Ch. 5) present an algorithm for finding the longest common subsequence (Hirschberg, 1975) that can be used for aligning two behavior sequences. In their case the two sequences are subject actions from a non-verbal protocol, and actions predicted by a simulation model. In the general case, the sequences can be (a) subject actions from a verbal or non-verbal protocol, and (b) actions predicted by a simulation model, and there can be more than one subject sequence (however we assume in this work that these will always be aligned pair-wise). This algorithm finds the longest common subsequence, that is to say, it finds the largest sequence of tokens (representing matches between tokens in the two streams), such that the tokens in the result have the same order in each sequence.

Other alignment algorithms. Hirschberg presents some additional algorithms (some recursive) for computing the maximum common subsequence that take up less space. The gravest flaw in using these more elegant algorithms is that they do not provide control over which subsequence is returned out of the (possibly) many maximally common subsequences. They may indeed prefer the same one that Card does. They are also more complicated, and speed and space is not of the essence in this application, clarity, readability, and modifiability is. Card1, which uses N^2 space, generally performs the match quickly enough.

Card's algorithm is also based on work by Sakoe and Chiba (1978), which represents the task of aligning a model of speech recognition with the speech signals presented for recognition. This is a

similar task, and can be represented as search with productions (Newell, 1980a). Pas-I (Waterman & Newell, 1971) and Pas-II (Waterman, 1973) proposed (but did not implement) an incremental approach to the match that included backtracking and partial matching. They proposed (Waterman & Newell, 1971) to move both traces ahead by a line, attempt a match, and incrementally advance the counters on the model trace until a match was found. Backtracking was reserved for continued mismatches. Ohlsson (1990) appears to have successfully used this algorithm by hand. A neural net has learned to match a Markov model's predictions to computer interactions in real time (Finlay & Harrison, 1990).

2.3.3 Tools for building and understanding models

2.3.3.1 Process model induction tools

Most process models induced from protocols are created by hand. There has been some work to do this automatically or semi-automatically with machine learning techniques. Semi-automatic generation is done in the event structure modeling domain (a sociological level of social events) by a program called Ethno (Heise, 1991; Heise & Lewis, 1991). It iterates though a database of known events finding those without known precursors. It presents these to the analyst, querying for their precursors. As it runs it asks the analyst to create simple qualitative, non-variablized token matching rules representing the events causal relationships based on social and scientific processes. The result at the end of an analysis is a rule set of 10 to 20 rules that shape sociological behavior in that area. In a sense, the analyst is doing impasse driven programming (i.e., what is the next precursor for an uncovered event not provided by an already existing rule?). After this step, or in place of it, the analyst can compare the model's predictions with a series of actions on a sociological level (a protocol in the formal sense of the word). The tool will note which actions could follow, and query the analyst based on this. Where mismatches occur, Ethno can present several possible fixes for configuration. By incorporating the model with the analyses tool in an integrated environment it provides a powerful tool. It would be a short extension to see the social events as cognitive events in a protocol.

Stronger methods for building models from a protocol are also available. Cirrus (VanLehn & Garlick, 1987) and ACM (Langley & Ohlsson, 1989) will induce decision trees for transitions between states that could be turned into production rules given a description of the problem space, including its elements, and the coded actions in the protocol. Cirrus and ACM uses the ID3 learning algorithm a variant of it (Quinlan, 1983).

Why is automatic creation of process model not done more? These tools look like a useful way to refine process models. These systems do not actually create process models. They take a generalized version of an operator that must be specified as part of a process model. It could be that finding the conditions of operators isn't the hard problem, but that creating the initial process model and operators is. It could also be that it is harder to write process models that can be used by these machine learning algorithms, but these methods should be explored further.

Their lack of use could be simply related to being new software systems. As new systems, they are probably difficult for people other than their developers to use, and they will have to go through several iterations of improvement (like most pieces of software) before they are ready for outsiders to use them. Future work should consider including a machine learning component, for they can help summarize knowledge level information.

2.3.3.2 Tools for understanding and building symbolic cognitive models

While there have been numerous attempts to create general cognitive modeling languages (e.g., Ops, Ops5, Ops83, IPS, Prism, reviewed in Neches, Langley & Klahr, 1987), there has not been many attempts to develop tools for manipulating and understanding the models created for these architectures. When models were small and simple, an additional level of interface was not needed. This is not true anymore. Cognitive models now often contain hundreds of rules (e.g., Browser-soar, Peck & John, 1992) to tens of thousands of rules (Doorenbos, Tambe, & Newell, 1992).

```
P: top-space
O: browse
==>G: (operator no-change)
   P: browsing
   O: find-appropriate-help
   ==>G: (operator no-change)
      P: find-appropriate-help
      O: change-search-criterion
      O: define-evaluation-criterion
      ==>G: (operator no-change)
         P: define-evaluation-criterion
         O: evaluate-evaluation-criterion
         O: generate-evaluation-criterion
      O: define-search-criterion
      ==>G: (operator no-change)
         P: define-search-criterion
         O: evaluate-search-criterion
         O: generate-search-criterion
      O: evaluate-help-text
      ==>G: (operator no-change)
         P: evaluate-help-text
         O: change-current-window
         ==>G: (operator no-change)
            P: mac-methods-for-change-current-window
            O: click-prev-index
            ==>G: (operator no-change)
               P: mac-method-of-click-prev-index
               O: click-button
               O: move-mouse
            O: drag
            ==>G: (operator no-change)
               P: mac-method-of-drag
               O: move-mouse
               O: press-button
               O: release-button
            O: page
            ==>G: (operator no-change)
               P: mac-method-of-page
               O: click-button
               O: move-mouse
            O: scroll
            ==>G: (operator no-change)
               P: mac-method-of-scroll
               O: move-mouse
               O: note-saw-criterion
               O: press-button
               O: release-button
         O: evaluate-current-window
         ==>G: (operator no-change)
            P: evaluate-prose-in-window
            O: compare-to-criteria
            O: comprehend
            O: read-input
            P: evaluate-items-in-window
            O: attempt-match
            O: read-input
         O: focus-on-help-text
      O: modify-search-criterion
      O: search-for-help
      ==>G: (operator no-change)
         P: search-for-help
         O: access-item
         ==>G: (operator no-change)
            P: mac-methods-for-access-item
            O: click-on-item
            ==>G: (operator no-change)
               P: mac-method-of-click-on-item
               O: click-button
               O: move-mouse
            O: double-click-on-item
            ==>G: (operator no-change)
               P: mac-method-of-double-click-on-item
               O: double-click-button
               O: move-mouse
         O: find-criterion
         ==>G: (operator no-change)
            P: find-criterion
            O: change-current-window [...]
            O: evaluate-current-window [...]
            O: focus-on-current-window
```

Figure 2-5: Example output of TAQL space graph.

There have been several disjoint attempts to provide a better interface for Soar on the level of production manipulation and understanding of the goal stack. The two previous graphical interfaces (Milnes, 1988; Unruh, 1986) provided an augmented description of the goal stack that let users click on objects to examine them. These systems did not retain any explicit model of the problem spaces and operators. There have also been three text editors extensions (by Ward, Shivers, and Milnes, respectively) designed for manipulating Soar on the production level. Each included simple commands for starting up a Soar process, editing productions, and loading them. These interfaces were not developed for very long, and were not widely distributed.

There has been only one tool for Soar that attempted to describe Soar's emergent behaviors on the problem space level. Version 3.1.4 of the TAQL macro language for Soar (Yost, 1992; Yost & Altmann, 1991) provides a textual description of the problem spaces and operators that it could recognize from the TAQL constructs making up the Soar model. It did not guarantee that they would be selected, or that the set it found was complete. An example of its output is shown in Figure 2-5.

Other typical cognitive modeling tools based on production systems, such as ACT★ (Anderson, 1983), and Ops5 (Forgy, 1981) come with only a command line interface for loading files and running the system. They do include debugging commands, such as which rules will fire next, but this must be explicitly requested by the user, and a match set is generally not available in a separate display. These systems are viewed as just an inference engine for a cognitive architecture. The production rules that implement the models are created and edited using a general purpose editor, which usually lacks the ability to directly add a new production to the interpreter, and to edit the production in a structured way. All of these production systems lack the ability to describe and manipulate the models on higher levels of organization, such as the problem space or knowledge level.

2.3.3.3 Knowledge acquisition tools

Many types of expert systems are designed to be process models of expert behavior in areas where algorithmic solutions are not available. Expert system development shells (KEE) are designed to build these process models of expertise. Their cousins, knowledge based knowledge acquisition (KBKA) tools, attempt to use an expert system recursively to monitor the process of model building and suggest places that need attention (KADS: Brueker & Wielinga, 1989; KEW: Shadbolt & Wielinga, 1990; Keats: Motta, Eisenstadt, Pitman, & West, 1988; Kriton: Diederich, Ruhmann, & May, 1987; Shelley: Anjewierden, Wielemaker, & Toussaint, 1990; an overview of the field is provided by the Fall 1989 SIGART Bulletin). KBKA tools include many of the features that a tool for testing models would need because their task, to build a process model that produces expert behavior starting with an analysis of the task and verbal protocols, is very similar to testing process models with protocols. Their strength is that they include in their environment, often in a highly integrated way, a process model. In these tools, the process model is the expert system that is being developed. They provide tools to manipulate the expert system, modify it, and run it on the task. They often include graphic depictions of their declarative knowledge (e.g., Keats: Motta et al., 1988), but they rarely, if at all, provide visual descriptions of the processes and the process knowledge.

Knowledge acquisition environments sometimes include the ability to tie verbal protocols or other texts to various facets of the model, showing where a feature came from, and serving as a note that a segment has had its knowledge extracted. Their task is only to create a model that performs the task, not to validate the model's performance against the input, so they all completely lack the ability to measure the comparison. As a group, when coding protocols they also make a fundamental misunderstanding about the comparison of similar levels in the protocol and model. They always code the protocol in terms of the static structure of the model, either as rules or data structures. When an expert is talking about rules to apply, that is appropriate. When the same tool is used to code verbal protocols given by an expert while they are performing the task, this coding is inappropriate. In terms of Figure 2-2, they are comparing the knowledge of the model to the process data of the subject. Like many cross-level comparisons, it is incorrect and only approximate, but in practice it serves them well. One system, Kriton (Diederich et al., 1987), claims to be able to do this automatically. How well it

does this is not clear from the short technical report.

2.3.4 Summary of useful tool features

Reviewing these tools for performing various subsets of testing process models with protocol data suggests several guidelines for future integrated tools.

1. Current automatic testing approaches carry too many constraints for general use. While it is possible to automatically compare a process model with data, several intelligent tutoring systems do so with non-verbal data, it does not appear to be currently possible with verbal data because of the difficulty of doing general natural language parsing. Natural language parsing is necessary to match verbal protocols to the process trace. Only two systems attempt to compare verbal protocols to models automatically. It is not clear that the parser in Pas-II is powerful enough to be applied in a truly automatic sense. The details provided on how the Kriton system does its parsing are not adequate to judge its performance.

   Systems that otherwise do automatic analyses must limit their general applicability by taking on one or more constraints to avoid natural language, such as using only well tested models and simple tasks that use a limited subset of language, or working with discrete data. Future general tools can only provide semi-automatic analyses until parsing technologies are further developed.

   The task of testing process theories with verbal protocols may present a unique opportunity to push natural language parsing forward. Unlike general parsing situations, the model being tested provides a strong theory of what will be said. The directly relevant knowledge structures needed for parsing, presumably the data structures used to perform the task, are available and updated by the process model as the task unfolds. Using multiple behavior streams to fit to the model would further restrain the parsing task.

2. Automatic and semi-automatic features aid reliability and speed. Tools that provide assistance doing the analysis that are automatic (completely autonomous) or semi-automatic (small user initiated analyses, or semi-correct analyses checked by the user) are highly praised by users, even if they are not always tied to the theoretical constructs being tested. People, even dedicated ones, do not like doing this type of work. Analyses done without tools that provide assistance are not repeated as often. The most automatic systems, the General Inquirer (Stone, et al., 1966) and the Anderson tutors, truly make the analyses they do routine.

   Semi-automatic tools may actually have been used more often because fully-automatic analysis requires the theoretical model of the analysis to be complete. The system can do this by having a weak but quite general model, such as the General Inquirer (Stone, et al., 1966), or by having a well developed model such as the Anderson tutors have. If the system does not provide all the specifications for the analysis, then the user must create them. If too much specification is required from the user, the system might not be used, which may be what happened to Pas-II.

3. An extendable word processor and a database facility are required. Most tools incorporate word processor functions to do such things as annote segments, correct transcription errors, and enter and edit labels. Systems that do not incorporate a word processor (such as PAW, Fisher, 1991), assume that one is available in the environment.

   While the broad approach to protocol analysis and model testing can be specified, in the end these are fluid tasks. A set of analyses must be provided, but additional, similar

analyses will be necessary. Systems should provide a macro language and interface to help automate repeated actions, to create modifications and extensions, and to integrate with other tools.

Simple database facilities are also required. These facilities need not be extensive, but the basics must be provided. Only Trace&Transcription (John, 1990) and some general tools appropriated to the task incorporate a complete database system (Oracle). Most tools provide some support in this area. Nearly all the systems support adding additional data fields to segment records. Those who do not suffer for it. For example, those that do can be expanded to include their verbal protocol as a separate field, even if their initial configuration does not. Simple aggregations by types is another very desirable database feature. These simple analyses appears useful for checking the ongoing analysis for correctness. More sophisticated systems include more extensive analyses built in, such as sequential lag analysis. The more advanced database functionality of listing segments by queries appears useful, although few systems currently support this.

4. Tabular displays present more data. Figure 2-6a shows a record based display, implemented as part of an initial version of Soar/MT. It is representative of the record based displays of most tools. Figure 2-6b shows a tabular based representation of the same data. The tabular approach represents a segment as a row, with its fields placed in separate columns, like a spreadsheet. This approach better supports the appropriate visual operators (Larkin & Simon, 1987) for finding the context of a segment and its associated fields. By only using one line per segment, and putting the labels only once at the top, more data can be displayed than in a multi-line record approach. This tabular format also supports the approach analysts have used when working on paper (Newell & Simon, 1972; Ohlsson, 1980; Young, 1973). Trace&Transcription and some of the general purpose tools support a tabular based visual layout. For a telling (and atypical) supporting example, see Newell and Simon (1972), Appendix 6.1 and its notes two pages later. The distribution of information across several pages makes this protocol hard to read.

5. Integrating the model support analysis. Tools should keep the models being tested close at hand. If a tool does, its manipulations can be directly based on the model being tested or built. This includes automatic analyses based on the model (such as features supported by data, features unconnected to other features). Having knowledge of the model is required for helping the analyst through automating the tasks or by providing smart features that partially do the task. This representation must be explicit. Several systems that use models (e.g., Pas-I and Pas-II) include them only implicitly in production rules. This precludes the tool from using the model codes the rules represent without running the model to find them.

   Systems that can automatically offer codes to assign to data points represent the weakest form of this ability. They know the names of the model components, but that is all. Most systems, including general purpose ones, either directly provide this low level of model manipulation or can be modified to do so.

   Pas-II represents an unfortunate position with respect to incorporating a model. It did not directly include a model that could be used to do an analysis. The model being analyzed was only available implicitly in the productions that users added to do the analysis, and not directly available. Pas-II could manipulate the data with respect to the model, but only if the user supplied these manipulations as additional productions based on a model external to Pas-II. It did not check directly to see that all the operations in a model had been supported by data, or suggest operators to code with.

   The model induction tools (ACM, Cirrus), and most knowledge acquisition tools can

directly access their model as a knowledge source for analysis. Indeed, in expert system development knowledge-based knowledge acquisition (KBKA) can base the analysis directly in terms of the model. Tools that include declarative model structures (e.g., Kriton and Keats) can reference the model directly. Having the model directly at hand provides KBKA with strong support for coding and data aggregation, but these tools do not completely incorporate comparisons because they have been designed for building models, not for testing them.

In many ways Trace&Transcribe is the best extant tool for testing process theories. Its major drawback is that it cannot reason about the model it is testing. Trace&Transcribe also does not explicitly represent the Soar models being tested, and the models are also not available from the current Soar implementation (5.2). After the alignment of the subject protocol with the trace, there are no further analyses that it can carry out on its own based on the model, such as noting which operators were supported, or where operators were not supported.

6. An interface is required for manipulating and understanding the model. The lack of development of interfaces and display representations for manipulating and understanding symbolic models can be contrasting with the development of tools for PDP models (McClelland, Rumelhart and the PDP research group, 1986; Rumelhart, McClelland and the PDP research group, 1986). PDP models can be difficult to interpret directly; their structure is implemented as an array filled with real numbers representing the connections between nodes. PDP software has mitigated this problem by letting modelers manipulate systems on the level that they think of them. Nearly all systems now allow the user to create models by drawing their nodes and connections (O'Reilly, 1991). The diagrams created this way are also used to describe the model's performance over time, such as the changes in the connections from learning are often visually displayed during a run (McClelland & Rumelhart, 1988). Displays to illustrate the importance and meaning of the connections have also been developed (Hinton & Sejnowski, 1986; Kolen & Pollack, 1988; McClelland et al., 1986; Rumelhart, et al., 1986; Touretzky, 1986).

2.4 Measures of model to data comparison

This section reviews the measures that have been used by analysts to improve and describe to others the fit of process models to data. Measures for comparing sequential data will primarily be examined, but measures for comparing aggregate data will be included when appropriate. A shorter, and slightly different view of measures useful for model building is available in Sanderson et al. (1990).

The review starts off by deriving what is needed from previously published and newly presented criteria for evaluating these measures. Taken together the criteria indicate that four different types of measures are necessary. These four types of measures include (a) a global measure of where the model mismatches the data, (b) a simple measure of fit to provide local guidance for improving the model, (c) a measure indicating how well the model will perform in the future, and (d) a measure indicating the degrees of freedom in the model. Some of these measures must also be persuasive to others, so this is listed as the fifth criteria. The ability to describe the behavior of the model in general terms also turns out to be important in comparing models to data, even though it is not itself a measure of fit. The most influential need, for measures to indicate where the model does not fit the data, particularly makes strong recommendations about which measures are useful.

The measures that have been used to evaluate the fit of process models can be categorized into four types: (a) non-numeric descriptive measures of the general fit, (b) numeric descriptive measures of the general fit, (c) measures of rule (or component) utility, and (d) measures based on inferential statistics. Previous uses of each of these types will be described, and they will be evaluated with respect to how

2-6(a) Example record-based display of model trace to data comparison.

```
1 ----------------
TIME: 12400   VIS:         TYPE: too-short
VNUM: 1   DURATION: 725  VERBAL>: I believe
MNUM:       MBANUM:        MOUSE>:
DC:       B EVID FOR:
  V EVID FOR:
  M EVID FOR:
  M REQUIRED:
Comments:
2 ----------------
TIME: 12406   VIS:         TYPE: v-coded
VNUM: 2   DURATION: 351  VERBAL>: write
MNUM:       MBANUM:        MOUSE>:
DC: 15    B EVID FOR: O: o82 (generate-search-criterion)
  V EVID FOR: O: o82 (generate-search-criterion)
  M EVID FOR:
  M REQUIRED:
Comments:
3 ----------------
TIME: 12409   VIS:         TYPE: v-coded
VNUM: 3   DURATION: 2210 VERBAL>: write
MNUM:       MBANUM:        MOUSE>:
DC: 21    B EVID FOR: O: o82 (generate-search-criterion)
  V EVID FOR: O: o82 (generate-search-criterion)
  M EVID FOR:
  M REQUIRED:
```

2-6(b) Example tabular display of model trace to data comparison.

```
T  Mouse actions  Window actions  Verbal  ST # Mtype MDC DC                       Soar trace
-+--------------+---------------+--------+-+-+-----+---+--+--------------------------------------------+
 0                               I believe  v 1 short
                                                          0 G: g1
                                                          1 P: p4 (top-space)
                                                          2 S: s5
                                                          3 O: browse ()
                                                          4 =>G: g19 (operator no-change)
                                                          5   P: p26 (browsing)
                                                          6   S: s39 ((unknown) (unknown))
                                                          7   O: find-appropriate-help
                                                          8   =>G: g43 (operator no-change)
                                                          9     P: p50 (find-appropriate-help
                                                         10     S: s59 ((unknown) (unknown))
                                                         11     O: define-search-criterion
                                                         12     =>G: g65 (operator no-change)
                                                         13       P: p72 (define-search-criterion)
                                                         14       S: s79 ((unknown))
 6                              write       v 2      v 15 15       O: generate-search-criterion ((write))
 9                              write       v 3      v 15
13                              write       v 4      v 15
  M(+x) (R of prog win)                            B4
            mouse line to pointer
                                                         16       O: evaluate-search-criterion
                                                         17     O: define-evaluation-criterion
                                                         18     =>G: g103 (operator no-change)
--**-emacs[SHAMO.SOAR]: example-types.spa    A36 ManUp <H]  (SPA) ----Top-----------------------------
```

Figure 2-6: Example displays for comparing the model's predictions with the data.

they serve the identified measurement needs. This section ends with a summary of the most useful measures.

2.4.1 Using criteria to develop a set of measurements

What are we trying to find out with these measures of models? Listing a set of criteria for each measure is a way to answer that question. Table 2-4 displays the eight most important criteria that have been put forward for desirable model measures. Taken together the criteria determine which measures to use. So, what are the questions that are typically asked about the data and the model? These can be grouped into five basic requirements discussed in order of importance. There appears to be two general questions that people ask: How well does a given model fit a given set of data?, and How can we tell if this is significant match? These questions will basically be rejected in the following sections and replaced with two others.

Table 2-4: The five major types of measures of model fit and the criteria supporting them.

(Criteria in []'s are later rejected.)

1. Globally showing where the model mismatches.

- The analytic testing criterion. The measure should indicate where the model does not fit the data (Grant, 1962).

2. Locally showing where the model mismatches.

- The inaccuracy criterion I. The measure should decrease for every false prediction (Grant, 1962).
- The accuracy criterion I. The measure should increase every time the model fits the data (Priest & Young, 1988).
- The inaccuracy criterion II. The measure should decrease each time the model does not fit the data (Priest & Young, 1988).
- [The accuracy criterion II. The measure should increase for every correct rejection (Ritter, this thesis).]

3. Knowing how the model will perform in the future.

- The prediction criterion. The value of the measure obtained from the data sample should provide an unbiased estimate of the value to be obtained from a larger sample (Priest & Young, 1988; Grant, 1962).

4. Representing the Fit vs. complexity tradeoff.

- The parsimony criterion. The measure should decrease for each additional variant procedure added to the system to account for the data (Priest & Young, 1988).

5. Being persuasive.

- [The numeric value criterion. The measure should return a single number falling in a predetermined range (Priest & Young, 1988).]

Globally showing in terms of the model where the predictions mismatch. In his seminal paper, Grant (1962) notes that scientists are not really in the business of testing theories just to put a stamp of approval on them, which is often presented as the questions of how well does the model match, and is it significant? Scientists are more like a parachute maker who wants to make a better parachute. The parachute maker tests parachutes to find out their weaknesses, and where to make them better. As scientists conducting ongoing research "[the theoretical scientist] is not accepting or rejecting a finished theory; he is in the long-term business of constructing better versions of the theory." Grant calls this approach analytic testing, it is "designed to tell me as much as possible about the locus and cause of any failure in order that I may improve my product." Analytic testing is then computed to tell where to improve the model, and not merely to provide a stamp of approval.

Usually the mismatches pointed out will be small errors, such as the order of performance of subtasks. Sometimes they will be big errors, such as completely different approaches to tasks or whole competencies not provided in the model. In either case we need to know the location and extent of both sizes of mismatches, and a way to see any systematic patterns in the errors so that we can fix the

model. Comparing the model with the data is thus two tasks, noting where the model is consonant with the data, but more importantly, noting where it is not consonant and needs to be improved.

In striving to predict more of the data, process models become more falsifiable, generally considered a good thing for a model. Popper (1959) argues that it is better to be falsifiable so that it can be more easily discarded if it is wrong, as it will be. What Popper should have meant, and what being falsifiable means for us as theory builders and improvers, is that theories should be falsifiable so that you can see where to improve them! Being falsifiable for us means not only that the predictions be concrete, but that the model make many of them, and that they be as detailed as possible, even including sequential information, so that where the model is wrong we can more easily and often tell where we need to improve it (Newell, 1990, p. 14).

Least we forget though, we also need to be able to characterize where the model performs well, so that we can know where to use it. In the parachute maker's terms, we need to know what objects and at what heights are the best places to use our parachute. Knowing the tasks where the model performs well is also useful for focusing attention on and characterizing tasks where it performs poorly. Finally, knowing where a model performs well will also be necessary for comparing the model with other models.

Locally showing where the model mismatches. In addition to global measures of how well the model fits the data, a local description of where the model mismatches is necessary to implement local improvements. This need not be a separate measure, but could be incorporated into a global measure if the global measure was sufficiently convenient.

The four accuracy and inaccuracy criteria essentially represent the four measures in signal detection, those of hits (subject matches model's prediction), misses (subject action not predicted), false alarms (model's prediction not matched), and correct rejections (model and subject correspond on inaction). These measures are not global measures of where to improve the model for they are only a count of places that could be improved. Strict numerosity is not required, merely that the measure gets "better" somehow for matches, and not only gets worse on mismatches, but points out where they occur.

Fit versus model complexity trade-off criteria. The final two criteria for measures are related to parsimony. The first is that the measures take into account parsimony, how many degrees of freedom were used to account for the data? Strictly speaking, this is not necessary for finding out where to improve a model, but for knowing when to stop improving its fit to a given data set. Degrees of freedom are normally reported separately (e.g., Chi-squared tests, T-tests, F-tests), and we will encourage that here as well. How many variant procedures have been added to the system to account for the data should be indicated, but other measures need not incorporate this directly.

Knowing how the model will perform in the future. Given this view of scientists as model builders and users rather than as model certifiers, the other important criterion for measuring a model is "How well does the model make predictions for the future?" This will be the main measure used for public display.

Being persuasive. One of the problems that has plagued researchers in this area is that they want to be able to persuade other scientists through their measures that their model fits the data well. This is a real need, and has distorted the choice of measures. In the past, various statistics from experimental psychology have been incorrectly applied in an attempt to do this by proving the models different from chance or not different from the data. These were never convincing, and rightfully so for they are, respectively, weak and incorrectly applied tests. The real need is to show how well the model will fit other data, not how likely it was to fit the current data as well as it did.

The numeric value criterion put forward by Priest and Young (1988), that the measure should return a single number falling within a predetermined range was to ensure that different micro-theories could be ordered for a particular technique. And it expresses the desire to provide a stamp of approval that Grant (1962) notes exists. This has to be rejected as an absolute requirement, for it assumes that

process model fits should be an ordinal number. And they cannot. Ordinal numbers cannot indicate multiple places where a model needs to be improved. Rejecting this also frees us from trying to find a way to combine all the accuracy and inaccuracy criteria into one measure.

If one was interested in incrementing a single numeric measure whenever the model fits the data, strictly speaking, the measure must also be incremented when the subject does nothing and the model does nothing as well. Correct rejections make sense when there are fixed items for responses, but it makes less sense when modeling a continuous process. It would be a difficult problem to decide on a reasonable method for enumerating all the places where the subject and model did nothing and to increment a measure based on this. This criterion can be rejected because the requirement of only needing to know where models mismatch frees the measure from including all the places where it does match.

Summary of required measures. The ability to show where the model globally and locally mismatches, and the ability to predict the quality of future performance based on where the model performs well are the three most important measures. We would prefer a unified measure, but will have to use several to cover the ground, particularly since different models can mismatch the data many different and subtle ways. Using multiple measures is acceptable because there are many compatible ways to improve something — we are not trying to put a stamp of approval on the model, where a single standardized test would be preferable. These measures do not have to result in a single number, and the numeric criteria of increasing the measure for every hit and so on, is taken loosely to mean to influence the measure appropriately in each case.

Presenting the measures that are used to improve the model and one that is a measure of the degrees of freedom in the model, if presented clearly, will hopefully convince others. These alone do not have to. The analyst also has the predictive power to report (which has been computed as the analysis has gone along), or to take home and use in applications that may also prove the model.

2.4.2 Description of measurement inputs

This subsection describes the possible inputs to the measures, including the two information streams, and the types of results of comparisons on the data level.

The two traces. As diagrammed in Figure 2-2, when a process model is run, it generates a trace of its external task actions and internal states and operators. These can be labelled M_1 through M_y. Each M_j is a trace of model actions on the appropriate level. It can be the rules that fire, or working memory elements, operator applications, or knowledge that has been applied. The choice depends on the theory level committed to and on the level of subject behavior available. If the model is based on rules rather than operators, then the trace could also include or consist solely of rule firings. In Soar models in general, and particularly the ones examined here, it is operator applications. Each model action has a simulation time associated with it. The model stream will include a fixed number of token types, specified ahead of time as part of the model.

On the other information stream, the subject generates a series of actions, S_1 through S_x. Each action should have a time stamp (in seconds or ms) associated with it. Not having a time stamp will preclude some of the analyses. As they are sequentially ordered data, they are a protocol. Different modalities, such as verbal utterances and eye movements, each represents a different information stream, but order across modalities is preserved. These information streams may contain an essentially unlimited number of different types of words.

The measures will be computed after the subject's actions (S_1 to S_x) are interpreted and aligned with respect to the model's predictions, (M_1 to M_y). As a simplifying assumption, each correspondence will be a full one; partial matches on the segment level will not be allowed. Measures may take into account multiple episodes for a subject, or multiple subjects, but the emphasis will be on fitting a single episode. Each measure or representation must represent each of the types of matches shown in

Table 2-5.

Table 2-5: Types of correspondences between the model's predictions and the data

1. Uncodable subject action. There may exist subject utterances too short to code or outside of the model being tested ("Hmm", "nice day"). If they are clearly outside the model, the analyst may wish to discard them from later analysis. Sometimes it will be useful to carry them along as comments because they serve as way posts, or they may be found to be data with respect to a more complete model. Measures and displays should be able to handle them as null points.
2. Uncodable model action. There may exist objects in the model's trace that represent internal state and operations. Not all model actions can or will be matched in a verbal protocol. Measures and displays should be able be able to handle these too as null points.
3. Simple hit. A subject's action and a model's action correspond one-to-one.
4. Multiple subject action hit. A single action in the trace may match multiple subject actions. This could be caused by a high level operator in the trace that represents actions larger than a typical segment, or else takes multiple descriptions or matches data in multiple modalities ("I'm pressing the space bar" and the keystroke action <space-bar>).
5. Multiple model action hit. A single subject action may correspond to multiple actions in the model. This is possible if the model performs the task on a finer level of detail than the data provides, or if the segment is incorrectly segmented. The analyst should consider splitting the segment when this occurs.
6. Miss. The subject action is not matched by a corresponding model action.
7. False alarm. The model produces an action that the subject does not perform. Overt task actions represent the most egregious example of this, and must be penalized. Unsupported internal actions of the model should not be penalized, but must be supported some other way. This can be done through appeals to necessity, or aggregate data. It may also be the case that portions of the model cannot be directly supported, but are required by the architecture.
8. Crossed in time. The model and subject actions would match, but are performed out of order with respect to other actions within the same modality, or across modalities. Actions matched out of order between the two streams will pose a dilemma of how to score them. A strict position is possible, that of only allowing monotonically increasing matches within a single information stream. This will be assumed within each behavior stream of a subject. Task actions, for example, clearly have an important ordering to themselves. Ericsson and Simon's (1984) theory of verbal protocol production asserts that verbal utterances are produced in the order that they enter working memory, and we will assume this for other protocols that report on internal state as well. This sequentiality matching requirement cannot be applied between multiple information streams until it can be supported theoretically or empirically.

When the model does not match the data. Process models will fail, they will make predictions that are not supported, and the subject will do things that the model did not predict. There will be subjects that the model cannot be said to model at all. That this will happen has been noted since their first use (Newell & Simon, 1972, p. 197). It is not the end of the world. No model gets all the data all of the time. Not getting verbal utterances (categorical data) matched may seem worse, but that is only because it does not look like the noise in numeric measures that we are used to seeing.

Table 2-6 notes the several approaches that have been used to deal with mismatches. Which approach to use will depend on the desired result of the analysis and the tractability of the model and subject. One approach adds new requirements to the modeling tools, and one suggests a useful constraint for the subject's experimental situation. A combination will often be required. It may be appropriate to minimizing the effect of mismatches in different areas in different ways. For example, the experimenter may choose a novel task domain in order to keep the effect of previous knowledge low, so that the mismatches are caused solely by the model in the area of interest, say problem solving.

Table 2-6: Ways to deal with mismatches

- Avoid mismatches (if you can).
- Reset the model: Conditional Prediction.
- Reset the subject.
- Reset the data.
- Reset the interpretation assumptions.
- Do nothing.

Avoid this (if you can). The cleanest and most desirable way to deal with mismatches is to avoid them (and this is the way your mother would tell you do deal with this, at least mine would).

- Choose regular task domains, where a model is available and it already predicts that the subject will be well behaved or driven by the task. The tasks must also be those where the subject's behavior does not depend on previously learned knowledge invisible to you. The common choice of novel problem solving domains reflects this constraint (e.g., the Tower of Hanoi). The drawback of this approach is that it encourages the models to stay close to home, to not attempt to model the unmodeled.
- Do small tasks at a time. By defining the starting state for the subject more often through the task definition, it is intrinsically easier synchronize subject's actions with the model's predictions, and it provides a greater number of tests of the predictions (although shorter sequences are tested). For example, studies of subtraction problems have benefited from this data set feature.
- Provide a good simulation of the environment. Providing a realistic model of the environment will allow model to come back on track. If the environment provides fixed responses to any model action (e.g., on the 1st mouse action, display the first menu), the model must match the subject to go on with comparison (Kieras, personal communication, May 1992).
- Repair the model. The mismatch often tells you something; so go and fix the model so it mismatches no more.
- Reexamine your interpretation of the model and its predictions if its predictions are not automatically derived. The interpretation of the data with respect to the predictions should also be examined.

Reset the model: Conditional Prediction. The most common approach for dealing with mismatches that start truly divergent behavior is to perform condition prediction (Feldman, 1962). Upon divergence in behavior, condition prediction specifies that the model's prediction that went awry gets noted as a mistake, and then the model gets reset to the state of the subject at that point, and the model makes a prediction from that point. The summary measures must then include the number of mismatches as a reported result. The errors a model makes result from either an incorrect model, or an incorrect

specification of the initial state. Conditional prediction allows these two types of errors to be treated separately (it does assume that the subject's state can be determined so that the model can be set to it). This approach has been used quite often in teaching programs that learn (Samual, 1959) and testing cognitive models (Feldman, 1962; Newell, 1972; Newell & Simon, 1961; Newell & Simon, 1972, Ch. 12; Newell, Shaw, & Simon, 1959; Samuel, 1959; Young, 1973).

This approach is theoretically justified. The actions mismatching after the first mismatch are consistent with it, given a different starting state (the state that results from the subject and model applying different operators), the subject and model will often end up in a different state, and choose different paths after that. There will be areas where this assumption is not desirable, where the length of the sequence of correct predictions is important, but this work does not take up that assumption. This makes the most sense when the model provides a set of equally likely actions or a set of actions by likelihood (Anderson, Conrad, & Corbett, 1989).

As noted by Feldman (1962), conditional prediction has many advantages: (a) It ensures that the errors at any point are caused by new mistakes rather than carrying on from old ones. (b) By reconsidering after each mismatch, conditional predictions provides a framework for testing each component of the model. By resetting the model at mismatched intermediate states, later parts of the model can be tested even if the initial performance is not perfected yet. (c) By removing the number of errors caused by previous errors, the model's predictions can be tested more often.

In a production system model, the model can simply be reset (e.g., Feldman, 1962; Newell & Simon, 1972), or it can be modified by inserting simple productions that correspond to each divergence (e.g., "if cycle = 3 then set operator to be add-3"). If the mismatches are truly fixed points of departure, and are represented as such, they will never be used again, and it is not worth attempting to generalize them. If several mismatches appear similar, it may be worth attempting to generalize them, providing a real fix if the problem occurs across subjects. (e.g., "if environment-attribute = value1 then set operator to be add-3"). This is the process that ACM (Langley & Ohlsson, 1989) and Cirrus (Kowalski & VanLehn, 1988; VanLehn & Garlick, 1987) attempt to automate. If the mismatches are the result of individual differences between subjects, then the system becomes a basic-model plus an individual difference modification (Miwa & Simon, 1992). One cannot necessarily tell when the attempted fixes are inserted which they will turn out to be. That they are correctly general can only be found out with more data, particularly, data representing the same situation.

Reset the subject. Given a suitable domain and social environment, the subject can be reset instead of the model. This ensures that the subject does not stray far from model. This is not generally applicable because it requires special domains where a task environment has license to reset the subject's state (e.g., tutoring). Therefore it is mostly used for closed analysis with well developed models. Anderson and his group, for example, do this in their tutors. The subjects are actually taught by resetting them.

Reset the data. Sometimes the mismatch will be so gross, and the model so persuasive, that data itself must be questioned. In areas outside psychology, doing this can be quite common (Heise, 1987; Heise, 1989). There are a few examples showing that this is possible (e.g., Feldman, 1962). And in the analyses reported in Chapter 7 we found two transcription errors and one interpretation error based on the model's predictions. They were noted as wrong, and reset. While it is possible, it must be applied selectively and carefully. Over-belief in the theory is a mark of zealots, and a theory can end up untested.

Reset the interpretation assumptions. In testing the predictions of any theory, assumptions are made about the mapping between the theory's predictions and the real world. In this relatively young area, how to map the predictions of cognitive actions to real world events such as verbal protocols and hand movements, is not yet well developed.

Do nothing. There will be areas where the model will not match, and the final approach is not to do anything about the mismatch except note it, and continue the analysis from the $n+1^{th}$ segment

(Ohlsson, 1990). This is, of course, the least desirable approach. There are two types of unmatched data that could be ignored. The most natural to ignore are actions that are outside of the scope of the model's coverage. Nose scratching, laughter, and meta-comments should not count against the model, but the amount of this noise should be included in summaries. The other type of unmatched data are subject behaviors that should be predicted by the model. These are more dangerous to ignore. The sequential and dependent nature of the actions will lead this approach to very poor performance, particularly if it is used exclusively. This is done for whole subjects when their data are not used because it is "too chaotic to analyze."

A note on resetting the model. Each of the measures that will be discussed are based on the raw comparison between the model's actions and the subject's actions. The model and the subject will mismatch, that is what the measures are there to measure. However, if the two traces completely diverge, the comparison stops being useful. Without local adjustments to reset the model or tutoring instructions to reset the subject to bring them back into correspondence, all of the measures will provide less information on how to improve the model. The model will start the next action with a different initial state than the subject does. After a divergence, the model ends up being tested with less of the data because the model's actions are based on a different set of initial conditions after the time of the divergence than the subject's are based on. All the measures, but for measures of parsimony, will assume that the model can be resynchronized with the data when needed.

Initial measures. In addition to the aligned traces, there are several simple building blocks based on the model's actions or the subject's actions that are used by more elaborate measures. The measures of the subject's data include the number of segments, the length of the protocol in words and seconds, and thus the word rate, as a check on the quality of the protocol. Similar measures are required from the model, including the number of model actions, a measure of the internal time of the model, such as the number of production firings, operator applications, or other constructs. Taken together, these two sets of measures show the temporal density of actions for support. Additionally, analyses of parsimony will require the number of rules or separable components in the model.

2.4.3 Non-numeric descriptive measures

The first set of measures examined are those that provide direct, non-numeric descriptions of the model's performance with respect to the data. They do not provide predictions of the model's future performance, but are used to tell globally where the model mismatches the data and could be improved.

Informal qualitative comparisons. The test can be performed in several ways. The most straightforward measure that has been used is to present the model's predictions and the subject's actions together but not necessarily aligned (Feldman, 1962; Newell & Simon, 1961). The analyst asks himself (or others), Do the two information streams appear to be different? When it was first proposed, it was put forward as a type of Turing (1956) test.[1]

After the subject trace and model trace have been aligned, it is possible to simply display the correspondences in a table or diagram. This comparison also has some bearing on the utility of each operator or rule that generates the actions, but this is limited by the viewer's memory.

The simplest way to present the correspondences is to just present the two information streams aligned side-by-side. It is often used by analysts in their work (e.g., Appendix I of Chapter 7 in this document, Appendix 6.1 in Newell and Simon (1972), and Appendix B of Young, 1973). Portions of the alignment are sometimes used when reporting results (Young & O'Shea, 1981). The model's trace is often compared to protocols in a simple, informal way (e.g., "This trace is remarkably similar to

[1]It actually is the popularized form and name of the Turing test. In the original Imitation Game, Turing calls for the test to be between a man imitating a women and a machine imitating a woman.

protocol D", Luger, 1981, p. 70). If the comparison is done on a high enough level, all the correspondences can be presented (Johnson, et al., 1981). This level of detail is necessary for local improvements. By presenting a sufficient number of rows together, some context and flavor of the match can be obtained.

Process models are not normally implemented to the level of producing verbal output, but in an interesting extension in the direction of the popularized Turing test, Ohlsson (1980) added to his model of linear syllogisms the capability of producing talk aloud trace. The model's output appeared nearly human when compared side by side with a subject's verbal utterances. Seeing them together and being able to note the similarities made the model more believable. In this case, it was interesting to note how mechanistic the subject's speech was.

This measure weakly supports showing where the model can be improved. Where model actions were and were not matched can be found through visual search, but a more global view is often needed of which model actions or knowledge structures were matched and how often they were matched.

Informal comparisons are most often applied by reporting aggregate descriptions of the subjects' and model's qualitative (e.g., Larkin & Simon, 1981, Simon & Simon, 1978) or quantitative (e.g., Carpenter, Just, & Shell, 1990) behaviors. No formal comparison is made, the two behaviors are just described, and the reader is left to judge on their own the significance of the match.

This can be a useful measure, because in addition to being able to test that the model can perform the task on a basic level, it also brings to bear all kinds of unspoken constraints on the task performance. It almost always can tell an observer something about where the model could be improved, particularly the global ways in which the model does not fit the data. In a certain sense, it is used informally by all researchers when examining a model's initial performance. It will indicate global and local places where the model mismatches. It will probably not provide a reliable measure of future applicability. If being fooled as part of a Turing test convinces, then it is also one way to be persuasive.

While this measure does not commit any errors, this type of Turing test fails to be complete enough to serve as the only measure of a model. Knowing more than it can tell about the fit is generally required. It also fails on its own to highlight where the model fails to match; it requires an intelligent observer. The observer also has weaknesses, and will not be able to bring all the constraints in the data to bear. Finally, it fails to make predictions about how well the model will perform in other situations.

Contingency tables. The simplest way to summarize the comparison is to aggregate the correspondences by model action in a table. The model actions that would match subject's actions and the actions that actually occurred in the model can be compared with an N-way contingency tables (e.g., Newell & Simon, 72, Figure 6.8). In this type of table, the row headings are rules or operators that would match the subject's actions, that is, could fire, and the column headings are rules that did fire. The firings and possible firings are totaled up from each segment, and the totals placed as counts in each cell. This is a useful diagnostic aid. Cell counts on the diagonal indicate that model components could and did match the subjects behaviors. Otherwise the cell counts represent misses for the rows and false alarms for the operator represented by the column. The counts can be aggregated over subjects (Larkin, 1981; Larkin, et al., 1980) or over episodes of a single subject (Newell & Simon, 1972).

They are useful diagnostic aids because they start to summarize the model's behavior with respect to the subject's, and start to suggest which parts of the model could be improved. It major drawback has been that it is difficult to compute by hand. It also does not directly indicate where the mismatches occurred. It would be more useful if the comparisons making up the cell counts were accessible as a set.

Visual displays of the correspondences. There are two particularly interesting displays for presenting the match between the model and data. This type of analysis (without a graphical display) is also supported in some knowledge acquisition tools (e.g., Kriton: Diederich et al., 1987; Keats: Motta et al.,

1988). They allow the analyst to manually (or automatically) indicate by showing for each knowledge structure the matching subject segments. No cumulative measures or complete listings are provided however.

Figure 2-7 shows the first display. Peck and John (1992) created a display that graphically depicts the correspondences between the model's structure and predictions, and the subject's actions. It is an operator application support graph that depicts the operator application order of a process model and a visual description of which actions of the model were successfully matched by the data, and which parts are either unused or are not supported by the present data set. This diagram helps the analyst answer several questions: What actions of the model were supported? At what portions of the episodes do these appear? What proportion of the time does the subject give verbal support when they could? If there is an order to where the model does and does not fit, this diagram may be able to suggest where it is occurring, and what model elements might be involved.

This graph has two shortcomings. The largest shortcoming by far is that it currently must be generated by hand for each episode and each time the model changes. This makes it too expensive to use routinely. The other problem is that while it tells us directly where the model is wrong, that is, what protocol segments are not predicted by the model, it could indicate more clearly how the model is wrong.

Figure 2-8 shows the second interesting display. In this display, Sakoe and Chiba (1978) present the match between the predictions of a speech recognition algorithm and the actual speech. The time course of the correspondence is given as a warping function, which can be used to directly compare the time course of the subject and the time course of the model. This graph is used to compare sequential, time dependent predictions, except they are working in a model of simple speech production instead of a cognitive model. The time scales used here are relatively smaller, 100s of milliseconds, than most of cognitive models that operate in the 10 to 100 s domain. This will be used to build a display in a later chapter.

This graph is related to one by Just and Carpenter (1992, p. 140) where they compare the time course of the match between the cognitive model of reading versus the subject's actions. The difference is that they put each time course on different graphs, which makes the comparison more difficult.

Summary. The general performance of a process model must be described in order for it to be understood, so the qualitative descriptions that make up the popularized Turing Test must be performed for every model, while both building and testing the model, and when describing it to others. This test requires a skilled observer, so it will be the last measure to be automated, when we can create machines not only intelligent enough to pass the Turing Test, but to judge others.

Graphic displays of correspondences are necessary and useful, as well as the summary tables showing how much of the model is used and matched by the data. They constitute theoretical predictions of the importance of the various model parts.

2.4.4 Simple numeric measures

Simple descriptive measures of the model's correspondence to the data can be computed. They are often the first step in an analysis. They are useful for measuring local improvements while developing the model, and as simple summaries of more detailed comparisons. The number of subject actions matched to model actions can be stated in signal detection terms as noted earlier. Hits are places where the model and the subject perform the same action; misses are situations where the model does not perform an action that the subject performs, and false alarms are actions that the model performs that the subject does not. These simple measures are often reported in model descriptions, They have included the number of actions to do the task for both subject and model (Simon & Reed, 1976), the percentage of subject actions matched (e.g., Larkin, et al., 1980; Larkin, 1981, Young & O'Shea, 1981), and the percentage of model actions matched (Larkin, 1981; Larkin, et al., 1980; Peck & John, 1992; Young & O'Shea, 1981).

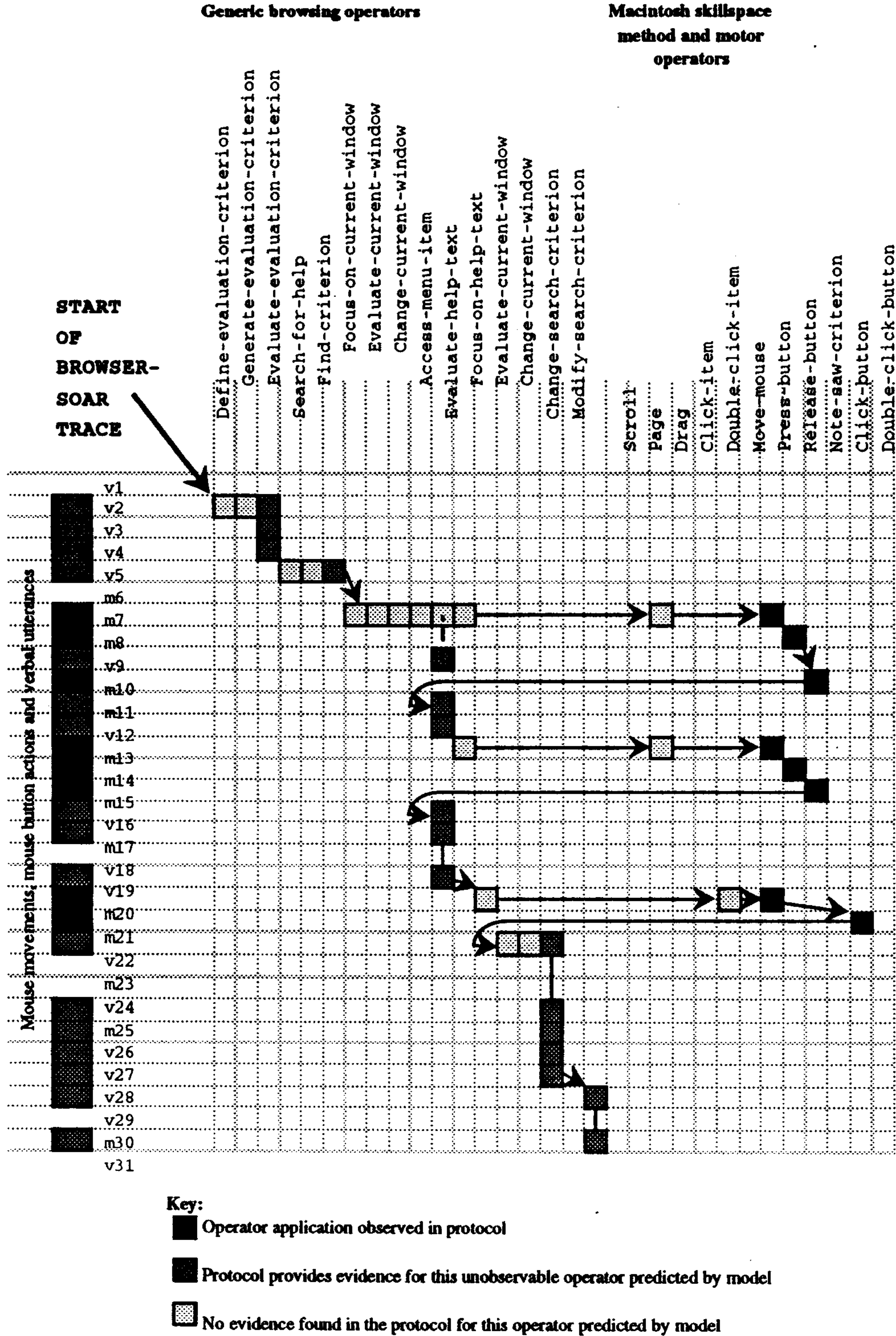


Figure 4. Representation of fit between observed behavior during the "write" browsing behavior excerpt (see transcription in Figure 2) and behavior predicted by the Browser-Soar model

Figure 2-7: Example operator application support graph, from Peck and John, 1992.

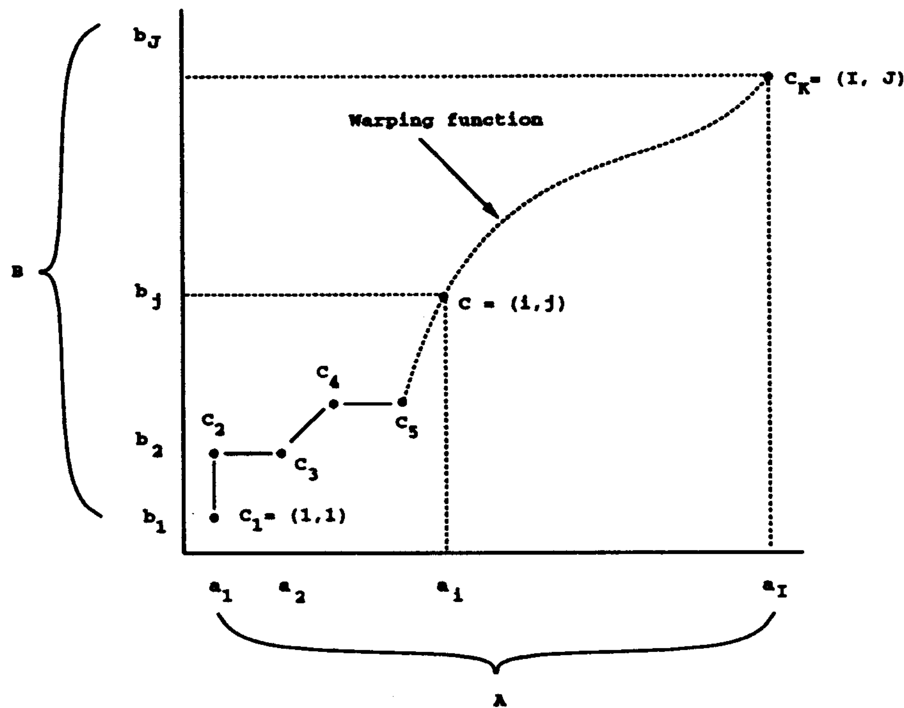


Figure 2-8: Example match over time display, taken from Sakoe and Chiba (1978).

More complex measures have been built from these building blocks in an attempt to combine them into a better summary measure. These measures include the error fit measure (fit = (Hits - FA)/(number of data points)) (Priest & Young, 1988; Young & O'Shea, 1981); and the micro-theory evaluation measure (u = (Hits - number of variant procedures)/(number of data points)) (Priest & Young, 1988). The raw numbers could also be combined in any two-way table of association that allows a missing cell, such as Signal Detection Theory (Swets, 1973; Swets, 1986) (more complete lists have been put together by Nelson (1984) and Reynolds, (1984)).

Simple aggregates and combinations of the match types are easy to compute. They often serve as useful summaries of the fit, and may be persuasive because they can be directly interpreted. If a general evaluation of the model is desired, all the component measures should be reported. Combining them requires additional assumptions about what the measure will be used for, and the relative costs of each type. Combinations cannot serve as measures for improving the model because they do not indicate where to improve the model, only a global measure of its correspondence. The raw components of these measures tell the analyst more about where to improve the model than their combination.

Any of the simple measure of the total numbers and percentages of hits, misses, and false alarms are useful measures for an analyst, it does not appear to matter which one is used, as long as it is readily available and helps indicate where to improve the model. By aiding others in understanding the global quality of the fit, they can also add to the persuasiveness, and should be reported when available. They must be augmented with other measures because on their own they cannot describe where to improve the model or predict how it will do in the future.

2.4.5 Measures of component utility

Measures of rule or component utility are used for measuring how much each additional component of the model adds to the model's performance. These measures are particularly designed to answer the question of the marginal utility of the last piece added. Did adding the last rule make the model fit much better or is it over-fitting the data, picking up just one action? This measure tells the analyst when to stop improving the model based solely on the current set of data.

For process models based on production systems, the most natural measure of the degrees of freedom in the model is the number of rules. In the extreme, it takes a single rule to do each action, and rules could be made for each data point. If fewer rules are required, the model would have less degrees of freedom. Nearly all of the measures included have used rules as their unit of model size, so we will just speak about rules. The only other apparent possibility is number of clauses, which would correlate with the number of rules, and their specificity: more specific rules would have more clauses. Different people may implement essentially the same model using different numbers of productions. These productions will also vary in generality. The number of productions has to be taken then as an upper bound on the degrees of freedom used (i.e., one might be able to create the model with less rules, but need not write more).

Simple counts. The raw number of firings per rule or instantiations per operator, without comparison to the data, is also worth using. As the number of applications per construct goes up, the importance of the rule presumably increases (e.g., Newell & Simon, 1972, Figure 7.29). As a measure of the model alone, it will help the analyst understand the model for later modification, and pointing out rules that have not fired and are perhaps incorrect. It can also be considered as a possible degrees of freedom measure.

Jackknife measures. The most difficult approach is to do a jackknife analysis based on the rules. In this analysis, each rule is taken out individually, the model is run and matched to the data, and a goodness of fit measure is computed, such as number of subject actions matched. By averaging the result and computing the sample variance, the difference in percentage of fit for each rule could be translated into the amount of variance that it accounts for. The significance of any rule could be tested with a plain F-test. Doing this for a process model with hundreds of rules (e.g., Browser-Soar) would be prohibitively expensive, even with tools to automatically do the task. A variant has been used on smaller models generating fixed responses, and they help illustrate the contribution of each sub-model (Young & O'Shea, 1981).

The cumulative hit curve. The cumulative hit curve (e.g., Newell & Simon, 1972, Figure 6.10; Ohlsson, 1980, Figure 6:8; Priest & Young, 1988) is a more specialized graph than a listing of how many times each model component was matched by data. It too depicts the number of data segments accounted for by each rule (or sets of rules), but the rules are sorted in decreasing order of data covered. Figure 2-9a depicts an example diagram. The curve will only reach 100% if all of the data are covered. Generating this curve simply requires keeping a list of rules and being able to aggregate the number of times they are supported by the data. It visually depicts the contribution each rule makes to fitting the data.

The diagram could be made better by more directly showing the measure this graph is actually designed to show, the incremental rule utility. In Figure 2-9a, computing the contribution of a rule requires a relative judgement between two unknown lines. Figure 2-9b shows a graph that shows the incremental amounts as an absolute measurement between a known line (the baseline) and top of a second bar for each rule.

Summary. Providing rule (or operator) counts, firing rates and the frequency of their support in the protocols appear to be useful measures. In addition to providing measures of the utility of the rules, the measures of fit can also serve as debugging aids for process models; rules that don't fire at all, or that fire all the time, are probably wrong in some way and need to be changed. A graphical display of what they match in the protocol is a way of visually displaying this information in a clearer form.

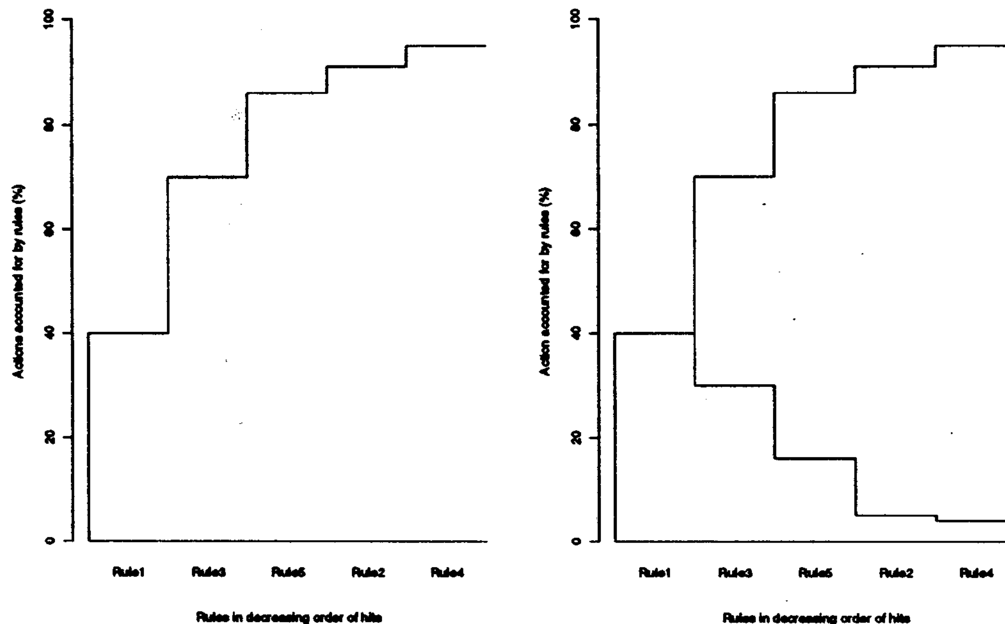


Figure 2-9: (a) Example cumulative hit curve (right). (b) Redesigned cumulative hit curve (left).

These measures have much in common with model based visual displays of the match, and summaries might better start there.

2.4.6 Inferential measures

A common form of persuasion in experimental psychology is reporting for effects the significance levels from inferential statistics. So researchers often looked there first for measures of the quality of model fit. These measures were designed to be a simple test of simple models, for example, that the numbers making up two means were sampled from different distributions. As process models are extreme hypotheses, inferential statistics do not apply. However, the measures that are computed for use in inferential statistics may prove helpful for highlighting where the model could be improved. There is also a general inferential statistic, the variance accounted for, that we can use to make predictions of future performance given multiple subjects.

Is the match better than chance? Examining the general questions of statistical significance first will simplify the remaining discussion. Can we tell if this model is significant?, that is, does it fit better than one might predict would happen by chance? Process models make a large number of predictions, that the multiple sequential responses of the model will match the multiple responses in the data. We are thus testing an extreme hypothesis. What chance is, in this case, is an impossibly rare event. The model will always (or nearly always) match many more tokens in the token sequence than chance would predict (Grant, 1962; Gregg & Simon, 1967). On the other hand, the models will often make predictions that turn out to be wrong, indicating that the model and data are distinctly different, and making the model's probability of being correct be zero. When sampling from different, overlapping distributions this result is not possible. So the answer to this question tells us little, and we must reject the measures of the probability of a model being correct. We can however, increase our belief that the model is more or less correct (the probability of the model) after it has been applied to more data, and the ability to move this informal belief, being persuasion, is one of the criteria for evaluating measures.

Do the model and subject appear not different? Perhaps the question of statistical significance can be redeemed by posing the opposite statistical question of does the model appear to be different from the subject. The two information streams could be compared to see if they are different, and an inferential measure used to determine if the difference is significant. We must dismiss this question too as irrelevant. Failing to reject a hypothesis is a weak claim, and is not a proof of similarity. The model will always (or nearly always will) be different than the behavior in some sense. This will be particularly true given a good data and large amounts of it. Poor experiments with small amounts of noisy data will be unfairly rewarded (this is why the data quality measure was first introduced). Most importantly though, the probability measures provided by the inferential measure does not tell us where to improve the model, nor do they tell how well the model will perform in the future. While we will not find these tests in themselves a useful measure, some of them may highlight useful things about the comparison in any case. The only use they have, is noting that the model is close to performing like the subject (but how close cannot be told).

Frequencies and transition rates. Two measures have been used to test the frequencies of action types and their transition rates. The Chi-squared measure has been misused in at least two different ways as an inferential statistic, but it could provide useful information. The most natural way to misapply it is to test whether the aggregate measures of the model's action types are different than the subject's. This would be an attempt to show that they are do not use the operators in different amounts (or similarly, a different number of items, using an F-test (Karat, 1968)). By accepting the null hypothesis it attempts to prove that they are the same. This measure has also been used to prove that the distribution of action types for a process model "could not have occurred by chance" (e.g., Koedinger & Anderson, 1990, p. 530). The second test makes more sense statistically, but doesn't tell us anything about the model, presumably, the model would not randomly apply its actions, but that is the point of a process model as opposed to a Markov model.

While it fails to be able to prove the model correct, the Chi-squared measure can be used to improve it. The analyst can examine the cell differences between subject and model, choosing to pay attention to the action types that are used in different proportions. This is not a complete measure, but it provides a summary, which is always helpful.

Sequential lag analysis (SLA) (Gottman & Roy, 1990) is also often used as a summary statistic that is useful for building process models as a way to find the major transitions or behavior loops. It can, however, be correctly used in a way similar to Chi-squared. It can note where the transition rates are different between the model and subject, and suggest areas of sequential behavior of the model to improve. It could be misused in ways similar to Chi-squared to test and improve process theories.

Variance accounted for. An inferential measure that can be appropriately applied is testing the significance of the correlation between the subject's actions and the model's actions (Grant, 1962). Tests are available for computing the significance of most types of correlations. Testing correlation rewards good theories by making their fit be significant. Significance is positively related to model quality, data quality, and data quantity. The partial measures contributing to the correlation that are low will indicate where the model could be improved. They will not indicate individual actions, but a category type, providing a global indication of where to improve the model. This is sometimes useful. As correlation can also be expressed as the variance accounted for by the theory, it makes predictions of how well future models will fit.

There are multiple features of a model that can be tested to see that they correlate with the subject's behaviors. Significance tests have been applied to correlations of reaction times on individual item tasks (Card, et al., 1983; John, 1988), to incremental reaction times (e.g., Just & Thibadeau, 1984), and to correlations of state transitions (e.g., Simon & Reed, 1976). There are special regressions available to account for the dependency of sequential measurements (Kadane, Larkin, & Mayer, 1981; Larkin, Mayer, & Kadane, 1986). Cautious modelers might test multiple facets of the model.

Individual subject and model differences. Given multiple subjects or multiple episodes per subject, standard statistical tests can be properly applied to the distribution of measures of fit are being sampled

from a presumably normal distribution. These measures would have to be numeric, and would include the variance accounted for and percentage of the subject's actions predicted, Standard tests (e.g., T-tests) are available to tell if a subject or model was matched significantly more poorly than others. Doing this would be useful to indicate which data set to pay attention to first, but not where in that data set. Visual inspection of the data might also be enough to tell which subject to start with.

In cases where the model is pliable enough to be over-fit, it may be useful to fit the model to one-half of the subjects, and then compare the model's fit to the second half of the data. Non-numeric measures would point out the differences in behavior and in fit between the two halves. In addition to telling more about the model, it may be especially persuasive to other scientists, who could recognize this schema from linear regression.

The differences in fit between two models could be tested in a statistical sense. The test results cannot be accepted as directly as they are when testing linear models however. The question of parsimony and other constraints on the models (if they differ) could greatly influence the final choice. How to weight the correspondence of a model to aggregate data like "human memory shows priming" is unclear.

Summary. Some inferential statistics have interesting submeasures that could be used to highlight where the match could be improved. But they must be used with caution. Checking to see if the model does not appear to be different from the subject may be useful to the analyst as an initial sanity check. It is not even a condition that all models must or can meet. In reporting this number as a result, it is accepting the null hypothesis, which is clearly wrong.

The model can be checked for significance with inferential statistics by testing the correlation between the subject's and the model's performance, through such measures as processing time, state selection, and strategy choices. As a stamp of approval, it is a weak test, most models of some merit will pass it. It is a good measure however, because it both provides help in finding where to improve the model, and given several subjects and a numeric measure of fit, they can give you estimates of how well, in numeric terms, the model will fit in the future. It will also be seen as persuasive.

2.4.7 A unified view: Criterion based model evaluation

A unified approach is also possible towards testing process models. What is actually desired is a model that is consistent with all that is known about human behavior, not merely the results of a small task, or a single subject's actual performance (Newell, 1991). The analyst and their audience need to know how well the model fits the data and what data it fits so that the model can be improved by adding more regularities and so that it can be compared with competing models, including variants of itself. When testing process models this suggests that in addition to fitting sequential data, it can be further constrained by having the model match aggregate data from other experiments. I will label this listing of the regularities accounted for as criterion-based model evaluation. In addition to matching the sequential predictions of a model to data, criterion-based model evaluation compares the model's performance with additional aggregate measures that would also apply to the task being modeled.

In criterion based model evaluation the data regularities and the model's correspondence are laid out in a listing or table. Figure 2-10 shows the type of format of regularities and the model's match for an example set of regularities modeled by FOKIBOFIT-Soar (Ritter, 1989) in the area of feeling of knowing for arithmetic problems. All kinds of regularities are possibilities, nominal performance, error rates and types, and of course the sequential behavior of particular subjects. Table 2-7 shows the five types of regularities that models have been asked to account for. The larger the number of regularities, the better. Various levels of fit, such as qualitative, quantitative, and not-at-all are allowed to describe the quality of fit. How well the model fits each regularity is included in the discussion of the quality of the model.

This approach is not new, but is merely being noted as an actual testing method that is particularly useful. Soar modelers often start with a list of criterion to model, and provide a scorecard of their

model's performance. For example, John (1988) provided a list of 29 regularities modeled by her typing model, Polk (1992) provides a list of 14 regularities accounted for by his syllogism model, and Newell's (1990) book is filled with lists of regularities that a unified theory of cognition must account for (e.g., Figures 1-7 and 5-14).

The FOK 8 — Model #6

Level of Qualitative Match	
XXXX	1. Strategy choice accuracy is pretty good.
XXXX	2. Only operator affects choice time.
X	3. Power law learning for answer time.
X	4. Power law strategy change.
XXXX	5. Linear relationship between learning and strategy choice.
XXXX	6. Frequency of the co-occurrence of the operands best predictor of strategy choice.
—	7. Abrupt changes in multiplication times.
—	8. Retrieving an answer does not guarantee later retrieval.

Figure 2-10: Example criterion table taken from Ritter (1989)

Rarely are lists of criterion used outside of the Soar community (for a counter example see Feldman, Tonge, & Kanter, 1963). For example, the development of Act★ (Anderson, 1983) has such lists implicitly in it, but no where does it include an explicit list of regularities that it covers, or of problem areas remaining to be addressed. For example, none of the 17 papers in the twelfth volume of Cognitive Science (1988) provide a list of regularities covered by their model. More typical tests of a model is to fit a single curve, a single data point, or even to informally exhibit similar behavior (Kaplan, 1987). Why this discrepancy? Are other scientists modeling simpler things? Are they doing it less well? Are the Soar researchers pedantic or irrelevant, or not communicating their methodology and more complex models? It appears that the Soar researchers are headed after more (but certainly not complete) unified theories (Newell, 1990), and with more data laid out in front of them need better ways to keep track of their fit to the multiple regularities. As other psychologists attempt to create larger models, they too will go towards more explicit listing of the regularities covered by their model.

2.4.8 Summary of measures

There are two basic results that the measure of model fit must provide. They must show where to improve the model and they must help predict how well the model will do in the future. There does not appear to be a single measure that will meet both criteria. Models can need improvement in lots of ways and make various types of predictions, so having several measures to do these tasks is acceptable and appropriate.

The analyst appears to end up needing three types of measures. First, a simple local measure to guide the improvement. While not providing the measure, signal detection provides useful terms for creating it. Some weighting of the relative importance of doing what the model does (hits and misses), while not doing what the model does not do (false alarms) is necessary. The degrees of freedom in the model, the number of rules or model resets, should also be kept in mind. These two measures should

Table 2-7: Types of data that have been used to test process models.

1. Sufficiency tests. The model is checked to see that it can perform the task in question in terms of the information it processes and its results, that is, be a functional model (e.g., the Logic Theorist (Newell et al., 1958) creating logic proofs).
2. General behaviors The general types of behaviors can be listed as regularities required in a model. These can include knowledge representations (e.g., visual schemas in geometry, Koedinger & Anderson, 1990), strategies (e.g, scientific discovery, Qin & Simon, 1990), operators and the problem spaces that they work in (the Logic Theorist: Newell et al., 1958). Learning and developmental effects are also powerful constraints (Anzi & Simon, 1979; Newell & Rosenbloom, 1981; Shrager, Hogg & Huberman, 1988; Simon, et al., 1991).
3. Qualitative effects of task stimuli The regularities can provide rather weak requirements for a model to meet. That different sets of problems be found hard or easy (Polk, 1992), and that dependent measures take on different values based on the task stimuli (e.g., typing non-words is not as fast as words, John, 1988)).
4. Quantitative behaviors (can be times or counts) The regularities can be that the model performs tasks at a set speed, or that task must be performed not only at relatively different speeds, but that the relationship in time be a given ratio (John, 1988). Similar constraints are available in relative response rates as well (e.g., Polk, 1992).
5. Sequential action correspondence The perhaps the strongest constraint available is that of sequential behavior — that the model performs a set of actions in a set order (e.g., Peck & John, 1992; Newell & Simon, 1972; Larkin, 1981). Currently, this constraint has only been taken to predict the order, but it could easily be extended to include time between predicted steps as well.

be directly combined. If the subject is being tested and not the model, then some measure combining the relative costs of hits, misses and false alarms is all that is required.

Second, for larger data sets, diagrams and graphs would be useful addition. They are often very compelling ways to display large amounts of information (Larkin & Simon, 1987; Tufte, 1990). They present more information than a single number, and can provide a more global description of the model's performance with respect to the data, helping to find where to improve the model. No visual cliches are available yet to describe the fit of process models' predictions, but there is no reason to believe that they are impossible to obtain.

Third, the analyst needs a measure of how well the model will fare on future data sets. The form this question has often taken is to prove that the model performed better than chance. Process models are extreme hypotheses, and there are no inferential measures for proving them. The only answer is that we end up like all other science, with arguments and evidence and no neat statistical proof. For example, we cannot "prove" special relativity with statistics, just provide arguments for it (which may include the odds of certain measures appearing by chance, but these are just part of the argument, not the argument itself), and show how well it fits the data (Jefferys & Berger, 1992). We can however, test how well the model's performance correlates with the data. This measure's virtue does not lie in providing a stamp of approval — it is a very weak test, that most models will pass. Its virtue lies in that it predicts how well the model will perform in the future, and by examining its constituent parts, it tells where the model could be improved.

The answer to being persuasive now appears to be simple. The model's strengths and weaknesses can

be explained to others with what is used to study and improve the model by their author. This approach may be convincing because of its disarming honesty, but if the measures do work as designed to tell the modeler where the model fits, doesn't fit, and how will it will fit in the future, they should equally well communicate these features to others. Finally, it is worth listing all the data regularities that the model accounts for.

2.5 Previous models of process model testing

This section reviews previous methods for testing process models as put forward in method sections of papers and books, or as implemented as computer programs. These have often been presented as methods for using process models as a way for doing protocol analysis, but are in reality ways of testing process models with protocol data.

The methodology presented in Newell and Simon (1972). Newell and Simon (1972; Newell, 1968) presented a method that has often been misclassified as merely protocol analysis. Its actual goals were to create and test rule-based process theories. Table 2-7 notes the steps in their testing methodology. Ohlsson (1990) presents model-based trace analysis as an interpretation and more explicit description of this methodology. Ohlsson's interpretation emphasizes the subject's behavior as a path through the problem space.

In this methodology, the protocols (verbal and non-verbal) are first summarized into states in a problem-behavior graph (PBG). An initial problem space for solving the task is created based on the task description alone. The rules making up the problem space are compared with the PBGs by hand. Tallies are kept of such measures as when the rules should have fired and when they did. This methodology (e.g., Newell & Simon 1972, p. 165) did not use timing data except for the order of the segments.

Table 2-8: Steps in protocol analysis method (Newell, 1968).

1. Divide the protocol into phrases.
2. Construct a problem space.
3. Plot the Problem Behavior Graph (PBG).
4. Create a production system.
5. Conjecture individual productions.
6. Consolidate the production system.
7. Plot the production system against the PBG.
8. Determine a conflict resolution rule.

Newell and Simon's methodology is incomplete, and does not focus on creating a running program and testing it. The emphasis of their methodology is on "improving the technology for developing theory, rather than for validating theory." (Newell, 1968, p. 183). Although Newell and Simon do not emphasize the routine aspects of testing process theories and protocol analysis, it is inherent in their work. They often use multiple subjects and multiple episodes. But it also takes huge amounts of time (Simon or Newell comment somewhere?). This methodology, based on detailed comparisons by hand, is not realistically set up for large amounts of data and more complicated models.

The theory implemented in Pas-I and Pas-II. Pas-I (Waterman & Newell, 1971) and Pas-II (Waterman, 1973; Waterman & Newell, 1973) specified a theory of building and testing process models as

software systems for automatically performing protocol analysis. The analyst created a series of rule sets that would rewrite the verbal protocol into codes, aggregate the codes into problem behavior graphs, and then these graphs could be displayed, or used to test process theories.

Pas-II supported many of the requirements to be noted in the next chapter. It attempted to automate the complete process, from segmentation through model building. It was designed for routine use, and emphasized automating the analysis. It supported the idea of semi-automatic analysis. But Pas-II failed to support several other requirements that can now be pointed out as important. The reasons for its lack of success, as noted earlier in the review, were equally related to its implementation and testing methodology. The simple parser could probably not provide automatic parsing if it was applied to additional domains. It did not closely incorporate the model and its constructs; this required the analyst to enter similar analysis rules in several steps. It lacked a visual interface and displays are now seen as central to this process. The numerous steps proved tedious. The proposed alignment algorithm between the predictions of a production system and the PBG was based on an incremental and continuous match with backtracking when the correspondence was provably wrong.

Competitive argumentation. VanLehn, Brown, & Greeno (1984) put forth a technique for testing and presenting computation theories of cognition that directly applies to process models. The technique primarily consists of noting how the fundamental principles underlying a model (and not just the implementation details) account for the data, and how similar principles would fail to account for the data. By presenting the principles in this manner, it may be possible to note which model components are necessary. They also call for the establishment of critical facts for cognition, facts that models must account for in order to be considered. The set of these critical facts can be used to compare different variants of a theory, showing why the best one is the one that is necessary because it accounts for the largest number of critical facts. This approach is consonant with the idea that unified theories of cognition should play the "anything you can do I can do better" game (Newell, 1990, p. 507). They present an example argument in their (1984) paper, and apply it elsewhere (VanLehn, 1983; VanLehn, Jones, & Chi, 1991).

Competitive argumentation consists of six components (Vanlehn, 1989):

1. A learning model.
2. Data from human learning.
3. A comparison of the model's predictions to the data.
4. A set of hypothesis (specifications for the model's performance, such as "Students [as realized by a model] expect a lesson to introduce at most one new "chunk" of procedure").
5. A demonstration that the model generates all and only the predictions allowed by the hypotheses.
6. A set of arguments, one for each hypothesis, that shows why the hypothesis should be in the theory, and what would happen if it were replaced by a competing hypothesis.

These are laudable goals that good summaries of models will provide whether or not the models are presented through competitive argumentation. In order to explain which parts of a model accounted for the data, the model must be understandable to the analyst, and when data are accounted for, the model components responsible should be noted. In problem solving behavior, it is probable that much of the critical data will be sequential in nature, and until we deal with process models and their predictions routinely, we will not see the critical data as clearly as we need to. In the end this can be seen as higher level presentation technique, not a testing technique for routine use, although gathering this information does test a model. The methods of competitive argumentation are completely worthwhile, but don't directly address how to test the sequential predictions.

ACM and Cirrus. ACM (Langley & Ohlsson, 1989) and Cirrus (Garlick & VanLehn, 1987; VanLehn & Garlick, 1987) start with a problem space of operators and their effects. They use the coded data to create a process model by specifying the application conditions for the operators. The operator application conditions are added based on information measures, so the model is undergoing a type of test as it is built. This testing process is not one that can necessarily be followed by hand, and appears to assume complete coverage of the subject's actions with operators in the problem space. The output is a decision tree of when each operator will be applied. The subject's actions that are covered by operator actions are presumably ignored, although if they are it would be simple enough to flag them as uncovered. It is not clear how they combine the models over multiple episodes or over multiple subjects.

Given a declarative representation of the problem space including an explicit description of the operators and a description of the environment's features, ACM and Cirrus's aggregate the operator application conditions. This is a feature worthy to include in any tool. They show that machine learning techniques can help summarize the subject's performance, in a form that can be turned into useful model components. The application rules that are learned can provide useful summaries; sometimes these rules will be incorrect reflecting a small sample size. The machine learning algorithms can do some of the abduction task of creating a model, but it is not clear where the difficulty lies, in creating the operators, or in defining the constraints on their application.

ASPM. Analysis of symbolic parameter models (ASPM) takes a process model realized as input/output tables, with a given finite set of parameters, and attempts to fit the model by finding the best set of parameters (Polk, 1992). It is possible to test a model against sequential data (as a series of responses), but it is much easier to test a model against single responses. ASPM can test only well developed models. Model fitting is done with known parameters. The input/output tables must be complete; all operator interactions must be noted there. Exhaustive search of all the sets of parameters is avoided by taking advantage of the structures implicit in the operator tables. ASPM assumes that the model is fixed, and it is the parameters that change. For sufficiently developed models, ASPM guarantees the best fit, but no direct indications of where the fit is poor. So it fails to provide a direct way to improve the model through testing.

Summary. With the description of the other methods now complete and with the short description of TBPA in mind from the introduction, I can note a few interesting similarities and differences between it and previous methods for testing process model's sequential predictions.

The major difference between TBPA and model-based trace analysis (Newell & Simon, 1972; Ohlsson, 1990) is that TBPA compares a full and actual trace of the cognitive model with the protocol, not just productions that could apply, and it works on a higher conceptual level than productions. In addition, model-based trace analysis sees the trace more as a way to analyze the protocols; the segments are hand coded to correspond to one or more of the model's operations (this is a type of analysis that we may wish to support as a preliminary or partial analysis). TBPA sees the trace primarily as predictions to be tested against the protocol data. Because TBPA incorporates an architecture (Soar) that can make time based predictions, it can also compare the model's speed with the subject's speed in doing the task, potentially an important source of constraints on models.

Unlike Pas-II (Waterman, 1973; Waterman & Newell, 1973), TBPA does not attempt to be fully automatic. TBPA directly and explicitly includes the process model and a declarative representation of it that can be used to assist and summarize the comparison. Most importantly, the analyst is not expected to modify the analyses by writing additional production systems, but to string together completed tools.

The presentation techniques of competitive argumentation (Vanlehn, 1989), and the model building techniques of ACM (Langley & Ohlsson, 1989) and Cirrus (Kowalski & VanLehn, 1988; VanLehn & Garlick, 1987) are just that, ways to present and build models. They include and indicate some useful functionalities for any environment that manipulates process models, but none that we must require in order to test them.

The earliest methods for testing process theories were implemented by hand. Although they were often used on large data sets, they were not designed for routine use, that is, applying the same model to multiple data sets, or in sufficient detail to automate them. The later, more automatic systems have generally specified a method that can only be applied to well developed models, and ones that have an excruciatingly detailed specification. TBPA is unique in presenting a method for improving an initial weak model to a stronger model through routine testing, which is what Grant (1962) believes is the basic process in science.

2.6 Summary of lessons for process model testing methodology and tools

Based on this survey of the previous uses of protocol data, the tools for manipulating protocols, cognitive process models, and comparing them, and the measures of model fit, we can enumerate several guidelines for a methodology for routinely testing process models with protocol data.

1. Graphic or tabular displays are required. The amount of information studied, generated, and manipulated when dealing with process models and protocols requires that a graphical presentation can be available. The presentation can done in tables or in diagrams. This requirement applies to the model, the model's behavior, both verbal and non-verbal protocol data, the correspondence between the model and data, and the residuals of any measurement.

2. Automate what you can, avoiding known pitfalls. There is a lot of bookkeeping and analysis tasks that are often done by hand in manipulating protocols, models, and their correspondences. Many of these tasks can be automated in a straightforward way, and they should be, such as the alignment of unambiguous actions. There are other tasks which look similar to these that cannot be easily automated. These will require separate research endeavors. Attempts to automate them will derail work on model testing or even general environment building.

   Parsing, that is, attempting to automatically interpret ambiguous data, particularly verbal data, in terms of a model, is perhaps the largest pitfall. Natural language parsing is not a solved problem. Attempts to include it in model testing have failed and taken much effort to do so. Simpler parses are possible though. Eye movements are now routinely translated into attended areas, and menu clicks by definition translate mouse movements into task actions.

3. The environment must be flexible. The environment the analyst works in must be flexible. A structured process and tool set can be presented, but many analyses will not be supported. Simple support for this starts with the ability to add fields to a display, and ends with the ability to create new analyses within the environment through a macro language. When the environment breaks down, the analysis is either not done, or the analyst must move the data to another environment.

4. Keep the original verbal data available. Verbal data contain a lot of information for model building and for model testing. The original data should be kept around for reference even after they has been coded; the model's performance may force them to be reinterpreted. It is a secret weapon — to gain new inspiration, "clear away an evening, and sit down and reread some protocols" (Newell, 1991).

5. Incorporate the model being tested. Analysis environments must explicitly incorporate the model they are building or testing if they wish to specifically test the model. PAW (Fisher, 1987; Fisher, 1991) only includes operator names, so it can only do tests based on operator names and their pattern of occurrences. Pas-II does not explicitly include a model, but allows the user to put pieces of it in various places, so the tests must be created by the user. SAPA includes the model directly, and thus can test the model

directly.

6. Know where the model is wrong so that you can improve it. Grant's (1962) position of scientists as model builders and improvers is persuasive. Improving the model requires knowing where the model is wrong, not if it is significant. Finding out where it is wrong requires generating the model's predictions, bringing them into close alignment with the data, and keeping the model around so that you can find what components are leading to difficulties (and conversely, which components should be left as is), and modifying the appropriate ones. This is basically the approach that is presented as trace based protocol analysis, presented in the next chapter.

Appendix to Chapter 2: Review of the Card model alignment algorithm

In algorithm theory the task of aligning the model's predictions with the subject's actions is equivalent to the longest common subsequence task. In its most general terms it is an NP-complete problem (Garey & Johnson, 1979). For a fixed language (which exists in this case, the types of behavior actions are fixed), or for a fixed number of sequences (which is also the case, there are two, the subject's and the model's behaviors), the problem is solvable in polynomial time (Wagner & Fisher, 1974) and polynomial space (Hirschberg, 1975). The task specification assumes that the globally best match is required, that there are no special correspondences that must be tied together, and that all subject data are used (and there are not any bits discarded as noise). These could, of course, be handled through simple extensions.

How the alignment is produced must be clear. The final alignment must be editable; even if the alignment is done completely and automatically, the analyst may have to jump in and redo parts by hand, either to correct the alignment because the specification of alignable objects was wrong, or to play what-if games. Our main interest in finding this subsequence is to use it to actually align the model's predictions with the appropriate data, rather than as a measure of similarity as it is often used. This gives us slightly different interests and needs. There are several that are simple and direct.

A potential problem is that the maximally common subsequence may not be unique — there may be several possible — and that we may have preferences about which one is returned. If we are just interested in the length of the maximally common subsequence, or the percentage of each sequence matched, which subsequence returned doesn't make any difference. For this task we have two preferences. First, we would like the longest subsequence that starts from the front. Since both the model and the subject move forward in time, the match must begin there. We believe that a priori the model's earlier predictions will be more accurate, and as modelers, we most often and most easily adjust the model's behavior starting with its initial actions. As we iterate through the cycle of match, modify the model, match, the earlier matches will be more stable, and require less modifications over time.

Second, the model's actions should guide the match. It represents the theoretical terms of the task, and in most cases there will be less model actions than subject actions, and these actions are less noisy, for our models don't have additional processes to add noise to their behavior like subjects have. Since the actions must actually match each other, and earlier actions are preferred to be included in the subsequence over later actions that also match, this constraint is also satisfied, although no additional changes will be required to the common implementations of the algorithm.

There are also requirements on the two types of data streams to be aligned. They cannot be ambiguous if the alignment is to be done automatically. This is a known problem of verbal utterances, but can also be found in a model trace if the token "add" is used to refer to two different types of constructs. Partial coding of the verbal data may be desirable here; coding polysemous words to a specific meaning. Having multiple data streams will also help; non-verbal discrete actions surrounding verbal utterances will help constrain the match of the verbal utterances.

The algorithm presented by Card, Moran, and Newell implements this algorithm by computing a matrix of possible matches, and then walks through this matrix generating the longest subsequence that matches. The initial step creates a matrix of counters, called SCORE, of size NUMBER-OF-OBSERVED by NUMBER-OF-PREDICTED. It then compares in order each token in the observed sequence against each token in the predicted sequence. Where there are matches, the counters are incremented to represent the longest possible sequence starting from Matrix(Observed-token, predicted-token). The final value of SCORE (SCORE[NUMBER-OF-OBSERVED, NUMBER-OF-PREDICTED]) is the size of the longest possible common subsequence. The second and final step traverses the matrix backward, starting at its most extreme point, generating one of the possible maximum length subsequences. Consider the example in Figure 11 matching DUC and DUDUDU.

```
Matching   predicted: D U C
and        observed:  D U D U D U
would generate the score matrix:

           D U D U D U
         0 1 2 3 4 5 6

    0    0 0 0 0 0 0 0
          \
  D 1    0 1 1 1 1 1 1
            \
  U 2    0 1 2 2 2 2 2
              |
  C 3    0 1 2-2-2-2-2

Alignment returned by Card1:

   Predicted sequence:      - - - - C U D
   Observed sequence:       U D U D - U D
```

Figure 11: Example alignment by the Card1 algorithm. The two strings being aligned are "DUC" and "DUDUDU".

How well the match was performed, the percentage matched, may be scored with Card's formula

Length(common-subsequence) / Max(length(observed), length(predicted)) (1)

This may be a bit pessimistic, in that it assumes that the longer sequence should be completely matched. An alternative formula 2 divides by the length of the shorter sequence, telling how many actions were matched that could be matched, taking the sequences as givens. The subsequences will be used for editing, so each of these measures are only used to inform the analyst how much editing by hand may remain to be done.

Length(common-subsequence) / Min(length(observed), length(predicted)) (2)

Chapter 3
Requirements for testing process models using trace based protocol analysis

This chapter first defines trace based protocol analysis (TBPA), a methodology for testing and revising process models with protocol data. This an attempt to specify and formalize the techniques used by Newell and Simon (1972) and Peck and John (1992) to analyze and validate their models. As noted in Chapter 1, the steps in the TBPA testing loop are (a) deriving the model's predictions, (b) testing the model's predictions by interpreting and aligning protocol data with respect to them, (c) understanding the results of the interpretations in terms of the model, and finally (d) modifying the model based on the test results.

The second goal of this chapter is to include the requirements for supporting TBPA within a software environment. The requirements based on the individual steps are supplemented by requirements based on the need to integrate these steps, to support this methodology computationally, and to test process models in general.

The most important requirement arising this way is to provide an integrated environment based on the model being tested, so that results from each step of TBPA can influence and summarize the other steps. In order to make TBPA more routine it will also be important to automate as many tasks for the analyst as possible.

It will not be possible to automate all the tasks, so the analyst must be supported in performing the remaining tasks. The environment should have a uniform interface. The environment will also have to be general, for all of the sub-tasks and their order cannot be specified in advanced. The environment must provide a path to expertise. The user must be able to use the environment and learn how to use it. And, perhaps the central problem in this task, the analyst must understand, manage, and manipulate large amounts of information, so they should be provided tools that assist with these tasks.

The role of integrating an intelligent architecture is also discussed. Because process models are implemented as knowledge within an architecture, an intelligent architecture will be incorporated within the testing environment. The architecture will provide the terms for aggregating model support. It is also worth considering if the architecture could be applied to the task at hand, and be used to automate all or parts of the testing process.

3.1 Definition of trace based protocol analysis (TBPA)

3.1.1 The inputs to TBPA

Routine trace based protocol analysis begins with a functional process model and protocol data to test it already in hand. Creating the initial model and gathering the data are by definition outside the scope of a methodology for testing process models. There are, however, minimum requirements that the inputs must meet.

3.1.1.1 A 0th order functional model

The first requirement for testing a functional process model is the model itself, a 0th order model that can perform the task. The model may be a preliminary model based solely on task analysis, and the task at hand is to test and improve the model (open analysis). Alternatively, the model may be a well developed and previously tested model, in which case the task may be to determine how well the subject matches each of several models (closed analysis). The model must provide predictions for each type of data to be used in testing it (e.g., verbal utterances and motor actions). The model may also have an associated simulated or real environment that responds to the actions of the model. TBPA assumes the model is in hand, but it is worth noting that the requirements for testing will also support

some of the processes for creating an initial model noted in Chapter 2, such as informally examining protocols.

3.1.1.2 Transcribed protocol data

The other required input to TBPA is previously transcribed and segmented protocols (such as video or audio tape, keystroke logs, eye-movement records). These must be put in a form that can be compared to the trace produced by the model. It must be transcribed and at least roughly segmented. Automatic analysis programs or the analyst may resegment it, but in routine analysis some initial segmentation is a necessary prerequisite. Marginally relevant segments, such as experimenter remarks, utterances too short to compare, and other extraneous material should be included and annotated as such; these items can provide context for understanding the segments. As two of the lessons noted in the review, it is desirable to include both the verbal and motor sub-streams, and time stamps are necessary for certain analyses.

3.1.2 The TBPA loop and its requirements

Process models built within symbolic cognitive architectures must be improved with a testing loop, they cannot be improved with closed form or automatic iterative analyses. Automatic iterative fitting methods are available when the direction and type of modifications to a model can be directly specified by fitting it to data (e.g., linear and logistic models (Afifi & Clark, 1984)). Improving a process model's fit is currently only possible by having the model generate predictions by performing the task, interpreting the data with respect to these predictions, then modifying the model, and then repeating the loop. The interpretation cannot be completely automated yet, nor can modifications to consider be completely prescribed based on the fit to the data, both of which are required for automated analysis.

What are the elements of the loop of testing process models with protocol data? How is this type of model validation specifically done? Figure 3-12 shows the relationship between process models and the protocol data used to test them. The analyst starts with a functional process model in hand and protocol data to test it.

The testing steps implicit in Figure 3-12 are shown explicitly in Figure 3-13. (1) As the model performs the task its actions are recorded and its internal states and operations traced. These actions are predictions of what will be in the protocol data. (2) The protocol data must be interpreted and aligned with the model's predictions, generating a sequence of correspondences made up of sets of the model actions and their corresponding data. These correspondences may contain a variety of mappings. A full list was presented in Table 2-5. (3) These comparison results must be analyzed to summarize where the model mismatched the data, and how to improve the fit. The summary may directly or indirectly indicate how the model can be modified to more closely match the data. (4) If the results are clear enough, they can be used to modify the model to more closely represent the behavior. The modifications will not necessarily correct all of the problems, so they must be tested through comparison with the data by going back to step 1.

Constraining the model with additional sources of data. TBPA tests only the sequential predictions of the model for a given task and subject. This represents only one level of comparison of the two information streams of model and subject noted in Figure 2-2, that of comparing actions on the *protocol* and *trace* level. In addition to testing the sequential predictions of a model, it will often be desirable and natural to bring the model's behavior into contact with additional data from other sources. For example, this can be done by comparing the model's aggregate performance with aggregate data from other studies, creating a less local model, one that is consistent with more data (Newell, 1990). If the aggregate measures can be related directly to the model, they may appear directly as constraints on the model's construction, such as incorporating a learning mechanism.

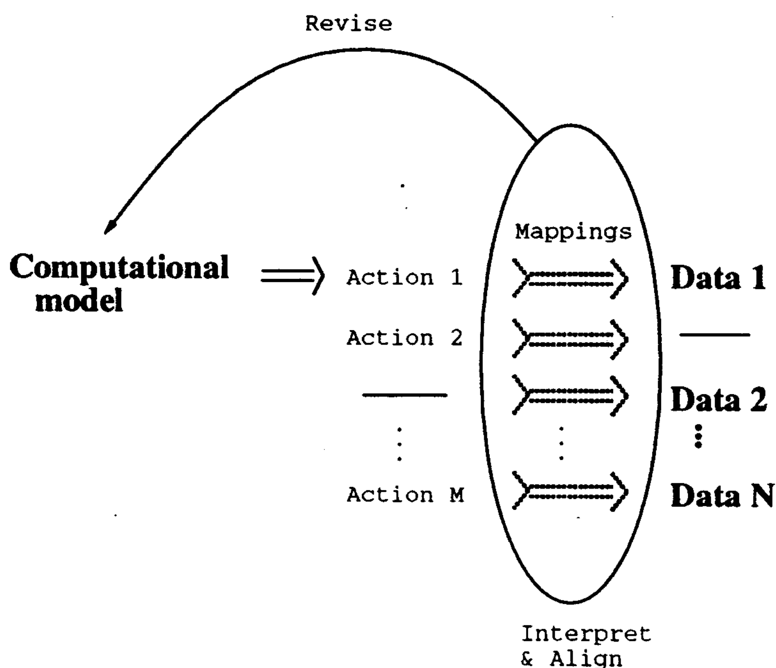


Figure 3-12: Diagram showing the inputs (in bold) to trace based protocol analysis (TBPA): A computational model and transcribed and segmented protocol data.

3.1.2.1 Step 1: Run the model to create predictions

As the model is run, a complete trace of its actions must be produced in a form that can be used to interpret the subject's actions. The trace is not an input to the testing loop because revising the model and reproducing the trace is part of the loop.

The trace should include all of the items in Table 3-9. The trace must be interpretable by both the analyst and the machine because at various times the comparison will be done manually and automatically.

(a) The trace should include unambiguous predictions for the content and timing in each information stream. The trace elements must be unambiguous. Automatic interpretation and alignment algorithms must be able to use the trace to automatically interpret the data. Within a computational environment the initial interpretation and alignment will be performed automatically for unambiguous comparisons, and it may be possible to perform a rough cut at interpretation on slightly unclear data points. The elements must also be easy for humans to interpret. Until the process is completely automated, a human analyst will need to be able to understand the trace to correct any errors and to do the final interpretation of ambiguous subject actions using the trace.

The trace must also include predictions for each protocol stream. The terms of these predictions will be dictated by the architecture and the data interpretation theory used (such as verbal protocol theory). Given the Soar architecture these will be terms of states and the operators to modify and create those states. Given a different architecture, such as Ops5 (Forgy, 1981), the model's actions used to interpret the data would be production applications, and the rules that fired would take the place of operators and must be included in the trace.

Including the model's simulated external actions (or actual, external actions, such as drawing on the

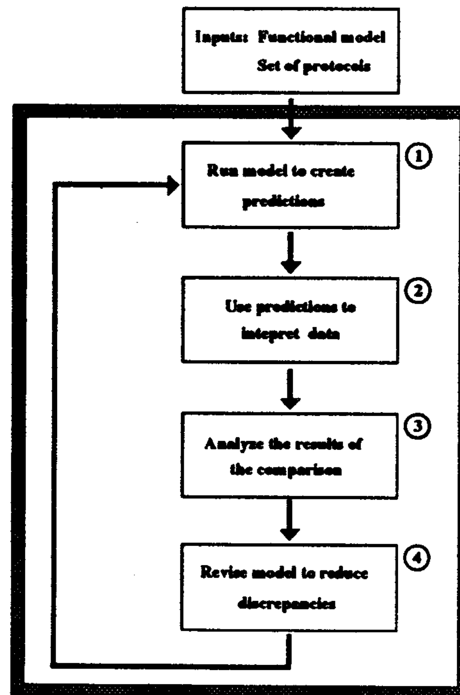


Figure 3-13: Diagram of the steps in testing process models with TBPA.

Table 3-9: Requirements for the process model's trace.

(a) Include:
  (i) Unambiguous predictions for each subject information stream (external and internal actions)
  (ii) Time stamps for each action.
(b) Be readable by the analyst.
(c) Provide various levels of detail.
(d) Provide aggregate measures of performance.
(e) Be deterministic even if the model is not.

screen) and the environment's responses to the model's actions provides information about the external task state and provides the context of the model's actions, so they should be kept in another synchronous data stream for reference.

The trace must include a simulation time stamp for each action if the speed of the model or architecture is to be tested. These time stamps, if unique, can also serve as identifiers.

It used to be believed that for problem space models, the trace did not have to include both the states and the operator applications because they are equivalent (Newell & Simon, 1972, p. 157). A listing of operators will indicate how the state gets modified, and a listing of states will indicate which operators have been applied. Since then our operators and states have grown up. Operators are less declarative, they are now context specific, they can be learned and their results can be modified through learning. The operators that are applied are actually instantiations of a semantic operator type, so the theoretically relevant instantiated features must also be included. For example, the trace of the add operator must include the value of the two addends.

(b) Be readable by humans. Analysts will also need to read the trace, both as predictions of behaviors

to be found in the subject's data, for debugging as a description of the model's performance. For example, the trace should support finding the context object's name and the goal depth.

Many cognitive modeling architectures are also implemented as AI programming languages. Soar is, for example. A trace useful for model building, especially debugging, is not the same as that needed for interpreting data. Programmers can handle more ambiguity and prefer a terser trace. A trace used for interpreting data may require considerably more detail than is used in a trace for model development.

(c) Provide various levels of detail. The trace must be able to present all the structures required to interpret the subject's actions. As a model is developed, and for different models, representations may differ and different levels of detail may be modeled. For some analyses and interpretation tasks one should be able to hide both whole streams of predictions and selected portions or levels of detail within a stream. How tightly the predictions are compared may also change as the model is developed. Initially the model may only match the subject's actions by operator name, such as add. Further development may allow the addition of operands that match. This will require an adjustable and flexible trace to provide in the simulation information stream an appropriate level of detail at a comparable behavior size.

(d) Provide aggregate measures of performance. The sequential data used to directly test the model's sequential predictions could completely test the model if all the sequences are perfectly matched. If they do not, comparing aggregate measures of the model with aggregate data measures may help characterize which types of actions are contributing most to the mismatch and where the model could be improved. Comparisons of the aggregate data may require additional displays to display aggregate aspects of both the subjects' and the model's behavior.

Subjects' aggregate measures are provided by standard analyses, and any tools for manipulating models should support aggregating the model's performances directly from the model as well. The right place to put these measuring devices, such as operator application counts, production firing counts, and cumulative simulation cycles, is in the architecture. It should not be necessary to hook up to experimental apparatus, or additional simulated experimental apparatus to take the measures.

(e) The trace must be deterministic even if the model is not. If the model might take multiple actions at a given point, the trace should indicate this. Often human behavior appears to be non-deterministic, and many architectures explicitly incorporate components to add stochastic behavior. Just because the system being modeled (the subject) appears to incorporate noise does not mean that the model must include noise as part of its process. So a necessary simplifying assumption is that of determinism, at least for this work.

It would be quite reasonable to make the trace sensitive to the data in a principled way, such as preferring the subject's choice when choosing between equal alternatives.

If the model incorporates a random component the best alternative to a deterministic trace appears to be a trace that indicates the range of possibilities at each step, with the trace summarizing the possible behaviors as options not taken along with their probabilities. This would complicate the interpretation process, requiring any dependencies in operations or states to be represented explicitly. If the model really was stochasticly implemented, it would probably be best to base the interpretation on a running model that could be reset at each choice point. Analyzing this type of correspondence is not attempted in this thesis, but may be an easy extension to this work.

There are several reasons for not incorporating a random component. In addition to having to modify our current architecture to add a noisy component: (a) A model that performs randomly is more difficult to represent and manipulate. The analyst must keep in mind not only the current trace, but what other traces could have appeared. (b) With a stochastic model in hand, there may be no way to see that a more deterministic model is possible, and no way to implement it. (c) When the model performs stochasticly, an additional layer of complexity is required either in the analysis or in the

model to aggregate the model's possible action sequences. The model actions must be represented aggregations of sets of behaviors; comparisons between the model's predictions and the data must be based on convolutions of probabilities rather than on direct comparison of actions. (d) The analyst ends up testing an even more extreme hypothesis: "This particular performance of the model matches this set of actions, and both are sampled from larger pools."

Determinism is an assumption many can believe in (Newell, 1990; Newell & Simon, 1972). The subject's actions may appear to be random, but it is just as likely that our models are incomplete. There are at least four ways to explain apparent non-determinism: (a) unincluded initial mental state information, (b) unmodeled environmental cues that are used in the task, (c) individual differences in previously learned information, (d) finer grained behavior rules than modeled (e.g., if A -> B sometimes, and A -> C sometimes, then perhaps what is required is if A.before-lunch -> B, and if A.after-lunch -> C).

3.1.2.2 Step 2: Use the predictions to interpret the data

Protocol analysis really is about model building and testing, not simply annotating sequentially ordered data. The core step in TBPA then is testing the model's predictions by using them to interpret data. Table 3-10 lists the requirements to perform this step. As theoretical constructs, the predictions provide a language for interpreting the data. The interpretation process may not be straightforward. When there are multiple interpretations of each action, finding which data segments are predicted by which model actions may involve problem solving. Where the interpretation is difficult or breaks down indicates where the model must be revised or extended.

The alignment process must handle concurrently all of the relevant information streams, including: (a) Each information stream of data collected from the subject (e.g., motor actions, verbal statements, eye-movements), (b) Corresponding predictions for each mode from the model, (c) Responses of the subject's environment, (d) Responses from the model's environment or simulated environment. Some of these, because they will be unambiguous, will help constrain the comparison. Others, like verbal protocols, will be ambiguous, and require more effort. At times the analyst may wish to hid some of these streams, and may desire to collapse several into each other. But they all must be available.

Table 3-10: Requirements for using the model's predictions to interpret the data.

(a) Display and support editing the correspondences.
(b) Automatic alignment of unambiguous actions.
(c) Support matching ambiguous actions.

Display and support editing the correspondences. The alignment does not do any good if it only exists inside a data structure and cannot be examined by the analyst. After any alignment automatic or manual, the analyst must be able to view the correspondences and the information streams that the correspondences are made of. As noted in the review, there are many small annotations and analyses done by hand with the protocols, the model's trace, and their correspondences.

As noted in the review of protocol analysis tools, most tools only display a few segments and their accompanying fields at a time. An example of this type of limited display was shown in Figure 2-6a. In order to understand each mapping, more context is needed than a record-based display can provide.

The analyst may also want to play what-if games with the alignment not supported by the automatic aligner. Upon occasion it may be necessary to add whole new fields, and to hide existing ones. For example, the verbal data must be kept available after coding. If the data do not get condensed by coding, it serves us little to pitch the raw data. If the data are greatly compressed, we may find that we are losing information as the model changes. After the data are coded however, the display may be more easily interpreted with the verbal information temporarily hidden.

A tabular display, as shown in Figure 2-6b, with its much larger content, is required in order to see the context of the model's fit, and to start to see how it varies across the episode. In all, the analyst will need a full visual editor to manipulate these data structures.

Interpret and align the data with respect to the predictions. This is the basic task in the loop, that of interpreting the protocol segments in terms of the model's predictions. Generally, the subject's overt task actions are compared with the task actions of the model, and the subject's verbal reports while performing the task about their mental structures and operations are compared with the trace of the internal states and operators of the model. For unambiguous data, such as mouse clicks or key presses it may appear as just a matching process. For verbal protocol, it will be a challenge to interpret the utterances in terms of the model's actions and states.

The transcription of the subject's actions may not use the same terms as the simulation, so an interpretation function will have to be provided as part of the model to support automatic alignment algorithms. Unambiguous actions (such as external task actions) may be directly comparable through something simple like, "the operator *Click-mouse* is equivalent to the transcribed mouse action 'C'".

The less ambiguous data, most often overt, task-based motor actions, should be aligned with respect to the model's predictions first. Once aligned, the less ambiguous data will help constrain the interpretation of the more ambiguous data. Even though the predicted actions and the actual action sequences may have different lengths, and will often not map one-to-one, this should be a straightforward matching process with unambiguous data. The types of behaviors that can be aligned with each other, the types of matches that are possible, and an algorithm for performing this alignment task were described in Chapter 2.

Support interpreting ambiguous actions. A simple interpretation function cannot in general align the subject's verbal protocols (describing internal mental states) with a trace of the model's internal states. This is a general parsing problem, albeit one with the knowledge structures in hand, but a parsing problem none-the-less. More ambitious interpretation functions are required, and this close juxtaposition of knowledge structures and verbal utterances represents a unique opportunity for natural language parsing, but taking advantage of this is not possible as part of this effort. By providing a semi-automatic and partial alignment that can later be cleaned up by hand a simple interpretation process may still be useful to the analyst with more ambiguous data streams. The analyst may also need to edit the correspondences generated automatically; the automatic alignment algorithm may be faulty, or called with incorrect comparison descriptions.

3.1.2.3 Step 3: Analyze the results of the comparison

Once the model's predictions have been used to interpret the data, it will be necessary to have a global view of all the places where the predictions fared poorly and where they fared well. The correspondences must be summarized in terms of the model if they are to be used to improve the model. The requirements for performing all of the levels of this analysis are shown in Table 3-11.

A scientist is interested in what varies to make an episode and what remains the same across episodes. The model being tested (as the architecture plus knowledge) is what holds the analyses together. The subject's actions cannot hold the analysis together, they are fixed for a given episode and cannot change, and across episodes they vary. The model's trace can serve as a summary for a single episode, but across episodes it too varies. The structures in the model, the objects that generated the trace, are what must serve as the summary of the invariant relationships found in the data.

Table 3-11: Requirements for analyzing the comparison of the data with the model's predictions.

(a) Show where the data does not match the predictions.
(b) Aggregate the results of the comparison in terms of the model.
(c) Interpret the test results as clues for modifying the model.

Show where the data does not match the predictions. One of the first measures an analyst needs to know is which subject actions have not been matched by the model's predictions. In order to quickly test simple modifications analysts need a simple, direct measure that allows these data points to be picked out and summarized. These simple, local measures of fit for immediate use should be provided automatically. Despite the often expressed desire for such a measure, as noted in the review section, this cannot be a global measure of model fit that can be used to compare two (or more) models so that one can be definitely preferred. The choice of model will, in general, be dictated by other factors, such as parsimony, expected usage, and fit to other constraints.

Aggregate the results of the comparison in terms of the model. The analyst needs to see a model-based description and summary of the comparison. As noted in the review of measures, failures in the interpretation process are particularly interesting. These can be initially identified by looking at the two information streams and noting holes in their correspondences, but several different summary views are also required. The analyst will need to be able to summarize globally which types of model actions got matched and which did not, to view the relationship of the matched actions to the model, to view the matched predictions with respect to time, and still be able to directly examine the matched (and unmatched) trace items after they have been aggregated into sets of summaries. As the models get larger and the data sets used to test them become larger as well, more efficient and useful displays will have to be developed. Graphic displays of model support, like the one used by Peck and John (1992) (shown in Figure 2-7), will be needed.

Support interpreting the correspondences as clues for modifying the model. Given a listing of the correspondences, the next task is to interpret these as clues about where and how to modify the model. In many ways this is an abduction task, rather like creating the initial model, but finer grained. This task will require seeing the context of the unpredicted mismatches to aid in their interpretation, and to recheck the match to avoid breaking other model components that are currently supported.

The discrepancies are a new source of data; task analysis represents a type of mean of behavior, the differences are the deviations from that, giving us access to higher order components of the qualitative data. The mismatches, if summarized, can provide an indication of how the model can be improved. For example, Newell and Rosenbloom (1981) showed that plotting learning data against different learning curves provides sets of signature differences.

There will be many fairly obvious ways to interpret the mismatches themselves. Examining the types of mismatches from Table 2-5 suggests the interpretations shown in Table 3-12. Additional types of mismatches and patches are also available from more specific descriptions of the model and its implementation as a production system (Corsaro & Heise, 1990). The non-obvious problems and the ability to combine the changes in the most parsimonious way will continue to make this an abduction task and real science. The environment should assist in this task by providing useful summary displays of the comparison, including the mismatches, their types, and related model components and behaviors.

Interpreting the subject in terms of the model. The analysis stops at this point if it is a closed analysis. The result is an interpretation of the subject's behavior with respect to the model's predictions or a classification of the subject behavior as belonging to a particular model chosen from a set of models.

3.1.2.4 Step 4: Revise the model to reduce the discrepancies

The final step in this testing loop is to use the results of testing the predictions with data to modify the model. As listed in Table 3-13, this step can be broken down into two requirements, first, to understand the model, and then second, to modify it based on understanding the model and knowing where its predictions do not fit the data.

Supporting understanding the model. Before modifying a model, the analyst must first understand it. This understanding may be supported in several ways. Users may gain understanding from playing with the model, running it and watching its actual performance. Viewing the model's structure may

Table 3-12: Some of the model modification clues based on the types of matches in Table 2-5 and their aggregation.

- Verbal utterances may reference knowledge structures and operations not yet incorporated in the model (misses). If the verbal utterances are sufficiently clear, it may be possible to simply add the representations and actions to the model.
- The order (crossed in time) and existence (misses and false alarms) of non-verbal actions may differ between the predictions and subject actions. These differences may be reconciled as different strategies to perform the task. Individual differences in subjects' knowledge and strategies may have to be included in each model as Miwa and Simon (1992) suggest.
- Actions the model performs more often than the subject performs them (misses) may indicate operations that need to be specialized and applied to more specialized situations.
- Model predictions that are not matched at all (false alarms) must always be considered for removal. Certain non-verbal structures will be difficult to report (Ericsson & Simon, 1984), or the predictions may be a non-reportable aspect of the architecture, such as goals in Soar. They may simply just not be observable. Or they may simply be not necessary.
- Slips and irrelevancies in the subject's data stream outside of the scope of the model (uncodable) may be ignored. When they clutter a display, the analyst and system must be able to ignore them.

Table 3-13: Requirements for modifying the model.

(a) Display the model so it can be understood.
(b) Modify the model based on the comparison.

also be useful. Both of these activities will require that the information be presented in terms of the model and underlying architecture.

As noted in Chapter 2, most cognitive modeling tools are research systems designed to show that a given architecture is sufficient for modeling intelligent behavior, not for routine application. Most do not yet provide good interfaces for running and viewing models, and this is true for Soar in particular. Established metaphors have not been developed for explaining models automatically yet, so one will have to be developed for them.

Modify the model based on comparison. The final task of testing process models is to incorporate the modifications that are suggested. The model must not only be generally ductile, but must also be modifiable based on the prescriptions found through the testing process. A partial list of abilities to modify the model on this level would include adding, modifying, and removing components, running the model, and examining its internal state. These abilities are essentially the same set of tools required to write new models, and could be used to help provide the initial model.

Process models perform the task by adding knowledge to an architecture for intelligence. Soar itself is built upon production systems, a type of AI programming language. Therefore, these cognitive models can and should be viewed as AI programs. Manipulating and modifying them should be viewed as an AI programming task requiring an AI programming environment.

3.2 Supporting TBPA with an integrated computer environment

This section notes the requirements that arise from integrating these steps, and the requirements necessary to support the methodology with a computer environment. The aids must be integrated, easy to use, extendable, and learnable. Automation of the most routine tasks and pathways to expertise are the primary ways to reduce the difficulty and extend the power of this environment. The analysts have limited processing power, working memory, and time, and large amounts of data to examine. They may not use this environment all day, every day, so even the advanced users will need assistance. Table 3-14 lists the requirements directly based on supporting TBPA with a computer environment. These requirements apply to each part of the environment, and to the environment as a whole.

Table 3-14: Requirements based on integrating the steps and supporting TBPA with a computational environment.

(a) Provide consistent representations and functionality based on the architecture.
(b) The environment must automate what it can.
To support the user for the rest of the task:
(c) Provide a uniform interface including a path to expertise.
(d) Provide general tools and a macro language.
(e) Provide tools for displaying and manipulating large amounts of data.

3.2.1 Why an integrated environment is needed

There are two orthogonal reasons why an integrated environment is required. The first is the desire to represent the results in terms of the model and its predictions. This will directly support using the results of previous steps to perform later steps. As noted in the review, it is useful to integrate the data and the model in various analyses. The component systems will be able to pull information from all the environment's programs using their data formats and functionality. This is required if the analyses are to present the results with respect to the model, and display the model with respect to its structure.

The second reason an integrated environment is required is that the analyst will not complete each step before moving on to the next. A normative description of this methodology has been presented so far as performing step 1 to step 4 completely and in order. The actual analyses may be less ordered. The analyst will not simply run the model (task1), then do the complete alignment (task2), and so on. There will be multiple cycles and partial cycles through the different sub-tasks. Often the analyst will end up performing part of the loop, only to find that a previous analysis was incomplete or wrong, or that another facet of the model could be changed without completing the testing loop. In the end this story will appear true, but initially the steps are just tasks to be achieved rather than a plan (Suchman, 1983).

In this vein, all the sub-tools should live in the same environment to encourage small exploratory analyses, and examination of the multiple data types and paths. Any macros the analyst creates must be able to call the other tools and reference their data. As an example of the flexibility and integration required, in addition to complete analyses, the environment should support pseudo-revision of models, trying out simple partial changes to the model's trace, such as changing the grain size of the analysis, without requiring a complete functional implementation in the process model. Figure 3-14 diagrams this. In this form of pseudo-model revision, the environment should support generating a new model trace by hand, inserting or deleting an operator not yet in the model in the trace, and testing it as any other model trace against the data. The analyst will need to perform tasks like this in general, inserting constructs into the model before there is functional code to support them and comparing proposed but not yet implemented operators with the subject's data.

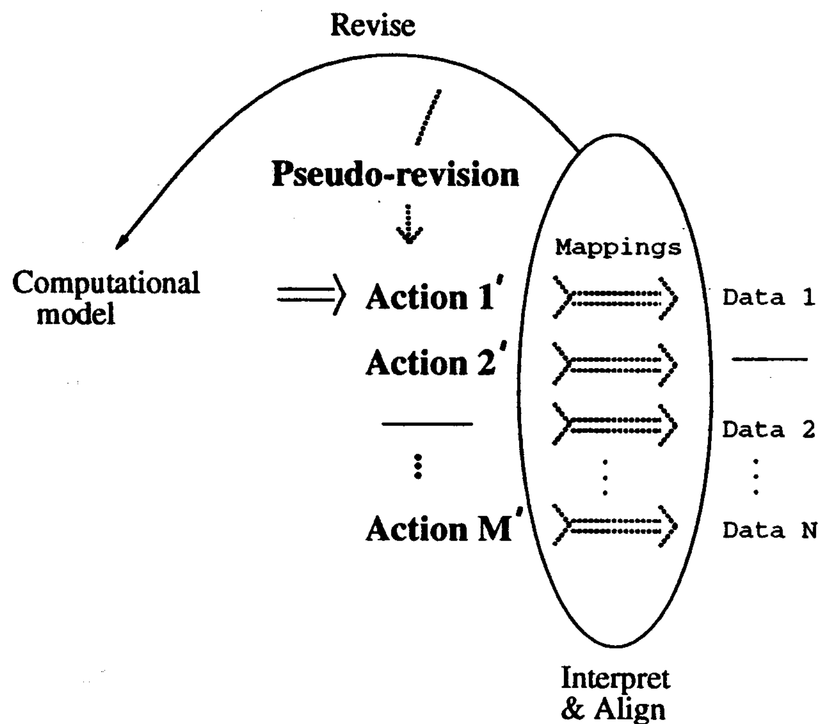


Figure 3-14: Diagram illustrating direct trace modification as a form of pseudo-model revision.

3.2.2 The environment must automate what it can

There are several aspects of the testing environment that will make it difficult to use. The amount of data and the allowed manipulations are large. The environment will be made up of several different systems, for modeling, matching, manipulating, and graphing. The users will not be novices at the task, but will be novices with respect to the environment. They will need assistance, but of a very different kind than typical novice users, who are novices at both the task and the supporting environment.

As noted in the review, it is not currently possible to automate the whole testing process in its general form. However, performing the analysis quickly will require that the system automatically assists in some ways in performing each of the steps. This assistance is also required to reduce the cognitive load and remove repetitive actions. Automatic actions will make the task less tedious, and is a necessary step towards making the testing automatic. This level of assistance will be achievable.

3.2.3 The environment must support the user for the rest

Although we may dream the dream of intelligently automated analysis, currently and in the foreseeable future, not every action can be automated. The analyst will end up cleaning up the automatic analyses, and performing those too difficult to automate.

General tools and a macro language. While a general methodology for testing process models with protocol data can be laid out, the examination of the previous examples of analysis presented in Chapter 2 shows that many different submethods and techniques are used. The general and fluid needs of this task require general tools, supporting many different analyses. A macro language must also be

provided. Within a given analysis, there will often be unanticipated subtasks that must be applied numerous times. These may be simple changes, such as replacing one word with another, or more complicated tasks such as conditionally reinterpreting and realigning subsequences of the predictions with the data.

Providing uniform interface and a path to expertise. The testing process is a difficult task itself. The computer environment to support it should not add to this burden, but lighten it. The first step is to provide an interface that is uniform across the tools. The second step is to provide a path to expertise. This path starts with making the environment menu driven for novices. Experts will want to perform more rapidly by using keystroke accelerators and the basic commands for macro creation. Novices will progress to experts through on-line copies of manuals, and help on the menu item, keystroke, and function level.

Display and support manipulating large amounts of data. As noted in the review chapter, fully testing process models will require better displays of data than are currently used. The analyst has on hand too much data, and cannot understand it when it is presented only as text. If the interface is done well enough, the raw amount of data that has to be manipulated will become the limiting factor in performing the analysis.

The data that have to be manipulated includes (a) the protocol statements and any preliminary analyses or coding of the segments, (b) the trace from the model, (c) the basic content of the model, including a snapshot of all short-term knowledge for any given time, and all long-term knowledge, (d) the correspondences between the protocol and the trace. The analyst will also need the ability to characterize the essential features of all of the above data sets. The environment must also contain information on itself, such as which analysis is being run, and which files are open. Each data type will need its own display, but several of these data sets are made up of elements from the model and the subject's actions, and the displays must incorporate their relationship.

The displays for two of the data sets must receive particular attention. First, the analyst must be able to understand how well the model fits. They must be able to examine and understand the comparison between the model's predictions and the subject's actions. Where and how to improve the fit between the model's predictions and data are difficult to see without a special "looking glass". Second, the analyst needs to understand the model, its structure and behavior so that they can modify it.

3.3 The role of an intelligent architecture in the testing process

There are three roles the architecture can play in creating an integrated environment for performing TBPA. The first is providing an interpreter for the knowledge that makes up the model. The second is providing a language for summarizing data supporting the model being tested. If the architecture deals with problem spaces, states, and operators, like Soar does, then the support must be in terms of problem spaces, states, and operators. If the architecture is based solely on rules, then the support must be in those terms. The third role that an intelligent architecture can provide is the promise of more extensive automatic analysis. The analysis task is also intelligent behavior, and is susceptible to being modeled with a functional model. Once modeled, the model can be used to perform the task.

3.3.1 Soar: The architecture used in this environment

Below a particular level of detail, which we are now at, the specific architecture that will be used in this work must be specified. It will be integrated within the environment, and to a certain extent influence the methodology through the types of tasks that can be approached.

General requirements for a cognitive architecture could be noted here, but this has already been done in some detail by Newell (1990). However, it is worth noting that the requirements for the architecture, even the general requirements, get highlighted through testing. For example, the general requirement that the architecture support learning applies to models of behavior as short as several

hundred seconds. Ohlsson's (1980, p. 184) attempt to model solving linear syllogisms was held back by his model's inability to learn between and during problem solving episodes.

Soar, an architecture for implementing process models. Soar (Laird et al., 1990; Laird, et al., 1987; Newell, 1990) is used as the architecture in this work. Soar is a proposed unified theory of cognition realized as a relatively well developed software system. For the purpose of this paper, Soar is not so different from previous architectures for cognition (e.g., Newell & Simon, 1972, Ch. 14; Simon's architecture (Simon (1979; 1989) or Erikson & Simon (1984)). Soar is just created with more constraints from psychology in mind, with additional abilities and improvements from AI algorithms.

There are other candidate architectures. The primary ones include ActR (Anderson, in press), CAPS (Just & Carpenter, 1987; Thibadeau, et al., 1982), the family of PDP models (McClelland et al., 1986; Rumelhart, et al., 1986), and numerous less developed architectures (SigArt, 1991). Most of the techniques covered here are not theoretically limited to Soar; any cognitive architecture that can produce a trace of predicted actions or mental states could have been used. Soar is a particularly good one to start with; most proposed architectures are less developed as computer systems, less psychologically oriented, or both. However, whichever one is chosen, its features will get embedded in the environment, in many places and in many ways.

The key points of Soar that the reader must understand to follow this thesis are simple and few. Soar uses knowledge represented as productions (Newell, 1980a; Young, 1979) to formulate behavior as goal directed search in and through problem spaces, so all long-term knowledge, such as knowledge for operator application and selection, is realized as production rules.

By their application operators move the system from one knowledge state to the next. When progress is stopped by lack of knowledge or conflicts in the available knowledge, a situation-based impasse is declared, and the architecture takes that as its next goal to solve. This goal in turn is approached by selection of a problem space to solve it, an initial state of knowledge, and an initial operator to apply. The selected items are known as context elements. If progress is again impeded by lack of available knowledge, another goal is created, a problem space is selected and so on. A subgoal hierarchy, representing working memory, is created this way. When an impasse is resolved, a new production (a chunk) is learned. It represents the knowledge used to solve the problem (the condition of the new production) and the information needed to avoid the impasse (the action). This chunk will prevent similar impasses from occurring in the future.

Each object in Soar is represented as a set of attribute-value pairs. Object names are also represented with attributes. Many objects will be created with the name attribute filled, but they need not. When an attribute is not provided, this can also be tested. Each object is unique though, and at the time of its creation, it is given a unique ID, such as G23.

The choice of which object to select for a given slot, when there are multiple choices, is decided with a preference calculus. The object must first be *acceptable*. If there is a single object also with a *best* preference or an object that is *better* then the all the other selections, then it is selected. More complicated combinations are possible. Full details are given in the Soar description and manual (Laird et al., 1990; Laird, et al., 1987; Newell, 1990).

A macro language has been created to manipulate Soar on the problem space level. TAQL (Yost, 1992; Yost & Altmann, 1991) consists of a set of constructs that correspond to the natural actions on the problem space level, such as operator proposal and comparison between two objects. The constructs are compiled into Soar productions by the TAQL compiler at load time. The TAQL compiler includes the ability to provide predictions of the problem space calling order based on the TAQL constructs loaded.

The Soar architecture has a basically deterministic implementation. The selection between problem spaces, states, and operators may be stochastic when the choices are equivalent, but upon their selection they are implemented deterministicly in a context sensitive manner (e.g., the operator *add*

does different things with different numbers, but always the same thing with the same numbers). This assumption simplifies any comparison with data.

3.3.2 Making functional models examinable

But where is the Soar model so that we might know it, and support it with data, and later automatically modify it? How does one get at a Soar model? Soar, as it comes out of the box, does not have explicit problem space level structures; operators and other problem space level objects are only implicitly available in the productions. The problem space level structure appears as emergent behavior when the system is run, and disappears after that. Attempts to derive them from any process other than running them are only approximate.

In general terms, keeping track of what the model exists in the productions is an agent tracking problem (Ward, 1991). In his thesis, Ward proposed that the agent tracking system would predict the agent's behavior by applying the same knowledge as the agent being modeled has. In Ward's domain of intelligent tutoring systems, this approach performed well, perhaps because the expert knowledge (the presumed knowledge of the agent) was fixed, and the model was complete or nearly so.

In general, there appear to be problems with this approach. The agent tracking system cannot examine all the possible behaviors currently available in the model because they have to be evoked within a functional model. The agent tracking methodology suggested by Ward provides a mechanism to evoke them, but the agent tracker is not guaranteed to find all of the model's behaviors. Near misses, where subject actions were close to being evoked in the model but were not, are particularly important to find because they represent the parts of the model that will need to be modified. Also, when the agent tracker does not predict the agents behavior correctly, only a locally driven backtracking mechanism is available to resynchronize the agent tracker. The Ward system was based solely on external actions required to perform the task matched to the model's external task actions. The match of the predictions to the data was one-to-one and onto, which is not often available, certainly not when verbal protocols are matched to internal states, or when the task allows multiple ways to perform a task.

This same problem has been seen by developers of expert systems (e.g., Brueker & Wielinga, 1989). The expert cannot elicit all of their behaviors upon demand. Creating a model of an expert involves some direct elicitation, augmented through observing them perform a variety of tasks.

The problems extending the agent tracking approach may be indicative of a general problem with functional models. Their knowledge used to perform a task is not inspectable until it is brought to bear to perform the task. When all of their knowledge is made examinable, it can no longer be used to perform the task because it is no longer directly in the architecture's terms, but in another language such as English — the classic declarative/procedural tradeoff. This provides an explanation for why external predictions and modifications of behavior will often be better than the system can perform on its own, and that in order to use functional models the user will need a non-functional, declarative model of them.

Figure 3-15 presents a schematic of an intelligent protocol analyzer. This indicates that if we want information about the model available in complete form and without running the model, we have to have a system to watch, cumulate the model's response and action history and summarize it. This information could be summarized by a separate, more declarative Soar model itself, or more simply, by a bookkeeping system just for keeping track of what problem spaces, operators (and so on) have appeared. The result, a list of operators applied perhaps, will itself also be a model. It still must get run (queried) to produce some output, only the necessary inferences are available much faster and in more complete form, and it may be a simpler model, one that is not functional, merely descriptive.

These simpler, descriptive, non-functional models, of what operators are available and what features will be seen in a state, must be ubiquitous in intelligent agents. They are necessary to know which button to press on a VCR, and they are also necessary for understanding other agents, when a full process model is not available, or too expensive to be applied. Normally the models are simpler than

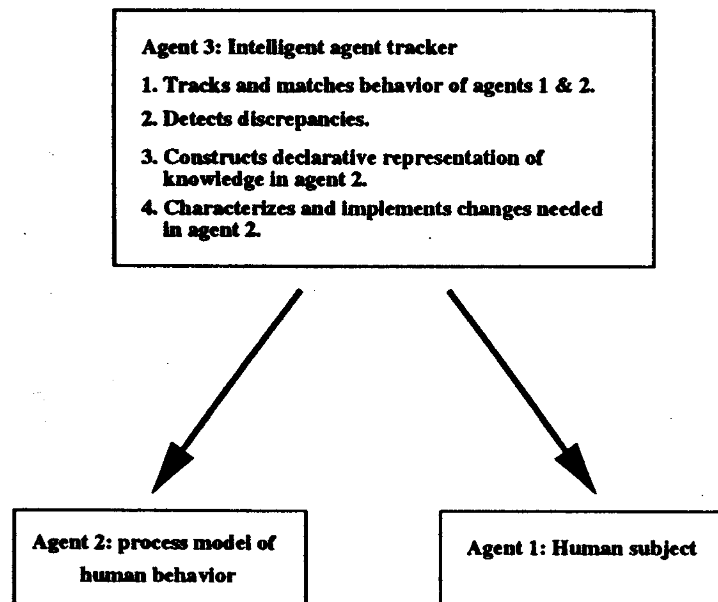


Figure 3-15: Grand design for an intelligently automatic protocol analyzer.

the process being modeled, but that is not always the case. Superstitious behavior is an example where the model includes extra rules that do not exist in the process being modeled, such as having to throw salt if you spill it.

Summarizing empirical support for the model. The declarative version of the procedural model created by running it is also used to summarize support for the procedural model. The individual rules are not the appropriate level. In Soar at least, they are the implementation, not the theory itself. The operators, because they represent objects in the architecture, are one of the appropriate levels. The knowledge level (Newell, 1980b) is probably the best, it represents the results in the most general terms, but it will also be the hardest to find automatically, and since it is presented completely without reference to an architecture, it is completely declarative.

The appropriate terms for accumulating empirical support are based on the architecture. In Soar, the architecture is the Problem Space Computation Model (PSCM) (Newell, Yost, Laird, Rosenbloom, & Altmann, 1990), and the terms will be of problem space level object's (e.g., operators) creation, proposal, selection, and application. These objects, the role they played in the current behavior, and the portions that were observed in the data must be identified by more than just name. It would appear that the most natural representation for this may end up being a production language, but probably a different, more inspectable one than the original model was implemented within.

So to summarize, the agent tracking system that wants to modify its agent model will need an additional, simpler, declarative model of the agent it wants to modify, and the declarative representation must be in terms of the architecture. As a declarative model, it will not be able to perform the task, but describe the procedural model in terms of the architecture. This simpler model can be used to suggest ways to manipulate the full functional model, and aggregates support for the procedural version.

3.3.3 Using the architecture to automate the analysis

An environment to support TBPA will include an architecture to implement the model that is capable of genuinely intelligent behavior. Why not put it to work to help do the analyses? It could learn by watching or be programmed. This thesis is only a small step in defining how model building and testing (in general, agent modeling) could be automated. But by working within a functional, unified architecture, it is possible to spin a story of how it could work all the way down — to completely automatic cognitive modeling.

The first step towards this vision was to define the operations required to do this task, which this chapter proposes to have done for TBPA. Next, the knowledge to perform this task must be put into an intelligent architecture. Soar could also serve as the architecture, and the knowledge to test and build models could be gathered like that for any cognitive model or expert system. Finally, the analyses environment should be made available to the architecture and the modeling knowledge. Soar learns. So perhaps an easier way to automate this task is through learning, by watching a series of analyses. As the more automatic Soar/MT watched a series of routine analyses over similar episodes be performed, it could follow along, learning how to run the analyses, and then driving the analysis programs itself (Newell & Steier, 1992).

This approach appears to be close, the next thesis, or at least then the one after that — the basic requirements have been identified, some of the simpler details have been automated, and a prototype of the environment that must be manipulated is in place. The most immediate step towards this would be to get a model to perform some of the simpler subtasks.

3.4 Summary of requirements and description of the environment's design

All the requirements for an environment to assist in testing process models are collected into Figure 3-16. These requirements are incorporated in the design of the Soar/MT environment, Model Testing capabilities in Soar. Figure 3-16 shows a schematic of the three major subcomponents of Soar/MT and the systems they are built on. Soar/MT consists of systems for testing the predictions with data, interpreting the correspondences with respect to the model, and manipulating the model. In addition to the overview provided here, each of the three main tools are described separately in the next three chapters.

Interpretation and alignment of the data with respect to the predictions. Spa-mode provides a structured editor within GNU-Emacs for editing protocols and their correspondence with a running Soar model of the task. It is based on the Dismal spreadsheet (Ritter & Fox, 1992), which provides a tabular display, and direct support for manipulating the two information streams (the trace and the protocol) as separate sets of columns. The basic spreadsheet has been extended to incorporate the ability to semi-automatically align the predictions with the data based on a simple regular expression equivalence parser. Additional commands let the analyst change and correct this simple alignment by hand.

Graphic display of the comparison. Two graphic displays of the comparison can be created automatically from the alignment data. The first display, based on Peck and John's (1992) model support graph, displays the data with respect to the model predictions that are matched. The second, new display graphically presents the relationship between the predictions and the data with respect to time. Signature phenomena of various ways the predictions can mismatch the data have been identified. They can be used as direct indicators of how to improve the model.

These displays are only examples of the types of displays that can be used to highlight how the model's predictions mismatch the data, and where it can be improved. A structured editor (S-mode) is provided to assist in creating additional graphs in S, the underlying graphic language. The S-mode editor is now joint work (Bates, Kademan, & Ritter, 1990).

Supported by	Requirements for the process model's trace.
New trace	(a) Include:
	(i) Unambiguous predictions for each subject information stream (external and internal actions), and
	(ii) Time stamps for each action.
New trace	(b) Be readable by humans.
New & old trace	(c) Provide various levels of detail.
SX graphic display	(d) Provide aggregate measures of performance.
na	(e) Be deterministic even if the model is not.
	Requirements for directly testing the model's predictions with protocol data.
Spa-mode	(a) Interpret and align the data with respect to the model's predictions.
Spa-mode	(b) Display and edit the protocol, predictions, and environment response streams and the correspondences.
	Requirements for analyzing the comparison of the data with the model's predictions.
Spa-mode/Fit graphs	(a) Show where the data does not match the predictions.
Fit graphs	(b) Aggregate the results of the comparison in terms of the model.
Spa-mode/Fit graphs	(c) Interpret the test results as clues for modifying the model.
	Requirements for modifying the model.
SX graphic display & Model fit graph	(a) Display the model so it can be understood.
DSI	(b) Modify the model based on comparison.
All parts of the Soar/MT environment	**Requirements based on integrating the steps and supporting TBPA with a computational environment.**
	(a) Provide interchangable representations and functionality based on the model being tested.
	(b) The environment must automate what it can.
	To support the user for the rest of the task:
	(c) Provide a uniform interface including a path to expertise.
	(d) Provide general tools and a macro language.
	(e) Provide tools for displaying and manipulating large amounts of data.

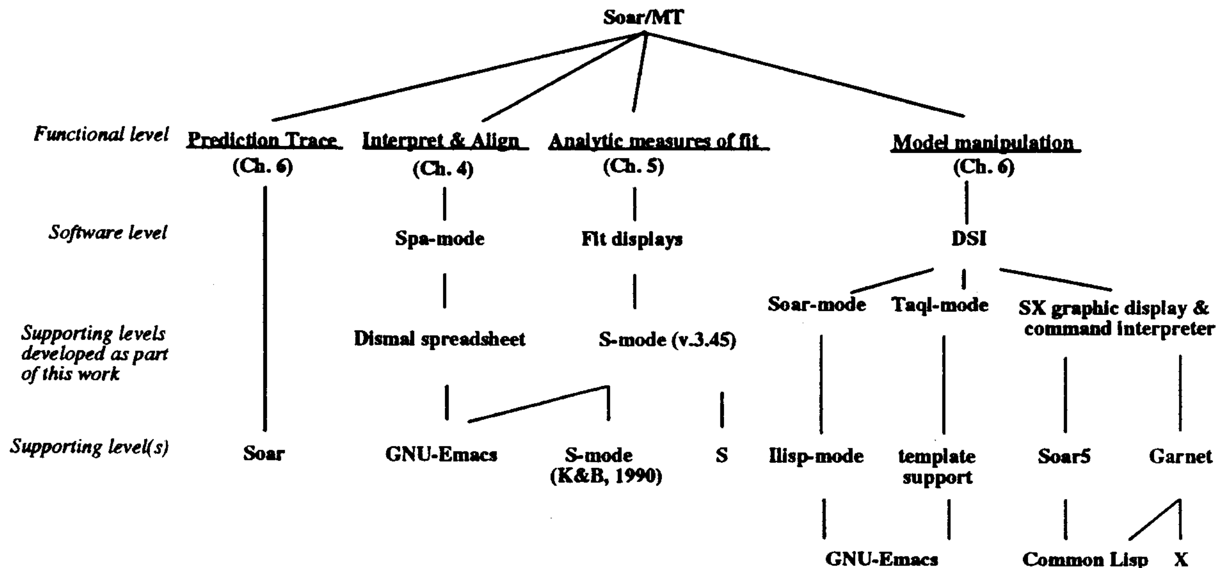


Figure 3-16: Requirements for an environment for testing process models and overview of the Soar/MT environment to support these requirements.

<u>Model tracing and manipulation.</u> The Developmental Soar Interface (DSI)[2] provides an interactive graphic and textual interface for running and manipulating process models in Soar. The DSI consists of three integrated yet independent pieces of software. They are designed to provide multiple entry points for users so that they may manipulate and examine Soar models in a natural and consistent way. For example, while examining the graphic display of Soar's working memory, users can run the model ahead a simulation cycle by typing a single character on the graphic display, and while editing the model in the accompanying editor, they can run the model though similar editor commands.

The largest module is the Soar in X (SX) graphic display. By displaying and storing the problem space organization over time, it adds to Soar in a real way the concepts of problem space level statistics, macrocycles, and user specified hooks. Problem space level objects and their working memory components can be examined by clicking on them. The models effect on working memory can be monitored in examination windows. An associated command interpreter and pop-up menu provide keystroke and keyword commands to manipulate Soar. The SX display also includes a modified trace designed for automatic interpretation and alignment with data.

The second module is Soar-mode, a structured editor and debugger written within GNU-Emacs, the latest version of the Emacs editor (Free Software Foundation, 1988; Stallman, 1984). It provides an integrated structured editor for editing, running, and debugging Soar on the production level. Descriptions of the productions that are firing or are going to fire can be automatically displayed. It includes for the first time complete on-line documentation on Soar and a simple browser to access it.

The third module, TAQL-mode, is a structured editor for editing and debugging programs written in the TAQL macro language. By providing TAQL constructs as templates to complete rather than syntactic structures to be recalled it decreases syntactic and semantic errors. After inserting templates users can complete them in a flexible manner by filling them in completely or only partially, escaping to the resident editor to work on something else or to edit them more directly. This leaves general editor commands available throughout the editing session. At any point in the process users can complete any partial expansions or add additional top level clauses, choosing from a menu appropriate to the construct being modified.

<u>A shared design.</u> The requirements based on providing a computational environment to support TBPA require a shared design. Some must be met by the environment as a whole and require a uniform design (a - interaction between the environment's components, and c - a consistent interface). The others end up being met in each tool individually and in different ways (b - automation, d - general tools, and e - support for dealing with lots of data).

The first requirement influencing the design is that the environment must be integrated — the tools must work together, sharing common data structures, and the analyst must be able to switch between them as needed. The glue that will bind them together will be that their data structures will all reference the same Soar model. Their communication link-ups will be straightforward because they will all be processes in Emacs. The tools can use these two features to pass data structures between themselves, and to let these functionalities incorporated in each tool be called by the other tools.

The second requirement, that of automating analyses, is supported in different ways in each tool. The interpretation and alignment tool includes a simple interpretation and alignment algorithm. The system charged with explaining the comparison creates graphs automatically from the alignment data. The model manipulation tool supports the user in keeping track of files and in loading and running the model.

To support the third requirement of providing a uniform, learnable interface, each tool is designed to be driven by similar menus, use similar keybindings, and have a consistent style. The tools that make up Soar/MT all provide a similar path to expertise by including the features presented in Table 3-15.

---

[2]Pronounced Dee Ess Eye.

Novices and casual users can start out using a menu to perform the analyses, and graduate to using the keystroke accelerators provided on the menus. When they need additional information, on-line help is available for each command, as well as a complete manual that is available on-line or as hardcopy.

Table 3-15: The features that all parts of the Soar/MT environment share as aids for ease of use and learnability.

(a) Menus to drive the interface.

(b) Keystroke accelerators available and automatically placed on menus.

(c) Help provided for each command on request.

(d) Hardcopy manuals also available on-line through the menu.

To support the fourth requirement of providing general tools, each system provides general building blocks, and includes an open software architecture that can be used to create macros and new analyses.

The final requirement of providing tools for displaying and manipulating large amounts of data is supported in the environment through a combination of approaches based on graphic and textual displays. Diagrams are taken as the basic building block because they offer the ability to explicitly and directly display large amounts of data (Tufte, 1990). They also provide the ability to group related information, and to perceptually support necessary inferences and operations in performing a task, which are effort-full and require additional time when the information is presented as text (Larkin & Simon, 1987). Text is also used where there are not perceptually operators and representations available, or the information consists primarily of propositions (and one must remember text can be considered as graphics, particularly when it is manipulated with a mouse). This means choosing graphics for information that is well represented graphicly, and text for information that is well represented textually. Both will be useful.

An organization scheme that will be used to display large amounts of data is to provide a recursive interactive display. High level structures (top-nodes) are directly displayed. They can be unpacked interactively and recursively to get details upon request. The display must display all levels well so that the structure is apparent. By providing the ability to let users directly manipulate the data points and model bits, the displays can also serve as an interface to the data. This is the central feature of outline processors.

The relevant theoretical objects will also be directly manipulable. The environment will allow the analyst to directly manipulate and examine the appropriate objects on the theoretical level. In each step of the analysis they are different, in the trace they are the model's actions, in the match and analyses they are the correspondences, and when modifying the Soar model they are productions and problem space level objects.

II Supporting the TBPA methodology: A description of the Soar/MT environment

Chapter 4
A spreadsheet for comparing the model's predictions with the data

Spa-mode is a spreadsheet facility to support the model-testing requirements related to interpreting and aligning a process model's sequential predictions with protocol data. These specific requirements are shown in Table 4-16 along with the global requirements of integration, automation, and support for non-automated tasks that must also be satisfied. Some of Spa-mode's initial design and iconography comes from Trace&Transcribe (John, 1990), but it goes significantly further in its representations and in the tools it provides for interpreting and aligning the transcribed protocol data with respect to the predictions. Spa-mode provides a simple alignment algorithm that can automatically align trace-elements with unambiguous protocol segments. There are several other commands that can be used to clean up the alignment, and to align the model's predictions with less clear protocols (e.g., verbal protocols). Spa-mode also includes the ability to code protocols with operator names taken directly from a running Soar model, a loaded TAQL program, or from a saved file of previously used operator names.

Spa-mode is built on the Dismal spreadsheet (Ritter & Fox, 1992), which is implemented as a set of extensions (a major-mode in Emacs terms) to the GNU-Emacs editor (Free Software Foundation, 1988). Dismal includes most of the major functions that one now expects from a spreadsheet, such as (a) the addition, deletion, clearing, and yanking of cells, rows, columns, and range; (b) formula entry and evaluation; (c) movement within the spreadsheet with keystrokes and mouse movements; and (d) the ability to format each cell's display. The difference is that Dismal lives within the GNU-Emacs environment. The GNU-Emacs environment provides the ability to cut and paste between files, a complete text editor, a language for writing user functions (GNU Emacs-Lisp), and complete on-line help.

Table 4-16: Requirements supported by Spa-mode

Requirements for using the model's predictions to interpret the data
(a) Display and support editing the correspondences.
(b) Automatic alignment of unambiguous actions.
(c) Support matching ambiguous actions.
Requirements for analyzing the comparison of the data with the model's predictions
(a) Show where the data does not match the predictions.
Requirements based on integrating the steps and supporting TBPA with a computational environment
(a) Provide consistent representations and functionality based on the architecture.
(b) The environment must automate what it can.
To support the user for the rest of the task:
(c) Provide a uniform interface including a path to expertise.
(d) Provide general tools and a macro language.
(e) Provide tools for displaying and manipulating large amounts of data.

The largest need addressed by Spa-mode is to interpret and align the protocol data with respect to the model's sequential predictions. Aligning predictions to data by hand is tedious and error prone, so some assistance is warranted for unambiguous data such as motor actions. Providing alignment automatically and completely is beyond current natural language parsing technology. However, other data (e.g., mouse clicks) are discrete and relatively unambiguous. These data can be automatically aligned with process models' predictions.

Providing assistance through automatic alignment of discrete protocols along with a semi-automatic tool for aligning verbal protocols appears to be a good compromise. There are several data streams that would be well suited for this. The model's overt motor interactions with the world can be directly aligned to overt world interactions of the subject, as well as codes built from verbal utterances to the model's operations or states. Verbal utterances may be partially aligned to model operations, and their

```
T  Mouse actions  Window actions  Verbal  ST # Mtype MDC DC                       Soar trace
-+--------------+---------------+--------+-+-+-----+---+--+------------------------------------------+
 0                              I believe  v 1 short
                                                       0 G: g1
                                                       1 P: p4 (top-space)
                                                       2 S: s5
                                                       3 O: browse ()
                                                       4 =>G: g19 (operator no-change)
                                                       5   P: p26 (browsing)
                                                       6   S: s39 ((unknown) (unknown))
                                                       7   O: find-appropriate-help
                                                       8   =>G: g43 (operator no-change)
                                                       9     P: p50 (find-appropriate-help
                                                      10     S: s59 ((unknown) (unknown))
                                                      11     O: define-search-criterion
                                                      12     =>G: g65 (operator no-change)
                                                      13       P: p72 (define-search-criterion)
                                                      14       S: s79 ((unknown))
 6                             write       v 2     v 15 15       O: generate-search-criterion ((write))
 9                             write       v 3     v 15
13                             write       v 4     v 15
  M(+x) (R of prog win)                            B4
              mouse line to pointer
                                                      16       O: evaluate-search-criterion
                                                      17     O: define-evaluation-criterion
                                                      18     =>G: g103 (operator no-change)
--**-emacs[SHAMO.SOAR]: example-types.spa   A36 ManUp <H]  (SPA) ----Top----------------------------
```

Figure 4-17:
Example display of a model trace aligned with data (taken from the Write episode of Browser-Soar). Left-hand columns "T" (time of subject's actions) through "MDC" (matched decision cycle) are one meta-column, and columns "DC" and "Soar trace" on the right are another meta-column. The right-most simple column of the left meta-column (in this case the H column) is indicated to uses in the editor's mode line (the bottom line of the figure) as "<H]".

alignment will often be constrained by the less ambiguous data streams.

4.1 Displaying and editing the correspondences

Most importantly, the analyst needs to view the correspondences, and annotate and edit them as appropriate. Providing all the capabilities normally associated with a spreadsheet takes care of most of these requirements. This includes the ability to resegment by adding additional cells and breaking a segment into several segments. This set of capabilities provides the bookkeeping abilities mentioned in the review.

Spa-mode uses a tabular display, noted as necessary in the review. The tabular display helps the analyst see how to line up the predictions and to understand the alignment by providing the context of each match, along with a visual operation (scanning a row) to identify the prediction and data that are paired. The tabular display also shows how much of the model is matched and unmatched, and it starts to show patterns in the alignment by sheer ink usage. This tabular display, with the field names shown as column headings, also allows more data (context) to be displayed on the screen. Typical users can now see up to 60 lines of the comparison. The tabular display reflects the underlying matrix organization of the data into rows of segments that each include several fields displayed as columns. Automatic alignment programs and semi-automatic tools have a uniform and appropriate data structure holding the segments and protocols to manipulate.

Spa-mode adds to Dismal the idea of meta-columns. During alignment operations there are two sets of columns that need to stay aligned with respect to each other. A meta-column ties these columns together so that cells within a meta-column remain aligned. This is necessary when the model's predictions or the data span more than one simple spreadsheet column. For example, most models will use two columns to hold the model's predictions and their simulation cycle. These two simple columns would be placed in a meta-column. In the example analysis in this thesis the columns on the

right are the data columns, and the columns on the left are associated with the trace, but this need not be the case.

Making the most of the visual space. In addition to using a tabular display, clever screen design within the spreadsheet can take further advantage of the tabular display to make more of the visual space. Additional information can be presented through overlapping use of columns (i.e., since the mouse movement and verbal utterance columns will never both be filled within the same segment (row), they can be placed so that long values of each run over into the other). Keeping all columns flush right saves adding an extra blank column between filled columns from running together. Space can also be used more efficiently by removing leading digits on numbers[3], removing extraneous whitespace in strings, and abbreviating mouse movement codes. Taken together, these improvements allow approximately 40% more information to be displayed on the spreadsheets used in Chapter 7 than an initial design based on the Excel versions used by Peck and John (1992).

The types of alignments. Once the predictions have been aligned with the data, either automatically, semi-automatically, or completely by hand, they must be presented in an interpretable manner. Figure 4-17 shows an Spa-mode display with fictitious data and model predictions. The simple columns A through H are a meta-column of subject data, and the simple columns I and J are another meta-column used to represent the model's predictions. In this display, lines 0 through 10 are used as a header, but like any other spreadsheet, these rows are not fixed as header rows. Line 11 holds a ruler, which names the columns. This too can be adjusted to any row, or omitted. When the user scrolls, the contents of the ruler row are redrawn as the top line of the display.

This example includes all of the ways Spa-mode can display how the model's predictions are matched by the data. The display should be capable of representing the types of correspondences noted in Table 2-5, and we will find that it can represent all but one. The way Spa-mode represents each type of match is noted in Figure 4-17.

There is one type of correspondence mentioned in Table 2-5 that cannot currently be represented in Spa-mode, a subject action that is matched by multiple model actions. The current representation assumes that what matches a given data segment is a single action of the model. This lack of representation serves as a useful constraint — segments that match more than one model action probably are not segments. Either the model is more fine grained than the data, or the segment should be considered for resegmenting.

Simple measures of fit and simple analyses. Another requirement Spa-mode starts to address is interpreting the alignment. This includes summary statistics of the comparisons, the time course of the match, and the ability to display the types of matches previously noted in Table 2-5. (A more global, model-based view will also be necessary, and this is covered in the next chapter.)

Formulas in Spa-mode can directly support some low level analyses of the comparison, such as counting the matches and numbers of operators matched. All of the simple measures presented in Section 2.4.4, such as *goodness = hits - false-alarms*, are directly supported. Additional measures, such as the number of model actions matched and number of words in a protocol have been added as Formulas or special functions as well.

A data-base facility (somewhat similar to the database facilities in Excel) comes with GNU-Emacs. *list-matching-lines* (a function) shows in a buffer all segments that match a given regular expression. A specifiable number of lines surrounding each matched line can also be included. Once the contents of an Spa-mode buffer are copied into a scratch buffer, there are additional functions provided within GNU-Emacs for manipulating the resulting buffer, such as *delete-non-matching-lines*, which deletes all lines except those containing matches for a regular expression, and *delete-matching-lines*, which

[3]This suggests that a currently unavailable but useful format for number series in spreadsheets where the leading digits are repeated would be to present just their trailing digits.

```
   A     B                C            D   E F  G    H  I                    J                             K
 +--+------------+----------------+-------+-+-+-----+---+--+-------------------------------------------+-----------+
 0 Example spa-mode file prepared for the thesis.
 1 Last edited    14-Oct-92
 2
 3 T is timestamp of action in s.
 4 MOUSE EVENTS is the user's mouse movements. MTYPE is type of match
 5 WINDOW EVENTS are responses from the system.MDC is matched DC.
 6 VERBAL is verbal protocols.                 DC is decision cycle in Soar model.
 7 ST is Segment Type                          SOAR TRACE is the literal Soar Trace
 8 # is segment number
 9                             [This is a fabricated example trace and behavior.]
10
11 T  MOUSE EVENTS  WINDOW EVENTS  VERBAL  ST # MTYPE MDC DC                  MODEL TRACE
12 0                             I believe v 1 short                                                     Uncodable S
13
14 6                                 write v 2     v  15 15  .  O: generate-search-criterion ((write)) Multiple hit
15 9                                 write v 3     v  15                                               Multiple hit
16 13                                write v 4     v  15                                               Multiple hit
17
18                                                       45  .  O: change-search-criterion ((draw))
19
20                                                       56  .  .  O: scroll (keyword)
21                                                       57  .  .  =>G: g451 (operator no-change)
22                                                       58  .  .  . P: p458 (mac-scroll-method)
23                                                       59  .  .  . S: s467 ((to-be-found write))
24 17 M(+x) to (keyword dn arrow)        mm 7     mr  60 60  .  .  . O: move-mouse (keyword down)      Hit
25
26 17 C            keyword menu scrolls    mb 8    muc                                                 Miss
27
28                                                fa     71  .  .  . O: move-mouse (keyword down)      False alarm
29
30 21          perhaps I should draw instead v 9   v  45                                               Crossed
31
32                                                       83  .  .  O: change-current-window ()        Uncodable
33                                                       84  .  .  =>G: g696 (operator no-change)        model
34                                                       85  .  .  . P: p703 (change-window)            actions
35                                                       86  .  .  . S: s711 ((to-be-found write)
36                                                       87  .  .  . O: scroll (keyword)
37                                                       88  .  .  . =>G: g724 (operator no-change)
38
39 --**-emacs[SHAMO.SOAR]: example-types.spa    A36 ManUp  <H]  (SPA) ----Top---------------
```

- An uncodable subject action, one that cannot be interpreted with respect to the model, is shown on line 12 (as numbered on the left-hand side) — a verbal utterance that is too short to code. It is indicated by the lack of a matching model trace and the code "short" in the match type (MTYPE) column.
- Uncoded model actions are shown in lines 18, 20 to 23, and 32 to 37. They are indicated by model traces without corresponding subject actions.
- Hits are shown in lines 14, 24, and 30. The match is indicated in columns E through H. Column E notes the type of the match of the segment (ST), which in this case is mouse movement, or mm. The type of correspondence (column G, MTYPE) is of a matched mouse movement. The simulation cycle that is matched by the mouse movement is shown in Column H as the matched decision cycle (MDC).
- A multiple subject action hit occurs on lines 14 through 16. In this case, the MDC and DC columns are not always the same. The Matched decision cycle column ends up with multiple entries for the decision cycle 15.
- A miss is shown on line 26. The subject has clicked the mouse, and there is no corresponding action in the model's trace.
- A false alarm is shown on line 28. In this case, the model has performed an overt action that has to be coded, but there is no corresponding subject action.
- A pair of actions that are crossed in time is shown in Lines 30 and 24. The corresponding behaviors cannot be directly aligned while keeping them both in order, so the matched decision cycle column is used as a reference for the last subject action matched.

Figure 4-18: Types of correspondences that can be represented in Spa-mode.

does the opposite.

Taken alone, the spreadsheet approach only starts to provide a vehicle for understanding the comparison between the model's predictions and the data. More global, even more informationally dense model based displays will be necessary to understand the comparison at a higher level. Potential solutions to this requirement, diagrams and other measures, are presented in the next chapter.

4.2 Automatically aligning unambiguous segments

In order to align the two meta-columns, the user specifies which tokens in each information stream match through a list of regular expressions. For example, in the data shown in Figure 4-17, in the transcribed mouse actions "^C$" matches in the Soar trace "O: click-button". ("^C$" is a beginning of string (^), a "C", and an end of string ($).) The trace matching pattern could also have the beginning and end marked, but it is not necessary. The alignment algorithm is then called either directly as a command, or from a menu, and an initial match is computed. The alignment algorithm then displays the matches one-by-one to the user for verification. After viewing each proposed match, the user can accept it, decline it, or escape to the next step by accepting all the remaining matches. After the set of matches has been approved, the alignments are passed off to a program to actually align the meta-columns in the spreadsheet.

The algorithm used by Spa-mode to compute the alignment is based on the Card1 algorithm (Card, Moran, & Newell, 1983, Appendix to Chapter 5) explained in Chapter 2, and presented in more detail in that chapter's appendix. Card1 is a straightforward implementation to solve the maximum common subsequence problem (Hirschberg, 1975; Wagner & Fisher, 1974), except the output will not just be the maximum length, but the actual matched subsequences that will be used to align the two meta columns kept in the spreadsheet. Because the two information streams may use different tokens (sometimes even verbal utterances), the extensions must include the ability to specify what constitutes a match. The extensions to the algorithm presented here are labeled as the Card2 algorithm.

The alignment starts at the first prediction and subject action matched, not at the first action in either information stream. The relationships of unmatched items at the beginning or end cannot be specified because they are not aligned. Various later displays and analyses have to adjust their analyses not from the first time stamp, but from the first matched time stamp.

The Card2 algorithm. The output of the Card1 algorithm suffers from a simple error that can be simply fixed. The small error is that the sequence Card1 returns is reversed. This is not normally a problem. However, Spa-mode uses the returned sequence (and its internal references to the original sequences) to align a protocol and the predictions. This is easily remedied by changing the pointers used to create the list. Figure 4-19 shows the first improvement to the algorithm, now called Card2.

Potential problems avoided: Multiple possible match sets. There may be several possible "best" alignments. Since we are interested not just in how much could be aligned, but in using the alignment to understand how it could be improved, which possible match set gets chosen is a real concern. Card2 satisfies the requirement of starting the match at the front by returning the subsequence that starts closest to the front of the two input sequences. That is, if sequence (a) is APA, and sequence (b) is A, then the common subsequence that is returned matches the first A. As a longer exemplar, consider aligning the two simple strings DUC and DUDUDU. Card2 would return an edit list (referenced by position) that would call for aligning the first D's together. This results in aligning the initial predictions, which is a more stable alignment if changes to the strings tend to come at the rear: If additional D tokens were later added to the shorter list, the alignment will change less.

Incorporating additional constraints in the Card2 algorithm. Finally, we find in some data, that while we would like the match to favor the front in general, that there may be additional conditions on the match, or on the choice of which possible maximum common subsequence we prefer. For example, when analyzing the Browser-Soar data covered in Chapter 7, we prefer to have the last possible item in

```
Modified version of algorithm on P. 191, Card, Moran & Newell (1983):
New elements are now inserted at the end of the output array rather
than the beginning.

i <- predlength; j <- obslength;
k <- predlength + obslength - Score[PredLength, ObsLength]       ; K set to last
                                                                   ; element
until (i = 0) and (j = 0) do
  if (i<>0 and (j=0 or (score[i-1, j] > score[i-1, j-1])))       ; NB. missing )
                                                                   ; in original
  then
        PredSeqResult[k] <- PredSeq[i]
        ObsSeqResult[k] <- nil
        k <- k - 1; i <- i - 1;
  elseif (j<>0 and (i=0 or (score[i, j-1] > score[i-1, j-1])))   ; NB. missing )
                                                                   ; in original
        PredSeqResult[k] <- nil
        ObsSeqResult[k] <- ObsSeq[j]
        k <- k - 1; j <- j - 1;
  else
        ObsSeqResult[k] <- ObsSeq[j]
        PredSeqResult[k] <- PredSeq[i]
        k <- k - 1; i <- i - 1; j <- j - 1;
```

Figure 4-19: A simple fix is applied to the original Card1 algorithm to return the edit sequences in the correct order without explicitly reversing the returned list.

a series of equal tokens to be used in the alignment. In that case, the earlier M's (mouse movements) represent mistaken applications of the M (mouse move) operator, while the final one more closely matches the subject's behavior. For example, in matching XMDU to YMMMDU, Card2 would provide the sequence:

```
XM--DU
YMMMDU
```

One might prefer (and in analyzing Browser-Soar we do prefer) that we get the following match:

```
X--MDU
YMMMDU
```

Doing this directly within Card2 would require a relatively large extension to the algorithm. The structure of the current algorithm would require either look-ahead search and access to the sequence values or else a new data structure.

We are able to remedy this with a clever manipulation directly based on manipulating the edit list returned by the Card2 algorithm. Instead of taking the items on the list as absolute, a separate function applied after the initial alignment extracts the value in the cell to be edited, and searches between the two edit references on either side for the last occurrence of the matched item's value before another match. If there are multiple identical items in a row, we can find the last one this way and use it in the alignment list instead.

Possible extensions to Card2 and other matching algorithms. Implementing the interpretation and alignment algorithm with a dynamic programming approach would allow more flexibility by separating the system into a generator and tester that would be easier to modify. The choice of objects to align could be modified to prefer to match more salient events first, or to avoid more than a given

amount of offset between the two information streams, or to give up if that limit was exceeded. It might provide a more natural algorithm to modify to prefer to match certain tokens next to each other if possible, and so on. Finally, one might actually prefer a locally driven match, where the system offers the user the choices incrementally. Incorporating this type of match should be explored in future work.

More complicated token comparison processes. In the original maximum common subsequence problem, the comparison between the two information streams is that of unity: are the tokens the same tokens? In the applications in which Spa-mode will be used, the information streams will often have equivalent tokens, but they will perhaps go by different names in each information stream. The mouse-move operator may be called just that, but the transcription system may note the user's movement as "M(+x23-y23)" to indicate a mouse movement down and to the right. The comparison step in the Card algorithm has been modified to do a more flexible comparison based on pairs of regular expressions, such as "mouse-move" matches "M(*)", where * represents "matches anything". The basic functionality of matching regular expressions was available directly from GNU-Emacs. With regular expressions, the analyst can also start to compare in a simple way keywords expressions with verbal utterances. The matching process also serves as the location to incorporate a more robust natural language matching process.

More ambitious functions to interpret the data using the model's predictions can be imagined. The model's predictions represent the actions and knowledge structures that one expects to find in the subject's actions and verbal utterances. Because there already exists a strong model of what will be said and done, the knowledge structures for a general parser are available — this is a much more restricted case of natural language parsing than those that start out with no preconceptions of what will be said and attempt to parse one sentence. The point of this work was just to get to this point. It will be the pleasure of the next project to take advantage of this possibility.

The scope of the natural language parsing problem makes it clear that a simple keyword parser will not be adequate to interpret the data with respect to the predictions completely automatically even in this restricted form. However, the parser need not be perfect in this application to make it worthwhile. If the parse is only approximate, as long as it does not go completely off track, it will make the testing more routine and repeatable.

4.3 Interpreting ambiguous actions

Additional capabilities are needed when the results of the automatic alignment tool need to be cleaned up, or the model's predictions have to be aligned with ambiguous protocols. In addition to the standard capabilities of a spreadsheet, the analyst will also need to manipulate the two meta-columns. Spa-mode provides two functions for aligning the meta-columns by hand. The most powerful function aligns the line the cursor is on with the line that is marked.[4] The analyst can also insert blank rows into each meta-column individually. This allows offsetting meta-columns while maintaining alignment within each meta-column.

Simple coding of the protocol. The review noted that most tools for manipulating protocols include the ability to assign codes to protocol segments as a step in model formation. Spa-mode includes this ability as well. Through a menu or keystroke command, a segment can be assigned one of a set of preselected codes. These codes can be entered by the user, taken directly from a loaded, but perhaps incomplete, Soar model (using a variety of tools in the graphic display), a loaded TAQL program (using its inherent facilities), or from a saved file of previously used operator names. The set can be accumulated from several of these sources and saved out for later use. A protocol interpreted this way

---

[4]Mark is an Emacs term for a previous selected location. The user can also toggle between the two locations, *point*, the location where the cursor is now, and *mark*, a previously selected location, with a command bound to a keystroke called exchange-point-and-mark.

can be analyzed like any other, including the ability to count matched segments and to perform the analyses presented in later sections and chapters.

4.4 Supporting the global requirements

In addition to the direct requirements of aligning the predictions with the data and starting to interpret their comparison, there are five requirements that Spa-mode must also support that arise from integrating the steps of TBPA and supporting them with a computer environment.

4.4.1 Providing an integrated system

Spa-mode supports multiple entry points to its own functions and to those of the other modules within the Soar/MT environment. Users can interact with the spreadsheet through keystrokes commands, the menu, and with the mouse. Users can also manipulate the model while in the environment. Through the menu or through keystroke commands users can run the model while viewing the correspondences in the alignment, jump to the running model, and cut and paste its trace directly into the spreadsheet. The segments can be coded with operator names taken directly from the model. Once the data has been interpreted and aligned, the correspondences can be used to create displays within S and S-mode, as discussed in Chapter 5 on the analytical measures of model fit.

The source code for all these systems is publicly available and runs within the same environment as Soar, Soar-mode, TAQL-mode, and the graphic display functions, so further integration would be straightforward.

4.4.2 Automating what it can

Spa-mode begins to automate the interpretation of subject's actions with respect to the model's predictions through the Card2 alignment algorithm. In addition to this, there are several minor functions for which Spa-mode provides automation. These include the ability to renumber segments, and to count the total number of words in a range and just those that match a regular expression.

4.4.3 Providing a uniform interface including a path to expertise

By keeping its design similar to popular spreadsheets, Spa-mode is able to take advantage of users' familiarity with spreadsheets in general. In addition to this inherent aid to being usable, a path to expertise is provided that is uniformly applied to each part of the Soar/MT environment. Table 3-15 displays the features that all parts of the Soar/MT environment share as aids for ease of use and learnability.

This approach starts with menus to make information available to recognition memory. Users can query the menus (by typing a "?" or a space) to display the available keystroke accelerators that are automatically placed on the menus' help displays. Exploration is further supported through help on individual functions, and on-line copies of manuals available through the menu.

State information is displayed to the user. This is not complete, and there is much more state information than can be displayed, but a conscious effort has been applied to display more than has been done in the past. The bottom line of Figure 4-17 includes the following mode line that accompanies each Spa-mode file.

```
--**-emacs[SHAMO.SOAR]: example-types.spa    A36 ManUp  <H]  (SPA) ----Top-------------
```

The leading two asterisks indicate that the buffer includes changes that have not been saved. "emacs[SHAMO.SOAR]:" indicates which computer Spa-mode is currently running on. The file name comes next. A36 is the current cell, and ManUp indicates that updates of changed cells are now only done upon the users request. "<H]" indicates that the H column is the last column of the first meta-

column. SPA indicates that an actual Spa-mode buffer is being examined, and not, for example, a dump of its text into a plain text file. Finally, "Top" indicates that the cursor is at the top of the buffer.

4.4.4 Providing general tools and a macro language

As a spreadsheet, Spa-mode is a general tool. The layout of the information streams and their components are reconfigurable by adjusting the location, width, and alignment of columns. The ability to put formulas into cells allows many analyses to be performed directly. Spa-mode exists within GNU-Emacs, and has access to simple commands to create temporary keyboard macros, and has direct access to the more powerful GNU-Emacs lisp, which Spa-mode itself is written in. This Lisp can be run interpreted or compiled, and the source code for Spa-mode is available for modification or interaction. Others have found developing extensions and macros straightforward.

Hooks are places to customize a system's behavior by calling a user supplied function at a set point, such as at startup, or after a file has been loaded. The standard set for GNU-Emacs modes have been included with Spa-mode. The user supplied function, if any, is called when Spa-mode is loaded or initialized.

4.4.5 Displaying and manipulating large amounts of data

The tabular display of the information streams, their arrangement side-by-side, and the ability to print out the correspondences, provides an unparalleled ability to view the model's performance with respect to the data it is attempting to model. In order to keep the columns' contents clear, as the user scrolls through the alignment, a ruler displaying the associated labels of column is placed at the top of the display. When the file is printed out, this ruler is placed at the top of each page.

The requirements for manipulating the information are fairly well met by the standard spreadsheet actions that are supported. Cell values are displayed in the spreadsheet. The user can click and point or use keystrokes to move between cells. When the user moves to a cell, the expression generating that value is displayed.

Support direct manipulation of the relevant theoretical objects. In the case of the interpretation and alignment task, the relevant theoretical objects are the actions in the two information streams and their correspondences. The individual actions can be manipulated in a natural way as cells in a spreadsheet. The correspondences are represented less well. Simple correspondences made up of single actions matching other single actions that can be aligned on the same row, are easy to manipulate. Rows are spreadsheet's atomic data type. Finding their component structures is easy, and they are automatically kept aligned through the various transformations that Spa-mode provides. More complicated correspondences, made up either of multiple actions or that are crossed with respect to other actions are harder to interpret and manipulate. In the current implementation they too are kept aligned, but finding their component structure is not automated, and sometimes requires visual search. Part of the problem is that there currently is not a well worked-out representation for these correspondences.

4.5 Summary

Spa-mode supports the initial requirements for testing a model by providing an algorithm for automatically interpreting and aligning protocol data with respect to the model's predictions. When the simple alignment tool fails, which it will, there are additional features that support the analyst in aligning the two sequences by hand. When the process model is still being developed and its trace is not yet available, the analyst can semi-automatically code segments separately using the operator (or state) names that will later appear in the trace.

Spa-mode provides a tabular display of the comparison that supports simple, initial visual inspection of the comparison. A visual pattern for describing each type of comparison is developing, but it is probably not in its final form. The underlying spreadsheet provides formulas for computing simple

aggregate measures on the comparison. As a system, it is becoming more robust daily, and has been used for all analyses and tables in this report.

Comparison with similar tools. Compared with Excel or other commercial tools, Spa-mode includes some new commands not supported in other spreadsheets, such as better movement commands (e.g., to the next filled cell, last filled cell in column), and a (presumably) better macro language (GNU-Elisp Lisp). More importantly, however, Spa-mode is integrated with the rest of the tools for testing process models, and the source code of Spa-mode is available so it can be integrated further. Finally, no extra hardware or software license is required to use it. In every other way, Spa-mode is a weaker spreadsheet; it is slower, less robust, and lacks many of the built-in features of other spreadsheets.

Future work. Spa-mode is adequate for a prototype, but further work will be required before it is of general use. There is a potentially very large user community for it (anyone who uses GNU-Emacs), and a large cadre of developers who might pick it up and improve it.

The missing features related to testing process models includes rather general features, such as the ability to split a buffer in two, better math functions, faster operation, and the ability to cut and paste ranges of cells with the mouse (in addition to being able to perform this with keystroke and menu commands). Spa-mode should also include more functionality specific to testing process models, such as a better way to choose sets of items to align and faster alignment of the cells through faster row and cell insertion and deletion.

Chapter 5
Visual, analytic measures of the predictions' fit to the data

This chapter describes a family of graphic displays for analyzing the interpretation of the data with respect to the model. These displays are designed to support the requirements listed in Table 5-17. While many measures presented in Chapter 2 provide useful starting points for analyses, they fail to be completely adequate for analyzing models and data the size we wish to consider, and they fail to summarize the comparison in terms of the model. Because it can present a large amount of information clearly, a graphical approach is better. Three approaches are used.

First, a version of the operator support display invented by Peck and John (1992) is automated. It shows which model actions were supported by data, and their position within the model. Given the aligned data and model actions in the Spa-mode spreadsheet, this display of the support for each operator can be created automatically by the analyst. The analyst can click on the correspondences in the display to learn more about them.

Second, a graphic display that presents the relative processing rate of the model and the subject is provided. Given the aligned data and model actions in the Spa-mode spreadsheet, it too can be created automatically to show the time course of where the model's predictions and the data do and do not correspond because the two have performed different amounts of processing. Here too the analyst is provided with associated information by clicking on the data points, and can find the actions that take disproportionate time and, presumably, effort.

Finally, an environment is provided to assist in editing and designing additional versions of these displays. More displays will be necessary. What is an appropriate display may vary with the model, and there are many ways for the data to not match the model. Several other approaches can now be imagined for displaying the interpretation of the data with respect to the model.

Table 5-17: Requirements supported by the graphic comparison displays and S-mode.

Requirements for analyzing the comparison of the data with the model's predictions.
- (a) Show where the data does not match the predictions.
- (b) Aggregate the results of the comparison in terms of the model.
- (c) Interpret the test results as clues for modifying the model.

Requirements based on integrating the steps and supporting TBPA with a computational environment.
- (a) Provide consistent representations and functionality based on the architecture.
- (b) The environment must automate what it can.

To support the user for the rest of the task:
- (c) Provide a uniform interface including a path to expertise.
- (d) Provide general tools and a macro language.
- (e) Provide tools for displaying and manipulating large amounts of data.

5.1 Creating the operator support display automatically

The operator support display of Peck and John (1992), previously presented in Figure 2-7, provides a display relating the model's trace to the problem spaces and their hierarchical organization. The indentation of the operators are ordered hierarchically and in order of use during a typical episode. This display, while it was not designed to do so, also supports the requirement to understand the model. The line connecting each trace element begins to show some of the regularities and periodicities in the model. More importantly, however, it shows which model actions are supported by data, and the type of data they are supported by. It also shows where the model's predictions are not

matched by data and data that are not predicted. Note, however, that the sequential order of the data is not shown in this display.

The original version of the operator support display took an 8-hour day to construct by hand (Peck, 1992). Figure 5-20 shows the automated version that can be created in minutes from the alignment data in the alignment spreadsheet. There have been a few additions and deletions between Peck and John's display and this version. The automated version presents the course of behavior from left to right, rather than top to bottom. This lets the operator names that are matched to the data to be presented in a more readable form, with complete names, and there is room to indent them to represent their organization by problem space. The automatically created version, when it is presented on-line, is interactive: the analyst can click on each data point or model action to see the other fields of data (such as the verbal utterance) of the actions making up the correspondence. Peck and John (1992) included a summary of the support for each model action and data segment in a column on the margin; the current version does not yet, but should.

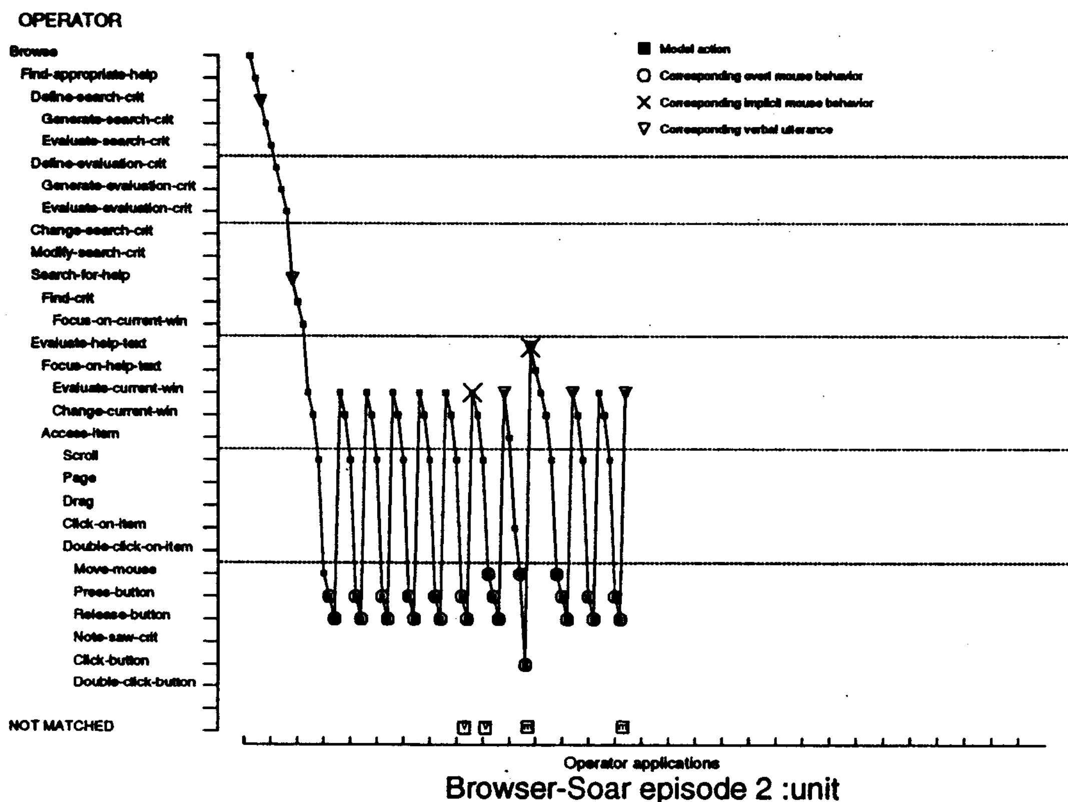


Figure 5-20:
Example operator prediction support display taken from the Unit episode of Browser-Soar. The model's operators are shown on the left-hand side, indented according to their depth in the problem space hierarchy. The connected black squares represent the model's performance. Corresponding data are represented by overlapping symbols. Unmatched data are placed at the bottom of the display as if it matched the *Not matched* operator.

5.2 Understanding the relative processing rate

In order to improve the model the analyst needs to see the global patterns of where the model's predictions do and do not match the data. One way to further understand the sequential predictions of the model would be to view the interpretation of the data with respect to time. A display showing the types of the correspondences (taken from Table 2-5) and the time each action occurred in their respective information streams would emphasize the sequential nature of the model's predictions and the data, the relative processing rate of each, and highlight sections of behavior that could be brought into closer correspondence. This display was initially motivated by the difficulty of understanding the relative rate of actions between the model and subject as they were depicted in the spreadsheet, but seeing that this relationship existed and could perhaps be more clearly displayed.

The order and temporal location of the correspondences between the model's actions and the data can be presented in a display showing their relationship to each other through time. Sakoe and Chiba (1978) first did this for doing speech recognition, matching a model of speech production against the actual recorded speech. Figure 5-21 shows an interpretation of their figure. Their display (and the matching process that generated it) required that each model prediction match a data point, and while it could admit noise, the process did not permit fundamentally wrong actions to be excluded, or an out-of-order match to be included. Cognitive models are usually not yet accurate enough to assume that every action in the model will have a one-to-one match to the data like they assume; multiple subject's actions may match a single model action, and some model actions may not be supported by data. But their display inspires a similar display with a warping function with loosened requirements on the match and with an augmented representation.

5.2.1 A display for comparing the relative processing rate

Figure 5-22 shows a chronometric fit display similar to Sakoe and Chiba's (1978) that presents the warping function for a cognitive process model. Each graphical element (shown in a legend in the upper left) represents a pair of corresponding model and subject actions. The current display presents the correspondences in order that the subject's actions occurred. A similar display presenting them in order of the model's actions could also be created. This display does not include the restriction of one-to-one matches, but allows model actions to match multiple subject actions (it would even support matching multiple model actions to a single data point, but as noted in the review, subject segments should match only one model action, or else they may not be segments).

This display presents the time of the corresponding subject actions on the x-axis in seconds, and the time of the corresponding model actions on the y-axis in terms of decision cycles. The time for both model and subject begin with the first match. Their relationship before the first correspondence cannot be computed. It is possible for later subject actions to match earlier model actions if the subject does not report all actions in order, or if different modalities are reported with different amounts of lag. Unmatched subject actions are unconnected, and put at the bottom of the display along with a label indicating the type of information that was not matched. They are positioned on the subject's time axis with the time they occurred. Overt actions of the model, such as mouse movements, that are not matched, are represented in a similar manner. Unmatched, non-overt model actions, such as internal state transitions, are not displayed. They will not necessarily be matched, so their lack of correspondence with data tells us less according to our theory of measurement (Ericsson & Simon, 1984), and it is best viewed with Peck and John's display of operator support.

This display provides many of the criteria for measures of model fit noted in Chapter 2. Better experiments are favored, for they provide a larger or more informationally dense display of the model's predictions fit to the data set. These denser displays provide more information on how to improve the model, and should be more persuasive. Typical mismatches produce signature patterns on this display; these are noted below.

<u>Identifying outliers: Computing a linear regression on the match.</u> In Figure 5-22 a least squares

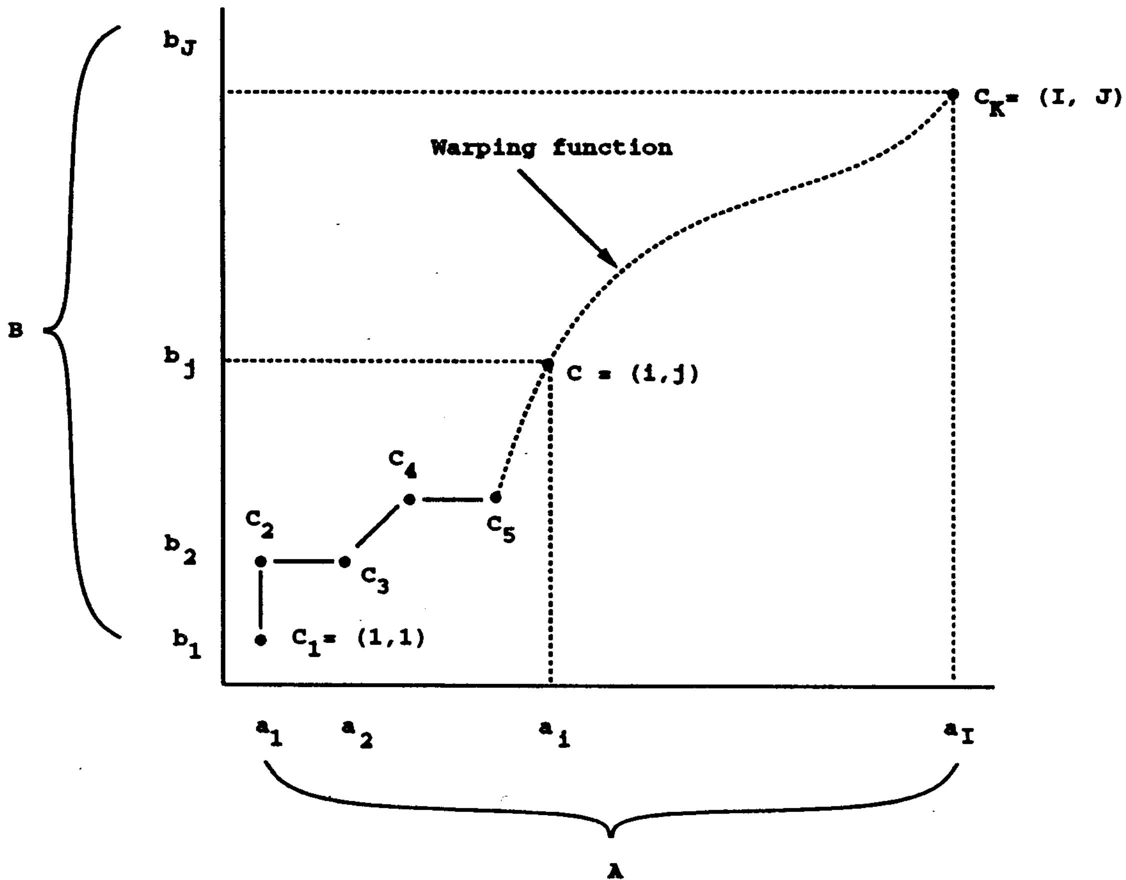


Figure 5-21:
Depiction of Sakoe and Chiba's (1978) correspondence diagram from their speech recognition task. The A axis represents the times of the subjects utterances, and the B axis represents the times of the model's predictions. The places where they correspond are represented by the C terms. The relationship of all the correspondences is seen as a warping function between the axis.

regression based on the correspondences, representing the warping function between the data and the model's predictions is drawn as a solid black line. While in each episode the correspondences may not be well fit by a linear relationship, theoretically the correspondences between the model and data should be a linear relationship. This line can be used in several interesting ways. The regression line helps to show where the fit is poor, highlights outliers, and gives a standard statistic, variance accounted for or r^2, that could be compared on a per subject and per model basis (e.g., Thibadeau, Just, & Carpenter 1982; Just & Carpenter, 1985). It also summarizes the relative processing rate of the model to the data, providing an empirical measure of the theoretical model processing cycle rate in seconds that should be more robust than simply dividing total model time by total subject time.

The regression line also makes several predictions. It can be used to find the mean percentage deviation from predicted for each subject action matched, mean absolute deviation (MAD), and root mean square deviation (RMSD). Measures like these can be used as part of engineering models of human performance to predict human performance (John, 1988), and to predict how accurately the model's predictions will be in the future.

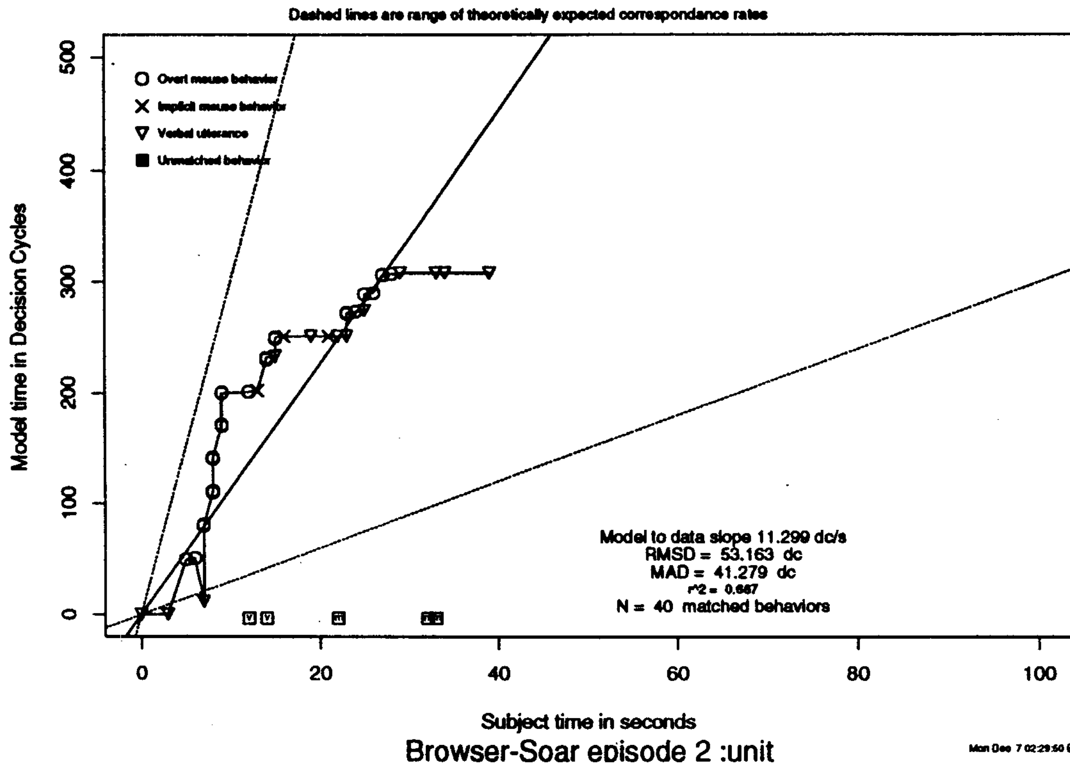


Figure 5-22:
Example relative processing rate display based on decision cycles taken from the Unit episode of Browser-Soar. The straight, solid line is a least-squares regression line through all the correspondences. Its slope is the relative rate between decision cycles and seconds. The dashed lines indicate the expected range for this measure. The location and type of the correspondences are marked on the connected line.

The model's processing rate as measured could also be compared to the theoretical decision cycle rate, but this is not completely known for the architecture used here. The Soar theory predicts the decision cycle rate only within an order of magnitude (Newell, 1990), that the rate of decision cycles will be between 3 and 30 cycles per second. To facilitate comparison with the empirically derived rates, these theoretical rates are presented as dashed lines on the display. Particularly if the regression is redone taking into account the dependent nature of the measurements, the regression results can serve as useful initial measure of the decision cycle rate. Just and Carpenter (1985) perform this analysis for the CAPS architecture, finding a 200 ms cycle rate.

The main use of the regression line, however, is for highlighting the systematic deviations. The rate of match between the two action streams should be a linear relationship, with the slope determined by the relative relationship between the model's cycle time and the actual time in seconds. Points that fall above or below the line indicate sets of behaviors that are not being performed with commensurate effort (or that follow such situations). These outlying points indicate where the model's behavior could be improved. Grant (1962) suggests three ways to use the regression line to find outliers: (a) examine the curve and points as they stand, noticing outliers, (b) draw error (95% confidence) regression lines, and look for points outside them, and (c) draw error bars for each point. In general, examining the plain line for outliers appears to be sufficient.

Signature patterns of model modifications. Table 5-18 lists the signature visual patterns that can appear on this display, and the indications they provide for how the model should be improved. How, or whether to remove them through modifying the model will be based on the purpose of the analysis and other factors. While the analyst can find out information about these outliers by clicking on them, the indication of how to modify the model is indirect. There is not an equation or set of complete rules for prescribing how to modify the model based on the correspondences, or what will happen based on the modifications. The model must be modified and refit. If there are prescriptions that can be drawn from these displays, they are not yet discovered, they will only come from further use and validation through experience with these visual displays.

Table 5-18: Signature correspondence patterns indicating types of model mismatches.

- Horizontal regions in the correspondences line indicate sequences of actions where the model performs the task too quickly relative to the subject. The model's performance may need to be expanded, or it needs to be done in a more cognitively plausible way. Alternatively, the subject may be performing more slowly than the subjects used to develop the model. This may be seen as a limitation of processing capacity of the subject, that the subject's full attention has not been paid to the task, or that the subject is less practiced at that portion of the task.
- Vertical regions in the correspondences line indicate sequences of actions where the model is performing slowly compared with the subject. The model is performing too much work.
- Downward right diagonal lines, indicate sequences (or pairs of actions) where the model and subject may be performing subtasks in different orders. This may indicate an individual difference in the subject in preference order for two operators, or if the actions are of different modalities, it may reflect reporting lag between different subsystems.
- Verbal statements that appear substantially separated to the right of their corresponding overt behaviors, indicate a lag in protocol generation, sometimes making protocols locally retrospective.
- Unmatched subject actions indicate that the model may not be performing the task completely or correctly. They may also indicate that the subject performs the task in an inefficient manner.
- Unmatched model actions indicate unnecessary actions, particularly if the subject performed the task correctly.

Examining learning within an episode. The linear regression line also supports a simple, initial examination of learning within an episode. If the Soar model is run with learning off, and if the subject learns information that transfers within the episode, then the fit of the data to the model's predictions would be concave upwards (the subject's performance speeds up relative to the model's performance). If the fit is concave only at the start of the episode, that indicates unmodeled startup effects.

Limits to the regression line. There are three problems with this regression line and its use. First, and most importantly, while it can point out outliers, it does so in a history dependent manner. If a series of actions represent a poor match, subsequent actions will also appear to be outliers with respect to the regression line. For example, in Figure 5-22, the actions from approximately 20 seconds to 40 seconds follow the same slope as the regression line, but are visibly offset.

Second, while the interpretation and alignment process results in an interpretation that forces the regression through the origin, this distorts the regression assumptions (e.g., Neter, Wasserman, &

Kutner, 1985, Ch. 4). The residuals no longer necessarily sum to zero, and the fit of the later data points is not as good as those near the origin.

Third, the value of r^2 returned by this regression is suspect because the data violate the independent measures assumption. Linear regressions assume that each data point is independent, and in this case, they are not. The regression is fit to a time-series, and the high values of r^2 may not hold when a regression that corrects for the dependencies is performed (Kadane et al., 1981; Larkin, et al., 1986).

Computation of relative processing rate using operator applications as the model's unit of time. In addition to decision cycles, there are other possible theoretically interesting measures of the model's effort. It is possible to modify the time-based comparison display to use a different time unit for the model. Soar models will have at least three main units of model time available to them, decision cycles, elaboration cycles, and operator applications. This is not an exclusive list, the selection of new states and problem spaces, and the rate of chunk creation and application could also be considered as possible units of model time. Figure 5-23 shows an example of using operator applications as the unit of model time. This time measure abstracts away some of the time information. The ratio of decision cycles to operator applications is not specified, and may not in the end be a useful measure. If the effects of decisions to select goals, problem spaces, and states, are relatively minor or correlate highly with operator selections, then the displays will appear to be very similar. If they are divergent, this may have indications for the architecture, or this may merely indicate that operator application rates vary between subjects or tasks. In either case, it tells us more about how to improve the model and illustrates the flexibility of the environment to explore new measures of model fit.

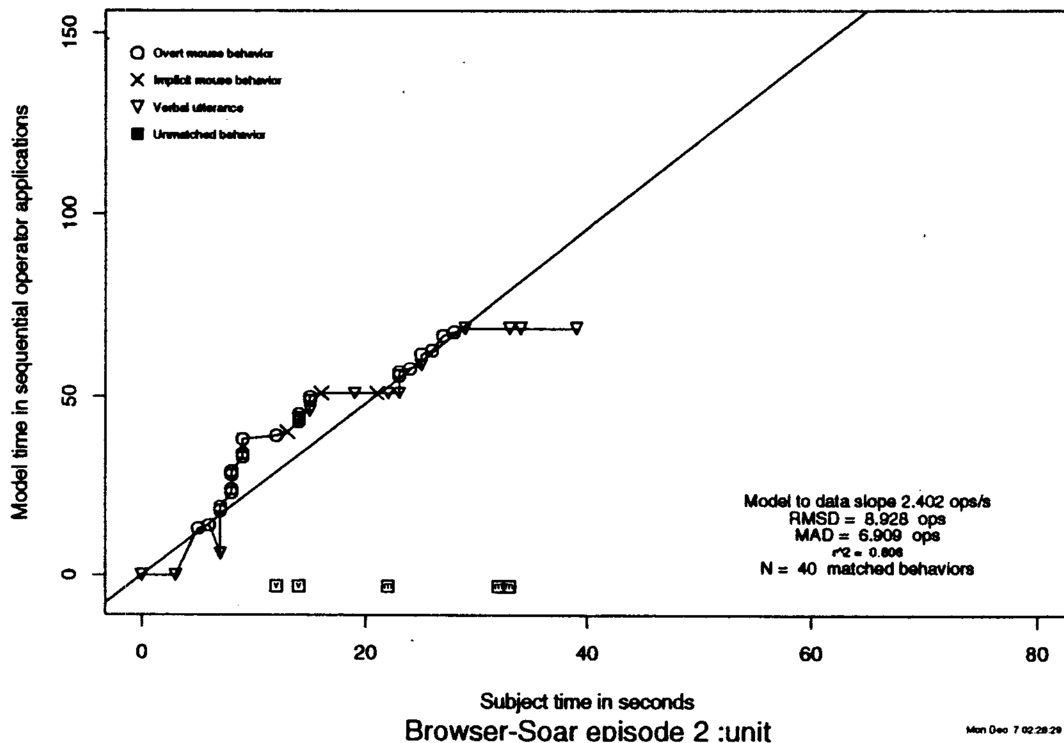


Figure 5-23: Example relative processing rate display based on operator applications taken from the Unit episode of Browser-Soar.

5.2.2 Using the relative processing display to test the sequentiality assumption of verbal protocol production

The relative processing display supports evaluating two features of Ericsson and Simon's (1984) theory of verbal protocol production. The first is the sequentiality assumption, that structures are reported verbally in the order that they enter working memory, and that "Information required as input to some process or operation will be verbalized before the output of that operation is verbalized" (Ericsson & Simon, 1984, p. 233). If the two parts of this assumption hold, subjects are not going to provide a verbal report of something that has sat around in WM for a long time and then delivered, like a letter stuck in a wall in the post-office. This assumption might also be extended to non-verbal protocols.

The second feature that can be examined with the model's predictions of working memory contents is whether the utterances are retrospective or prospective, and the time lag (or lead) of the utterances. Matched overt non-verbal task actions (e.g., clicking a button that has to be clicked to perform the task) can provide fixed data points for computing the offset of the verbal utterances.

In testing both of these features, we are also testing our models. If these reasonable assumptions are consistently violated by the models, in addition to calling into question the assumptions, we must also question the models.

The sequentiality assumption. There are two ways to test the sequentiality assumption. The most direct way is to examine working memory contents directly with neurophysiological tools. This is not yet possible. The other way is to use a model to predict what is appearing in WM. The model's goal stack is a model of what is in working memory, and can be used to test the sequentiality assumption.

The relative processing rates display supports this analysis visually. They represent the correspondences of the protocol segment to the model's predictions as temporally ordered connected symbols. The sequentiality constraint can be quickly checked by examining a display and finding only positive or zero sloped connections between all the protocol segments. Negatively sloped connections between two segments indicate a pair of verbal utterances that violate this assumption.

In addition to verbal protocols, this assumption can be extended to other data streams from the subject, for example, such non-verbal protocols as mouse movements or key presses. The testing process will be the same, except different symbols representing different data streams will be examined.

The cross-modal sequentiality assumption. One might also expect to see a difference in correspondence between verbal utterances and overt task actions. The Ericsson and Simon (1984) theory says that subjects report on information in WM (Ericsson & Simon, 1985, p. 264) along with reporting on inputs before outputs. In valid protocol (their talk-aloud vs. think-aloud utterances), only working memory items used in the task will be reported, and objects with verbal representations will be more easily reported than those that must be translated first (Ericsson & Simon, 1984, pp. 95-100). Therefore, the overt task actions and reports of mental actions may not occur in order. The overt actions (mouse clicks, moves) might not be verbally reported at all, they may not exist in WM or they may not exist in a form that is easily reported verbally. If the overt actions are not reportable verbally, they will still occur as overt actions, while verbalizable information in WM may have to be buffered, or may be reported as the overt actions pre-conditions or post-conditions.

It is not clear what the direction and size of this offset between verbal and non-verbal protocols should be. If the items reported truly are only operator inputs and outputs, then verbal utterances will be presented in order with overt actions, or with a lag, where an overt action occurred while information had not been told yet. If the utterances include goal statements about overt actions to be done, then the utterances may be prospective of the overt behavior.

The amount and direction of lag may be an indicator of protocol quality. If the lag between structures entering working memory and being reported is too long, the protocols are retrospective. The analyst must postulate the uses of long term memory processes that are actually producing the utterances,

particularly if the working memory elements reported on have also left the working memory by the time of the report. If the lag is positive the protocols are prospective, and may be including introspective comments. The lag may also indicate tasks (or subtasks) that have a non-verbal representation. The subject's utterances will include covert activity to translate the structure into a verbal representation.

5.3 Creating additional displays

One might now imagine creating numerous types of displays like these, sequential displays based on other time measures of the model and displays that presented the correspondences in a data dependent order. A more extensive sample list is presented in Table 5-19. It is not quite clear in each case what the display will look like, but some of them surely will be interesting and useful. More importantly, it is now possible to consider creating them automatically from the data.

The idea of a single description that shows how the model could be improved now seems small-minded. An analyst trying to understand and improve their model will want many types of displays. In order to create such displays, the analyst will require an underlying system that provides the functions to make them, and an environment for creating and modifying them. S provides the functionality to create the displays. S-mode provides an interface for creating additional displays as functions to be called on a dataset.

Table 5-19: Further displays for summarizing the fit of data to model predictions

- Cumulative model predictions matched over time.
- Scatter plot of correspondence times (rise and run in the relative processing rates display).
- Correspondences with primary order coming from the data instead of the model's predictions.
- Correspondences presented over time with respect to problem spaces or elaboration cycles (instead of operators or DCs) vs. subject time.
- Problem behavior graphs (depictions of states rather than operators).
- A matrix showing production conflicts.

5.3.1 S: An architecture for creating displays

These displays have been implemented as functions in S (Becker, Chambers, & Wilks, 1988; Chambers & Hastie, 1992), an interactive, exploratory, statistics and graphing package. S provides a set of general and easy-to-use facilities for organizing, storing, and retrieving data structures such as matrices. In addition to the numerous built-in numeric and graphing functions, libraries of advanced data analysis routines (such as ANOVAs and logistic regressions) are available from a fairly large and friendly user community. S is programmable, and users can create functions to perform set analyses, to combine smaller analyses, and to create interactive graphic displays. Other flexible, programmable graphing packages, such as GNU-Plot, could also have been used, but S is perhaps the most powerful and appears to have the largest user community.

While S provides all this functionality, it suffers from two disadvantages. First, it does not so much have an awkward interface, but a primitive one: a simple TTY command line. There is no facility to edit the functions in a structured way, treating them as first class objects to be loaded, edited, and run.[5]

[5] S does provide a command that will call a plain editor on a single, named function.

Secondly, S is the only piece of software in the testing environment that is not freely available. This problem is attenuated by the wide distribution of S.

5.3.2 S-mode: An integrated, structured editor for S

In order to create these displays that are functions in S, a structured editor created within GNU-Emacs is provided, called S-mode (Bates et al., 1990; Smith, 1992a). S-mode has been joint work with Kademan, and Bates over the last three years, and Smith for the last nine months. S-mode provides a structured editor to write, load, and edit S programs, and an improved command line interface.

Most analysts will use S to create displays in an edit-test-revise cycle. When programming S functions, S-mode provides for editing S functions in GNU-Emacs edit buffers. Unlike the default use of S, where the editor is restarted every time an object is edited, S-mode uses the current Emacs process for editing. In practical terms, this means that one can edit more than a single function at once, and that the S process is still available for use while editing. Error checking is performed on functions loaded back into S, and a mechanism to jump directly to the error is provided.

S-mode also provides mechanisms for maintaining text versions of S functions in specified source directories. These objects can be manipulated by the user as first class objects, and be examined, edited, and loaded. S-mode provides an interactive command history mechanism, including a quick prefix-search of the history list. To reduce typing, command-line completion is provided for all S objects and keybindings are provided for common functions. Help on individual S functions and on S-mode itself are easily accessible, and a paging mechanism is provided to view them. Finally, an incidental (but very useful) side-effect of S-mode is that a less literal transcript of the session is kept for later saving or editing than S provides by default. A complete listing is available in the manual (Smith, 1992a), and a summary is available in Table 5-20.

Table 5-20: Functionality supported by S-mode.

- Edit S object in a buffer.
- Display help on a function or variable.
- Jump to the S process.
- Load a single line.
- Load the current function.
- Load the current function and go to the S process.
- Load a file of S functions.
- Reformat the current function.
- Complete an object name by querying the S process.
- Move to the beginning (or end) of a function.
- Execute the previous command.
- Insert a function template.
- Automatic matching of parentheses and braces.

5.4 Supporting the global requirements

In addition to the direct requirements of aligning the predictions with the data and starting to interpret their comparison, there are five global requirements that the displays and S-mode also support.

5.4.1 Providing an integrated system

The S-mode and the graphing functions are integrated with the other portions of Soar/MT environment. The data for the displays can be dumped from the spreadsheet into text files and read into S data structures for creating the displays. This step of transferring the data could be more automated, but the path for passing this information is well worn and can be performed relatively quickly. Although the interaction is on the level of files, because of the file manipulation facilities GNU-Emacs provides, it is easier than it sounds.

The functions for creating the displays were designed using S-mode, and S-mode provides a fine environment for routinely calling the functions to make the displays. If at some point an order of magnitude more data is used, on the order of hundreds of subjects, it may be useful to use the batch processing facilities of S to create these displays, where the commands are not executed interactively but as part of a file of commands.

5.4.2 Automating what it can

The graphing functions have automated the display and global analysis to a great extent. What used to take a day to display can now be performed in minutes. Once created, a display can be called with nearly no cost to the analyst, so using many displays to understand a model is possible. S-mode automates many of the actions necessary to create additional displays as S functions. A table similar to Table 6-25 could be created for S-mode.

5.4.3 Providing a uniform interface including a path to expertise

The initial set of displays provided do not require any expertise beyond knowing their inputs, and this is provided with their definitions. As a prototype, this has been an adequate level of documentation. If more users start to use them, the documentation will have to be improved.

S-mode, like Spa-mode, provides a path to expertise similar to that provided by other components in the Soar/MT. S-mode can be menu or keystroke driven. The keystroke bindings of the menu commands are available to the user, and are displayed to him or her after they have been completed.

Documentation is also available as hardcopy. The S-mode manual (Smith, 1992a) and a reference card (Smith, 1992b) are available through the S-mode menus, and obtaining on-line documentation on functions takes only two keystrokes.

5.4.4 Providing general tools and a macro language

S and S-mode provide a very general set of capabilities for performing exploratory analyses and creating new displays for model testing. S commands can be combined to create functions to draw nearly any display imaginable. The creation of these displays is supported through S-mode, as well as applying them to each data set.

Hooks are places to customize a system's behavior by calling a user supplied function at a set point, such as at startup, or after a file has been loaded. The standard set for GNU-Emacs modes have been included with S-mode. The user supplied function, if any, is called when S-mode is loaded or initialized.

5.4.5 Displaying and manipulating large amounts of data

These graphic displays directly support examining a large amount of the model's performance and examining the relationship between the model's predictions and the data. The displays include the ability to examine individual data points based on their location. The displays are based on the model, and can use their representation of the model to make clear which model objects have generated predictions found in the data, and which have not.

Supporting direct manipulation. The displays and S-mode directly support manipulating the main objects of interest on this level, the data points within the displays, and the displays themselves. The functions to create new displays (and their components) are first class objects in S-mode, and allow the analyst to load and manipulate them directly.

On a lower level, the points on the displays are inspectable; clicking on them (after an appropriate function call) will print out the Soar trace, the verbal utterances, their time stamps, and other information, known about that point. Selected sets of points could even get thrown into spreadsheet for further analyses (but currently this is not supported). These interactive displays also serve as a way to understand large data sets by hiding irrelevant fields, but allowing them to be recalled on demand.

5.5 Summary of measures and recommendations for use

The two measures presented here (the operator support display and the relative processing rate display) provide the analyst with ways to view the correspondences between the data and predictions with respect to the model and the time course of the correspondences. Both types of displays provide visual patterns indicating where to improve the model and where the model is consistent with the data.

These displays do not primarily provide local, immediate information about the comparison, but this too are included as a separate block of text on the displays, and the local comparison is also available in the matching tool, Spa-mode.

These displays are not the only ones possible for depicting the comparison, the model's predictions, the data, or various combinations of these, they are only a starting point. There are many ways the model can fail to predict the data, and there are many facets to the relationship between models and the data, so many different types displays will be required by an analyst wishing to improve their model. Further displays can be created using S and the S-mode interface, and several new displays can already be suggested as potentially useful.

Problems with these displays. There remain at least three problems that apply to both types of these displays. When interpreting and using these displays analysts will have to keep in mind: (a) While the architecture treats all operators the same, the modeler and model may have different sized operators (with respect to correspondence to the subject's actions) and different levels of theoretical commitment to particular operators. Some operators may be represent placeholders for complicated operators, such as *read*. These differences are not currently represented. The relative processing rate could vary quite a bit when the grain size changes between operators in this way. (b) Not all analysts will be committed to time based comparisons. They may be interested in other facets of their models, which will need additional displays. (c) Before they are learned, Soar operators are implemented hierarchically. It is not clear how to represent in the displays when hierarchical operators are in effect, and their calling order.

What do these displays have to say about comparing two models? Given these displays, comparing how well the data fits the two models does not come down to comparing two numbers, but can now be based on more analytical (but less straightforward) process of comparing the models' performances in more meaningful ways. As indicated by the displays, the models will have regularities associated with them, places where each model's predictions match the data more closely than the other, but only a metric outside of the comparison can order these comparisons. The models' proponents, however, can now point to diagrams showing the comparison, and describe more clearly and fully how and where

their models match the data.

Future work. While these displays use the models and its structures to help in the analysis, the question remains of how to extend them. It may be possible and desirable to incorporate additional components, such as the number of rule firings, the number of matches per rule[6], and to aggregate across subjects or episodes. It would also be desirable to incorporate measures of rule and operator utility more directly, and to understand, display, and manipulate the degrees of freedom in the models.

[6]Although generally Soar models are not described on the level of rules and rule firings, but the higher level cognitive structures, such as operators, that the rules are creating.

Chapter 6

The model manipulation tool -- the Developmental Soar Interface (DSI)

> "Realizing programs with GPS on a computer is a major programming task. Much of our research effort has gone into the design of programming languages (information processing languages) that make the writing of such programs practicable."
> Newell, Shaw, & Simon, 1960

In the past, the implementation of Soar as a program has failed to fully support many of the requirements noted in Table 6-21 as necessary for testing the sequential predictions of cognitive models. The basic Soar interface was only a command line, and the commands were simple. Most commands did not provide default values. A default editor, GNU-Emacs, was perhaps assumed, but the editor was not tailored to Soar and no help was provided for manipulating productions or higher level objects in the model. The emergent structure of the model, such as problem spaces and operators was ephemeral, and only existed in the trace. After the goal stack exited a problem space, it did not exist until it was entered again. The trace itself was flat, it was printed out, and that was that.

Table 6-21: Requirements supported by the Developmental Soar Interface.

Requirements for the process model's trace
- (a) Include:
  - (i) Unambiguous predictions for each subject information stream (external and internal actions)
  - (ii) Time stamps for each action.
- (b) Be readable by the analyst.
- (c) Provide various levels of detail.
- (d) Provide aggregate measures of performance.
- (e) Be deterministic even if the model is not.

Requirements for modifying the model
- (a) Display the model so it can be understood.
- (b) Modify the model based on the comparison.

Requirements based on integrating the steps and supporting TBPA with a computational environment
- (a) Provide consistent representations and functionality based on the architecture.
- (b) The environment must automate what it can.

To support the user for the rest of the task:
- (c) Provide a uniform interface including a path to expertise.
- (d) Provide general tools and a macro language.
- (e) Provide tools for displaying and manipulating large amounts of data.

The Developmental Soar Interface (DSI) provides an interactive graphic and textual interface to support the requirements shown in Table 6-21 related to using, understanding, and manipulating the Soar model being tested. The DSI consists of three integrated yet independent pieces of software. They are designed to provide multiple entry points for users so that users can manipulate and examine the models in a natural and consistent way, no matter which module of the DSI they are working with. For example, while examining the graphic display users can run Soar ahead a simulation cycle by typing on the display, and while editing productions they also can run Soar ahead a simulation cycle though similar commands in the editor. Novices and casual users can interact with each tool through a menu. Experts will learn common commands from the menus because the keystroke equivalents are displayed there. Further details will come out as how the requirements are taken up in turn.

The Developmental Soar Interface (DSI) adds several new concepts to Soar: the idea of *interlocking* tools, each component can use the other tools' representations and capabilities; *problem space statistics*, keeping track of how often problem space objects are selected; a *macrocycle*, the ability to run the model not in terms of decision cycles, but in terms of the architecture, such as to the next

problem space selection or the third operator to be applied; and *hooks*, the ability to modify Soar's behavior at set points such as initialization or to trace actions, such as at the end of the elaboration cycle.

<u>The Soar in X (SX) graphic display.</u> While displaying the Soar goal stack the SX graphic display creates a representation of the model. This new representation of the running model (in itself a model) is used to represent the problem space level objects and keep statistics on their use. This representation allows the analyst to directly manipulate problem space level objects. Clicking on problem spaces and their subcomponents allows their working memory components to be displayed in an examination window. These windows can exist during a run and the model's working memory can be monitored in examination windows as it performs a task. An associated command interpreter and pop-up menu provide keystroke and keyword commands to manipulate the model. A special command line interpreter, tailored for running Soar, is also provided. A complete description of the functionality is provided in the SX manual (Ritter & McGinnis, 1992). While the new graphic display copies little of the code directly from previous instantiations of the DSI (Milnes, 1988; Unruh, 1986), it copies some of their ideas, particularly that a graphical interface is doable and desirable.

A trace of the Soar model designed for use with automatic interpretation and alignment systems is also provided, either with the SX graphic display or with Soar-mode. Its most important feature is that it provides the models actions in an unambiguous format, putting each selected object's name and attributes in fixed fields. It also includes features that make it more compact to fit on a limited width screen, and more easily read by other programs (subfields separated by tabs). The improved trace is also more interpretable by human analysts because it indicates the goal depth of each element of the trace with a number of dots separated by spaces instead of just with the number of spaces.

<u>Soar-mode.</u> The second module is a structured editor and debugger written within GNU-Emacs, called Soar-mode. It provides an integrated, structured editor for editing, running, and debugging Soar models on the production level. Productions are treated as first class objects. With keystroke (or menu) commands productions can be directly loaded, examined, and queried about their current match status. Listings of the productions that have fired or are about to fire can be automatically displayed. Soar-mode includes and organizes, for the first time, complete on-line documentation on Soar and a simple browser to examine this information. A complete description of the functionality is provided in the Soar-mode manual (Ritter, et al., 1992).

<u>TAQL-mode.</u> The third module, TAQL-mode, is a structured editor for editing and debugging TAQL programs written as an extension to GNU-Emacs. TAQL is a macro language for writing models in Soar on the problem space level. By providing TAQL constructs as templates to complete rather than as syntactic structures to be recalled, it decreases syntactic and semantic errors. After inserting templates users can complete them in a flexible manner by filling them in completely or only partially, escaping to the resident GNU-Emacs editor to work on something else or to edit them more directly. This leaves general editor commands available throughout the editing session. At any point in the process users can complete any partial expansions or add additional top level clauses, choosing from a menu appropriate to the construct being modified. A complete description of the functionality is provided in the TAQL-mode manual (Ritter, 1991).

6.1 Providing the model's predictions in forms useful for later comparisons and analysis

The first set of requirements that the model manipulation tool must support is related to deriving the sequential predictions of the model in a usable form. It must provide two versions of this, the first is the direct predictions used to interpret the protocol data. These predictions primarily need to be machine and human readable, but there are other requirements discussed below. The second version is an aggregation of the predictions in order to understand the model's general performance, and for comparison with aggregations of the subject's data.

6.1.1 Providing predictions for comparison with the data

The requirements for the model's trace are listed in Table 6-21. The improved trace, initially provided with the graphic display and now available separately, substantially improves several of the requirements, but several remain a problem. Aggregate measures are taken up in the next subsection. Figure 6-24 shows how these requirements have been met. The original Soar trace is shown in 6-24(a). This version is slightly ambiguous. In decision cycle 3, the name of the problem space (SOME-SPACE) and its traced feature (VALUE1) are not distinct. If the problem space did not have a name, the value would appear in the first position. The bottom of the figure lists the improvements to the trace shown in Figure 6-24(b).

6-24(a) Original Soar 5 trace:

```
0    G: G1
1    P: P2 (TOP-PS)
2    S: S4 (TOP-STATE)
3    O: O6 (WAIT)
4    ==> G: G2 (OPERATOR NO-CHANGE)
5        P: P3 (SOME-SPACE VALUE1)
6        S: S6 (VALUE2)
```

6-24(b) Modified Soar 5 trace:

```
0/   G: G1 ()
1/   P: TOP-SPACE ()
2/   S: S4 ()
3/   O: WAIT ()
4/   => G: G2 (OPERATOR NO-CHANGE)
5/   .  P: SOME-SPACE (VALUE1)
6/   .  S: S6 (VALUE2)
 (tabs are indicated with a /)
```

Improvements to the Soar trace for use in TBPA

- An unambiguous name reference is placed at the front of each line in the trace. The object's id is used if there is none. Now only the traced fields are in the parentheses, which, as an option, can be removed if there are no traced fields for a given object.
- A leading tab or spaces (user selectable) is inserted after the decision cycle number, so that trace is parsable by spreadsheet programs.
- A period (.) is placed in the indentation for each impasse level down to directly indicate the goal level.
- The goal stack indentation width and symbol are adjustable to aid where compact presentations are needed. The goal indicator is initially "==>", but it also can be changed to "=>" or "~~>".
- The generated id of the object has been moved to the back of the trace, and as an option it can be removed entirely (except it is used as the name on nameless objects).

Figure 6-24: Original and modified Soar trace.

(a.i) Be unambiguous. The new trace removes several ambiguities and retains the decision cycle number of the original trace. The name and traced attributes of the selected object have fixed positions. The use of the object's ID when a name is not available may not turn out to be the best

choice; it may be better to insert "no-name" or some other distinct marker that can be more easily interpreted than the ID as the lack of a name.

(a.ii) Include a simulation time stamp for each action. Both the new and old trace include a time stamp for each action in the architecture's own terms of decision cycles. The only difference is that the time stamp in the new trace, because it can be separated with a tab, can be read directly into spreadsheets.

(b) Be readable by the analyst. The addition of the dots for every level down in the goal hierarchy should make the trace more readable. Besides making the trace less ambiguous for machine use, presenting the name and traced attributes in a less ambiguous way should also make the trace more readable for the analyst. There have been proposals for putting the traced attribute names in the trace in addition to displaying the values. This might clutter the trace, but it should be provided as an option.

(c) Provide various levels of detail. Plain Soar provides most of the necessary variations in the level of trace detail. As noted in Figure 6-24, several additional ways to modify the trace have now been provided. These modifications were necessary to create a narrow enough trace to fit the predictions into the spreadsheet. There will be other ways to manipulate the trace so this task is not complete. How to represent the environment's responses and when to include them was not touched by this improvement.

One specific level of detail that can be manipulated is whether operators, states, or both are included in the trace. Newell and Simon (1972, p. 157) believe that problem spaces can be characterized by the states that are seen or the operators that transform the states, one can be derived from the other. Both the old and the new trace primarily display objects only at the time they are selected. Because operators are nearly always clear if not complete at the time of their selection, both traces provide rather complete pictures of the operators. At the time of their selection, states are almost always empty, and undergo further transformations as operators are applied to them. Adequate depictions of states remains a problem for both the new and old traces.

(e) Be deterministic even if the model is not. The new and old traces are only as deterministic as Soar is. A small, clear improvement would be to design a simple way to display the alternative selections in the trace when one item is chosen from many indifferent selections. The graphic display already provides this for objects with examiner windows.

6.1.2 Aggregating the model's performance

The behavior of a model can be aggregated by an external system that examines the model's external actions, or, in the case of computer programs, by inserting instrumentation into the system itself. The method used in the DSI is to aggregate the behaviors with an internal system, based on the data used to create the display.

The primary level for aggregating Soar model's performances is the problem-space computational model (PSCM) (Newell, et al., 1991). Additional measures could be (and are) taken at other theoretical levels, such as rule firings. The aggregations on the PSCM level are counts of object selection on that level, of goals, problem-spaces, states, operators, and chunks created, although, strictly speaking, chunks are on the symbolic level.

Figure 6-25 displays an example output of these aggregate counts. It is not clear that the way chosen is the best way to present this information, but it serves as a starting point for discussions and further design. After a time stamp, the initial block provides a listing of all the problem spaces found so far, and the number of operators in each of them. In this example, the problem spaces are taken from a set loaded into the graphic display besides those that have been selected.

The second block of information provides the complete selection counts for each PSCM level object known, even if it has not been selected since the last restart of Soar or call of *reset-PSCM-stats*. On each line is shown (a) the count of the selections, (b) the type of object, (c) its name or first selected

ID, (d) in parentheses, the actual name or "no-name" if one was never provided. Problem spaces also have the number of chunks that have been assigned to them. This can happen through the normal course of learning while running, or by placing previous learned chunked (or plain productions) on the list of chunks. The update function then assigns the productions to a problem space based on the problem space's name in their condition or other means (this assignment process is covered in more detail in a later section). An indentation of a single space occurs after goals and problem spaces to indicate a choice point. A similar level also could be created for states, but most problem spaces use only one initial state so it has never been found necessary. Objects with the same indentation, such as the operators in the *Compare-positive-integer* problem space, have all been selected for the same context slot.

The objects in the SX graphic display are primarily identified by their name. Objects without names are essentially identified by their relationship to the most recently selected object at the time of their creation. This implicit naming process will break down given sufficiently complicated goal stack constructions, but none have been observed so far.

This identification scheme raises several interesting questions about the architecture and what counts as a unique object in it. The current counting system relies on the name attribute of objects to be provided and on the names being unique. In this representation, if objects of the same type and relationship to the goal stack (e.g., two operators in a given problem space) do not have names, then they cannot be differentiated. The underlying structures are also available, so a more complete algorithm could be used to differentiate them.

This counting scheme breaks down when keeping track of goals. The system assumes that all goals that have the same goal type (e.g., tie or no-change), impasse object (e.g., operator), and the most recently selected context element (e.g., the top-space problem space), are the same goal. They may be different, for example, the number of tied operators in a tied impasse. Whether this represents a real difference in the architecture and a problem in the representation is not clear.

The problem of tracing embedded structures is highlighted in this display. For example, it is clear that the first *less-than-or-equal* operator in Figure 6-25 is testing two numbers. How the actual numbers are represented in the operator is obfuscated by the large number of parentheses.

Implementing *pscm-stats* suggests that counting objects on the problem space level is not yet clear. How many operators are there in a system? Sometimes a given name can occur in multiple problem spaces, but it represents different operators, and sometimes the same name can occur in multiple problem spaces and really be the same operator. Other systems avoid this problem by deciding how to name objects and then enforcing the distinction or lack of it. A position on this has not been taken within the Soar community. *pscm-stats* currently assumes that an operator cannot occur in more than one problem space. How to reliablely represent operators that appear in more than one space remains a problem both conceptually and in the software.

When printing out the calling tree and the counts of each problem space, *pscm-stats* will print out the operators used in the space and their counts each time. If a problem space is used to solve two different impasses (as defined by the higher level problem space and goal type), its selection count and its operators selection count will get printed twice. When this occurs it is misleading for two reasons. The first reason is that it implies that all of the problem space is used to resolve each impasse. This may not be the case. The second problem is that the total number of selections printed out can easily become two to three times the actual selection count.

6.2 Displaying the model so that it can be understood

The SX graphic display (Ritter & McGinnis, 1992) makes visual representations of Soar models real in a sense not available before, actual triangles get drawn for problem spaces[7], circles for operators, and

[7]Unless the user hides them, which they can do.

```
PSCM Level statistics on November  22, 1992

22 problem spaces, with a total of 11 operators.
Ops  Problem space
 0   EVERY-SPACE
 2   ANALYSE
 1   ANALYSE
 5   COMPARE-POSITIVE-INTEGER
 0   MEMORY
 0   MEMORY
 3   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   JOHNSON
 0   <N>
 0   COMPARE
 0   COMPARE-INTEGER
 0   STRIP-LEADING-ZEROS
 0   CUMULATE

The actual selection counts and calling orders:
 2 G: g1 (g1)
 3 .P: johnson (johnson) (0 chunks)
 1 . S: s8 (no name)
 7 .  G: (operator tie) (g372)
 7 .  .P: analyse (analyse) (13 chunks)
 7 .  : S: s8 (no name)
 4 .  . O: analyse-op (analyse-op)
 4 .  .  G: (operator no-change) (g377)
 4 .  .  .P: analyse (analyse) (6 chunks)
 4 .  .  . S: s8 (no name)
 3 .  .  .  G: (state no-change) (g362)
13 .  .  . O: less-than-or-equal(((((7) ((1)))) (((5) ((0)))) none))) (less-than-or-equal)
 7 .  .  .  G: (operator no-change) (g390)
 7 .  .  .  .P: compare-positive-integer (compare-positive-integer) (0 chunks)
 6 .  .  .  . S: compare-positive-integer (s320)
 6 .  .  .  . O: move-left (move-left)
 6 .  .  .  . O: direction-right (direction-right)
 8 .  .  .  . O: less-than(((((3))) (((2))) none))) (less-than)
 4 .  .  .  . O: equal(((((3))) (((2))) none))) (equal)
 2 .  .  .  . O: move-right (move-right)
 3 .  . O: create-slot((j12 no)) (create-slot)
 4 . O: count-objects-smaller (count-objects-smaller)
 3 . O: memory (memory)
 3 . O: count-objects-greater (count-objects-greater)
 1 .G: (goal no-change) (g7)
```

Figure 6-25: PSCM level statistics for approximately 100 decision cycles of the Sched-Soar model (which is shown in Figure 6-27).

so on. While our initial hope and many viewers' first reaction is that this standardizes the visual representation of Soar, this is not so. One should not view the current display as canonical, but as an approximation. Further work and suggestions from others have and will shape it, as well as its own inherent successes and failures. As a graphic display, it can be driven by a menu or keystrokes from its display windows. As part of an integrated environment, it also can be driven by keystrokes in the editors.

The graphic display can be used in two ways, as a normative display of what problem spaces may exist in the model and their relationships to each other, and as a descriptive display of the goal stack

contents while the model is running. Both types of information can be displayed simultaneously (the display can get a bit complicated) to see if the model normative behavior is correct.

Garnet. The graphic display, the Soar Command Interpreter, the dialog boxes, and the pop-up menu are built out of components provided by the Garnet user interface development environment (Myers et al., 1990). Garnet is "a comprehensive set of tools ... for implement[ing] highly-interactive, graphical, direct manipulation user interfaces" (Myers, Guise, Dannenberg, Vander Zanden, Kosbie, Marchal, Pervin, Mickish, & Kolojejchick, 1991). It stands singularly above (egregious) other graphic interface toolkits because every feature needed (or nearly every) is provided, and it is built correctly to be extendable on the right levels. Garnet provides object-oriented, constraint-based representation that allows graphical objects to be specified declaratively, and then maintained automatically by the system. The iterative behavior of objects is specified separately. The Garnet group, headed by Brad Myers and located at CMU, provides excellent support. They intend to continue extending Garnet for the next three to five years.

It is hard to imagine building a graphical interface like the SX graphic display without a powerful and well-supported interface design toolkit such as Garnet. It substantially contributed to the ease of programming of this work. Its modular design allowed it to be modified to run four times faster. Its only drawback is its size, and perhaps its speed (the problem may be with the SX code, not Garnet, or inherent to graphical interfaces). The Soar image nearly doubles when Garnet and the graphic display are loaded.

6.2.1 Normative displays of the model

Figure 6-26 provides an example display showing the problem spaces, their normative calling order, and some of the chunks that are learned in MFS-Soar (Krishan et al., 1992), a system for formulating mathematical programming models from a problem definition. The arrows indicate the nominal calling order, and the type of relationship between the two spaces. This is often a simplification, for often the relationships are not between two problem spaces, but between a problem space and an operator or other objects.

Problem spaces can be placed on the screen before a run by explicitly creating them. This can be done with functions in an initialization file or as a menu command. Problem spaces also can be placed on the screen through running the model. The default is that problem spaces remain on the display after they have been created. Most often it is desirable for problem spaces to stay in the same place on the screen across and during runs. This can be done by "anchoring" them. This means that they will appear in the same place each time they are entered. Anchored problem spaces are indicated by an asterisk (*) on their bottom left corner. However, this can be overridden when they are created by modifying the initialization file, or by removing the anchored indicator in an examiner window. If an initialization file is not loaded, problem spaces appear in a series of straight lines, but can be moved around if desired, and their configuration can be written out to a file for later reloading.

Displaying the amount of knowledge in each problem space. The SX graphic display also can depict an approximation to the amount of knowledge in each problem space. Just as the learned productions (chunks) can be associated and displayed with their problem spaces, so can the original productions. By displaying the productions associated with each problem space, the graphic display is also displaying the amount of knowledge in each space.

Figure 6-27 shows a normative display of the problem spaces and initial productions for Sched-Soar (Nerb & Krems, 1992). It was drawn by loading in a set of previously found and arranged problem spaces and their connections. Then Sched-Soar was loaded. All its productions were set to be chunks, and were assigned by the system to a problem space. If a problem space did not already exist to hold, the SX graphic display would create one. Not all problem spaces are connected. The problem spaces shown were derived from the productions loaded. The unconnected problem spaces are part of the function package and are not actually used by Sched-Soar.

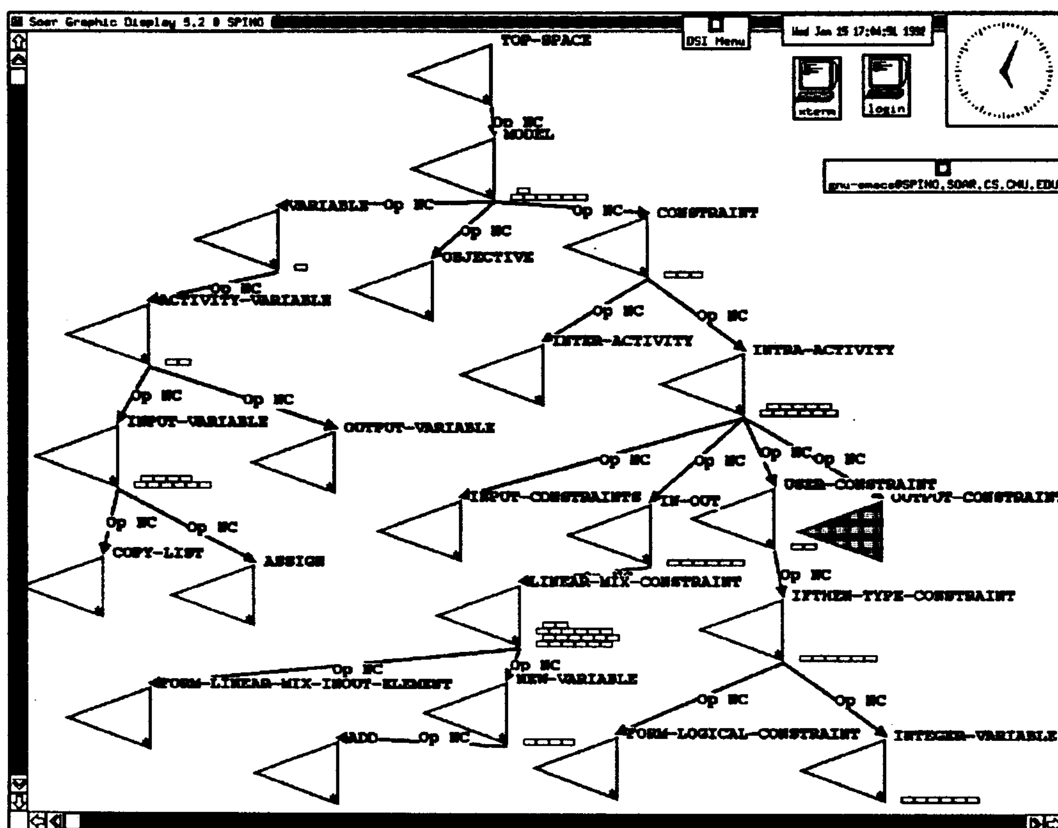


Figure 6-26:
The problem space structure of MFS-Soar (picture taken by David Steier). Learned chunks (small bricks) shown on chunk walls to right of each problem space (triangles). Lines between problem spaces labeled "OP NC" stands for operator no-change impasses in the higher space that are resolved by lower level spaces. The grey fill in the problem space on the right-hand side, *Output-Constraints*, indicates that it has recently been selected to be moved or to have its contents displayed in an examiner window.

Shown at the top of the display, the space *Every-Space* holds the productions that potentially can apply in every space because they do not contain explicit references to any single problem space. Sched-Soar is unusual in that it has so many. Upon inspection of the productions (by clicking on them), the productions are found to be predominately those that support the Soar function operator package (Rosenbloom & Lee, 1989) that Sched-Soar uses. Several problem space selection productions are also placed here, as well as several productions that would live in the *Johnson* space, but appear to have had their problem space name accidentally left out, and a few for state tracing. Most spaces contain the productions that could apply in them. For example, *Compare-positive-integer* and *Memory* contain a fair number of productions. The large number of *Johnson* problem spaces are used for look-ahead search.

The knowledge that can be applied in each space is not always displayed. Knowledge can migrate through learning, and this is represented by lines of connectivity, and later through chunks. Not all the knowledge that can be used by re-entrant problem spaces is shown. Only the highest version of each problem space is used to hold the knowledge for all of the instantiations that might be created. In some problem spaces, when an impasse incurs, an instantiation of the original problem space may be

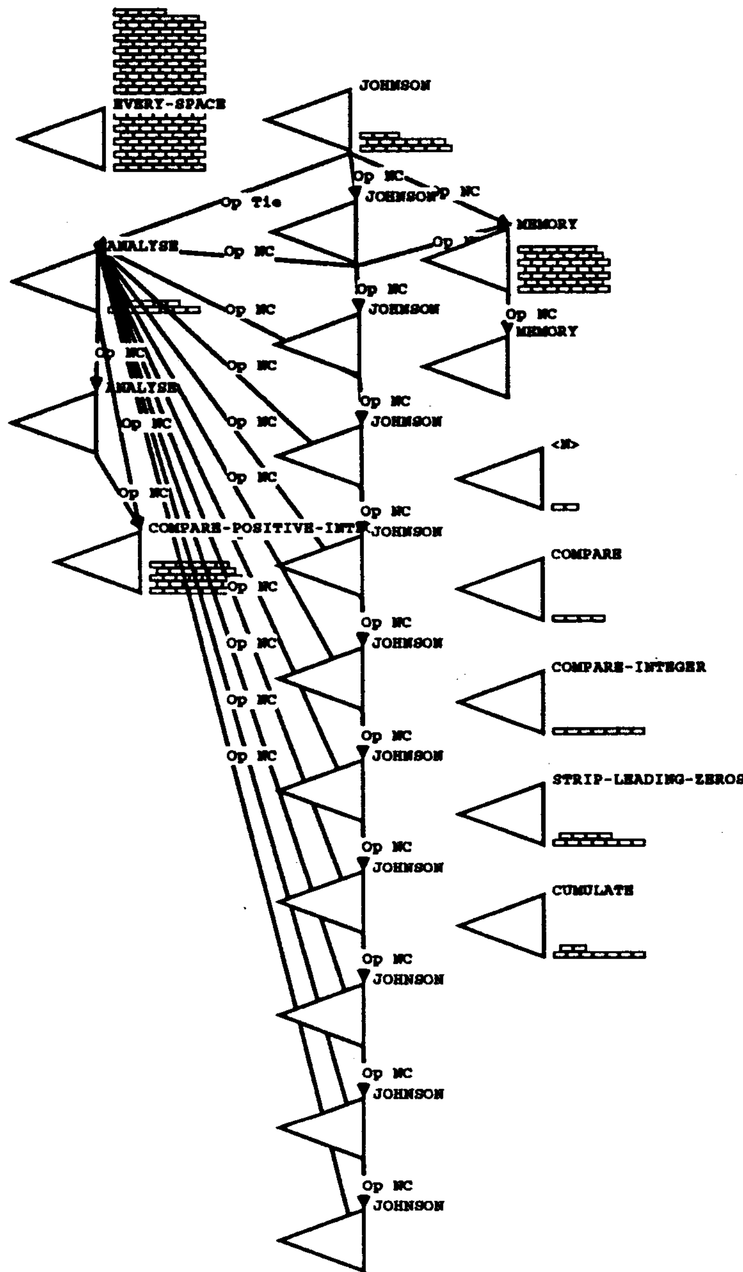


Figure 6-27: Normative display of Sched-Soar showing the productions in each problem space as chunks on the chunk wall to the right of each problem space.

instantiated and selected again as a problem space to resolve the original impasse. These are re-entrant problem spaces. In Sched-Soar the Johnson problem space (named after the original algorithm's designer) is re-entrant, and several, but not all, of the concurrent instantiations that would exist during problem solving are shown.

The knowledge in each problem space has to be measured in terms of productions. Although this certainly appears to be an imperfect measure, there is no other coherent metric. The generality of the

productions might be measurable through the number of clauses, but in the quest for accuracy, even that should be adjusted for the frequency of the features tested in the environment. The number of operators is another possible metric, but they vary even more than productions in size and generality.

Assigning chunks to problem spaces. The algorithm that assigns productions to problem spaces is a simple one that uses heuristics to classify which problem space to place a production in. It is used when new chunks are learned, and when both previously learned productions (chunks) and hand-written productions are loaded at a later time and are noted for display as on the chunk wall. The SX display first attempts to find a problem space name in their condition. If one is found, then the chunk/production is assigned to that problem space. If one does not exist, the addition algorithm next checks for an operator name in the conditions. If one is found, SX checks each problem space in the order they were created. The first problem space that has an operator by the same name is used. Next, if there is an active goal stack, then the lowest active problem space is used, which is where a learned chunk would have placed its results. If a problem space cannot be found by any of these means, then the production is placed in a dummy graphic problem space called *Every-Space*, indicating that presumably (and this is an assumption) the production could fire in any space. In practice, the production will often have conditions that can only be matched in a subset of the problem spaces.

6.2.2 Descriptive displays of the model's performance

Although most figures in this document are normative descriptions, for most users, the SX graphic display primarily serves as a descriptive display of the models' behavior by graphically displaying the goal stack and its contents. Starting with the top goal, each context level element that is selected gets displayed as a graphic element, and they can be examined with the working memory walker described in the next section.

Because the problem space level objects persist over time in the SX graphic display, a declarative model of the structures in the productions is created. This can support simple discoveries about models. Until Soar and then TAQL were run with the same graphic display, a mistake really, the developers of TAQL and Soar did not know that they used different top level problem spaces. TAQL uses *Top-space* and Soar uses *Top-PS*. In the graphic display, they appeared as two different problem spaces — in a textual display this difference went unnoticed for a year.

Figure 6-28 shows Sched-Soar during a run. The problem space names and locations have been loaded from a previously created description. If the problem spaces were not preloaded, they would appear in several columns top to bottom starting in the upper left corner. The black lines connecting problem space level objects in the display indicates their selection order in the stack.

Selected context item. The context element last added to the stack, such as a state or problem space, is treated as the "selected" context element and is shaded. Clicking on a context element that is not the latest one added (i.e., not "selected") also will select it and display its name if it is not displayed. When Soar is running, the graphic window will scroll to make the selected context object visible if auto-scroll is turned on. Figure 6-26 includes a selected problem space. In Figure 6-28 the selected context item is the *Less-than-or-equal* operator in the *Analyze* problem space.

Problem spaces. Problem spaces are displayed as triangles. Their names are displayed at their upper left hand corner. Any traced attributes are displayed after the name separated by a colon. Problem spaces can be moved around with the mouse, and when double clicked upon, a problem space examiner window will be created. The bold text in their examiner windows can be moused to create further examination windows of goals, operators, and states, and of their substructures.

Goals, states and operators. Goals are displayed as large circles. Their ID is displayed by default. Their type (impasse and attribute, e.g., operator no-change) is displayed on their creation, and it gets smaller when a problem space is selected to make room for the problem space triangle. States are displayed as squares. Their name is not displayed by default. Operators are displayed as small circles. Their name is displayed by default. These types of objects, when double-clicked, will display their

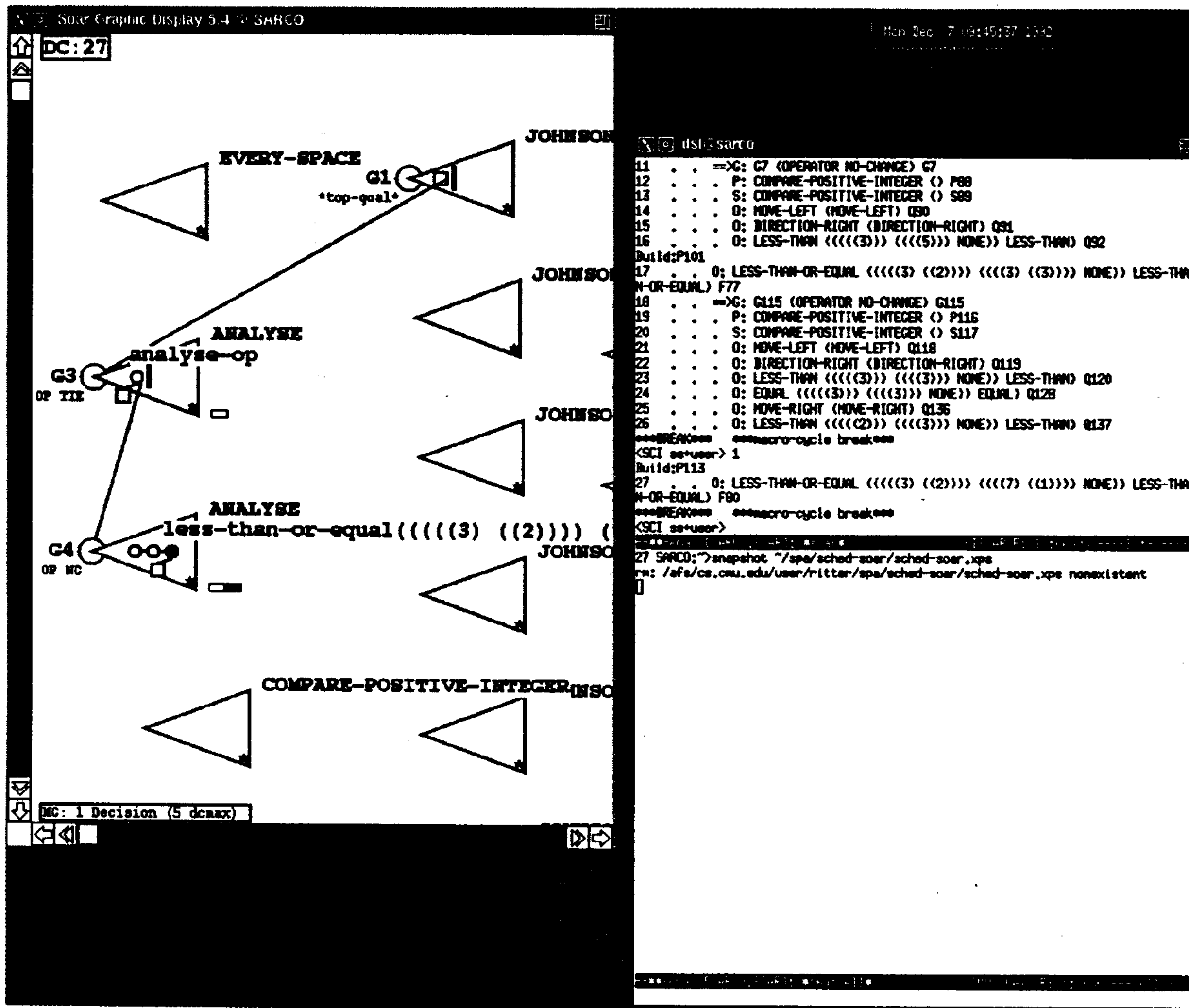


Figure 6-28:
Example descriptive display of Sched-Soar at decision cycle 27. The chunks reported as belonging to each space are not learned chunks, but are the model's own productions loaded as chunks and assigned to spaces based on the algorithm presented in Chapter 6 on the graphic display.

contents in a simple examiner window as shown in Figure 6-29.

Chunks. Chunks are displayed to the right of the problem space that the SX display believes that they will apply in. They are displayed as a dark black box on the decision cycle that they are created and later as a hollow box. When chunks fire, they explode visually, and, optionally, beep. They also can be set to display their ID when they fire or are created. To make it clear which chunks fired, the exploded chunk remains until the beginning of the next decision cycle. Similarly, newly created chunks remain dark after their creation until the beginning of the next decision cycle. The small block in black next to the *Analyze* problem space in Figure 6-28 is a newly created chunk, and the white filled block is an old chunk.

6.2.3 The working memory walker

Besides examining the global structure of the models, users will need to examine the structure of the components. Table 6-22 lists the requirements that users need from this type of display to examine working memory. This display task is similar to displaying other graph and structure examiners. Often graphs like this are examined by displaying the whole graph on a sheet. The user can open and close leaf nodes, and if the graph is too large to display all at once, the user is provided with a window on the graph that can be scrolled around.

Table 6-22: Requirements for the working memory graph examiner.

- Click on objects to examine them.
- Hide objects.
- Do not require lots of scrolling.
- Examine memory all the way down.
- Look at multiple objects at once, perhaps from various levels.
- Hide sibling subtrees.
- Hide parent links that are not informative.
- Update structures as Soar runs.
- Run quickly enough not to significantly degrade performance.
- Be relatively easy to implement.

A design to meet these needs does not appear to require a single large window to display the graph. Actually, a single window design cannot meet these requirements, so a different design was tried here. The global display was extended so that users could click on the objects that represent the global structure and have them open up into similar windows, all the way down. This design appears to satisfy all the requirements in Table 6-22. Figure 6-29 provides an example display examining a tied operator and its substructure in Rail-Soar (Altmann, 1992).

A window displaying the selected item can be created by typing "e" for examine on the display (also *:e* or *e* in the Command Interpreter), by selecting the "Examine selected item" option on the pop-up menu, or by double-clicking on the desired object. Items in bold text in problem space examiners and all objects in other examiner windows can be clicked on to create further examination windows, all the way down. If a constant value is selected to be examined, the examiner beeps when the constant is selected to be opened. The traced attribute values that would normally be displayed in a trace are displayed as the object's name when it is created, and used as the window title when the object is examined.

Since this display has been implemented, a few users but not many, have noted that it would be useful to be able to modify Soar's working memory directly with this tool. This has not yet been implemented, it is not crucial for few users have noticed it, but this capability might support new debugging methods, and should be added in the future.

The examiner windows during a run. Examination windows contents are always updated after every macrocycle and by default after every decision cycle and elaboration cycle. They can also be updated by calling *update* or *up* in the Soar Command Interpreter. There is no such thing as a free update, so if a user wishes to update less often, they can do two things. For a single modeling session or part of one, they can select that as an option from the *DSI and Soar parameters* dialog box. For a long term

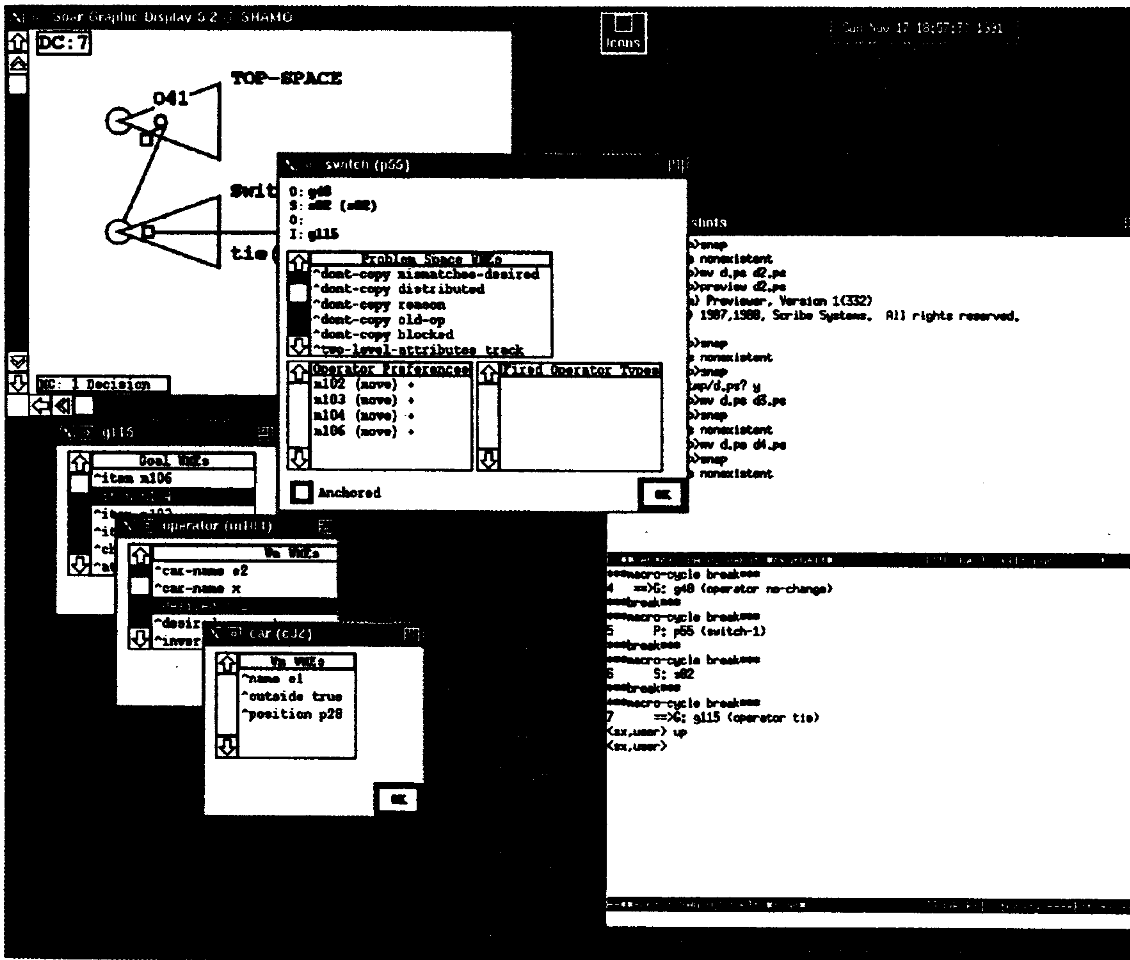


Figure 6-29:
Example display of examiner windows of Rail-Soar (Altmann, 1992). The *Switch* problem space has been opened, and the impasse goal *g115* has been opened from it. From within that examiner window (labeled "g115") the *m104* operator was opened, and then the *desired* attribute of that, *Car c32*, has been opened from within the operator examiner by clicking on it. A Soar-mode editor is on the right.

change they can modify their initialization file.

Providing a visual display of the contents of working memory while the model is running can be very informative. For example, during a demo of NL-Soar with an examiner open on the top goal, it was observed that the top goal had two top-level problem spaces to choose between. This was not known to the NL-Soar implementors, and was caused by a duplicate production creating acceptable, indifferent problem spaces.

6.2.4 A pop-up menu and dialog boxes to drive the display

Figure 6-30 shows all the dialog boxes and the pop-up menu that can be used to run and modify Soar and the SX graphic interface. The SX graphic display is in the upper right. Moving clockwise, the first object is the pop-up menu that the user obtains by clicking on the graphic display. By default the menu will stay up until it is iconified or exited, but the user can set the menu to be a true pop-up only

menu.

Each item consists of a "menu label" followed by the keystroke accelerator equivalents (if any) available for typing on the graphic window, or typing to the new Soar Command Interpreter. If multiple commands are available, they are separated by a "|" between types and by commas within a type. The menu support running the model in a variety of ways, including a new unit called a macrocycle. A macrocycle is a user set-able amount that can be measured in decision cycles and in problem space level units such as "until the 3rd operator has been selected". This menu is also used to access all the dialog boxes. The menu also includes some general graphic commands, such as examining a graphic object or taking a snapshot.

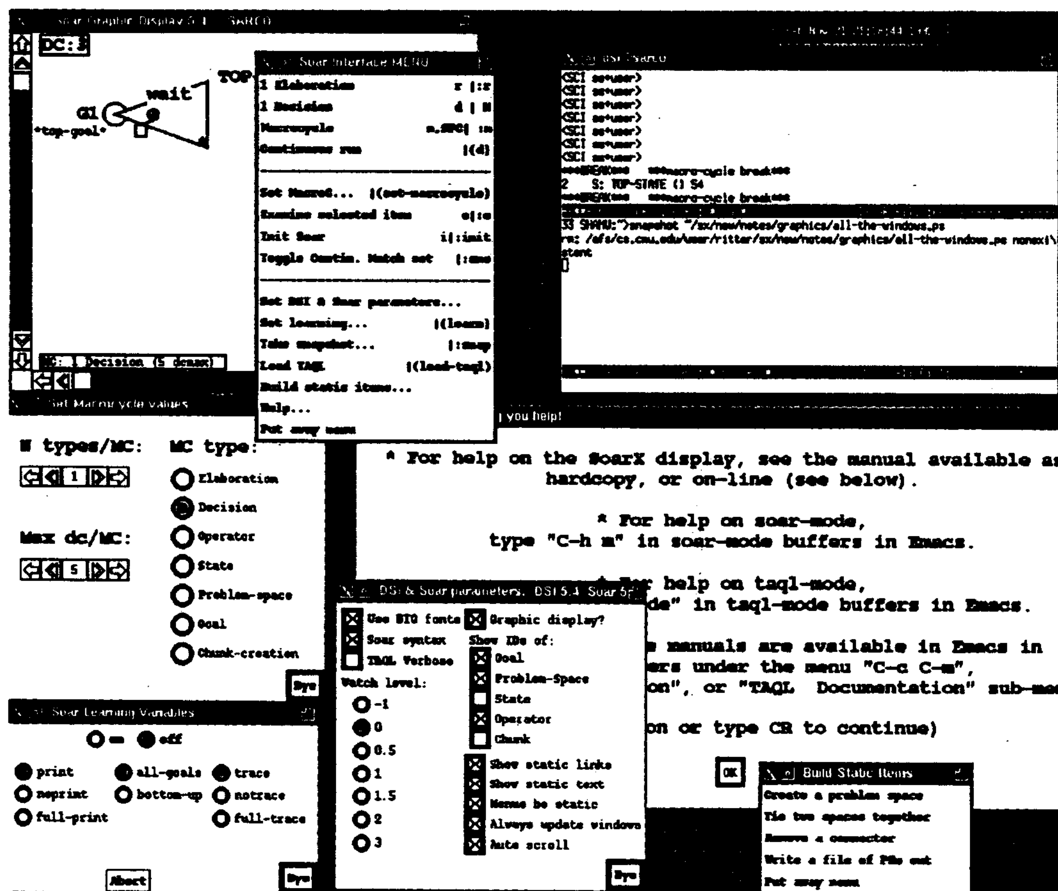


Figure 6-30:
The pop-up menu and dialog boxes within the SX graphic display. Moving clockwise, the pop-up menu is followed by a GNU-Emacs window, which has the Soar process buffer as one of its windows. The DSI help window is below that, partially obscured. This help window is accessible from the pop-up menu, and provides general guidance for how to get help, mostly through Soar-mode. At the bottom right is the static display menu that allows the user to create static views of a model on the problem space level. To its left is a dialog box for modifying some of the Soar parameters, and some of the graphic display's parameters. Next to that, on the bottom and left, is a dialog box for setting the Soar learning algorithm. Finally, there is a dialog box for setting the macro-cycle.

6.3 Creating and modifying the model

The analyst needs to create and modify the cognitive model by writing knowledge as productions or TAQL constructs. The ability to informally test the models for functional performance even before comparing it with behavior must be included in this requirement. As structured, integrated editors for Soar and TAQL programs, Soar-mode and TAQL-mode support these needs. They are integrated with Soar — they provide a facility to start up a Soar process and can communicate directly with it. In particular, Soar-mode provides a command line interface that augments the Soar Command Interpreter when it is available and replaces it when it is not. They are structured because they are designed to treat the structures in Soar programs, productions, and the structures in TAQL programs, TAQL constructs, like other structures within the editor. Users can move between them, cut and paste them, directly load them, and examine these structures as they appear to the Soar process.

6.3.1 Soar-mode: An integrated, structured editor for Soar

Soar-mode (Ritter, et al., 1992) provides a set of commands to manipulate Soar objects more directly and allows the user to start a Soar process. The user is provided menu items and keystroke commands that can quickly pass various sized portions of Soar tasks to the connected Soar process. Table 6-23 lists the major functionalities provided by Soar-mode.

Novice users can drive Soar-mode (and TAQL-mode) with a menu. After each command is executed a description of any equivalent keystroke accelerators is displayed to the user, providing a path to expertise. The user can also query a menu (select the "?" item that is provided or type a space) for a list of the keybindings of the menu items.

Soar-mode is built on top of a Lisp editing mode for GNU-Emacs called ILISP (McConnell, 1992), which is similar to, and emulates many of the functions in the Lisp machine programming environment (Greenblatt, Knight, Holloway, Moon, & Weinreb, 1984). The underlying functionality of that mode and GNU-Emacs are also available.

Table 6-23: Overview of the functionality offered by Soar-mode.

- A structured editor for Soar productions and for loading productions, regions, and files directly into a running Soar interpreter.
- The ability to treat Soar problem spaces and operators as levels in an outline, performing the usual outline processing functions on them.
- Commands to test and examine productions bound to keys and mouse buttons that are smart enough to tell which productions they are in or over.
- Complete on-line documentation for Soar, Soar-mode, the Soar default productions, and the Soar source code.
- Functions to generate and maintain informative source code file headers.
- Tags file support for Soar productions (i.e., find-production-source-code) to enable fast and easy retrieval of production's source code.
- Support for running one or more Soar processes in separate buffers, and commands for interacting with these subprocesses.
- Support for Common Lisp programming (this is the system underlying the current implementation of Soar 5, and may disappear in later releases when Soar moves to C).

6.3.2 Taql-mode: An integrated, structured editor for TAQL

Taql-mode (Ritter, 1991) builds upon the basic capabilities in the GNU-Emacs editor and a template system extension (Ardis, 1987) to provide users with the ability to enter TAQL constructs by filling in a template. When users execute the command to insert a template, they are offered the menu of templates shown in Figure 6-31. Figure 6-32 shows an example template as it would initially appear in a buffer. During expansion, commands to expand the current TC are explained in the mode line (the reverse video line at the bottom of each buffer) or the message line (the line at the very bottom of an GNU-Emacs display). Often the user is simply queried with yes/no questions about inclusion of optional clauses and expansion of clauses. At other times, they are presented with a menu similar to the selection menu. The heart of the templates is entered as text. The ability to auto-complete names upon a keystroke command, already extent in Emacs, is highlighted through display on the Taql-mode menu, and by rebinding it to a new key. Encouraging the use of auto-completion helps keep variables spelled the same way each time, and cuts down on the number of keystrokes to enter a TAQL construct.

```
    PROBLEM-SPACE-PROPOSAL-AND-INITIALIZATION:
propose-space:
propose-initial-state:
propose-task-state:
    OPERATOR-PROPOSAL:
propose-task-operator:
propose-operator:
    OPERATOR-SELECTION-and-EVALUATION:
prefer:
compare:
evaluate-object:
evaluation-properties:
operator-control:
    OPERATOR-APPLICATION:
apply-operator:
    GOAL-TESTING-and-RESULT-RETURNING:
goal-test-group:
result-superstate:
propose-superobjects:
    ELABORATION:
augment:
    OTHER-TEMPLATES:
the-OSU-production-templates:
sp:                       ; the simple sp
TAQL-program-template:    ; Yost's outline
```

Figure 6-31: TAQL-mode templates menu.

6.3.3 The Soar Command Interpreter

The SX display is run with the new Soar Command Interpreter (SCI). It provides a better command interpreter, one tailored to Soar. The prompt of the Soar Command Interpreter has three fields: a Soar Command Interpreter title ("SCI"), characters indicating the current reader syntax, and the current lisp package. This prompt is easily changed. The read table in Soar interprets commas as preference syntax; Lisp normally interprets them as part of the backquote macro. In the prompt, "ls" indicates that the Lisp interpretation is used, while "ss" indicates that the Soar syntax is used. For example, the prompt "<SCI ls:user>" indicates that the user is running the Soar Command Interpreter, the Soar reader is set to Lisp syntax, and the current lisp package is the user package. The SCI accepts keywords that specify an action for the graphic display or Soar. These commands can begin with or

```
(propose-space {propose-space-name}
    {space-proposed}
    {subspace-function-clause}
    {?when-conditions}
    {?copy-clauses}
    {?rename-clauses}
    {?new-actions-specs}
    {?use-superspace-top-space-or-id-clause}
)

  ? indicates optional clauses,
  ! indicates mandatory expansion (usually user doesn't see this)
  plurals indicate multiple copies may appear, e.g. when-conditions.
```

Figure 6-32: Example TAQL-construct template.

without a colon. Table 6-24 lists the most important commands in the SCI.

Table 6-24: Most important commands in the Soar Command interpreter (SCI).

- The ability to run ahead based on the problem space level, such as next operator.
- Short cuts for toggling the reader syntax and the lisp package.
- Pop up an examination window on the currently selected PSCM level object.
- Run ahead one macrocycle. The default value for a macrocycle is 1 decision cycle. Any open windows on PSCM items are updated each macrocycle.
- Any number runs the model N macrocycles.
- Type the initial letter of any problem space level object (goal, problem space, state, operator, chunk) to run to the next new occurrence of that object.
- Redo the last successful command.
- Take a snapshot of the display for inclusion in documents like this one.
- When the user types "help" or "?", help is provided as a listing of the keywords and their effects. The help message is automatically generated from the commands.
- Anything else gets read, evaluated, and printed.

6.4 Supporting the requirements based on the whole process and its size

Besides the direct requirements of aligning the predictions with the data and starting to interpret their comparison, the DSI supports the five global requirements based on the whole process and its size.

6.4.1 Providing consistent representations and functionality

In the DSI, while each of the tools can stand alone, they also know about the others, and can interact appropriately with them. For example, commands executed from the menu on the graphic display window can request buffers to appear in Emacs. (In the best of all possible worlds, if the other tool is not present, something appropriate still happens.) Similarly, commands in Soar-mode can run commands in Soar directly. In each tool and across tools, some care has been taken to provide

multiple entry points. That is, each command is available in each tool and often in a variety of appropriate and similar ways. For example, there are several ways to run the init-soar command; one can type *(init-soar)*, *:init*, or *init* to the Soar Command Interpreter, choose *Init* on the graphic display menu, or type an "i" on the graphic display window. Help is provided with each tool to facilitate learning the other entry points. For example, the graphic menu item for init-soar includes a listing of the other expressions of this command in the other modules.

Because they can communicate, the various modules in the DSI are also able to use each others display. Users can request objects be displayed graphically from the Soar Command Interpreter, and the graphic display, when chunks are clicked on, can display them in Emacs buffers. As an additional example, Soar has been augmented with a command called continuous-match-set. This command sets up machinery so that after every elaboration cycle Soar prints out which productions will fire on the elaboration cycle (the match set). If Soar-mode is available, they get displayed at the top of a separate, scroll-able buffer. If Soar-mode is not available, they merely get put in the trace.

The components of the DSI also interact with Spa-mode and the measures of fit. Upon the user's request, Spa-mode can query the graphic display to obtain a listing of the operators in the current model, and the trace can be inserted in the spreadsheet. Spa-mode can then use these for exploratory coding of data. The displays of fit organize their data using the names of the operators obtained from the graphic display as labels on the display.

6.4.2 Automating what it can: Keystroke savings

The model manipulation interface does not offer any large pieces of automation such as automatic alignment or display creation. What it offers is a large number of small automations. Models can be loaded more quickly, some pieces of functionality are directly accessible. The largest small improvement has been to create functions to perform frequent tasks, and bind them to keystrokes and command names in Soar-mode, the Soar Command Interpreter, and the SX graphic display.

The keystroke model of Card, Moran, and Newell (1981; 1983) predicts that as a first order effect, the amount of time performing a task will be proportional to the number of keystrokes needed to perform the task. Table 6-25 shows the savings for several common tasks that Soar-mode provides over interacting with a plain Soar process.

The savings appear to be considerable. The measures in this table are only an approximation of the true savings because they include many simplifying assumptions. The measures do not include the time to plan, but it should be small for most of these actions, and the interactions with Soar-mode are more direct and should require less planning. Some of the more complicated commands not shown in Table 6-25, such as running the model to the next problem space, would offer further savings because they would require many more keystrokes and would include several mental operators.

6.4.3 Providing a uniform interface including a path to expertise

The DSI has been designed to accept multiple entry points and names for commands. Many commands can be executed in a variety of windows, with a variety of names. You can choose the way that best suits you, and the work that you are currently doing. For example, you can init-soar by typing to the command interpreter ":init", "init" (as long as the variable init is unbound), or (init-soar), by selecting init-soar on the graphic display pop-up menu, by typing "i" on the graphic display window itself, or by typing in Emacs, ESC-x init-soar.

Each command across the multiple possible entry points is consistent: they share the same name, or when appropriate, they use (so far) single letter abbreviations. While several toolkits are used, only one designer has integrated them, so while perhaps screwy, a method to the madness also should be observable (Brooks, 1975).

Menu driven for novices, keystrokes for experts. Each component of the DSI (SX graphic display,

Table 6-25: Keystroke savings for Soar-mode accelerator keys, the Soar-mode menu, the SCI, and the SX graphic display compared with the default Soar process. (All measures in keystrokes unless otherwise indicated.)

	PLAIN SOAR PROCESS	SOAR-MODE			
COMMAND	Keys	Keys	Speedup	Menu	Speedup
Load file (using 7 char long name)	24	3	8.00	7	3.42
Excise production	25	3	8.33	7	3.57
Load production					
with keys	14	3	4.66	7	2.00
with mouse	7	3	2.33	7	1.00
Trace production	24	4	6.00	7	3.42
Production matches?	31	4	7.75	7	4.42
Continuous match set (just look for Soar-mode)	8	1	8.00	1	8.00
Run Soar 1 DC	9	3	3.00	na	na
Open on-line Soar manual	49	7	8.00	7	7.00
Find out reader syntax	14	9	1.55	na	na
View function documentation	35	3	11.66	7	7.00

	PLAIN SOAR PROCESS	SCI		SX Display	
COMMAND	Keys	Keys	Speedup	Keys	Speedup
Run model 1 decision cycle	5	2	2.50	1	5.00
Find out reader syntax (just look for SCI)	14	1	14.00	na	na
Examine an object (spr)	9	2	4.50	2	4.50
Initialize Soar	12	2	6.00	1	12.00

Soar-mode, and TAQL-mode) can be menu driven and keystroke driven. Menus lay the commands out for the user, users need not memorize them. Each menu also displays the equivalent keystroke shortcuts. If the user does not know how to do something, they can check the menus. The graphic display menu is available by clicking the middle mouse button, and then selecting an item with any mouse button. In Soar-mode and TAQL-mode, Control-C Control-M will bring up a menu of commands and sub-menus, and in later releases of GNU-Emacs this will be saved to provide menu functionality. Menu items can be selected by typing their first letter. Further explanations and key binding information can be obtained by typing a "?" or a space. After the command is executed, the keybinding is echoed in the message area.

Previously there was little documentation for Soar on-line, including the manual ("someone might take it and improve it"!), and the documentation for individual functions were awkward to obtain; the user had to type the cumbersome command "(documentation '<function-desired> 'function)". This is not uncommon for modeling systems, Lisp often comes that way out of the box. We consider on-line documentation to be a useful adjunct to hardcopy versions, so Soar-mode includes a uniform documentation accessing mechanism available as a menu item. Users can now obtain the main Soar manual and other manuals (such as the editor manuals and release notes) via the main menu.

6.4.4 Providing a set of general tools and a macro language

The DSI is designed to support a general activity, inserting knowledge into a Soar model, and is itself general. It can be used to create any Soar model, and is designed to be able to display any Soar model. Macro-languages and an interpreter are available for each component. Common Lisp is available with

the graphic display, and GNU-Emacs Lisp is available with the structured editors, Soar- and TAQL-mode. The source code is provided for each component, so what is poorly documented or not documented in sufficient detail can be found in the source code.

Hooks are places to customize a system's behavior by calling a user-supplied function at a set point, such as at startup, or after a file has been loaded. Several have been added to Soar5, and the standard set (loading and initialization) for Emacs modes have also been included. The appropriate user-supplied functions, if any, are called after Soar is initialized, after each decision cycle, and after a macrocycle.

6.4.5 Displaying and manipulating large amounts of information

Objects that the programmer (or Knowledge Engineer if you prefer) has in mind, such as productions, TAQL constructs, emergent objects in Soar that appear as members of the goal stack or attached to a subpart of it, are treated as first class objects that can be directly loaded, excised, run, and examined.

The SX graphic display uses a new, node-based algorithm for browsing the working memory structures in the goal stack in a natural manner, and for displaying how the contents change while the model runs. The structures inherent in a model, most notably the problem spaces (states and operators too, but they are not shown as nicely), are examinable after a run in the graphic display, and their names and frequency of appearance are available from the *pscm-stats* command. Which structures are in the stack is graphically depicted.

The structured editors provide support for manipulating the productions and TAQL constructs directly. Direct manipulation of Soar models on the appropriate level provides a significant drop in the number of keystrokes required.

6.5 Lessons learned from the DSI

In addition to providing an environment to support manipulating the model, its initial use unrelated to testing process models provided several lessons about the usability of Soar software and the behavior of Soar models in general.

6.5.1 The relatively large size of the TAQL grammar

Codifying and supporting the creation of TAQL constructs in a structured, template driven editor required enumerating them in a formal grammar. Table 6-26 displays the sizes of each version of the TAQL grammar with respect to several other languages that template-mode provides. Included for comparison purposes are set of templates used at The Ohio State as part of Taql-mode. These templates are based on the problem space level operation templates that were included in the Soar 5.2 manual (Laird et al., 1990) as plain text. From left to right, the columns display the raw size of the templates, the total number of nodes in the grammar, and the number of grammar nodes automatically expanded for the user as the templates were completed, and the size of each set of templates in nodes relative to the smallest template set, excluding any auto-expanded nodes.

This table shows the relatively large size of the TAQL grammar. It is quite possible that the coding of the TAQL grammar is more thorough than the coding of the other grammars, and an examination of the grammar for Emacs Lisp confirms that it is missing perhaps half of the special forms. However, the TAQL 3.1.4 grammar itself is not complete, with approximately 90% of its constructs represented in the templates. The size of its grammar may have impeded TAQL's acceptability and learnability.

6.5.2 Behavior in Soar models is not just search *in* problem spaces

Models of human behavior in Soar have often been described exclusively as search *in* problem spaces. Table 6-27 lists several places where the behavior of Soar models have been described this way (and

Table 6-26: The size of the TAQL grammars within TAQL-mode and the programming languages supplied with the underlying template-mode.

	Raw size in char.	Relative size to elisp (in chars)	Total Nodes	Auto-expand Nodes	Relative size to elisp (in nodes)
TAQL (3.1.2)	29.40 k	10.50	238	99	5.34
TAQL (3.1.3)	31.10 k	11.10	287	93	7.46
TAQL (3.1.4)	35.80 k	12.78	306	98	8.00
Soar (SPs)	11.60 k	4.14	31	0	1.19
C	2.70 k	0.96	34	0	1.30
Pascal	3.40 k	1.21	44	0	1.69
Elisp	2.80 k	1.00	26	0	1.00

yet there are other descriptions where the relationships between problem spaces and search in Soar models includes other alternative formulations, e.g., Yost & Newell, 1989; Newell, 1991; Waldrop, 1988). Even the cover of *Unified theories of cognition* (Newell, 1990) presents a schematic of this type of search. If the behavior of the models is viewed this way by their authors, it will color their thinking, and percolate out to other audiences, as indicated by the last quotation.

Table 6-27: Descriptions of Soar and Soar model's behavior as search *in* problem spaces, presented in chronological order except for the final quote (All italics in original).

- "Soar is organized around the *Problem Space Hypothesis* (Newell, 1980b), that all goal-oriented behavior is based on search in problem spaces." Rosenbloom, Laird, & Newell, 1988, p. 229
- "The Soar architecture is based on formulating all goal-directed behavior as search in problem spaces."' (The Soar group, 1990)
- "The search through the [problem] space can be made in any fashion", Newell, 1990, p. 98.
- "Soar formulates all tasks in *problem spaces*, in which *operators* are selectively applied to the *current state* to attain *desired states*." Lewis, Huffman, John, Laird, Lehman, Newell, Rosenbloom, Simon, & Tessler, 1990, p. 1035.
- "All tasks are formulated in Soar as search in problem spaces, where operators are applied to states", Simon, Newell & Klahr 1991, p. 435.
- "One of the most unique characteristics of Soar is its view of all goal-directed cognitive behavior as search in problem spaces. Each problem space consists of a set of states and a number of operators to move from state to state. Given a goal to achieve, Soar first selects an appropriate problem space, then selects an initial state, and then selects an operator that it applies to that state to get a new state. This process continues until a state that satisfies the goal is reached." Ward, 1991, p. 13.
- "The basic premises [of Soar] are these: ... 4: That all intelligent activity can be characterized as search through a problem space;" (Norman, 1990)

What is search in a problem space? Search in Soar would appear to describe primarily two types of behavior. The first is the application of numerous operators in a single space. Backup, when

necessary, would be performed by other operators to modify the state. Soar4 models often used this technique when they performed search. These operators could also use other knowledge sources through impasses to other problem spaces. If the operators were all indifferent, there would not have to be a conflict leading to a tie between operators and the associated impasse. Search in this instance would have a large number of operators applied per problem space, and a large number of states.

The other way that search could be performed would be to have several available operators in a problem space, but not have them indifferent to each other. An impasse would arise of which operator to apply, and a goal stack of problem spaces used in look-ahead search would be created, like the one in Sched-Soar shown in Figure 6-27. This type of behavior would also result in numerous operators in the lookahead space, and a large number of instantiations of the lookahead space.

Types of behavior that are probably not best described as search in problem spaces are situations where there is a series of operators applied, and each operator is the only possible operator and where the operator is readily available. That is, where there is no uncertainty involved in the creation, selection, and application of the operator. That is not to say that such situations will not arise in problem spaces, or that it cannot be represented in terms of the problem spaces, just that these are not situations best characterized as search *in* problem spaces.

Visual displays of search. With the graphic display having provided dynamic pictures of several model's goal stacks and counts of how many operators the models use and how many operators are used in each problem space, we can now make the argument that search within a single problem space does occur, but it is not the only mode of activity and is too weak of a description of how current models in Soar use knowledge. The graphic display's representation of the goal stack shows that the models are not just performing search in a problem space. Observing the goal stack for Browser-Soar (Peck & John, 1992), Seibel-Soar (Ritter, 1988), Sched-Soar (Nerb & Krems, 1992), MFS-Soar (Krishan et al., 1992), NTD-Soar (John, et al., 1991), NL-Soar (Lehman, Lewis, & Newell, 1991) and Rail-Soar (Altmann, 1992; Newell, P., Lehman, Altmann, Ritter, & McGinnis, forthcoming) indicates that most of the time these models do not apply many operators in a row before subgoaling, and instantiate nearly as many problem spaces as they do operators. After much worry and concern about how what happens when operators walk out the rear of problem spaces, it does not seem to happen all that often. Indeed, only two systems (Red-Soar: (Johnson & Smith, 1991), Able-Soar, Jr.: (Levy, 1991; Ritter, 1992)) have seriously overrun the current limitation of being able to display four or five operators in a problem space before they are no longer graphically in the triangle.

Several models do perform explicit search as part of their behavior. Sched-Soar, Rail-Soar, NL-Soar, and Groundworld, at least, sometimes do it. For example, part of the structure of Sched-Soar's search can be seen in Table 6-25 and Figure 6-27. Other models do not perform any search on the problem space level. If the operator support displays for Browser-Soar are examined (the Appendix to Chapter 7), one can conclude that Browser-Soar's behavior is routine (and this is indeed what Peck and John intended and claim). The operators are applied in a very orderly way. A system that was performing search that depended on the information it found would presumably be less regular.

Table 6-28 presents other possible measures for characterizing behavior as search: the number of operators, the number problem spaces, and instantiations of operators and problem spaces over a typical task episode (as defined by their authors) for several Soar models. In each case, the number of different operator types in each problem space is relatively small (the largest average ratio is approximately 4 operators per problem space in Red-Soar), and the average number of instantiated operators per instantiated problem space is small too.

The proportion of goals that are operator no-changes are shown for each of the programs in Table 6-28. Several of these programs do use lookahead search, but the ratio of operator no-change impasses suggests that these programs are not spending a substantial amount of their effort performing lookahead search.

There are also some unusual, very non-search-line behaviors exhibited by the models in Table 6-28.

Table 6-28: The number of operators, problem spaces, and instantiations of these per run for several Soar models.

Model	Descriptive			Instantiations					
	Ops	Spaces	Ratio (ops/ps)	OP nc goal ratio	Ops	Spaces	Ratio (ops/ps)	Max (ops/ps)	Space
Browser-Soar	31	18	1.72	0.87	238	52	4.57	7.22	Evaluate-items-in-window
Groundworld	33	15	2.20	0.92	531	74	7.17	179.50	Wait-external
Liver-Soar	55	20	2.75	0.72	208	44	4.72	15.66	Check-features
MFS-Soar	69	23	3.00	0.96	347	92	3.77	6.33	Input-variable
NL-Soar	18	8	2.25	0.56	122	52	2.34	6.33	Check-constraints
all learned	18	8	2.25	0.50	38	2	19.00	37.00	Comprehension
NTD-Soar	42	11	3.81	0.93	779	73	10.67	24.40	Sqagr
Rail-Soar	25	13	1.92	0.73	233	48	4.85	8.00	Eval-state
Red-Soar									
plain episode	107	27	3.96	0.94	1258	130	9.67	154.00	Rule-out
"Searchy" episode	109	29	3.75	0.86	923	126	7.32	81.50	Match-hyp-to-antigram
Sched-Soar	11	4	2.75	0.69	866	187	4.63	3.25	Analyze

Red-Soar uses 154 operators in the rule-out space to check constraints when typing blood. The goal stack and pscm-stats in the SX graphic display indicate that Groundworld (Stobie et al., 1992) performs one 9-step look ahead search, and then waits for approximately 270 operators. It is a program designed for a continued existence, and can keep running after its initial task is finished. NL-Soar, after it has learned a sentence, performs rather differently from its initial behavior. The "expert" behavior has no search whatsoever, and directly applies a series of 37 operators to understand the 10 words used in the example sentence.

Examination of the visual displays of these models suggests that they can best be characterized as a set of behaviors, including search *through* problem spaces, hierarchical decomposition of problem solving, as migrating and combining knowledge sources, and as search within a single problem space. In a fully learned Soar model, actions just happen automatically in the top space, which is not a search space at all then. The problem spaces used for search have disappeared. Search may remain on other levels. There may be the results of previous searches guiding behavior that can be seen as degenerate search; there may be search going in the external environment; there may be search being performed in the Rete net to find which productions to fire. But in many cases there is not search being performed on the problem space level. These other searches are not wrong, but they must be included in the explanation of behavior of Soar models.

6.5.3 Soar models do not have explicit operators

Problem spaces and their objects, such as operators, do not exist in Soar models in an explicit sense. Within a running Soar model, neither the model nor the modeler can obtain a list of all the problem spaces and operators that exist. They are only available to an observer (including the model itself) by watching the system perform over time, and a history of their appearance and use is not saved automatically (except by the SX graphic display).

The "operators" (or any problem space level object) that are selected for application are not Operators (capital O). A chain of the same operator in the graphic display, all in a row, illustrates that the Operator is not being applied, but instantiations of it are being created and applied. If the same operator was being applied, then a chain would not be an appropriate metaphor, but a moving dot would be. Operator preferences may really be preferences for a given operator, but perhaps they should be seen as operator instantiation preferences. Or how else could you prefer *add(3 4)* over *add(5 6)*? Both appear to be the *add* operator.

What is selected then? The objects selected are instantiations of a semantic object, or object instances generated by an implicit generator. Both the semantic operator and the operator generator are not available for inspection on the symbol level. They are knowledge level objects, and can only be manipulated on that level. The symbol level, which the Soar architecture provides, can only approximate them, and can only obtain them through effort and observation.

This difference between operators and operator instantiations may seem small, but it is necessary to disambiguate these differences for automatic model testing. When aggregating the results of testing the model the objects that are supported must first be identified, and represented across runs of the model. The instantiations are not the theoretical level objects they are often mistaken to be, and cannot aggregate support. Identifying architectural objects is also necessary to display them.

6.6 Summary

The Developmental Soar Interface supports creating models in Soar by treating model building more like an AI programming task. Users can load and run code more directly, manipulating productions as productions, rather than as portions of plain text. By integrating a Soar process within the editor, the textual representation of productions can be quickly augmented with features found only in the process, such as how well a production matches the current goal stack and its contents.

The DSI has added several key ideas to building models in Soar. The first is that the theoretical constructs of Soar models should be displayed. The SX graphic display provides a visual description of the model's structure and behavior over time, and the improved trace provides a better linear description. By aggregating the ephemeral trace over time, the SX graphic display can infer the structure of the problem spaces.

The second is that the user should be able to directly manipulate the theoretical structures. The two structured, integrated editors provide commands for creating, evaluating, and examining models in various ways on the production or TAQL construct level. The SX graphic display provides the ability to examine objects on the PSCM level, but not the ability to create them.

Implementing and using the DSI has provided some lessons on Soar programming languages on the behavior of Soar models. Implementing an aid for TAQL programming pointed out the relatively large size of the TAQL language. Using the graphic display has pointed out two features of Soar models that are more accessible with a graphic display of Soar model's behavior. First, that Soar models include other types of behavior than just search in problem spaces. Second, that within a Soar model, its basic structures, problem spaces level objects, do not exist in an explicit form. Users and systems that want to manipulate Soar models will have to create their own representations of them.

Remaining problems with the DSI. The main components of the DSI represented different levels of support for the user and had different levels of success. The two editors, Soar-mode and TAQL-mode, are well received, and will continue to be used by a large part of the community given normal software maintenance. The current version of the graphic display of the model's behavior and structure has several problems that will have to be fixed for it (or by future systems) to truly useful.

States remain essentially untraced. This is a problem both for testing predictions against protocols and for basic model building. What the necessary information is, how to let the user represent it, and how to provide it succinctly, remains a high priority design issue. Implementing the trace once it has been designed is probably straightforward.

Users have requested several extensions to SX display. These include the ability to remove working memory elements and to show how a single production matches over time, but the largest problem with the SX graphic display has been speed. This is the largest acceptability issue that the graphic display has faced. People who do not use it, do not use it because it unacceptably slows down Soar. It has only been truly acceptable where speed is not an issue, such as for teaching novices and for demos, but the lack of later acceptability has even encouraged some novices to not start to use it. As the Soar

architecture gets implemented in C, this system should get duplicated in C, but it is unclear that the relative slow down will not also occur there. Any display, but particularly graphic ones, may always offer at best a two-to-one slow down compared with the underlying application (Myers & Rossen, 1992).

Several graphic design issues remain. The dynamic structures of Soar in the goal stack are all represented fairly well. How to represent several of the static structures remains a problem, for example, how to nicely display the operators in a space; we use a simple way for chunks, can we find a similar one for operators? Representing the states that exist in a problem space suffers from a similar problem.

Finally, can we tie creating and editing productions to the graphic display? The ability to click on chunks and examine them has proved useful in exploring the types of chunks that end being assigned to *Every-Space*. Being able to go between a graphic and textual representation is appealing.

III Performance demonstrations of Soar/MT and Conclusions

Chapter 7

Performance demonstration I: Analyzing the Browser-Soar model faster and more deeply

Browser-Soar (Peck & John, 1992) is a model of a user using an on-line help system. Ten episodes totalling approximately ten minutes of a single subject's behavior have been used to test it. This chapter examines Browser-Soar in detail, duplicating and extending the previous set of analyses. By choosing to duplicate and extend an existing analysis, it includes a set of analyses known to be useful, and provides a reference point for measuring its speed and discrimination.

Soar/MT allows the sequential predictions of Browser-Soar to be tested more quickly and at a finer level than can be performed by hand using sheets of paper and a plain spreadsheet (Excel) to hold the correspondences. Displays showing the fit of the data to the predictions can be easily created, and provide additional insight for characterizing the model and how to improve its predictions.

Browser-Soar provides predictions of when structures enter working memory. This allows testing the sequentiality assumption of verbal protocol theory, that mental structures are reported on in the order that they appear in working memory. This assumption is found to hold for verbal utterances. Sequentiality can also be tested for mouse actions and they too are performed in order. However the verbal utterance and mouse action information streams do not initially appear to be sequential with respect to each other. The most likely cause for this discrepancy is that an approximation in the interpretation and alignment process was used. The data in the two information streams should be considered sequential. When this is done, the overt actions provide fixed reference points for computing the lag of the verbal utterances.

With a measure of the model's performance and fit in hand, a small modification of Browser-Soar suggested by the measures of fit is attempted, removing some problem spaces that might be redundant. This change does not drastically improve the fit, and this is shown clearly in the analytic displays. The resulting model, however, is more parsimonious with the effects of learning.

In nearly every case the results reported here duplicate what Peck and John already know about Browser-Soar, but they come at less expense and can be shown more compellingly with Soar/MT's displays.

7.1 Description of Browser-Soar and its data

While Browser-Soar and its data are explained fully elsewhere (Peck & John, 1992), an overview is presented here with particular emphasis on the aspects of the data and model that receive attention in this reanalysis. Because Peck and John have generously allowed me access to their original data, I am able to include additional descriptions of the data here.

Description of the data. The Browser-Soar data used to test Browser-Soar was gathered from a single user interacting with the cT programming environment on the Macintosh computer (Sherwood & Sherwood, 1984; Sherwood & Sherwood, 1992) to perform her own task arising out of her work, creating a graphing program for her own use. She was a non-professional but experienced computer programmer who had never used cT before the experiment. The episodes of interest occurred when she used the on-line help system to learn about cT. The data that Browser-Soar is tested with represents only a portion of the 85 episodes using the help browser that occurred during the three and one-half hours of behavior that was videotaped. A portion of the remaining data was used informally to help create the model.

The first four episodes were chosen to cover a wide variety of browsing behavior, and the remaining six were chosen randomly from all the browsing episodes that were videotaped. Each episode represents a different environment and goals.

As noted in Table 7-30, the ten episodes averaged 56 s in length and included a total of 58 verbal and non-verbal segments. Each episode included a mean of 126 words. This was not reported in the original analysis because computing the number of words per episode is something that is not easy to do, at least given a journeyman's familiarity with Excel.[8] The subject's behavior provided a high density of data, on average, over an action or utterance every second, and a relatively high verbalizations rate of 146 words per minute.

Three information streams from the subject were transcribed by hand into Excel spreadsheets, the verbal utterances, the mouse movements, and the mouse clicks. Verbal utterances were broken into separate segments when pauses of more than 100 ms occurred.

There are three special features of the Browser-Soar data worth noting. The first is that the types of mental information reported is small. The user generally only mentions search criteria, evaluation criteria, and the words that she is reading. Second, this is not a hard problem. In contrast to many tasks that have been modeled, using the computer interface is routine behavior for the subject and their internal representation of the task is not changing. Finally, there is what will turn out to be a useful mix of overt, necessary task actions (mouse actions) with verbal statements. The overt, motor actions will help disambiguate the verbal, and vice-versa.

Description of the Browser-Soar model. Figure 7-33 depicts the problem spaces in Browser-Soar and their relation to each other as drawn with the SX graphic display. Figure 7-34 presents the problem spaces as depicted by Peck and John (1992). All the problem spaces are related by operator no-change impasses except the *Selection* problem space, which is used to resolve operator ties in the *Find-criterion* space. Browser-Soar does not reuse any problem spaces, so the maximum goal depth will be six, not counting the top-goal.

The Soar learning mechanism is not turned on in the Browser-Soar model. Peck and John (1992) argued that there will be little learning observable in this set of tasks. The user is either performing as an expert, that is, will not be learning how to move the mouse, or is learning items that will not transfer between trials, such as how to print out a variable's value.

Based on a sample run (the "Write" episode, the subject was seeking information about writing information onto the cT screen) and assigning the productions to problem spaces graphically, the problem space level statistics function in the SX graphic display reports that there are a total of 18 problem spaces and at least 31 operators. The statistics based on a complete run are shown in Table 7-29.

Browser-Soar is actually a short progression of models based on testing and modify it with the ten episodes. During this progression Browser-Soar remained rather stable. Between the first and the tenth episode, two operators were added to Browser-Soar, and four operators application conditions were changed. Because there are so few adjustments, in this analysis Browser-Soar can be treated as a single program.

Comparing the listing of problem spaces in Table 7-29 with Peck and John's (1992) listing, it appears that either they do not include all the Macintosh method problem spaces, or the organization of Browser-Soar has changed since it was reported. Table 7-29 includes an additional operator more than reported by Peck and John (1992) (probably several more, because four of the *Mac-method-** problem spaces also would have operators). This missing operator could be the *Browsing-task* operator itself, or it could be an operator used in two spaces, which the SX graphic display would count twice. The difference in object counts is compounded by Peck and John's treatment of the model. They knew that they did not have an adequate model of reading, and did not attempt to match the model's behavior below reading the whole screen.

---

[8]Macros can, however, be created to perform this task (Schroeder, 1992).

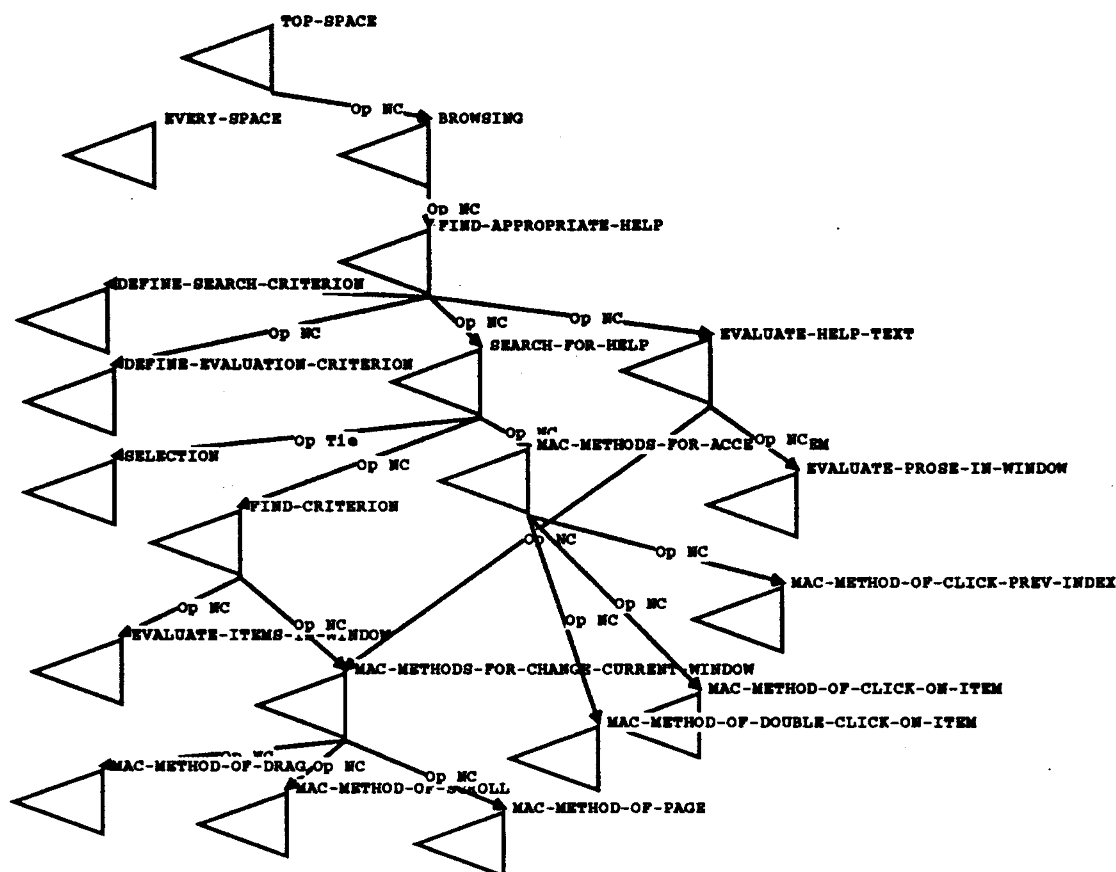


Figure 7-33: The problem space organization of the 19 problem spaces in Browser-Soar generated with the SX graphic display.

The static structure depicted in Figure 7-33 shows the normal dynamic selection and use of the problem spaces. It was created by loading Browser-Soar and running an episode. The problem spaces that were created were then rearranged from their location on a grid to the tree structure shown in the figure, connected together by hand, and annotated. Their organization was written out so that this structure could be used again.

Figure 7-35 shows the goal stack in Browser-Soar at decision cycle 17 of the Write episode. The selection and use of problem spaces moves roughly from top to bottom and left to right. At the start of the browsing episode, the Browsing space is selected and the *Find-appropriate-help* operator is applied. This cannot be directly implemented, so the *Find-appropriate-help* problem space is selected. Within this problem space, the operators *Define-search-criterion, Define-evaluation-criterion* are called to initialize the search. Both of these operators cannot be directly applied, and similarly named problem spaces are used to implement them.

The *Search-for-help* operator is applied once the search and evaluation criteria are defined. This operator also cannot be directly implemented, and the *Search-for-help* problem space is selected. Within this problem space, operators (and corresponding problem space to implement them) are applied to search the help screen (*Find-criterion*), and to select an interesting item to read about if it is found (*Mac-methods-for-accessing-item*). When searching for interesting items, the *Find-criterion* problem space uses two operators, one to evaluate items in the window, and one to scroll the screen

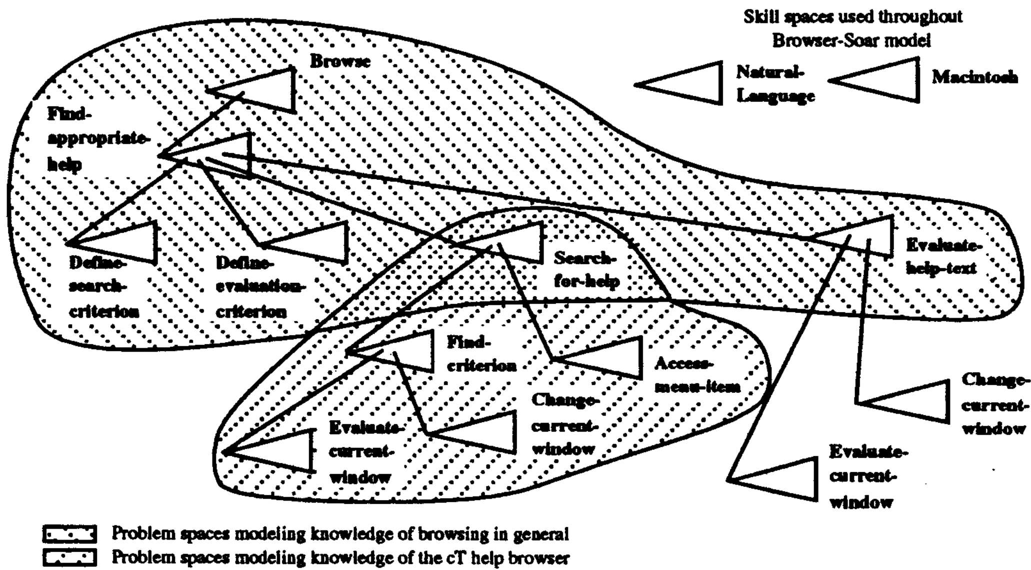


Figure 7-34: The problem space organization of Browser-Soar taken from Peck and John (1992).

when the end of the screen of items is reached (*Mac-methods-for-changing-window*). Both operators are implemented in their own space.

Once an item has been found and selected by clicking on it, the *Evaluate-help-text* operator is selected in the *Find-appropriate-help* problem space. This operator is implemented in its own space using two operators. The first operator selected will be to evaluate the help text by reading it. The other operator is the same scroll used to scroll the window of items to select from.

When they did their analyses, Peck and John grouped two of the Soar operators in *Evaluate-items-in-window* problem space that implement reading the computer screen into a higher level operator, *Evaluate-current-window*, not shown in the automatically derived figures. In Peck and John's operator support displays, the low-level operators, such as *Read-input*, do not appear, for all the coding was based on the higher level operator. This coding scheme was duplicated in the later analyses that are reported here, except that the lower level operators and problem spaces do appear in the automatic display and aggregate model statistics.

Figure 7-36 shows the number of productions used to implement each problem space. Approximately 430 productions are generated from the 193 TAQL constructs used to create these problem spaces. The exact numbers varied slightly between episodes. When the productions were sorted into problem spaces, a problem space was found for most productions. Productions and TAQL constructs that are included as part of Soar's default knowledge are not included in the counts or the display.

Productions without a problem space name directly in their condition were assigned to *Every-space*, 29 in all. *Every-space* is used to display productions that could fire in every space. Examining these with the graphic display indicated that 15 of them are for proposing new problem spaces based solely on the goal and its superstate, 12 are internal TAQL productions, one is used to note that all search-for-help operators are equivalent, and one prints out the search criteria whenever it changes.

The number of productions associated with each problem space is an approximate measure of the amount of knowledge in each problem space. One of the reasons this measure is approximate is because TAQL uses a production for each of its state edits. Only the user's productions are included in this display, so the lack of productions associated with the *Selection* space means that it only uses the

Table 7-29:
Problem space level statistics for the "Write" episode. The top block presents the problem spaces and operators represented in the graphic display. The selection counts for each goal, problem space, state, and opeator are presented in their hierarchical calling order.

```
PSCM Level statistics on November 27, 1992

18 problem spaces, with a total of 31 operators.
Ops  Problem space
 1   top-space
 1   browsing
 5   find-appropriate-help
 2   define-search-criterion
 2   define-evaluation-criterion
 2   search-for-help
 3   find-criterion
 2   evaluate-items-in-window
 1   mac-methods-for-change-current-window
 3   mac-method-of-scroll
 1   mac-methods-for-access-item
 2   mac-method-of-click-on-item
 3   evaluate-help-text
 3   evaluate-prose-in-window
 0   mac-method-of-drag
 0   mac-method-of-page
 0   mac-method-of-click-prev-index
 0   mac-method-of-double-click-on-item


The actual selection counts and calling orders:
 1 G: g1 (g1)
 1 .P: top-space (top-space) (3 chunks)
 1 . S: s5 (no name)
 1 . O: browse (browse)
 1 .  G: (operator no-change) (g19)
 1 .  .P: browsing (browsing) (16 chunks)
 1 .  . S: s39 (no name)
 1 .  .  G: (state no-change) (g3145)
 1 .  .  .G: (goal no-change) (g3152)
 1 .  .  . G: (goal no-change) (g3159)
 1 .  .  .  G: (goal no-change) (g3166)
 1 .  .  .  .G: (goal no-change) (g3173)
 1 .  .  .  . G: (goal no-change) (g3180)
 1 .  . O: find-appropriate-help (find-appropriate-help)
 1 .  .  G: (operator no-change) (g43)
 1 .  .  .P: find-appropriate-help (find-appropriate-help) (55 chunks)
 1 .  .  . S: s59 (no name)
 1 .  .  . O: define-search-criterion (define-search-criterion)
 1 .  .  .  G: (operator no-change) (g65)
 1 .  .  .  .P: define-search-criterion (define-search-criterion) (30 chunks)
 1 .  .  .  . S: s79 (no name)
 1 .  .  .  . O: generate-search-criterion((write)) (generate-search-criterion)
 1 .  .  .  . O: evaluate-search-criterion (evaluate-search-criterion)
```

(continued on next page)

default productions provided with Soar. It appears that it takes a minimum of three user productions to create a usable problem space.

Browser-Soar interacts with a simulation of the cT help browser. The simulation provides Browser-Soar with the contents of each window in the browser. The simulation does not take into account the length of time a mouse is held down; on each mouse click it scrolls to the same place the subject scrolled to in the same situation. If the model were to scroll in the wrong direction (which it no longer does, and perhaps never did), it would be up to the analyst to catch this.

Description of original Browser-Soar analyses. Peck and John's (1992) originally performed the alignment by hand, aggregating the correspondences into summary measures for each episode and for each operator. They used limited graphic displays of the alignment, relying mostly on a tabular representation. Their analysis also included a picture of the Browser-Soar problem spaces drawn by hand in MacDraw (their Figure 3).

Table 7-29: Problem space level statistics for the "Write" episode (concl.).

```
1  . . . . O: define-evaluation-criterion (define-evaluation-criterion)
1  . . . .  G: (operator no-change) (g103)
1  . . . . .P: define-evaluation-criterion (define-evaluation-criterion) (17 chunks)
1  . . . . . S: s117 (no name)
1  . . . . . O: generate-evaluation-criterion((value-of-something)) (generate-evaluation-criterion)
1  . . . . . O: evaluate-evaluation-criterion (evaluate-evaluation-criterion)
2  . . . . O: search-for-help (search-for-help)
2  . . . .  G: (operator no-change) (g1025)
2  . . . . .P: search-for-help (search-for-help) (17 chunks)
2  . . . . . S: s1043 (no name)
2  . . . . . O: find-criterion(keyword) (find-criterion)
2  . . . . .  G: (operator no-change) (g1049)
2  . . . . . .P: find-criterion (find-criterion) (27 chunks)
2  . . . . . . S: s1066 (no name)
2  . . . . . . O: focus-on-current-window (focus-on-current-window)
9  . . . . . . O: evaluate-current-window (evaluate-current-window)
9  . . . . . .  G: (operator no-change) (g2415)
9  . . . . . . .P: evaluate-items-in-window (evaluate-items-in-window) (85 chunks)
9  . . . . . . . S: s2432 (no name)
65 . . . . . . . O: read-input (read-input)
65 . . . . . . . O: attempt-match(12504) (attempt-match)
7  . . . . . . O: change-current-window (change-current-window)
7  . . . . . .  G: (operator no-change) (g2339)
11 . . . . . . .P: mac-methods-for-change-current-window (mac-methods-for-change-current-window) (34 chunks)
11 . . . . . . . S: s3042 (no name)
11 . . . . . . . O: scroll(help-text) (scroll)
11 . . . . . . .  G: (operator no-change) (g3054)
11 . . . . . . . .P: mac-method-of-scroll (mac-method-of-scroll) (21 chunks)
11 . . . . . . . . S: s3070 (no name)
4  . . . . . . . . O: move-mouse(help-text down) (move-mouse)
11 . . . . . . . . O: press-button (press-button)
11 . . . . . . . . O: release-button (release-button)
2  . . . . . O: access-item(keyword) (access-item)
2  . . . . .  G: (operator no-change) (g2527)
2  . . . . . .P: mac-methods-for-access-item (mac-methods-for-access-item) (4 chunks)
2  . . . . . . S: s2542 (no name)
2  . . . . . . O: click-on-item(12537) (click-on-item)
2  . . . . . .  G: (operator no-change) (g2548)
2  . . . . . . .P: mac-method-of-click-on-item (mac-method-of-click-on-item) (5 chunks)
2  . . . . . . . S: s2562 (no name)
2  . . . . . . . O: move-mouse(keyword unspecified) (move-mouse)
2  . . . . . . . O: click-button (click-button)
2  . . . . O: evaluate-help-text (evaluate-help-text)
2  . . . .  G: (operator no-change) (g2576)
2  . . . . .P: evaluate-help-text (evaluate-help-text) (26 chunks)
2  . . . . . S: s2592 (no name)
2  . . . . . O: focus-on-help-text (focus-on-help-text)
6  . . . . . O: evaluate-current-window (evaluate-current-window)
6  . . . . .  G: (operator no-change) (g3104)
6  . . . . . .P: evaluate-prose-in-window (evaluate-prose-in-window) (69 chunks)
6  . . . . . . S: s3122 (no name)
6  . . . . . . O: read-input (read-input)
6  . . . . . . O: comprehend (comprehend)
6  . . . . . . O: compare-to-criteria (compare-to-criteria)
4  . . . . . O: change-current-window (change-current-window)
4  . . . . .  G: (operator no-change) (g3027)
11 . . . . . .P: mac-methods-for-change-current-window (mac-methods-for-change-current-window) (34 chunks)
11 . . . . . . S: s3042 (no name)
11 . . . . . . O: scroll(help-text) (scroll)
11 . . . . . .  G: (operator no-change) (g3054)
11 . . . . . . .P: mac-method-of-scroll (mac-method-of-scroll) (21 chunks)
11 . . . . . . . S: s3070 (no name)
4  . . . . . . . O: move-mouse(help-text down) (move-mouse)
11 . . . . . . . O: press-button (press-button)
11 . . . . . . . O: release-button (release-button)
1  . . . . O: change-search-criterion (change-search-criterion)
```

The model trace and protocol were first printed out and interpreted and aligned by hand, with the correspondences and annotations entered into an Excel spreadsheet. Over the course of testing Browser-Soar with the ten episodes, few changes to the model were required. The first episode was used to create the initial model, and during testing of the next three episodes four additional operators were added and two were modified. During the analyses of the last six, the only changes required of the model were modifying two of the operators.

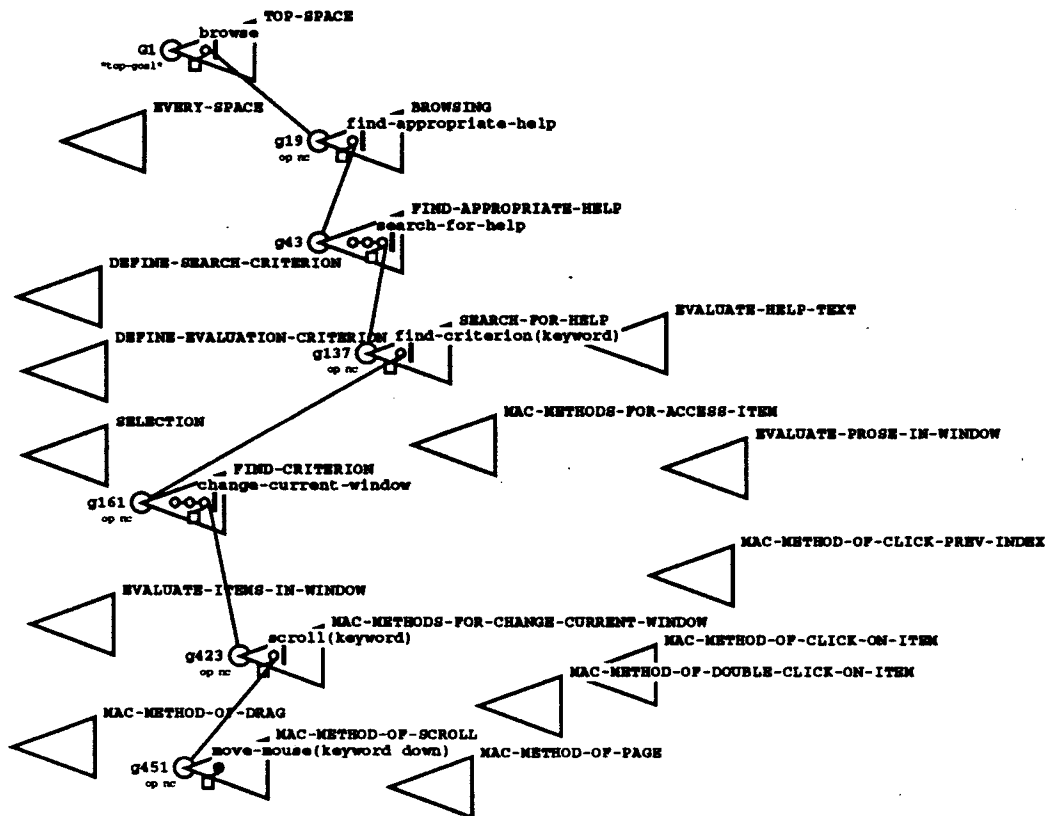


Figure 7-35: Browser-Soar during a run.

All the interpretation was done with respect to operator applications. This included overt, task actions necessary to perform the task, and internal mental actions necessary for deciding what to do and for understanding. The verbal utterances were interpreted with respect to internal operators or their results. Mouse clicks were always interpreted with respect to overt task actions. Mouse movements could correspond to either. When the model predicted that they were required to perform the task and they were used in a task specific way, they were interpreted as overt task actions. When the model did not predict their use, and the mouse pointer could be interpreted as over some part of the display currently being used or read, they were treated like eye-movements and interpreted with respect to an internal operator.

Peck and John's major analyses were to aggregate how many of the subject's behaviors were predicted by the model's actions, aggregating separate measures for directly observable operations, such as mouse clicks, and mental operators that are only observable through verbal protocols or movements of the mouse over words on the screen. Over 90% of the subject's actions and utterances were accounted for by the model's predictions, and the fit between data and predictions was judged to be very tight. These computations were computed by hand for each episode.

The percentage of operator predictions supported by the data were also computed. At 15% this initially appears to be a low rate. One must keep in mind that the trace of the Browser-Soar model provides more predictions than can be tested, even given the rich verbal and non-verbal data streams used to test it. Across all episodes they found indirect evidence for 57% of the operators that could not be directly observed.

An operator support display drawn by hand in MacDraw (their Figure 4, our figure 2-7) illustrated the

Figure 7-36: Browser-Soar problem space organization with productions shown by their problem space.

tightness of fit, but this was not done for all episodes because it took approximately a day to produce (Peck, 1992).

7.2 Producing richer analyses more quickly

This demonstration of the Soar/MT environment must show that it is possible to duplicate the previous tests of Browser-Soar, and that new, more useful analyses can be performed as fast or faster than have been done in the past. Supporting the analyses is the most important though, speed can come from faster machines or revisions of the software.

This demonstration will not include all the possible analyses that could be or were done by hand for Browser-Soar or for other process models, but it should be clear from this example that the analyses that were not performed are no harder, and are quite likely more easily performed with Soar/MT than by hand.

The emphasis of the analyses will be Grant's (1962) emphasis, finding out where to improve the Browser-Soar, not that all the improvements will be incorporated in this demonstration.

7.2.1 The interpretation of data with respect to the model trace done faster and tighter

Using the Soar/MT environment allowed the interpretation and alignment step to be performed more quickly. The first four episodes were used to debug the Soar/MT system. For later episodes, either analyst (FER or VAP) could go once through the TBPA loop, deriving the predictions, aligning them, and creating the global displays in 20 minutes to 2.5 hours, depending on the pre-existing degree of

alignment, length of the episode, and desired level of detail. The amount of time to analyze another episode is now much less than the initial ten hours needed to understand one.

In each episode the unambiguous data was aligned first. This took on average a minute to set up. During the 30 minutes it would take to rearrange the cells the analyst did not need to be present. The verbal protocols would then be partially interpreted, their locality would be bounded by the matched non-verbal actions near them. A complete listing of the analyses' results are shown in Table 7-30, and the visual, analytic measures created for each display are included as the Appendix to this chapter.

After two episodes of observing me work and working jointly, Virginia Peck (VAP), analyzed three episodes on her own except for creating the displays of model fit. She took approximately 100 minutes to perform these analyses from producing the trace to interpreting the data. Her time was a limited resource, and the software I was most interested in testing was the alignment capabilities, so I created the displays based on her alignments. She also attempted to analyze the last data set, Zcommand, but the unusual size (it is the largest episode by approximately a factor of two) disclosed some bugs in the Spa-mode.

The Card2 algorithm worked admirably. Across the ten episodes it correctly aligned all of the 296 predicted unambiguous mouse actions (mouse clicks and mouse movements). The ability of the algorithm to adjust the edit-list to align a predicted action with the last action in a series of similar subject actions substantially contributed to this performance. Without that modification the results would have been less, around 90%. For each episode the edit list used to align the two meta-columns was generated in under a minute. The alignment of the data with the predictions then took approximately 30 minutes for the Write episode. This alignment process does not require intervention of the analyst. If the two information streams were partially aligned this took less time. A single trial with a single subject on the Write episode, an average sized episode, took approximately 45 minutes to align by hand with Excel. Longer episodes take more than proportionally longer in Excel (Peck, 1992), up to several hours.

After the Card2 auto-alignment algorithm was run, the analyst (FER, VAP, or both) would go through the Spa-mode spreadsheet interpreting the remaining data with respect to the model's predictions. Because both data streams were completely included in Spa-mode, this resulted in a tighter match between the two information streams. Each correspondence included a line of Soar trace (including the decision cycle, the context element selected, and any traced substructures), instead of a coded operator name. These alignments included in the display Soar actions not matched. Figure 7-37 provides a partial example, and the appendix to this chapter includes a complete example for the Write episode.

These alignments generated in Spa-mode provide a more telling comparison of the predictions with the data. Including both data streams in a tabular display shows gaps where the model performed more or less actions (and thus took more or less time) than the subject. When we viewed the first episode aligned this way, we were somewhat surprised by the amount of Soar trace not aligned with subject data. It is also easy to find mismatched actions in this display. False alarms, actions by the model not matched by the subject's actions, which are not representable when the model's predictions are not directly included, can also be represented in Spa-mode. It remains slightly difficult to compare and aggregate the comparisons between episodes with this spreadsheet representation because of the large number of sheets of paper and dispersion of information across them.

7.2.2 Operator support displays created automatically -- as a set they highlight periodicity in behavior

An operator support display for each episode was generated automatically from the alignment data in the spreadsheets. These displays are shown in the appendix to this chapter as Figure 46, along with the displays for a modified model and episode called Better-array, which is explained later in this chapter. These displays originally took approximately a day to produce, so only four were created in the initial analysis (Peck, 1992).

```
    T MOUSE ACTIONS       WINDOW ACTIONS          VERBAL  ST #  MTYPE MDC  DC                      Soar Trace                                 Comments
  +--+---------------+----------------------------+---------+--+----+---+--+----------------------------------------------------------+----------+
180                                                                        91 . . . . . . O: attempt-match ()
181                                                                        92 . . . . . . O: read-input (scalex)
182                                                                        93 . . . . . . O: attempt-match ()
183                                                                        94 . . . O: access-item (hierarchical)
184                                                                        95 . . . ==>G: g676 (operator no-change)
185                                                                        96 . . . . P: mac-methods-for-access-item ()
186                                                                        97 . . . . S: s691 ()
187                                                                        98 . . . . O: click-on-item (1606)
188                                                                        99 . . . . ==>G: g697 (operator no-change)
189                                                                       100 . . . . . P: mac-method-of-click-on-item ()
190                                                                       101 . . . . . S: s711 ()
191 50 M(-y) 1 line to 'scalex' in "scalex Setting the Sc m 19   mr 102 102 . . . . . O: move-mouse (hierarchical unspecified)
192 50 C("scalex Setting the Scales")                   b 20   mba 103 103 . . . . . O: click-button ()
193                   mouse pointer to watch
194                                                                       104 . . O: evaluate-help-text ()
195 59                'scalex' help text appears                          105 . . ==>G: g725 (operator no-change)
196 59                         Let's look at 'scale-x'. v 21     v  94
197                   "scalex Setting the Scales" to bold
198                   mouse watch to pointer
199                                                                       106 . . . P: evaluate-help-text ()
200                                                                       107 . . . S: s741 ((accessed scalex) (mark-and-label-axes))
201 60                                                                    108 . . . O: focus-on-help-text ()
```

Figure 7-37:

Portion of the alignment of the protocol and model trace from the Axis episode. On each row: T is time of subject's actions in seconds. MOUSE ACTIONS is any mouse action. WINDOW ACTIONS are any responses from the actual cT system that the subject saw. ST is the segment type. VERBAL is any verbal utterances by the subject. # is the segment number. MTYPE is type of match, MDC is the decision cycle matched, DC is corresponding Soar decision cycle. Soar Trace holds the model's predictions.

Each display provides a visual depiction of the operator applications for the episode modeled, along with the support each operator received, if any, from corresponding verbal utterances, move actions necessary to perform the task, and mouse movements over screen items that were read. The indentation of the operator names corresponds to their problem space level, and roughly to which problem space they belong to.

For each episode. Individually the operator support displays indicate for each episode the level of support for the model's operators in that episode. Figure 7-38 shows the operator support display for the Write episode. It shows that most of the subject's actions could be interpreted by the model's actions. The verbal utterances mostly match the *Evaluate-current-window* operator, as do the mouse movements that are not required to perform the task.

We also can start to see that the subject's performance shows a definite periodicity. The cycle of evaluating a window, changing it through scrolling by moving the mouse and then clicking on the scroll bar occurs 13 times, with some variations. On the third cycle of examining help topics, the subject sees something that changes her search criteria. On the ninth cycle, she finds a topic that matches the criteria she was looking for, and selects an item for examination. On the remaining cycles she examines the help window. So the main loop is based on menu interaction, and there may be a secondary loop of revision of the search criteria. Just this episode is not enough to tell.

Ohlsson (1980) noted that he could find regularities in protocols that covered a shorter period of time than Newell and Simon (1972) used (200 s versus 1,000 s). This display shows that regularities can occur over shorter time periods. The point is getting enough data, not time. In this domain, in addition to verbal utterances, the mouse movements and mouse button presses help provide the required data density.

A few of the subject's actions could not be interpreted, and they are shown on the bottom as corresponding to the NOT MATCHED operator. Just examining the surface of this display does provide any insight into why they were unmatched, although two of the mouse movements appear to come after a click button operator. When the points and their context are examined by clicking on them (or by finding them in the original spreadsheet, but this is more work), the first is found to be a random mouse movement to a position that is not over something being read or in anticipation of a later click or move, the second the subject laughing, the third another random mouse movement, and

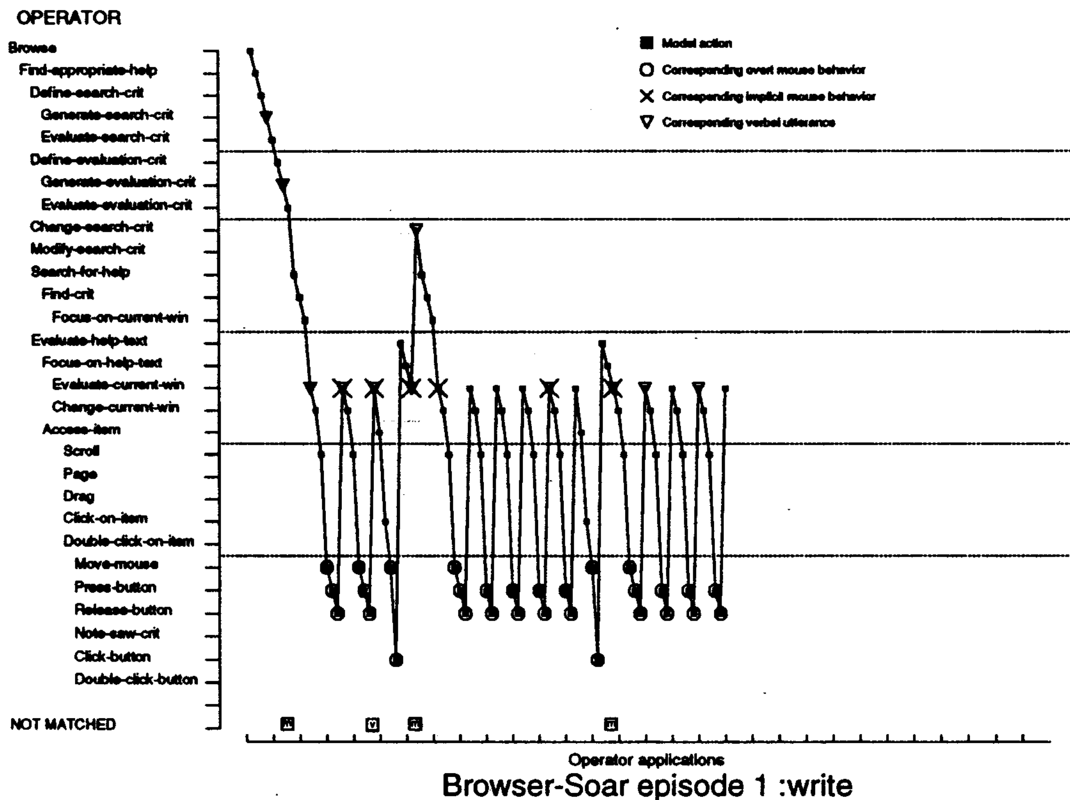


Figure 7-38: Operator support display for the Write episode.

finally, a movement that is interpreted as a mistake. The last mouse uncoded action is a mouse movement that falls short of a scroll bar, and is soon followed by a mouse movement to the position the model predicts.

Across episodes. As a group the ten operator support displays (included as Figure 46 in the appendix to this chapter) tell us even more, and the reader is encouraged to examine them before proceeding. The largest effect visible when viewing these en masse is the periodicity. The longest episode, Zwrite, looks like the display of an oscilloscope indeed.

When viewed together we also can start to characterize what operators are supported and with what types of evidence. We can see that the subject did not talk about every operation. (Many operators have no mark (∇) indicating a corresponding verbal utterance.) This is predicted by Ericsson and Simon's (1984) theory of verbal protocol production, so this is as expected, and the rate, in quantitative terms is probably acceptable as well. However, it is slightly surprising to see what this looks like, see just how little is said and supported in each single episode. Based on these displays, Browser-Soar appears too small grained indeed, much more detail is provided than in Newell & Simon's models where nearly every production firing could be matched against a verbal segment.

Across the ten episodes, the subject talked about operators that she should have talked about, and did not describe operators that she should not have. Higher level operators, such as setting up the search criteria and evaluating the window contents were often talked about. These operators manipulate verbal representations, so they should appear in the verbal protocol stream. The motor operators for actually scrolling the windows were never mentioned in the verbal protocol, and this is appropriate

given our measurement theory (Ericsson & Simon, 1984), for they would include non-verbalizable operators or information.

What is not verbalized? The *Change-current-window* and its implementation operators *Scroll, Page, Drag,* and *Click-on-item,* were never supported by verbal utterances, nor could they be directly supported by mouse movements or mouse button actions for they are themselves implemented with lower level primitives such as *Click-button* and *Move-mouse*. In the future, they must be considered for removal, unless other evidence, perhaps timing evidence, can be provided for them.

The mouse clicks also appear not to be in working memory. In no episode did the subject report that they were using the mouse. Based on the Soar architecture we would believe that they are motor operations, so we would not expect them to be directly represented. The external motor actions need to be set up, however, and the operator that does this remains unsupported.

New questions these displays raise. In each episode at least one of the operators that set up the search in the cT help browser, the first seven operators below the *Browse* operator, is mentioned at the beginning of an episode. Never are they all mentioned, and eight different combinations appear across the ten episodes. It may be possible to combine or rearrange these operators to provide more consistent support for a single operator or set of operators.

During both the Zwrite and Vars episodes, there is a long period of behavior where nothing is said. Similar periods exist in other episodes but there are verbal utterances and mouse movements to support the intermediate operators of reading the topic lists. Characterizing these periods in some way remains an open problem.

Several problems remain with this display. The indentation of the operator names hints at their hierarchical relationship to each other. The implementation of their relationship remains poorly specified and awkward. Operators can come from different problem spaces, and still appear at the same level in the hierarchy.

7.3 Where the model and subject process at different rates shown clearly

Relative processing rate displays were created automatically from the alignment data for each episode. A complete set of these displays is included in the Appendix to this chapter as Figure 47.

7.3.1 Processing rate display based on decision cycles shows that the quality of fit is high

The relative processing rate display can provide hints about how to improve the model within a single episode. Across episodes it can provide additional hints, and measures of the architecture can start to be taken.

For each episode. As an example, consider Figure 7-39, which shows the relative processing rate display (developed in Chapter 5) for the Write episode of Browser-Soar. Each correspondence between the model's predictions and the subject's actions is noted with a connected symbol. Each correspondence shows the relative times when the model and the subject performed the same action. The time that the subject performed the matched actions is represented in seconds on the x axis, and the time that the prediction occurred in the model's behavior is represented in model cycles on the y axis.

The number and relative linearity of the line of correspondences indicates that the predictions generated by Browser-Soar are relatively well matched by the subject's behavior. The number of unmatched subject segments, placed on the bottom of the display at the time they occurred is a relatively low amount, and there are no overt task actions performed by the model that were not observed in the subject. If there were any, these would go near the y-axis.

The squiggles and sections with extremely high or low slopes show where the fit could be better.

When the line becomes more vertical, it means that the subject has started to perform faster than the model, and when the line becomes more horizontal, it means that the model is performing faster than the subject. In Figure 7-39 both occur.

A regression line is provided to help judge the rate of correspondences, and it is used to provide some additional information as well. Its slope is the relative processing rate for that episode in decision cycles per second. The correlation it computes may be a prediction of how well the model can predict the time course of processing in an episode, but it is likely to show a falsely high correlation. Note that the relationship of decision cycles to seconds (the slope) is well within the range (indicated by the dashed lines at 3 DC/s and 30 DC/s) predicted by the Soar theory.

In each episode the correlation between the subject actions and predictions (measured in decision cycles and operator applications) is fairly high. The values of the slope and r^2 value for each episode are shown in Table 7-30. These values for r^2 values are comparable to well developed single response models (e.g., $r^2 = 0.79$: Thibadeau, et al., 1982; $r^2 = 0.94$: Just & Carpenter, 1985). Browser-Soar is near to making engineering level predictions of human behavior, as has been called for by John (1988).

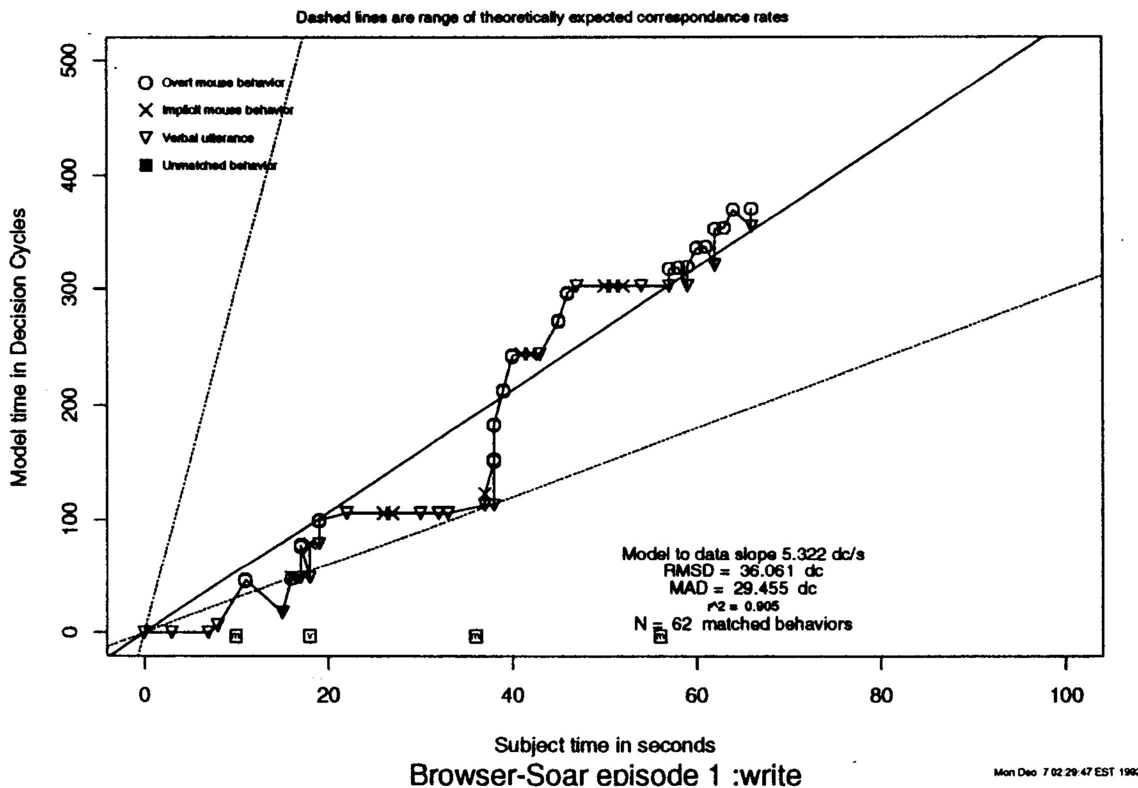


Figure 7-39: Relative processing rates display in decision cycles for the Write episode.

The first parts of display that give specific recommendations on ways to improve the model are the relatively vertical and horizontal sections of the line of correspondences. These sections represent periods where the model and the subject are processing information at relatively disproportionate rates. When the points on the horizontal section between 20 and 40 s are examined by clicking on them, one finds that they all matched to *Evaluate-current-window* operator. This operator is taking much longer for the subject to perform than it does for the model (this operator essentially reads at 100 words/s).

The model could be improved by incorporating a more complete operator to evaluate the current window, that is, read the help text.

The second part of the display to examine is the near vertical line at around 40 seconds. In this section the model is reading every word on a menu while the subject must be skimming the menu's contents, as suggested by the relative rates of processing. Here the model must be smarter about what it is doing, and do less processing than it currently does.

Table 7-30: Summary of raw measures for each episode and regression results.

Episode		Segments	Time (s)	Words N	Rate w/min	Raw DCs	Slope dc/s	DC-r2	Slope op/s	op-r2	DCs/op
1	write	62	66	113	102	399	5.32	0.90	1.18	0.93	4.50
2	unit	40	39	91	140	331	11.29	0.68	2.40	0.80	4.70
3	array	96	68	151	133	517	9.48	0.69	2.26	0.78	4.19
3'	array'	96	68	151	133	346	6.37	0.59	1.49	0.75	4.27
4	precision	21	25	58	139	146	6.82	0.19	1.95	0.34	3.49
5	marker	32	47	162	206	116	2.58	0.43	0.91	0.33	2.83
6	axis	46	83	245	177	173	1.53	0.80	0.45	0.70	3.40
7	labelx	23	34	52	91	77	2.04	0.51	0.61	0.53	3.34
8	circle	52	65	136	125	395	7.32	0.79	1.74	0.78	4.20
9	vars	69	27	44	97	805	35.21	0.90	6.21	0.90	5.66
10	xcommand	140	108	213	118	1529	16.62	0.58	2.76	0.66	6.02
	Sum:	581	562	1265							
	Mean:	58	56	126	146	449	6.92	0.65	2.05	0.67	4.23
	SD:	37	27	68	36	439	4.65	0.22	1.66	0.21	1.32
	Normalized SD:	0.63	0.47	0.54	0.27	0.98	0.67	0.35	0.81	0.31	0.35

From left to right the columns display for each episode the total number of subject data segments, the time of the subject data being modeled, the number of words uttered during the segment and the rate in words per minute, the slope of the least squares regression line on the correspondences in decision cycles per second, the r^2 for that line, the slope of the regression line in operators per second, the r^2 for that line, and the relative rates in the episode of decision cycles per operator. Aggregate measures do not include the Array' episode. Each episode is equally weighted.

Across episodes. Several known problems of Browser-Soar are shown in these displays. Seeing the problems occur in ten episodes is more believable than seeing it in just one episode. Over individual episodes the regression line matched to the correspondences provides a good prediction of the subject's search time. The results of the regression for each episode are shown in Table 7-30.

The rate of the architecture, in decision cycles per second, is slightly slower in Browser-Soar than the Soar theory predicts. Across all the episodes, as shown in Table 7-30, the average rate of decision cycles is six per second. The Soar theory predicts ten per second, plus or minus half an order of magnitude. As this is an average, the actual rate on a single episode can be much lower. This implies that the model is still slightly lean, performing less of the task than the subject is, or that the theoretical analysis of decision cycle rate is slightly high. The first explanation, that the model performing more efficiently or doing less of the task is consistent with but not as far off as other model results (John & Vera, 1992; Newell, 1972; Ritter, 1988; Ritter, 1989; Rosenbloom & Newell, 1982). The large variance in the rate may be cause for some concern, or may just be artifact of the known problems in the *Evaluate-current-window* operator, the *Read-menu* operator, and their ratio in each episode.

In none of the episodes do we find that the line of correspondences is concave upwards, indicating that the subject's relative rate of performance is increasing relative to the model. The displays tend to display the opposite effect, that the line of correspondences is concave downwards. In general, this would suggest that the model was learning and using what it learned (intra-trial transfer) more than the subject was. I believe, however, that in these analyses, this is caused by the order of menu reading and

text reading in this task and the relative performance of the model with respect to the subject. In each episode the basic task units are first to read a menu and then to read some help text, and this sequence may be repeated. The model is slower than the subject at reading menus (causing the line to become more vertical) and faster at reading help texts (causing the line to become more horizontal). This is probably what is causing the curve of correspondences to be concave downwards. Any within episode learning effects will not be visible until these larger problems are ameliorated.

There is often an initial horizontal segment in the first 5 to 10 seconds where the subject is taking much longer than the model. The Write episode, for example, displays this effect. We can find out from the operator support display that this region is exclusively where the selection and evaluation criteria are decided upon. It appears that these operators are too simple, at least in terms of the amount of processing that they perform. This mismatch is smaller than the text and menu reading rates, but probably does reflect a basic problem.

We also can note some problems interpreting this display. The verbal utterances have durations, and currently only their starting point is taken into account. All operators are treated as taking the same amount of time. If substructure will be added at a later point to an operator, the analyst can not currently represent that it should take longer than a simple operator.

7.3.2 The processing rate display can be based on other measures of the model's effort

The relative processing rate display can represent the model's rate of processing in measures other than the decision cycle rate. In this subsection a version using operator applications is used to test Browser-Soar. This display is the same display as the display based on decision cycles, except the model's performance is viewed with a different metric.

Figure 7-40 provides an example display of the relative processing rates of the model (in operator applications) and the subject (in seconds). A complete set, one per episode, is included in the appendix to this chapter. A regression line is still fit to the line of correspondences to indicate outliers, but an expected range is not provided because the Soar theory does not provide one — it will depend on how often problem spaces are entered and exited, which is based on the task at hand and the knowledge that can be brought to bear.

The results of computing the relative performance of the model in terms of operator applications are reported in Table 7-30. The operator application rate (in seconds) has a wider relative range and varies more than the decision cycle rate does; the normalized standard deviation of the operator rate is higher. The number of operator applications the model took to perform the task correlated as highly with the subject's performance as did decision cycles. The correlation is slightly higher, but it is not significant ($t(9) = 1.05$), nor does it appear to a large enough difference to be important. This is not too surprising, operator applications are caused by and correlate highly with decision cycles.

The known problems with the two *Read-text* and *Read-menu* operators can again be seen in these displays. Relative learning rates within an episode can also be examined, but again, any relationships found probably are due to the big bad *Read* operator.

This display does not appear to tell us anything new about Browser-Soar, but other models may see an effect here if operators are less directly used, or more behavior occurs in each problem space. Similarly, it does not imply that other measures of the model's effort, such as rule applications, elaboration cycles, or problem space selections, will not prove useful in some way. It is, however, the most likely measure after decision cycles to prove useful.

We can note a constant relationship within Browser-Soar with this display that appears constant across episodes: the number of decision cycles per operator as computed from the two regression slopes. It is not clear yet what this really means, it may mean nothing, but if a relationship appears constant, there may be reason for it.

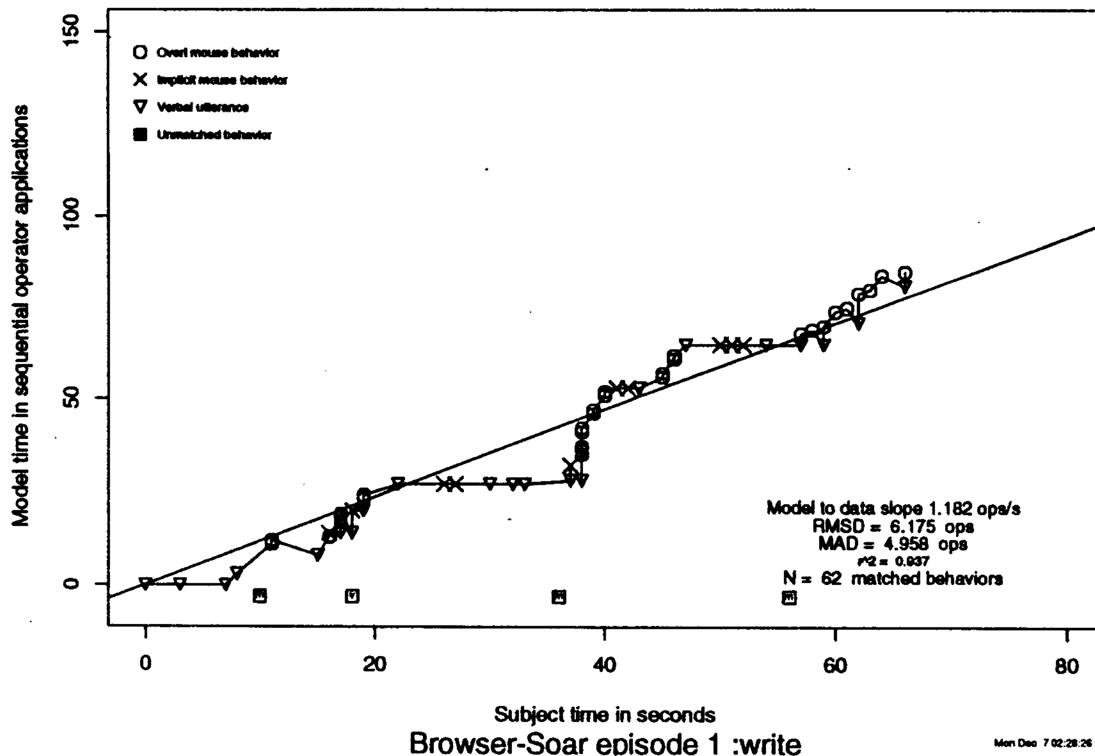


Figure 7-40: Operator applications vs. subject time display for the Write episode.

7.4 High level features of the Browser-Soar model made apparent

Examining Browser-Soar in the SX graphic display suggests further modifications based on how it models routine behavior. Performing a pseudo-model revision to incorporate the effects of learning suggests that Browser-Soar might be improved by using less problem spaces.

7.4.1 Browser-Soar as routine behavior is made directly visible

Search in a problem space means lacking knowledge about how to proceed, and search between alternatives where the solution path is unknown. The solution path in Browser-Soar is not unknown, or at least not substantially unknown. Most operators are the only one proposed, and most problem space impasses are resolved directly. We can see this in the graphic display while Browser-Soar runs. Figure 7-35, which shows Browser-Soar during a run, shows that there are not many operators applied in any one problem space. This is also visible in the problem space level statistics, few states are visited, and not many operators are applied.

Search, in Browser-Soar, when it occurs, also occurs as much as search through problem spaces for knowledge external to the initiating space. The name of "solution space" (Ohlsson, 1990) particularly here, makes more sense, with Browser-Soar more like a task (Ohlsson, 1990) than a problem. This result is noted by Peck and John (1992), but appears more clearly in these pictures and aggregate statistics than in the textual trace alone.

7.4.2 Noting Browser-Soar's large goal depth

The goal stack depth is relatively deep in Browser-Soar. As noted in Figures 7-33, 7-35, and 7-36, the goal stack often grows to be between four and six levels down from the top problem space. This appears to be a large number for what is described as routine behavior (but we have no real metric). In addition to the question of the depth of the goal stack, all the lower problem spaces for manipulating the mouse and screen represent expert level behavior in the subject, that is, behavior that does not significantly improve with practice. In Browser-Soar impasses still occur, and if learning was turned on, knowledge would migrate between them. In expert behavior, the lowest level of operators and problem spaces in Browser-Soar should not be visible because they have been learned by the problem spaces that use them.

7.4.3 Modifying Browser-Soar

With the learning constraint in mind, a modified version of Browser-Soar was created and tested using the pseudo-model revision method mentioned in Chapter 3. The modified version does not contain the lower level problem spaces that would have been learned. The actual output operators were migrated to higher level problem spaces, and intermediate operators and problem spaces that did not receive support from the data, such as the operators in the *Access-item* problem space. A complete listing of the modifications is provided in Table 7-31.

Table 7-31: Problem spaces and operators removed from the Browser-Soar model simulating the effects of learning.

- Browsing PS and OP,
- Find-criterion OP and PS,
- Mac-methods-for-change-current-window PS,
- Change-current-window OP,
- Drag OP,
- Scroll OP,
- Mac-method-of-scroll PS,
- Mac-method-of-drag PS,
- Mac-method-of-page PS,
- Access-item OP,
- Mac-methods-for-access-item PS,
- Click-on-item OP,
- Mac-method-of-click-on-item PS,
- Evaluate-help-text OP,
- Evaluate-help-text PS,
- Double-click-on-item OP,
- Mac-method-of-double-click-on-item PS, and
- All associated goals and states.

Figure 7-31 shows the problem space organization of this modified version of Browser-Soar. The organization can be compared with the original version shown in Figure 7-33. The new version has fewer problem spaces, and is flatter. The maximum goal stack depth of this version is four, with final

depths of two and three. It has 8 problem spaces compared with 17 problem spaces in the original Browser-Soar, 22 operators compared with 31 in the original, and a corresponding decrease in the number of intermediate states and impasses.

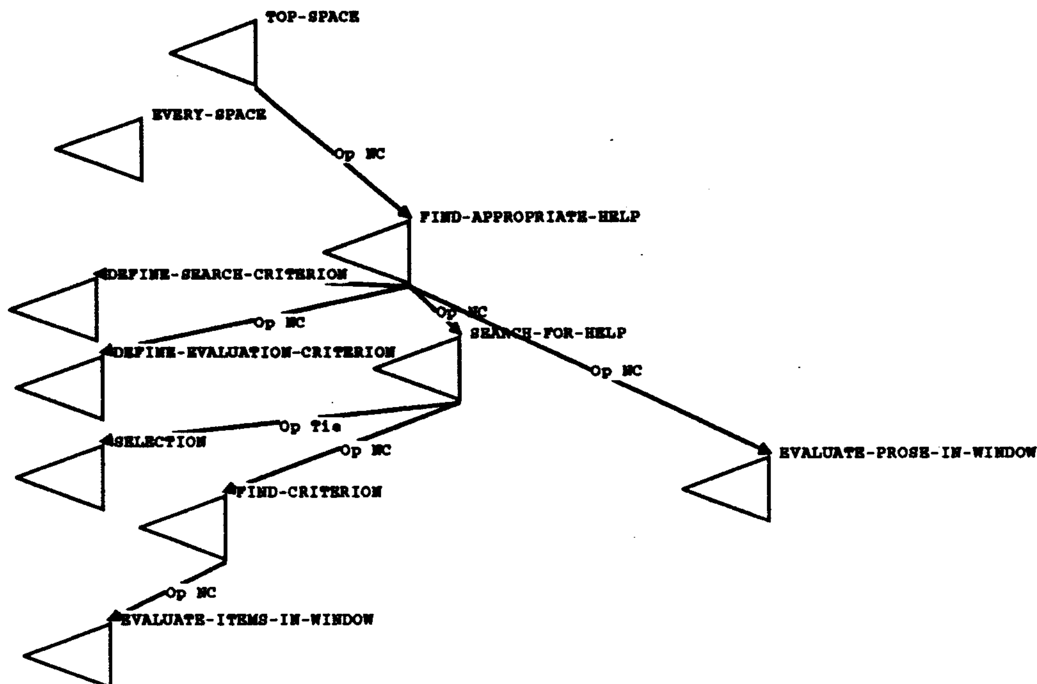


Figure 7-41: The nine problem spaces in the modified Browser-Soar (see Figure 7-33 for the original structure).

The revised model was not implemented on the production level, but was created using a more lightweight technique of trace revision. All the operator and problem spaces that were removed, were simply deleted from the trace for the Array episode, the second largest episode, and the trace was renumbered. This took approximately 20 minutes. These changes also could have been implemented by modifying the model, and rerunning it. Theoretically there would be no differences, in reality, actually editing the code instead of the trace probably represents an order of magnitude more work.

As the actual model was not modified, this represents an instance of pseudo-model based revision, where an aspect of the analysis changed in terms of the model, without the expense of completely implementing the changes on the production level.

7.4.4 Testing the modified Browser-Soar

After the revised trace was made, the two information streams were realigned from scratch. Because no model actions with support were removed, the realignment was essentially the same. It would have been faster to use the old alignment and modify it slightly, removing the empty cells, for no correspondences were cut, but I wanted to see what the total process could look like, and see how long a more modified model would take to test. The total time to perform the model manipulation, realignment, and generate the analyses was 2.5 hours.

Figure 7-42 shows the operator support displays for the two versions of Browser-Soar. The displays are essentially the same, the shape is the same, and the subject actions and the operators that they correspond to are all represented. The only difference is that the modified version is more compact; it

has less operator applications.

Figures 7-43 and 7-44 show the model fit displays for the modified version of Browser-Soar next to the original versions. These two displays show that the revised model has a denser level of support, the lines connecting the corresponding model and subject actions are closer together, and the RMSD and mean average deviation are lower. The rate of decision cycles to seconds ratio is also closer to the predicted mean, and visually the fit appears to be better. The modified version has slightly worse r^2, more so when the model time unit is decision cycles (.69 versus .59) than for operator applications (.78 versus .75). The correspondence rates in decision cycles and operator applications per second for the modified model also go down, as less is done.

It is hard to tell if these differences are important. It would perhaps become easier to tell after further revisions of the *Evaluate-current-window* operator, and with a more proper regression line (Kadane et al., 1981; Larkin et al., 1986). These results do point out that it is hard to distinguish learning on the single problem space level at this time grain. In order to clearly distinguish these two problem space representations we would have to look at more episodes, more subjects, or further constraints from data. Given the lack of real difference, parsimony would argue for using the simpler, modified version of Browser-Soar.

This analysis also calls into question the strict interpretation used. The subject must decide to move the mouse. The operators that were removed originally represented this choice. With a different interpretation function, these operators would have been supported and would not have been removable. As noted in the list of corrections available when the model's predictions mismatch the data (Table 2-6), the interpretation function can also change. This case raises the question of how to interpret data given Soar's hierarchical operators and state representation. This may remain a problem for some time.

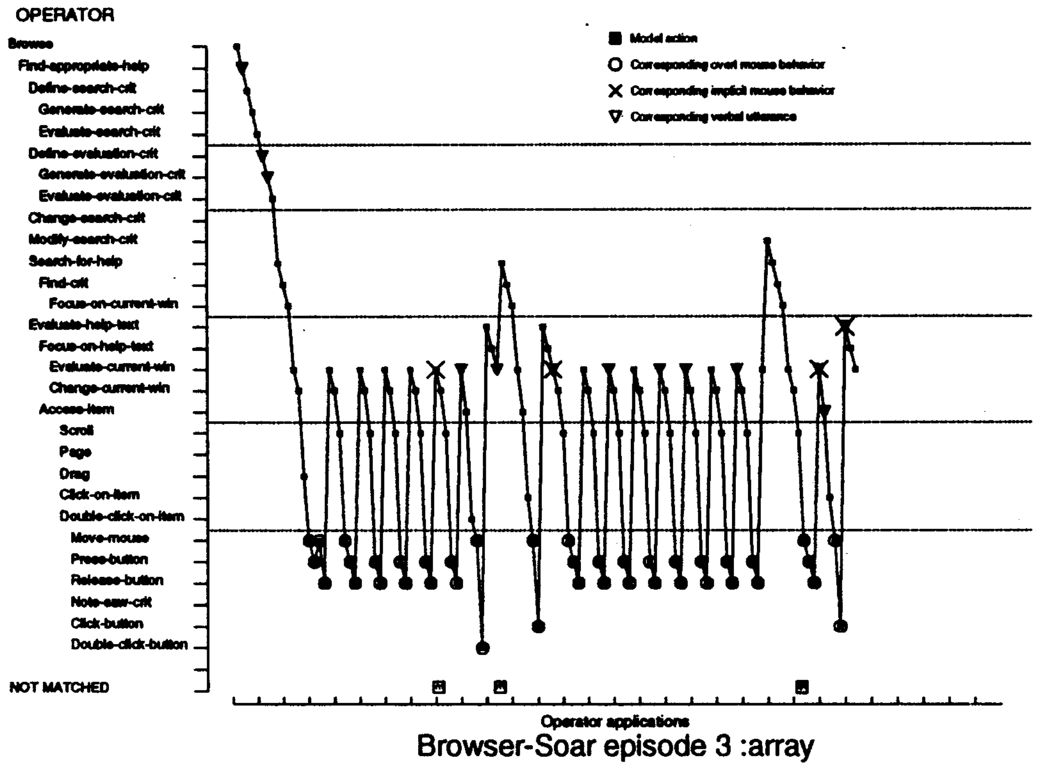
7.5 Testing and extending the sequentiality assumptions of protocol generation theory

As noted in their initial description, the relative processing rate displays allow the sequentiality assumption of Ericsson and Simon's (1984) theory of verbal protocol production to be tested. That is, if verbalizations are produced in the order that the corresponding data structures appear in working memory. There is another aspect to this assumption, that inputs to operators will be reported before their outputs, but is a more specific form that will not be directly tested unless we run into problems. A model of what appears in working memory is currently necessary to test this assumption. There are no other ways to tell when information enters working memory, and thus that it is reported in order. Having a model of the contents of working memory also allows use to judge if the verbalizations are retrospective or prospective.

Browser-Soar provides predictions of the contents of working memory while using a specific on-line help system. By examining the relationship of these predictions with the subject's verbal utterances in the ten Browser-Soar episodes, the sequentiality assumption can be tested.

The predictions of the external task actions (mouse movements and button presses) can also be compared with the contents of working memory, but because getting the order of the external actions the same for both model and subject is essential for performing the task, in a well developed model like Browser-Soar there is not likely to be many mismatches. What will be interesting though, is using the external actions to compute how later (or early) the verbal utterances are.

Finding that this holds will not be an iron-clad proof that this assumption holds. If it is an assumption, then it cannot be proven, only shown that we meet it. If it is treated more as part of the theory of verbal protocol production, then there may be similar models of browsing behavior where the information is reported in a different order, and that the current set of verbal protocols would not match sequentially.



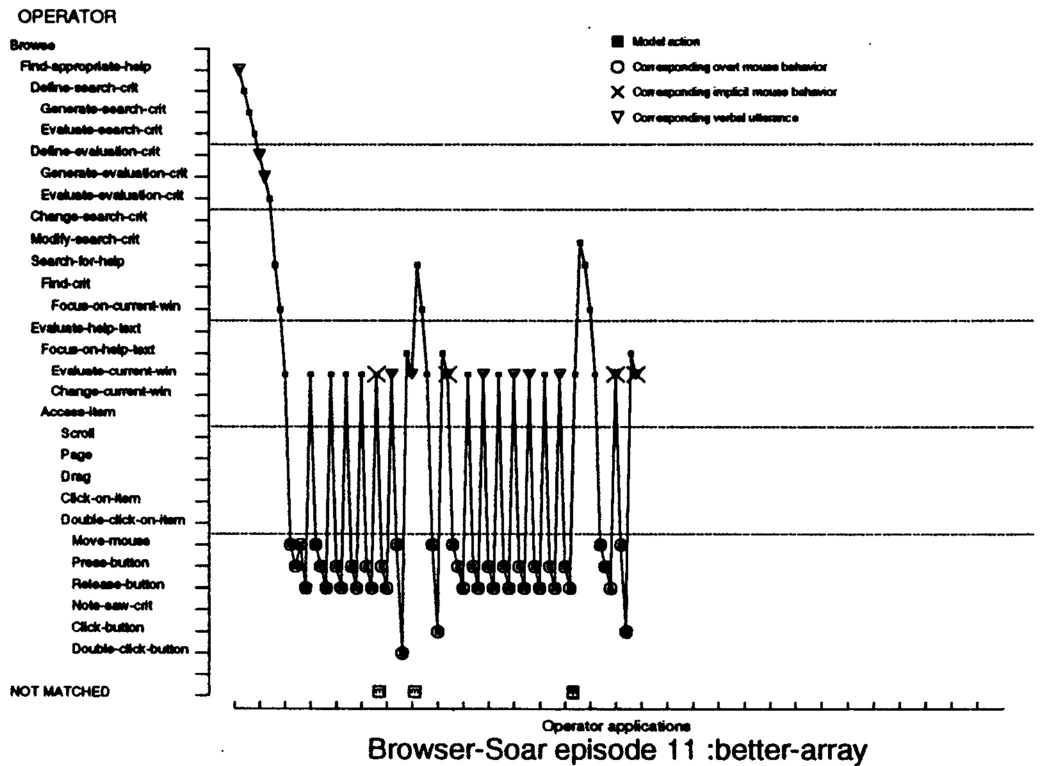
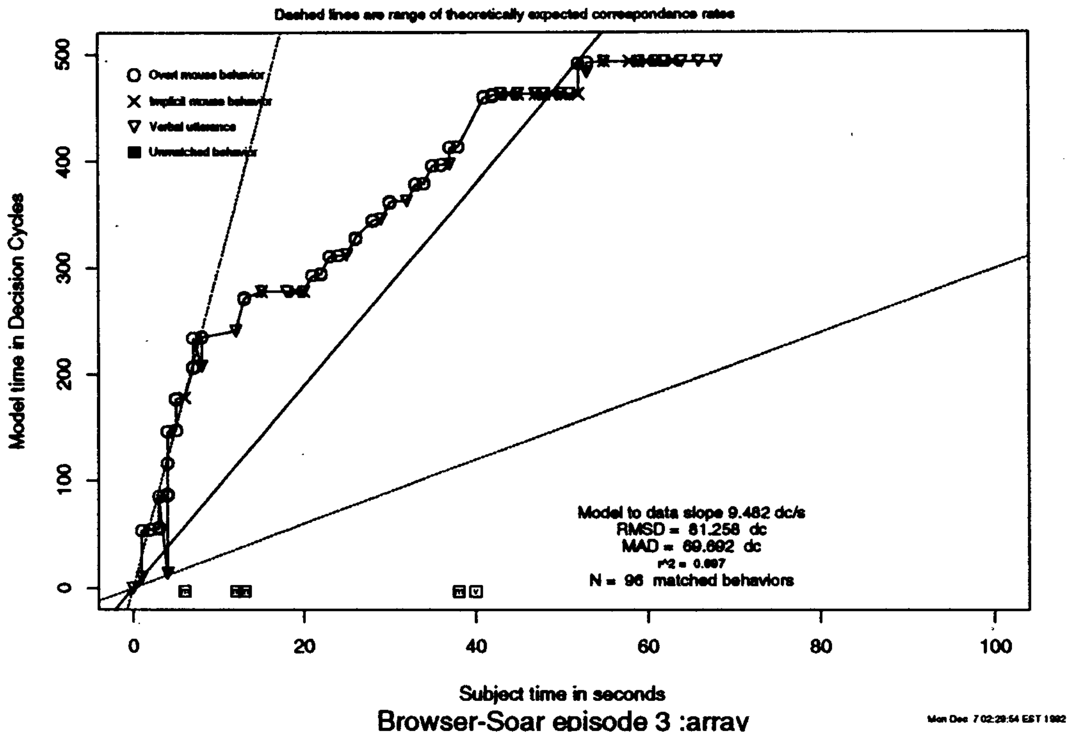


Figure 7-42: Operator support displays for the Array episode.
The original Browser-Soar predictions are on the top, and the modified version on the bottom.



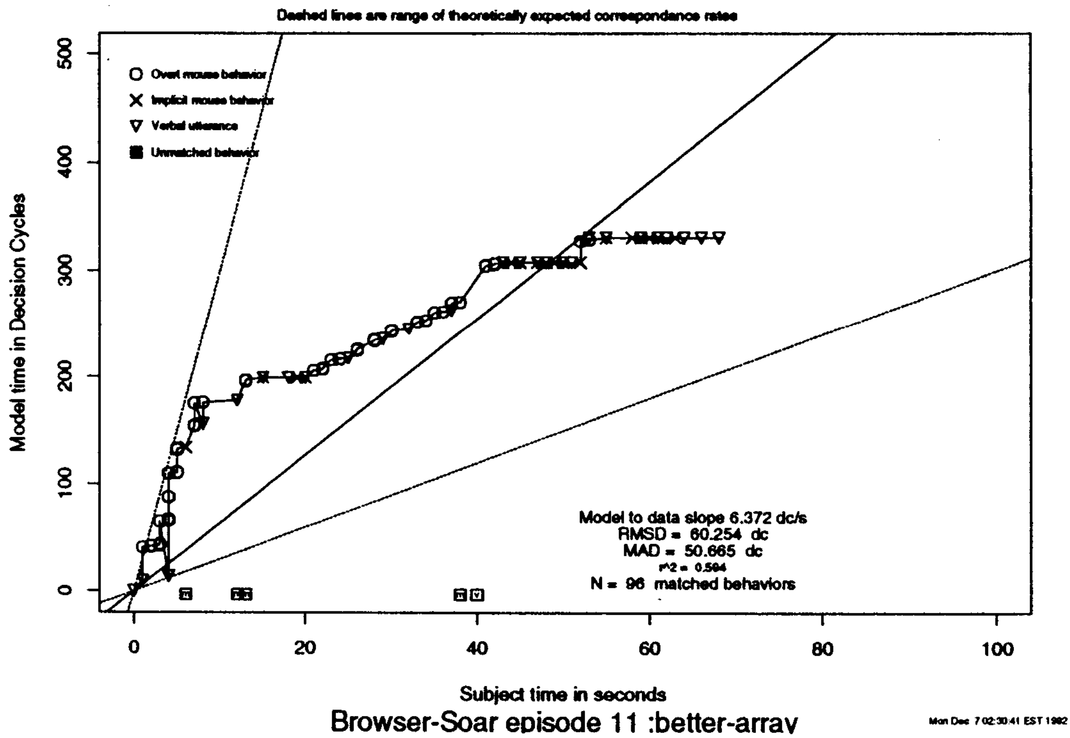
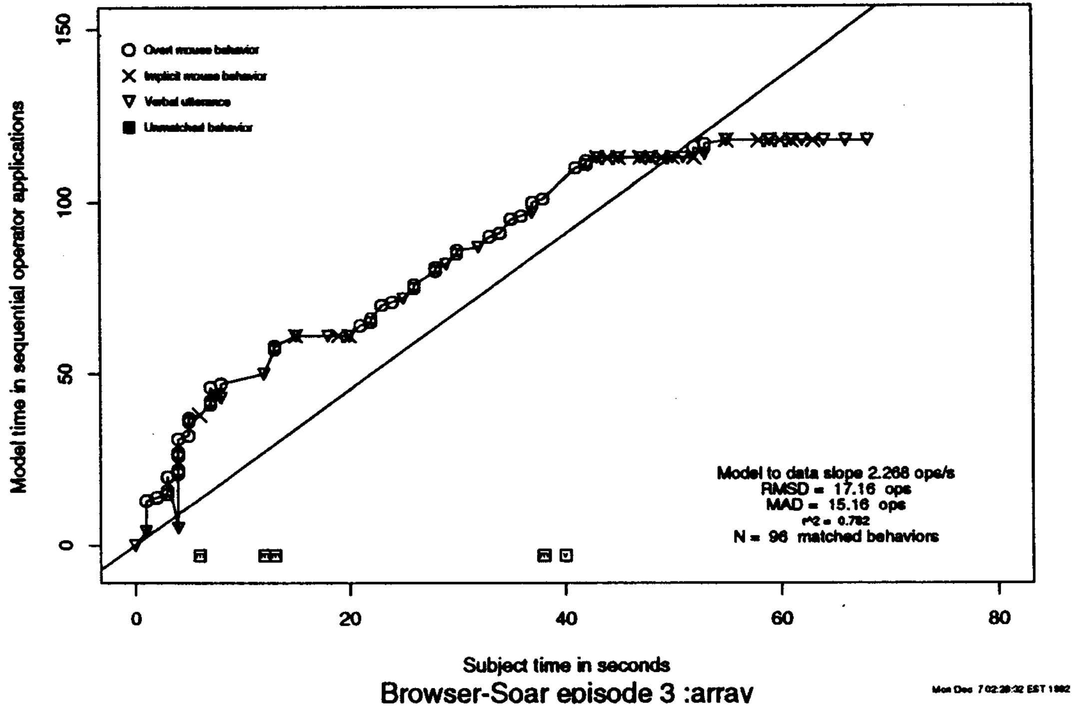


Figure 7-43: DC time based plots for the Array episode. The original Browser-Soar predictions are on the top, and the modified version on the bottom.



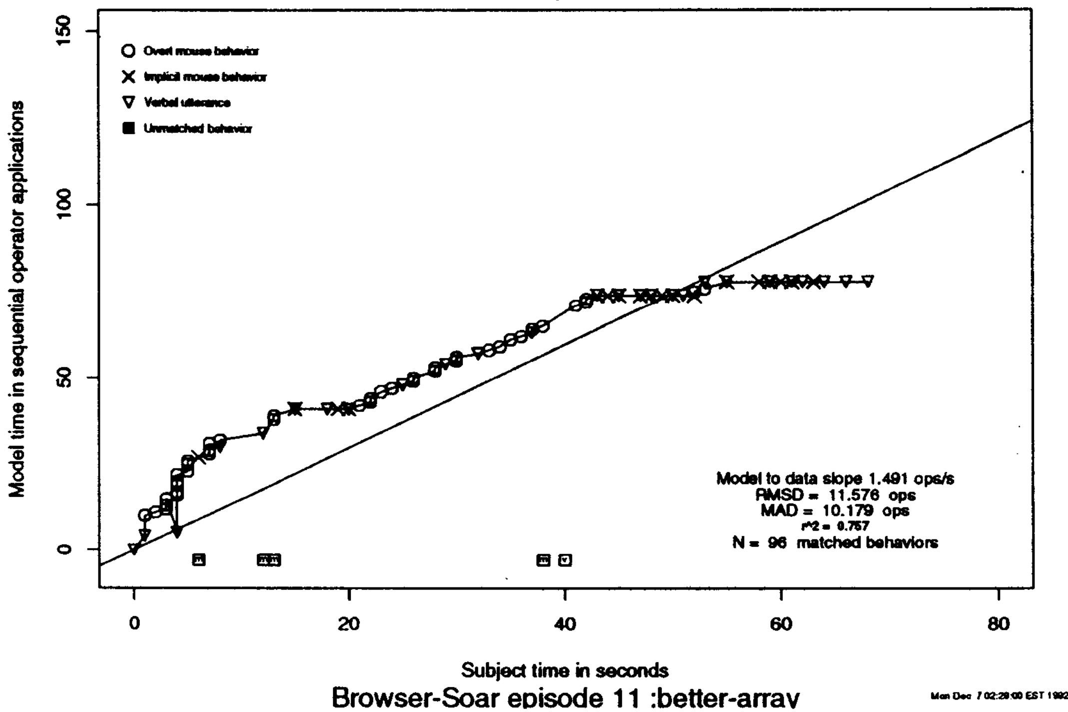


Figure 7-44: Relative processing rates displays based on operator applications for the Array episode. The original Browser-Soar predictions are on the top, and the modified version on the bottom.

7.5.1 Are verbalizations generated sequentially?

Of the 220 verbal utterances in the ten episodes, 195 can be aligned with the model's predictions. The remaining 25 are mostly too short to compare. The remaining segments make up 210 pairs of immediately sequential utterances that can be tested against the sequentiality assumption. This test can be performed by eye with the displays, and the initial analyses did this because it was so easy and direct. The final counts were taken from the data structure used to create the displays.

All 210 pairs follow the sequentiality assumption; for all the pairs, the later segment in each pair either matches the same model trace action as the first segment matches or a later model trace action. So this appears to be another constraint that Browser-Soar meets.

7.5.2 Are mouse actions generated sequentially?

In a similar way the mouse movements and mouse button actions can be tested for sequentially. Because these actions were used as fixed points to automatically align the subject's protocol and the model's trace, in order to match out of sequence they would had to have been moved by hand out of sequence, or items that could not be automatically aligned would have had to be aligned by hand.

Of the 404 mouse actions in the ten episodes, 373 can be aligned with the model's predictions.[9] These 373 actions make up 363 pairs of sequentially contiguous actions. Again, a preliminary examination of the displays showed that none matched the model out of order, and an analysis of the data base confirmed that.

7.5.3 Does the sequentiality assumption hold across verbalizations and mouse actions?

All the subject's actions can be tested for sequentiality. As explained in Chapter 5, this can be done by examining the connected correspondences in the relative processing rate displays. Starting from the first correspondence and moving along the line of correspondences, a connecting segment with a negative slope indicates that the second correspondence matched earlier in the model than the first correspondence, violating the sequentiality assumption. Simply examining the displays shows that several verbal utterances lag the mouse movements noticeably. Of the 624 total segments, 568 are aligned with the model's actions in the ten episodes.[10] These 568 actions make up 558 pairs of sequentially contiguous actions, and 21 pairs do not meet the sequentially assumption, that is, in these pairs, the second subject action is a verbal utterance that matches an earlier prediction than the first action that is a mouse action.

The lag of verbal utterances was computed by comparing the decision cycle number of the model prediction corresponding to the verbal utterance with the decision cycle of the previously matched mouse action. Figure 7-45 shows the distribution of these times. Across all verbal utterances in all episodes the average lag was 9 decision cycles, or roughly 1 second. This is an acceptable number, indicating that while some verbal utterances appear to have been produced quite late compared to the mouse movements, overall the subject was not providing retrospective reports.

Most of the verbal statements (144 out of 195) match the model's predictions sequentially, not matching earlier portions of the model than their proceeding segment. Based on their starting points these utterances can be considered as truly concurrent protocol — it is generated as the subject doing the task and it matched the predictions of the contents of working memory. The ends of the utterances have not been included in these analyses, although Peck and John included this length in their data set.

---

[9]An astute reader may note that there are five more mouse movements matched by subject actions in this analysis than in the original analysis reported by Peck and John. One of these discrepancies has been found so far, and it was a typo.

[10]An astute reader may again note that there are five more predictions matched by subject actions in this analysis than in the original by Peck and John. Even with a semi-automatic tool, analysts will make mistakes.

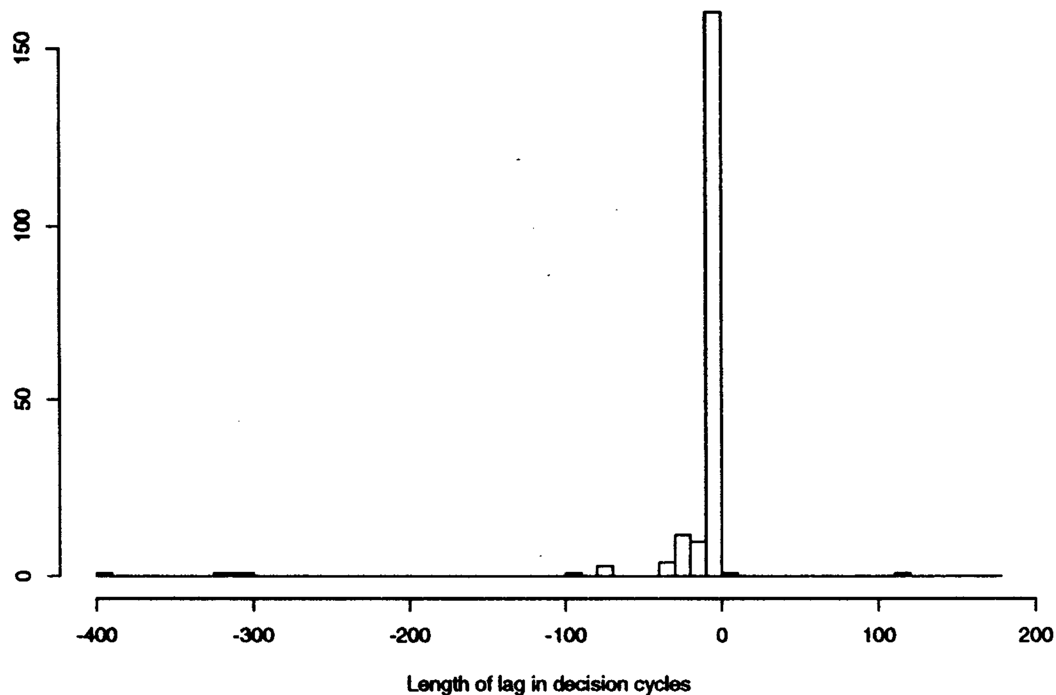


Figure 7-45: Histogram of the lags (in decision cycles) of the verbal utterances.

While these segments are not long generally, it is possible that their tail end ceases to be concurrent.

There are two prospective utterances, one in the axis episode, which upon inspection was an typo in alignment. The segment was properly concurrent, but misaligned by four decision cycles in the spreadsheet. The other utterance occurred in the Vars episode and is more interesting. It has a positive offset of 111 decision cycles (nominally 11 seconds). It is hard to see on the relative processing rate display because it is surrounded by several mouse movements, which is the cause of it being interpreted as early. When the segment is examined, it turns out that the verbal utterance is not so much prospective, but that the model's menu reading ability falls behind the subjects at that point, and the model has to perform an extra 100 cycles of work before it can match the verbal utterance.

The remaining 49 utterances all lag their previous segment, matching an earlier prediction. When an utterance lags, it lags on average 38 decision cycles, or roughly 4 seconds. Again this remains a modest amount. This amount of time is consistent with the amount of time items can exist in working memory. A very small number, three, lag over 300 decision cycles.

Characterizing the long lags Many short lags of the verbal utterances appear to be partly (but not completely) an artifact of the Browser-Soar model. The model does not read individual words but whole screens at a time, which leads to many of the short lags that occur late in an episode when the subject is reading a help text. Including predictions of reading individual words would remove this cause.

The three longest lags, however, are worrisome. They lag over three hundred decision cycles, and represent a mismatch on the order of 20 to 40 seconds. The problem space of the operator they match

has long been removed from the goal stack, and several other problem spaces on that level have been used as well. When these segments are examined they are found to be statements of the search or evaluation criteria that occur after the search has started and numerous items have been examined. While an operator put them on the state, at the point they are uttered, they clearly represent state information that has been guiding the search for some time. Other operators could be refreshing them, but if that is what lead to these utterances, then the operator used to interpret them is still the wrong one.

Finding this lag in the literature. The actual lag of verbal protocols has not been computed in this way to my knowledge. It requires an architecture that makes predictions about the time to perform a task, external actions to provide fixed points of reference, and the predictions must be aligned to this detail. We can see a lag in other data sets, however. The verbal data used to develop HI-Soar (John & Vera, 1992; John, et al., 1990) can be fixed relative to the performance of external actions. The verbal protocols lagged behind the external actions so much that they were ignored when testing the model.

7.6 Conclusions about Browser-Soar and the TBPA methodology

Having performed these analyses, we can summarize the results into several suggested changes to Browser-Soar, which is the point of testing a process model. In general, Browser-Soar performed very well. The operators in the model that performed best were the ones that are essential to browsing on-line help systems: manipulating the mouse, choosing windows, and evaluating text items. On a higher level, testing Browser-Soar also generated some lessons for the methodology and for the environment that should be incorporated into the environment.

This methodology was stretched in a particular direction through testing Browser-Soar. Browser-Soar and the data used to test it have some very particular characteristics: (a) very close matches, (b) very routine behavior and typical problem solving by the subject, (c) a highly interactive task, (d) mostly a mental task (the perception and motor actions were routine). This example application did not deal with every type of data. It is easy to name several data features that have not been touched: (a) very bad matches between data and model, (b) perceptually based reasoning, (c) how to create a model in the first place, or drastically revise it, (d) tasks that cannot be modeled as search through or in problem spaces, and (e) extremely long or short protocols. Adding any of these features to the data and task is likely to add further lessons and stretch the methodology in a new way.

7.6.1 Some conclusions about Browser-Soar

The analyses performed suggest several ways to improve Browser-Soar. Most, if not all, are known to the authors of Browser-Soar, but the importance and location of the changes should be clearer after these analyses. These changes are presented in Table 7-32.

Browser-Soar's ability to predict large amounts of the data should also be clearer as well. Chapter 2 put forward the idea that analytic testing would not only point out where to improve a model, but it also would make it more believable by presenting it more clearly. Several diagrams and tables were created in performing these analyses that should make the model more believable. There are more visual descriptions of the model (Figure 7-33), its performance (Figure 7-35), a rough measure of the amount of knowledge in each problem space (Figure 7-36), and a picture of the calling order of its operators (Figure 7-38). Aggregated measures of which operators and problem spaces are used and how often have been presented (Table 7-29). The analytic displays show when operators are supported, and by which type of data (Figure 7-38 and the Appendix to this chapter), and the relative processing rates of the model and subject over time (Figures 7-39 and 7-40, and the appendix to this chapter).

Table 7-32: Suggested changes to Browser-Soar based on analyses performed.

- Operators without evidence, *Scroll, Page, Drag,* and *Click-on-item,* must be considered for removal from the model, or be supported with non-protocol data such as aggregate timing results.
- Fitt's law should be included in the model of moving the mouse.
- A more complete *Read* operator for reading text that takes longer.
- A less complete *Read* operator for reading menus faster, more like scanning.
- Overall, the model's performance is slightly lean, but this must be reevaluated after some other problems, most importantly the reading operator, have been improved.
- Include learning, and decrease the goal stack depth.
- Include state information in the trace and match to it.

7.6.2 Some conclusions about the methodology

Performing these analyses pointed out that it is nearly always good to have context, and sometimes it is required. Just providing information on a single item is often not enough to understand the item. The item's context is also needed. In several places, particularly in examining the model fit displays, users can now click on a data point and get a segment and a selectable amount of its context displayed.

Different grain sized operators and different commitments to operators lead to problems in the analysis, and should be avoided if possible. Soar in particular, as a general architecture for intelligence, provides the ability to model every action. As a unified theory of cognition it highlights the desire to provide a complete model, covering all the data. By definition, some portions of each model will be weaker than others.

Soar models are much finer in their grain size than Newell and Simon's (1972) systems; more actions occur that cannot be tested, such as goals and many problem spaces. Other items might be found, but are not found in every episode. It may be desirable to omit these items automatically and appropriately when performing an analysis.

While Newell and Simon (1972, p. 179) propose that states and operators are equivalent, the reanalysis of Browser-Soar shows that they are only equivalent for information purposes. When the timing of the correspondences is included, they are not equivalent. States, and the information they contain, last much longer. It may be possible to continue to match verbal utterances primarily to operators, but when this breaks down, one must match to the state. Using the state properly is not a trivial task, and will require designing and extending the trace. It will require further mechanations in the interpretation algorithms to find the appropriate items to support in the model when this does occur.

No problem spaces or goals are used to interpret the subject's behavior. Together their creation and selection make up a substantial portion of the model's behavior. What it would mean to match their prediction is not clear, problem spaces may be supported by their operators and states, goals by the indication of a lack of knowledge in some way. If they will not be directly supported, the cognitive modeler may desire to removal them from the trace if not the model.

Finally, we see that testing the model points out that the model is not complete without rules describing how to interpret the data with respect to the predictions. For example, the *Page, Scroll, Drag,* and *Click-on-item,* were considered for removal because they were not supported. A more generous interpretation of the mouse actions might have included the decision to click (e.g., to *Page*) as being supported as well.

Appendixes to Chapter 7

1 Alignment of the Write episode of Browser-Soar

```
Wed Nov 25 14:35:42 1992 - Dismal (0.83) report for user ritter
For file /afs/cs.cmu.edu/user/ritter/spa/browser/write/write6.spa

To print use  "enscript -r -c -G -fCourier7 -L66  /afs/cs.cmu.edu/user/ritter/spa/browser/write/write6.dp"
----------------------------------------------------------------

   A      B              C              D    E F   G    H  I                          J                                      K
+--+---------------+------------------+-------+-+-+-----+---+---+--------------------------------------------+---------------------------------+
 0 vapeck   25-Sep-91 revised 13-Jan-92 -FER          3-Jun-92                                                         To do:
 1 From original transcription by dmr, 16-Mar-90, and verbal transcription by s.esch,  Jun-91
 2 Transcription of the 15-Jun-89 of browser tape 2
 3
 4 Browsing for information about writing the values of variables to the screen.
 5 Previous Goal:     define the 'loop' construct that will label the x-axis
 6 Current Goal:      to figure out how to write the value of 'ExpCondNames' in
 7                    order to label the x-axis with experimental condition names
 8 Windows:           program window out front, right side and bottom of help win.,
 9                    only the left edge of the commands win., right side of execution window
10 Program Window     line 1  "**********"    line 2  "unit   DrawGraph"    line 3   "gorigin   110?, 340?"
11 Execution Window: error box with message at the top
12 Help Text Window: line 1  "coarse 'Typing-Paper' Coordinates"
13 Keyword Menu       line 1  "coarse" (selected)"
14 Hierarchiel Menu: line 1   "at   Positioning a Display"   (no lines selected)
15 Commands Window:  (not used)
16 Cursor:            positioned at the end of the 'put' command line within the 'loop' contruct
17 Mouse:             located (+x) -3/4 in. from the end of the 'put' command line
18    (mouse is currently a line)
19                           See also /afs/cs/project/soar/member/vapeck/browser-soar/current/episodes/write/01-22.write.complete.log
20 69 total behaviors                          69
21 11 distinct operators evidenced in behavior                     Loaded from:
22                                                                 /afs/cs/project/soar/member/vapeck/browser-soar/current/LOAD-write.lisp
23 25 total verbals                                                Loading /afs/cs.cmu.edu/user/ritter/spa/browser/op-trace-additions.lisp.
24
25
26  last time verbal/mouse matches information used by an operator.
27 21 total mouse movements            MTYPE is type of match:
28 9   unnecessary movements that give evidence        87   Total
29 6   necessary movements
30
31 24 Total mouse button actions                        4     B4 is actions before model coverage.
32                                                      2     SHORT is too short to code.
33 TIME is timestamp of action in s.                   14     CONT is segment continued from previous line.
34 DURATION is length in ms of behavior.
35 VERBAL is verbal protocol.                          22     V is verbal coded.
36 Mouse Action is the user's mouse movements.          6     MR is mouse required (these are just movements).
37 ST is Segment Type                                  11     MI is mouse inferred.
38 # is segment number                                 24     MBA is mouse button actions (necessary by default).
39 MTYPE is type of match
40 MDC is matched DC.                                   3     MUC is mouse uncoded.
41 DC is decision cycle in Soar model.                  1     VUC is verbal Un Coded.
42 SOAR TRACE is the literal Soar Trace
43
44    0.94          percent subject data matched
45    0.09          percent model matched
46    0.10          seconds/decision cycle
47
48                  total words               113
49
50 Time starts at 12400
51  T  Mouse actions   Window actions    Verbal     ST #  Mtype MDC  DC                  Soar trace                     Comments
52 0                                     I believe   v  1 short
53                                                                 0 G: g1                                            From:  raw-trace3.txt, 17-Mar-92
54                                                                 1 P: p4 (top-space                                 Operator names cleaned up by hand,
55                                                                 2 S: s5                                            6-july-92 FER
56                                                                 3 O: browse ()
57                                                                 4 ==>G: g19 (operator no-change
58                                                                 5   P: p26 (browsing
59                                                                 6   S: s39 ((unknown) (unknown)
60                                                                 7   O: find-appropriate-help
61                                                                 8   ==>G: g43 (operator no-change
62                                                                 9     P: p50 (find-appropriate-help
63                                                                10     S: s59 ((unknown) (unknown)
64                                                                11     O: define-search-criterion
65                                                                12     ==>G: g65 (operator no-change
66                                                                13       P: p72 (define-search-criterion
67                                                                14       S: s79 ((unknown)
68 6                                     write       v  2     v  15   15       O: generate-search-criterion ((write))
69 9                                     write       v  3     v  15
70 13                                    write       v  4     v  15
71    M(+x) (R of prog win)                                  B4                                                   B-Soar doesn't model bringingup the
72                  mouse line to pointer                                                                              help win.
73                                                                16       O: evaluate-search-criterion           *** generated search criterion 'write' **
74                                                                17     O: define-evaluation-criterion
75                                                                18     ==>G: g103 (operator no-change
76                                                                19       P: p110 (define-evaluation-criterion
77                                                                20       S: s117 ((unknown)
78 14                                    Can I write v  5     v  21   21       O: generate-evaluation-criterion ((value-of-something))
79 15 M(+y) (top of screen)                                  B4                                                   unmodeled (before the task) mouse movement
80 15 M(-x-y) (portion of help win below prog win)           B4
81 16 C             help win comes forward                   B4
```

```
S1  T  Mouse actions   Window actions    Verbal      ST #  Mtype MDC  DC                    Soar trace                                 Comments
+--+---------------+-----------------+-----------+-+---+------+---+---+-----------------------------------------------------------+---------------------------------------------+
82 16 M(+x+y) (R of 'coarse' at top of keyword menu)mm  6   mms
83    --(+x-y) (just L of keyword down arrow)               cont
84                                                             22    O: evaluate-evaluation-criterion                  *** generated evaluation criterion
85                                                             23   O: search-for-help                                    'value-of-something' **
86                                                             24   ==>G: g137 (operator no-change
87                                                             25     P: p144 (search-for-help
88                                                             26     S: s155 ((to-be-found write) (value-of-something)
89                                                             27     O: find-criterion (keyword)
90                                                             28     ==>G: g161 (operator no-change
91                                                             29       P: p168 (find-criterion
92                                                             30       S: s170 ((to-be-found write) (value-of-something)
93                                                             31       O: focus-on-current-window
94                                                             32       O: evaluate-current-window
95                                                             33       ==>G: g239 (operator no-change
96                                                             34         P: p246 (evaluate-items-in-window
97                                                             35         S: s256 ((to-be-found write) (value-of-something)
-98                                                            36         O: read-input (coarse)
99                                                             37         O: attempt-match
100                                                            38         O: read-input (comment)
101                                                            39         O: attempt-match
102                                                            40         O: read-input (comp_x)
103                                                            41         O: attempt-match
104                                                            42         O: read-input (compute)
105                                                            43         O: attempt-match
106                                                            44         O: read-input (constant)
107                                                            45         O: attempt-match
108                                                            46         O: read-input (cos_x)
109                                                            47         O: attempt-match
110                                                            48         O: read-input (cursor)
111                                                            49         O: attempt-match
112                                                            50         O: read-input (detain)
113                                                            51         O: attempt-match
114                                                            52       O: change-current-window
115                                                            53       ==>G: g423 (operator no-change
116                                                            54         P: p430 (mac-methods-for-change-current-window
117                                                            55         S: s438 ((to-be-found write)
118                                                            56         O: scroll (keyword)
119                                                            57         ==>G: g451 (operator no-change
120                                                            58           P: p458 (mac-method-of-scroll
121                                                            59           S: s467 ((to-be-found write)
122 17 M(+x) to (keyword dn arrow)             mm  7   mr  60  60           O: move-mouse (keyword down)
123 17 D               keyword menu scrolls    mb  8   mba 61  61           O: press-button
124 18                 keyword menu scrolls
125 19                 keyword menu scrolls
126 20                 keyword menu scrolls
127 21                                 write   v  9    v  32
128 22 U               scrolling stops         mb 10   mba 62  62           O: release-button
129 22 M(-x+y) (R of item3, keywd menu:wrong (write imm 11   mi 63  63       O: evaluate-current-window
130 23                                 wrong?  v 12    v  63
131                                                            64       ==>G: g507 (operator no-change
132                                                            65         P: p514 (evaluate-items-in-window
133                                                            66         S: s524 ((to-be-found write) (value-of-something)
134                                                            67         O: read-input (wrong)
135                                                            68         O: attempt-match
136                                                            69         O: read-input (wrongv)
137                                                            70         O: attempt-match
138                                                            71         O: read-input (xin)
139                                                            72         O: attempt-match                           We are left matching operators
140                                                            73         O: read-input (xout)                       for we have not states.
141                                                            74         O: attempt-match
142                                                            75         O: read-input (zaltered)
143                                                            76         O: attempt-match
144                                                            77         O: read-input (zaxcont)
145                                                            78         O: attempt-match
146                                                            79         O: read-input (zarrows)
147                                                            80         O: attempt-match
148                                                            81         O: read-input (temptemptemp)
149                                                            82         O: attempt-match
150                                                            83       O: change-current-window
151                                                            84       ==>G: g696 (operator no-change
152                                                            85         P: p703 (mac-methods-for-change-current-window
153                                                            86         S: s711 ((to-be-found write)
154                                                            87         O: scroll (keyword)
155                                                            88         ==>G: g724 (operator no-change
156                                                            89           P: p731 (mac-method-of-scroll
157                                                            90           S: s740 ((to-be-found write)
158 23 M(+x-y) (keyword up arrow)              mm 13   mr  91  91           O: move-mouse (keyword up)
159 23 D               keyword menu scrolls    mb 14   mba 92  92           O: press-button
160 23 U               scrolling stops         mb 15   mba 93  93           O: release-button
161 24                                 no      v 16    v  63
162 24 M(-x-y) (2nd keyword from bottom, xin)  mm 17   mi  94  94       O: evaluate-current-window
163 24                                 Ha  ha haaa v 18 vuc
164 25                                 write.  v 19    v  94
165                                                            95       ==>G: g777 (operator no-change
166                                                            96         P: p784 (evaluate-items-in-window
167                                                            97         S: s794 ((to-be-found write) (value-of-something)
168                                                            98         O: read-input (user-vars)
169                                                            99         O: attempt-match
170                                                           100         O: read-input (vbar)
171                                                           101         O: attempt-match
172                                                           102         O: read-input (vector)
173                                                           103         O: attempt-match
174                                                           104         O: read-input (write)
175                                                           105         O: attempt-match
176                                                           106     O: access-item (keyword)
177                                                           107     ==>G: g878 (operator no-change
178                                                           108       P: p885 (mac-methods-for-access-item
179                                                           109       S: s893
180                                                           110       O: click-on-item (1000)                      1000 is an unnamed item
181                                                           111       ==>G: g899 (operator no-change
182                                                           112         P: p906 (mac-method-of-click-on-item
183                                                           113         S: s913
184 25 M(+y) (3 items up to 'write')           mm 20   mr 114 114         O: move-mouse (keyword unspecified)
```

```
 S1  T Mouse actions   Window actions    Verbal       ST #  Mtype MDC  DC                  Soar trace                                  Comments
 +--+----------------+-----------------+------------+-+--+------+---+---+------------------------------------------------------+------------------------------------------+
185 25 C               mouse pointer to watch         mb 21   mba 115 115        O: click-button
186 26                 'write' help text appears      io
187 27                 'write' becomes bold & moves   io
188                                                                 116     O: evaluate-help-text
189                                                                 117     ==>G: g927 (operator no-change
190                                                                 118       P: p934 (evaluate-help-text
191                                                                 119       S: s943 ((accessed write) (value-of-something)
192                                                                 120       O: focus-on-help-text
193 28                          convenient way to v 22      v 121 121       O: evaluate-current-window
194 29                                write out short   cont
195 30                                of text that loo  cont
196 31                                in your program   cont
197 32                                the text command  cont
198 32 M(-x-y) (3/4 dn help text scrollbar below elevmm 23    mi 121
199 33                                umm...       v 24 short
200 33 M(-x-y) (bottom R quad of help text win)    mm 25    mi 121
201 36                                show comman v 26     v 121
202 38                                are used    v 27     v 121
203 39                                to display  v 28     v 121
204 41                                so thats what I   cont
205                                                                 122       ==>G: g966 (operator no-change
206                                                                 123         P: p973 (evaluate-prose-in-window
207                                                                 124         S: s984 ((accessed write) (value-of-something)
208                                                                 125         O: read-input
209                                                                 126         O: comprehend
210                                                                 127         O: compare-to-criteria
211 42 M(+x+y) (mid of keywd scrollbar, over elev)    mm 29    mms
212 43                                is show com v 30     v 128 128     O: change-search-criterion ((accessed write))     *** changed search criterion 'write' **
213                                                                                                                        *** changed search criterion 'show' **
214                                                                 129     O: search-for-help
215                                                                 130     ==>G: g1025 (operator no-change
216                                                                 131       P: p1032 (search-for-help
217                                                                 132       S: s1043 ((to-be-found show) (value-of-something)
218                                                                 133       O: find-criterion (keyword)
219                                                                 134       ==>G: g1049 (operator no-change
220                                                                 135         P: p1056 (find-criterion
221                                                                 136         S: s1066 ((to-be-found show) (value-of-something)
222                                                                 137         O: focus-on-current-window
223 43 M(-x-y) (-1/2 in R of keyword 'zanscnt')      mm 31    mi 138 138        O: evaluate-current-window                  goes by but doesn't stop on zanscnt,
224 43 -- (+x+y) (keyword scroll bar, above elevator)         cont
225 43 -- (+y) (above keyword up arrow)                       cont
226                                                                 139          ==>G: g1092 (operator no-change           micro-codable as:
227                                                                 140            P: p1099 (evaluate-items-in-window      162 O: read-input (zanscnt)
228                                                                 141            S: s1109 ((to-be-found show) (value-of-something)
229                                                                 142            O: read-input (write)
230                                                                 143            O: attempt-match
231                                                                 144            O: read-input (wrong)
232                                                                 145            O: attempt-match
233                                                                 146            O: read-input (wrongv)
234                                                                 147            O: attempt-match
235                                                                 148            O: read-input (xin)
236                                                                 149            O: attempt-match
237                                                                 150            O: read-input (xout)
238                                                                 151            O: attempt-match
239                                                                 152            O: read-input (zaltered)
240                                                                 153            O: attempt-match
241                                                                 154            O: read-input (zanscnt)
242                                                                 155            O: attempt-match
243                                                                 156            O: read-input (zarrows)
244                                                                 157            O: attempt-match
245                                                                 158          O: change-current-window
246                                                                 159          ==>G: g1276 (operator no-change
247                                                                 160            P: p1283 (mac-methods-for-change-current-window
248                                                                 161            S: s1291 ((to-be-found show)
249                                                                 162            O: scroll (keyword)
250                                                                 163            ==>G: g1304 (operator no-change
251                                                                 164              P: p1311 (mac-method-of-scroll
252                                                                 165              S: s1320 ((to-be-found show)
253 44 M(-y) (keyword up arrow)                       mm 32    mr 166 166              O: move-mouse) (keyword up)
254 44                             so let's ju v 33     v 128
255 44 D                                              mb 34   mba 167 167              O: press-button
256                    keyword menu scrolls & stops
257 44 U                                              mb 35   mba 168 168              O: release-button
258                                                                 169          O: evaluate-current-window
259                                                                 170          ==>G: g1358 (operator no-change
260                                                                 171            P: p1365 (evaluate-items-in-window
261                                                                 172            S: s1375 ((to-be-found show) (value-of-something)
262                                                                 173            O: read-input (use)
263                                                                 174            O: attempt-match
264                                                                 175            O: read-input (user-vars)
265                                                                 176            O: attempt-match
266                                                                 177            O: read-input (vbar)
267                                                                 178            O: attempt-match
268                                                                 179            O: read-input (vector)
269                                                                 180            O: attempt-match
270                                                                 181            O: read-input (write)
271                                                                 182            O: attempt-match
272                                                                 183            O: read-input (wrong)
273                                                                 184            O: attempt-match
274                                                                 185            O: read-input (wrongv)
275                                                                 186            O: attempt-match
276                                                                 187            O: read-input (xin)
277                                                                 188            O: attempt-match
278                                                                 189          O: change-current-window
279                                                                 190          ==>G: g1547 (operator no-change
280                                                                 191            P: p1554 (mac-methods-for-change-current-window
281                                                                 192            S: s1562 ((to-be-found show)
282                                                                 193            O: scroll (keyword)
283                                                                 194            ==>G: g1575 (operator no-change
284                                                                 195              P: p1582 (mac-method-of-scroll
285                                                                 196              S: s1591 ((to-be-found show)
286 44 D                                              mb 36   mba 197 197              O: press-button
287                    keyword menu scrolls & stops   io
```

```
S1  T  Mouse actions    Window actions      Verbal     ST # Mtype MDC  DC                    Soar trace                                  Comments
 +--+-----------------+--------------------+----------+-+---+---+---+----------------------------------------------------------+------------------------------------------+
288 44 U                                               mb 37   mba 198 198        O: release-button
289 45                                       sure I know how   cont                                                               Continued from what matched dc 128
290                                                                    199       O: evaluate-current-window
291                                                                    200       =>G: g1622 (operator no-change
292                                                                    201         P: p1629 (evaluate-items-in-window
293                                                                    202         S: s1639 ((to-be-found show) (value-of-something)
294                                                                    203         O: read-input (ten)
295                                                                    204         O: attempt-match
296                                                                    205         O: read-input (text)
297                                                                    206         O: attempt-match
298                                                                    207         O: read-input (touch)
299                                                                    208         O: attempt-match
300                                                                    209         O: read-input (unit)
301                                                                    210         O: attempt-match
302                                                                    211         O: read-input (use)
303                                                                    212         O: attempt-match
304                                                                    213         O: read-input (user-vars)
305                                                                    214         O: attempt-match
306                                                                    215         O: read-input (vbar)
307                                                                    216         O: attempt-match
308                                                                    217         O: read-input (vector)
309                                                                    218         O: attempt-match
310                                                                    219       O: change-current-window
311                                                                    220       =>G: g1811 (operator no-change
312                                                                    221         P: p1818 (mac-methods-for-change-current-window
313                                                                    222         S: s1826 ((to-be-found show)
314                                                                    223         O: scroll (keyword)
315                                                                    224         =>G: g1839 (operator no-change
316                                                                    225           P: p1846 (mac-method-of-scroll
317                                                                    226           S: s1855 ((to-be-found show)
318 45 D                                               mb 38   mba 227 227           O: press-button                            these scrolls, all within 1 s in the human,
319                    keyword menu scrolls & stops                                                                              don't correspond to this novice like model.
320 45 U                                               mb 39   mba 228 228           O: release-button                          -- some of this will chunk up in the human.
321                                                                    229       O: evaluate-current-window
322                                                                    230       =>G: g1886 (operator no-change
323                                                                    231         P: p1893 (evaluate-items-in-window
324                                                                    232         S: s1903 ((to-be-found show) (value-of-something)
325                                                                    233         O: read-input (string)
326                                                                    234         O: attempt-match
327                                                                    235         O: read-input (syntaxlevel)
328                                                                    236         O: attempt-match
329                                                                    237         O: read-input (ten)
330                                                                    238         O: attempt-match
331                                                                    239         O: read-input (text)
332                                                                    240         O: attempt-match
333                                                                    241         O: read-input (touch)
334                                                                    242         O: attempt-match
335                                                                    243         O: read-input (unit)
336                                                                    244         O: attempt-match
337                                                                    245         O: read-input (use)
338                                                                    246         O: attempt-match
339                                                                    247         O: read-input (user-vars)
340                                                                    248         O: attempt-match
341                                                                    249       O: change-current-window
342                                                                    250       =>G: g2075 (operator no-change
343                                                                    251         P: p2082 (mac-methods-for-change-current-window
344                                                                    252         S: s2090 ((to-be-found show)
345                                                                    253         O: scroll (keyword)
346                                                                    254         =>G: g2103 (operator no-change
347                                                                    255           P: p2110 (mac-method-of-scroll
348                                                                    256           S: s2119 ((to-be-found show)
349 46 D                                               mb 40   mba 257 257           O: press-button
350                    keyword menu scrolls & stops    io
351 46 U                                               mb 41   mba 258 258           O: release-button
352 47 M(-x-y) ('showh' 2nd f/ top of list)           mm 42    mi 259 259       O: evaluate-current-window
353 47 M(+x+y) (-1/4in R of 'showb', the 1st item)    mm 43    mi 259
354 48 M(+x) (just R & below keyword up arrow)        mm 44    mi 259                                                            partial move to get ready to scroll
355 48 -- (+x) (up arrow)                             mm      cont                                                               again. Fits law!
356 49                                   show B show v 45       v 259
357                                                                    260       =>G: g2150 (operator no-change
358                                                                    261         P: p2157 (evaluate-items-in-window
359                                                                    262         S: s2167 ((to-be-found show) (value-of-something)
360                                                                    263         O: read-input (showb)                    This is a patched in trace
361                                                                    264         O: attempt-match                         from dc 263 to 277
362                                                                    265         O: read-input (showh)
363                                                                    266         O: attempt-match
364                                                                    267         O: read-input (showo)
365                                                                    268         O: attempt-match
366                                                                    269         O: read-input (showt)
367                                                                    270         O: attempt-match
368                                                                    271         O: read-input (sign)
369                                                                    272         O: attempt-match
370                                                                    273         O: read-input (sin)
371                                                                    274         O: attempt-match
372                                                                    275         O: read-input (sinh)
373                                                                    276         O: attempt-match
374                                                                    277         O: read-input (size)
375                                                                    278         O: attempt-match
376                                                                    279       O: change-current-window
377                                                                    280       =>G: g2339 (operator no-change
378                                                                    281         P: p2346 (mac-methods-for-change-current-window
379                                                                    282         S: s2354 ((to-be-found show)
380                                                                    283         O: scroll (keyword)
381                                                                    284         =>G: g2367 (operator no-change
382                                                                    285           P: p2374 (mac-method-of-scroll
383                                                                    286           S: s2383 ((to-be-found show)
384 51 D                                               mb 46   mba 287 287           O: press-button
385 51 U                                               mb 47   mba 288 288           O: release-button
386                                                                    289       O: evaluate-current-window
387                                                                    290       =>G: g2415 (operator no-change
388                                                                    291         P: p2422 (evaluate-items-in-window
389                                                                    292         S: s2432 ((to-be-found show) (value-of-something)
390                                                                    293         O: read-input (scalex)
```

```
S1  T Mouse actions    Window actions     Verbal      ST #  Mtype MDC  DC              Soar trace                           Comments
+--+----------------+------------------+-----------+--+--+-------+---+---------------------------------------------+----------------------------------------+
391                                                                294      O: attempt-match
392                                                                295      O: read-input (scaley)
393                                                                296      O: attempt-match
394                                                                297      O: read-input (set)
395                                                                298      O: attempt-match
396                                                                299      O: read-input (setfile)
397                                                                300      O: attempt-match
398                                                                301      O: read-input (show)
399                                                                302      O: attempt-match
400                                                                303  O: access-item (keyword)
401                                                                304  ==>G: g2527 (operator no-change
402                                                                305    P: p2534 (mac-methods-for-access-item
403                                                                306    S: s2542
404                                                                307    O: click-on-item (i2537)
405                                                                308    ==>G: g2548 (operator no-change
406                                                                309      P: p2555 (mac-method-of-click-on-item
407                                                                310      S: s2562
408 52 M(-x-y) ('showb', 3rd from bot, keyword menu)  mm 48    mr 311 311      O: move-mouse (keyword unspecified)
409    -- (+y) ('show')                                     cont
410 52 C                                               mb 49   mba 312 312      O: click-button
411                                                                313 O: evaluate-help-text
412                                                                314 ==>G: g2576 (operator no-change
413                                                                315   P: p2583 (evaluate-help-text
414                                                                316   S: s2592 ((accessed show) (value-of-something)
415                                                                317   O: focus-on-help-text
416 53                                 I don't kno v 50     v 318 318   O: evaluate-current-window
417 54                                 show binary, pro  cont
418 55                                 show expression.  cont
419 56                                 is an infinite f  cont
420 56 M(-y) (middle R side of help text)          mm 51    mi 318
421 57 M(-y) (a little lower)                      mm 52    mi 318
422 58                                 uhm...      v 53 short
423 58 M(-y) (a little lower)                      mm 54    mi 318
424 60                                 but I wonde v 55     v 318
425 62 M(+x-y) (just L of dn arrow for help txt win) mm 56   mme
426 63                                 for         v 57     v 318
427                                                                319   ==>G: g2615 (operator no-change
428                                                                320     P: p2622 (evaluate-prose-in-window
429                                                                321     S: s2633 ((accessed show) (value-of-something)
430                                                                322     O: read-input
431                                                                323     O: comprehend
432                                                                324     O: compare-to-criteria
433                                                                325   O: change-current-window
434                                                                326   ==>G: g2650 (operator no-change
435                                                                327     P: p2665 (mac-methods-for-change-current-window
436                                                                328     S: s2673 ((accessed show)
437                                                                329     O: scroll (help-text)
438                                                                330     ==>G: g2685 (operator no-change
439                                                                331       P: p2692 (mac-method-of-scroll
440                                                                332       S: s2701 ((accessed show)
441 63 M(+x) (down arrow)                          mm 58    mr 333 333       O: move-mouse (help-text down)
442 64 D            help text win. scrolls         mb 59   mba 334 334       O: press-button
443 65                                 markers     v 60     v 318
444 65 U                                           mb 61   mba 335 335       O: release-button
445                                                                336   O: evaluate-current-window
446                                                                337   ==>G: g2744 (operator no-change
447                                                                338     P: p2751 (evaluate-prose-in-window
448                                                                339     S: s2762 ((accessed show) (value-of-something)
449                                                                340     O: read-input
450                                                                341     O: comprehend
451                                                                342     O: compare-to-criteria
452                                                                343   O: change-current-window
453                                                                344   ==>G: g2787 (operator no-change
454                                                                345     P: p2794 (mac-methods-for-change-current-window
455                                                                346     S: s2802 ((accessed show)
456                                                                347     O: scroll (help-text)
457                                                                348     ==>G: g2814 (operator no-change
458                                                                349       P: p2821 (mac-method-of-scroll
459                                                                350       S: s2830 ((accessed show)
460 66 D            help text win. scrolls         mb 62   mba 351 351       O: press-button
461 67 U                                           mb 63   mba 352 352       O: release-button
462 68                                 okay        v 64     v 336                                                         Gin - This had been 332, a state?
463                                                                353   O: evaluate-current-window                          30-jun-92 FER
464                                                                354   ==>G: g2864 (operator no-change
465                                                                355     P: p2871 (evaluate-prose-in-window
466                                                                356     S: s2882 ((accessed show) (value-of-something)
467                                                                357     O: read-input
468                                                                358     O: comprehend
469                                                                359     O: compare-to-criteria
470                                                                360   O: change-current-window
471                                                                361   ==>G: g2907 (operator no-change
472                                                                362     P: p2914 (mac-methods-for-change-current-window
473                                                                363     S: s2922 ((accessed show)
474                                                                364     O: scroll (help-text)
475                                                                365     ==>G: g2934 (operator no-change
476                                                                366       P: p2941 (mac-method-of-scroll
477                                                                367       S: s2950 ((accessed show)
478 68 D            help text win. scrolls         mb 65   mba 368 368       O: press-button
479 69 U                                           mb 66   mba 369 369       O: release-button
480                                                                370   O: evaluate-current-window
481                                                                371   ==>G: g2984 (operator no-change
482                                                                372     P: p2991 (evaluate-prose-in-window
483                                                                373     S: s3002 ((accessed show) (value-of-something)
484                                                                374     O: read-input
485                                                                375     O: comprehend
486                                                                376     O: compare-to-criteria
487                                                                377   O: change-current-window
488                                                                378   ==>G: g3027 (operator no-change
489                                                                379     P: p3034 (mac-methods-for-change-current-window
490                                                                380     S: s3042 ((accessed show)
491                                                                381     O: scroll (help-text)
492                                                                382     ==>G: g3054 (operator no-change
493                                                                383       P: p3061 (mac-method-of-scroll
```

St	T	Mouse actions	Window actions	Verbal	ST	#	Mtype	MDC	DC	Soar trace	Comments
494									304	S: s3070 ((assessed show)	
495	70	D	help text win. scrolls		mb	67	mba	305	305	O: press-button	
496	72			well, I'll	v	68	v	370			
497	72	U			mb	69	mba	306	306	O: release-button	
498									307	O: evaluate-current-window	
499									308	==>G: g3104 (operator no-change	
500									309	P: p3111 (evaluate-press-in-window	
501									390	S: s3122 ((assessed show) (value-of-something)	
502									391	O: read-input	
503									392	O: comprehend	
504									393	O: compare-to-criteria	
505									394	==>G: state no-change	
506									395	==>G: g3152 (goal no-change	
507									396	==>G: g3159 (goal no-change	
508									397	==>G: g3166 (goal no-change	
509									398	==>G: g3173 (goal no-change	
510									399	==>G: g3180 (goal no-change	

2 Displays of each analytical measure for each episode of Browser-Soar

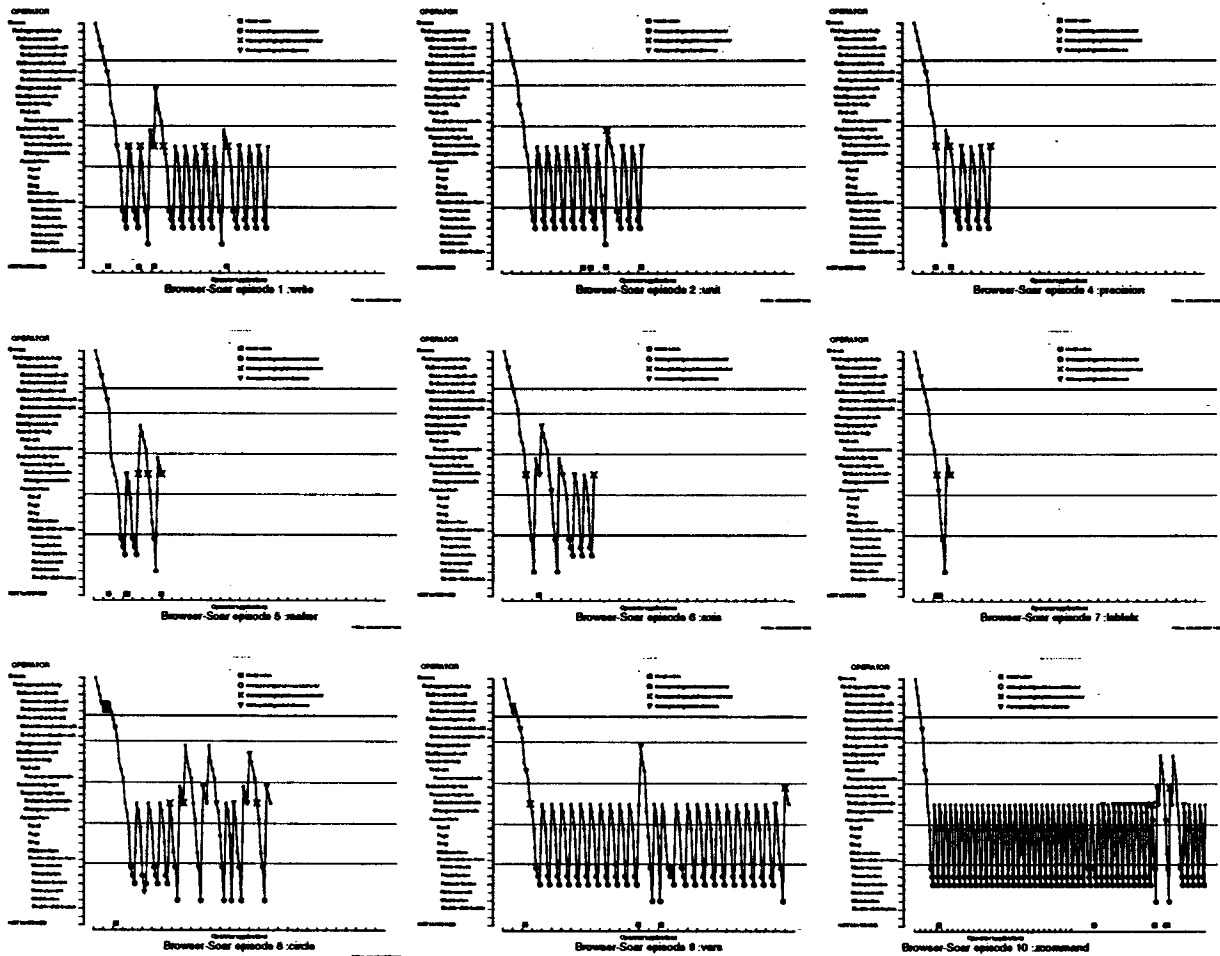
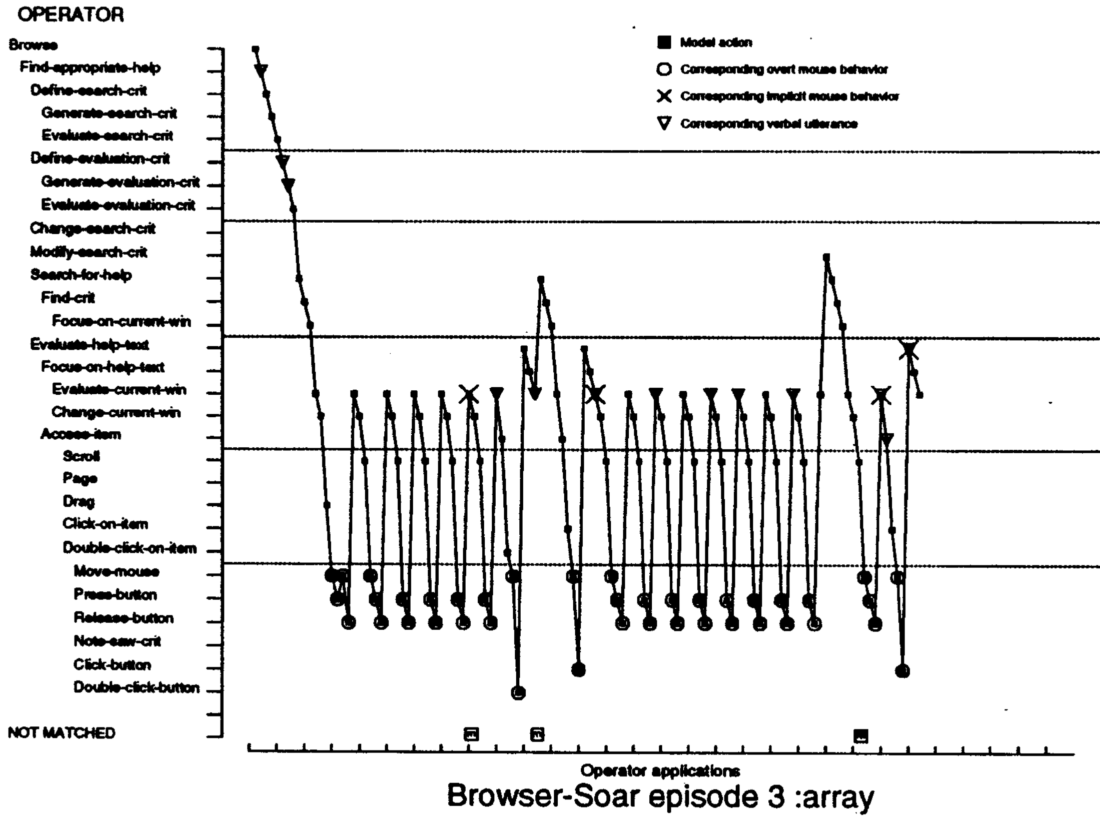


Figure 46: The operator support displays for each of the episodes.



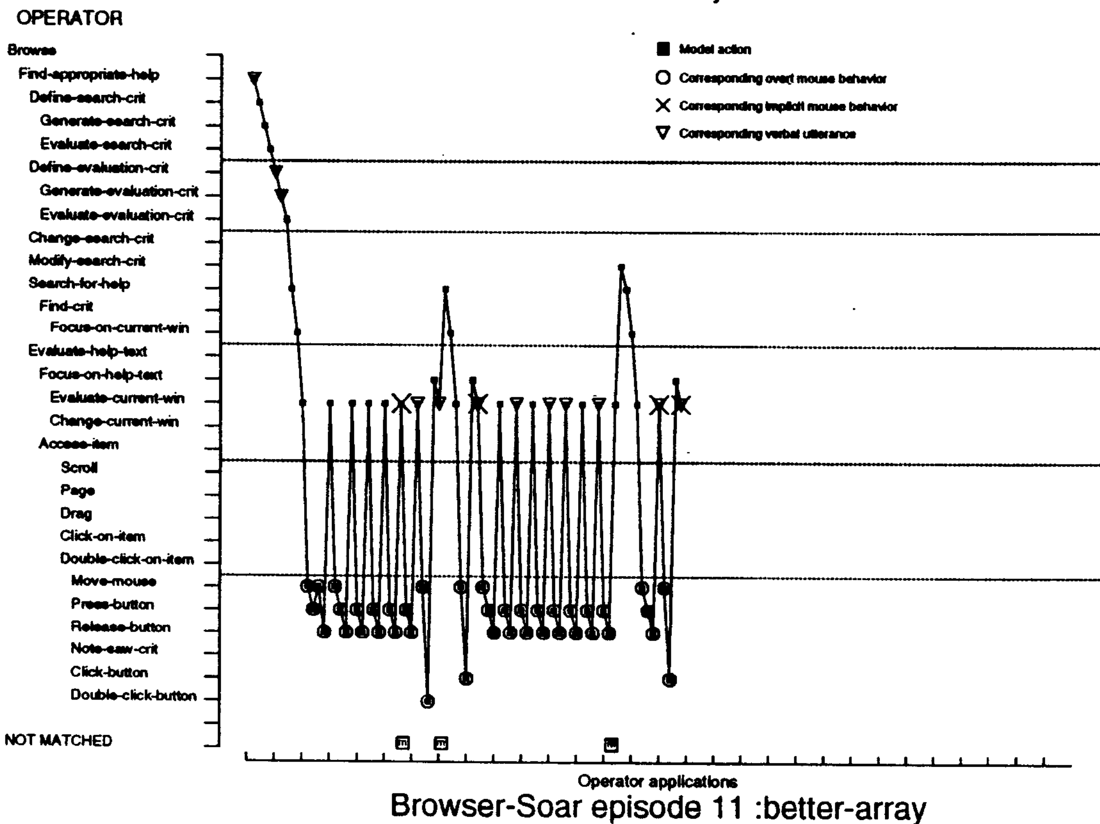


Figure 46: The operator support displays for each of the episodes (cont.).

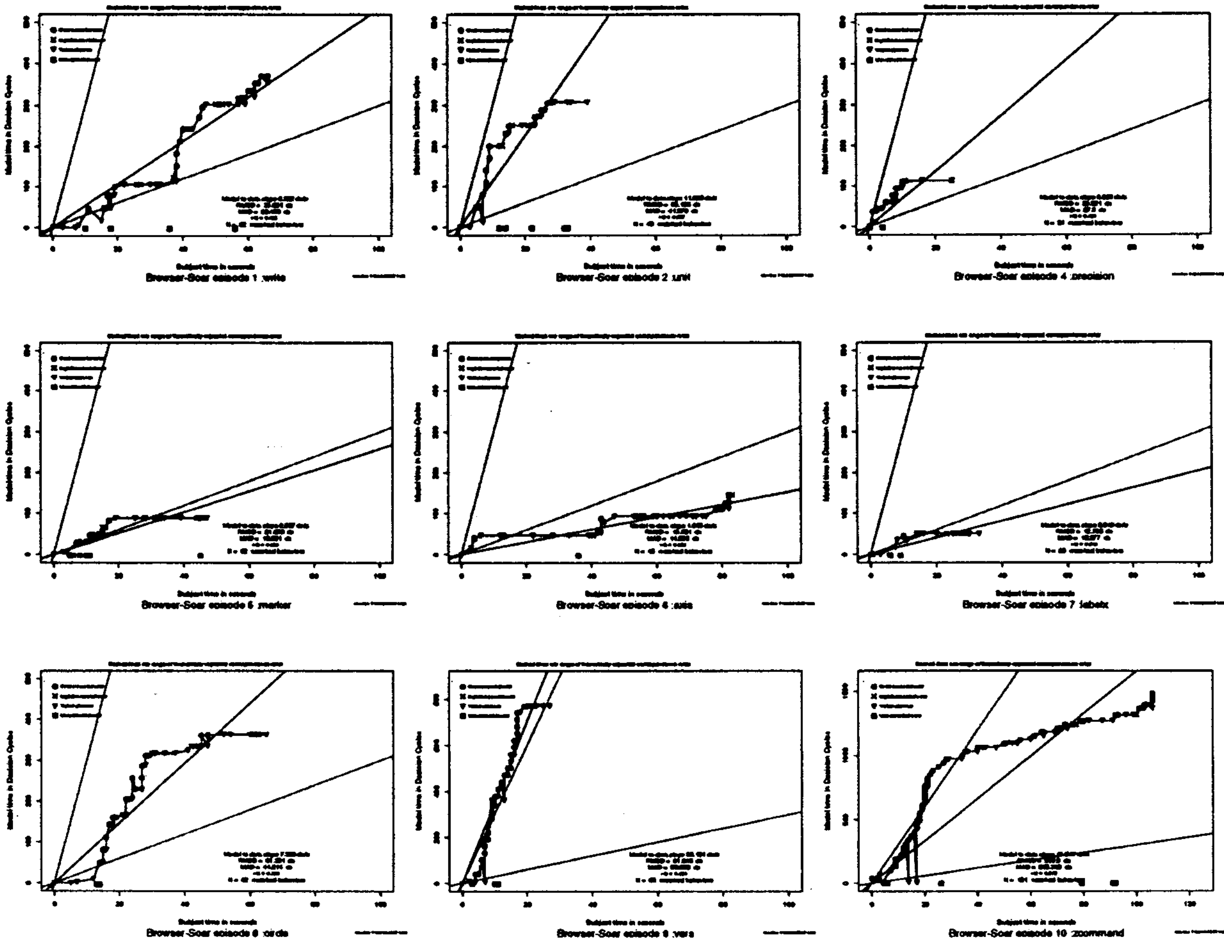
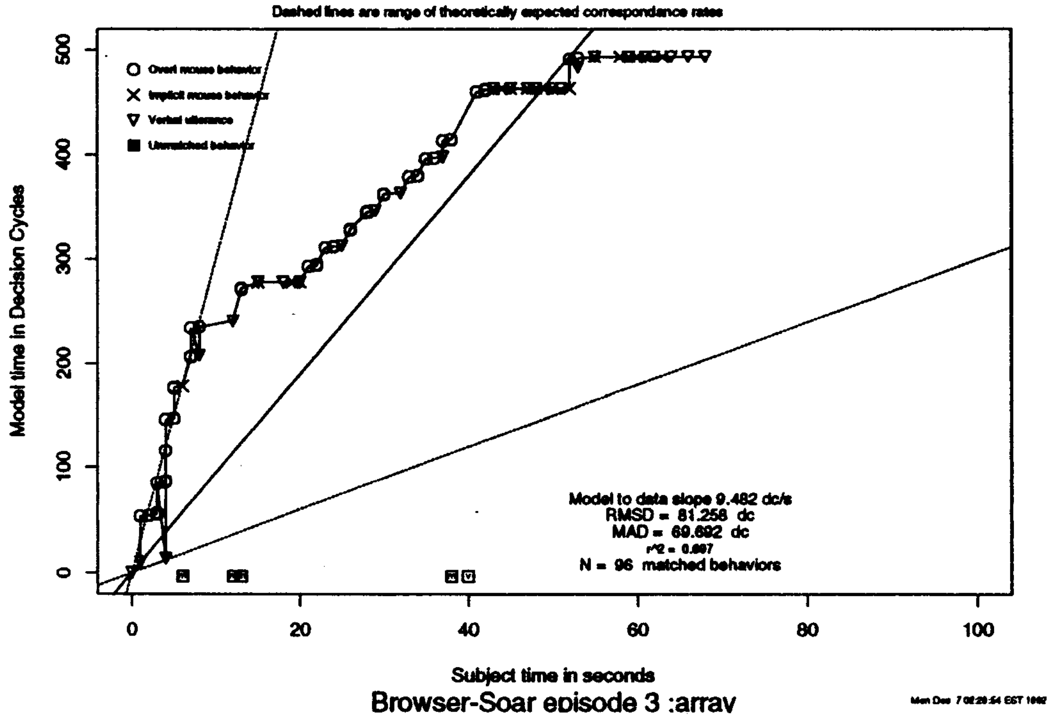


Figure 47: The relative processing rates displays based on decision cycles for each of the episodes.



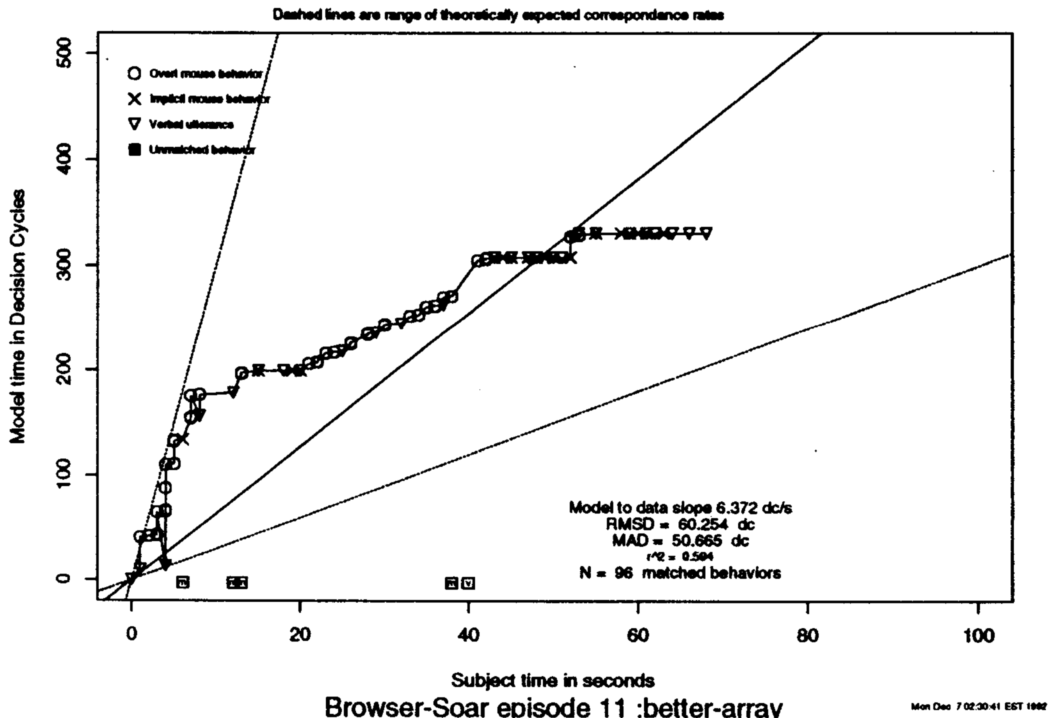


Figure 47: The relative processing rate displays based on decision cycles for of the episodes (cont.).

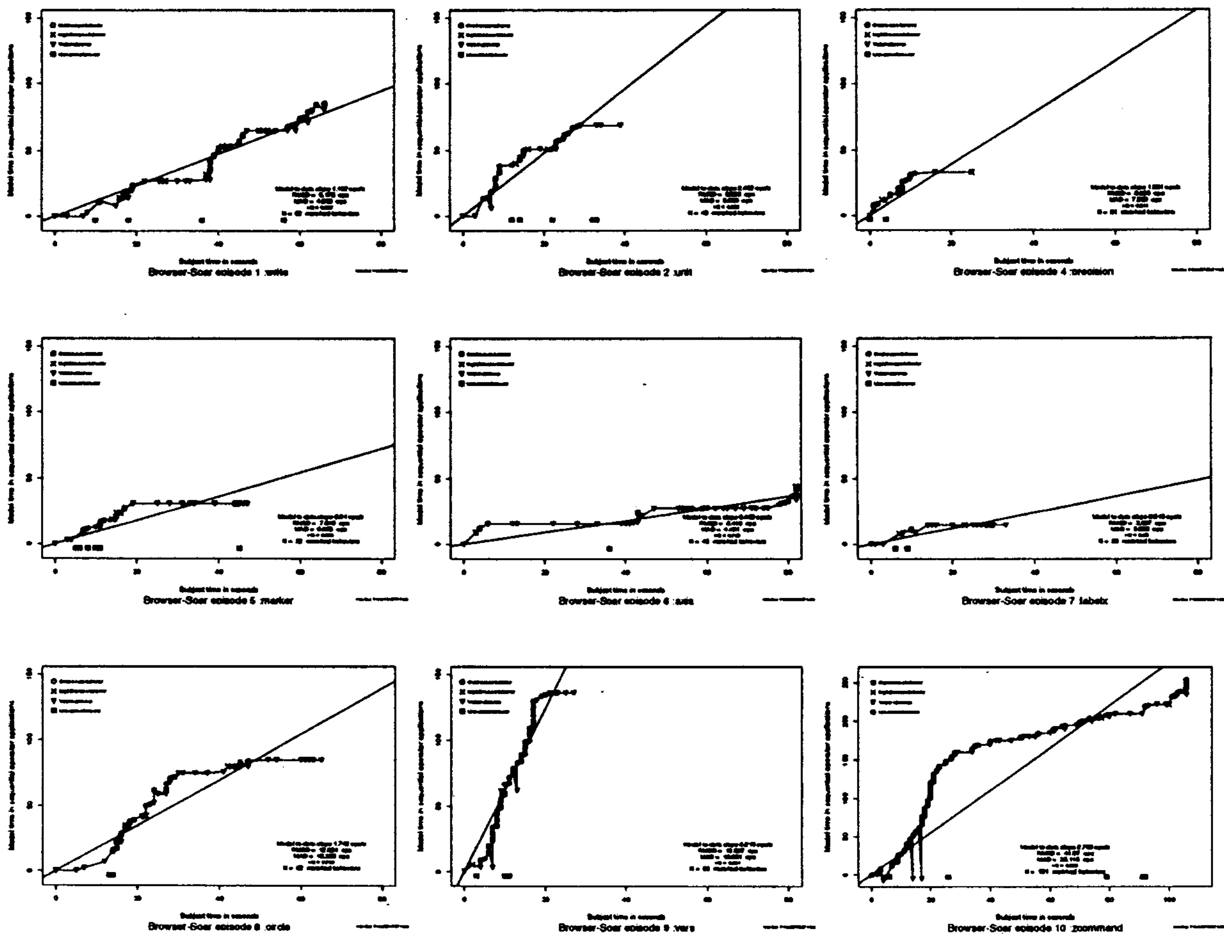
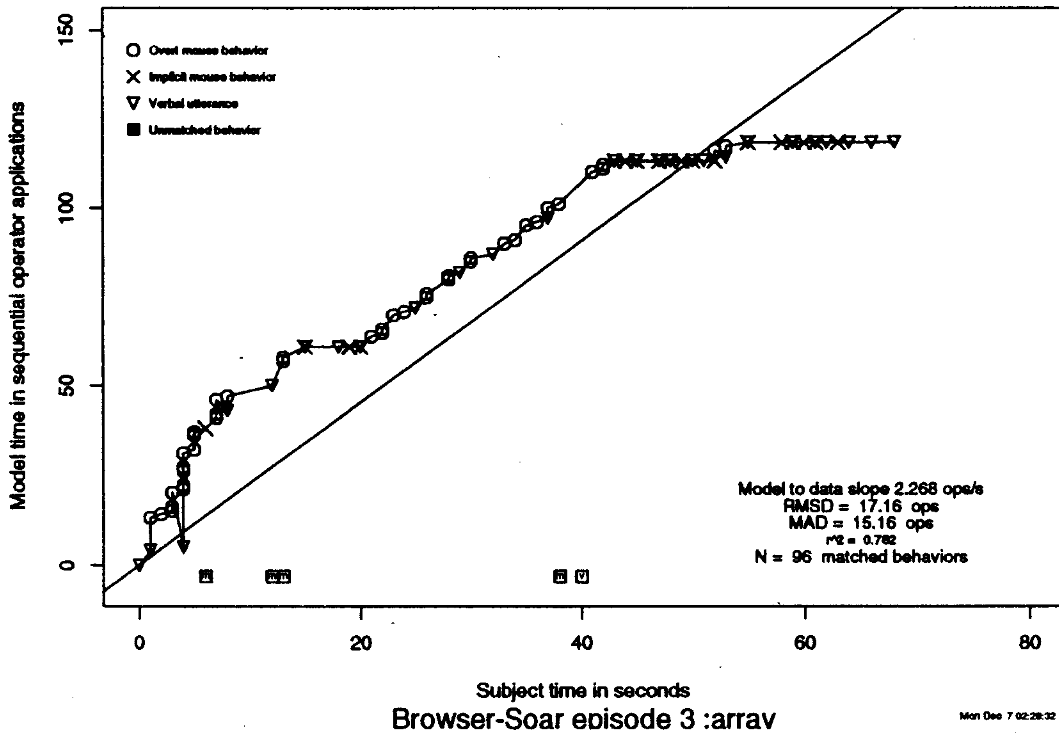


Figure 48: The relative processing rates displays based on operator applications for each of the episodes.



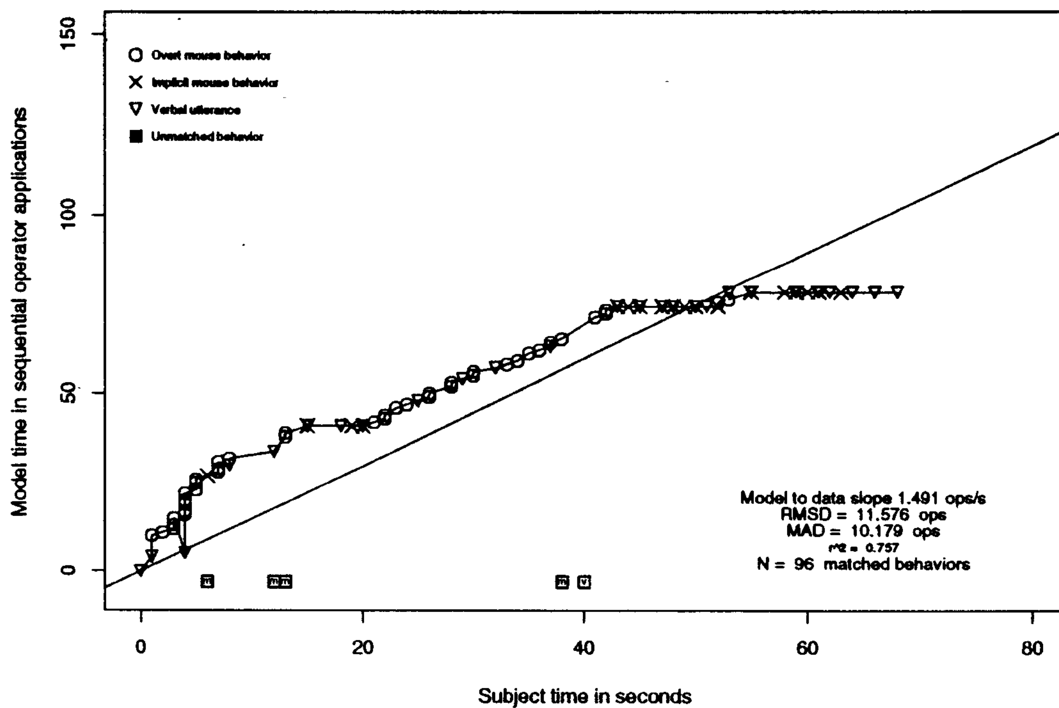


Figure 48: The relative processing rates displays based on operator applications for each of the episodes (cont.).

Chapter 8

Performance demonstration II: Use of Soar/MT components by others

While the environment is integrated, its components have been developed separately. As each component became available, it was spun off for use by others performing subsets of the tasks involved in model testing. The number of users of each tool, their comments, or both, provided feedback on how the various tools help perform (Tesler, 1983) specific tasks of model testing. Together they provides an estimate of the current and potential impact of the whole environment.

Spa-mode has had no use outside of this thesis. As noted earlier, the total environment, but for the displays, was used by V. Peck to perform two episodes of the Browser-Soar reanalysis. The underlying Dismal spreadsheet has had three to four additional users. It still has many problems, so a survey probably will not point out inadequacies not already known.

A survey was conducted of Soar users to find the strengths and weaknesses of the Developmental Soar Interface (DSI).

The other pieces of software either are not used by enough users (Spa-mode, Dismal), or they are so widely used that undertaking a survey is a more serious proposition (S-mode) than can be undertaken as part of this work. Portions of the DSI should no longer be considered pieces of developmental software, for out of the 60 Soar users responding to the survey, two-thirds now use some portion of it every time they use Soar.

8.1 Usage of the Developmental Soar Interface to develop Soar models

The three modules of the DSI (Soar-mode, Taql-mode, and the SX graphic display), have been through several releases. How to obtain them is explained in Appendix I. One or more of the modules are installed at each of the four principle Soar sites in the US, and at sites in Germany and the Netherlands, with over 40 researchers using one or more of the modules.

In the Fall of 1992, a survey (included as an appendix to this chapter) was sent to members of the Soar community identified through the Soar project's mailing lists, workshop attendance lists, and presenters at workshops, as most likely to use Soar in a routine way. In addition to the users directly targeted, an announcement of the survey was emailed to the general Soar mailing list, and an announcement was made at the Soar XI workshop in October, 1992.

Out of the 69 potential users identified, 63 returned a survey (a 92% response rate).. The three people who never actually used Soar were dropped from later analyses. If users that were personally known did not fill in an item, or misidentified a portion of the DSI, this was corrected. Of the people responding, 50 are current members of the Soar community, and 13 are former members.

Table 8-33 shows a listing of the usage patterns. The columns list the components used, with each row representing a single user. The rows are grouped by the sets of components used. The primary tool used is Soar-mode, with 37 of the 60 users reporting using it. The SX graphic display has only been used as a routine tool for debugging by its developer and two other users, but 14 people have used it to create pictures of Soar models and to give demonstrations of their models. Taql-mode has been used and put aside by several people as they became more familiar with the TAQL grammar.

In users' responses of why they did not use additional modules, the largest number of responses (14) was that they did not use TAQL, so they did not need Taql-mode. (This would not necessarily translate into 14 users if they used TAQL.) The next largest concern (12) noted problems with installation and not knowing how to use the tools. Speed (5) was also a concern, and this concern was not limited just to the graphic display, a few users thought that Soar-mode and Taql-mode were slow to load. Most potential users of the SX graphic display were put off by how much it slowed down the system, and while only half the users reported dissatisfaction with its speed, this does not mean that

Table 8-33: Survey responses categorized by usage pattern. Each row represents a user. Totals do not include "tried" users.

Components used	Frequency of usage: Soar	Soar-mode	SX	Taql-mode	Totals
EVERYTHING	daily	daily	Weekly	daily	7
	daily	daily	special	weekly	
	daily	daily	special	daily	
	daily	daily	daily	special	
	weekly	weekly	weekly	monthly	
	weekly	weekly	special	special	
	weekly	weekly	special	daily	
SX & SOAR-MODE	daily	daily	weekly		8
	daily	daily	special	tried	
	daily	daily	special		
	daily	daily	special		
	daily	daily	special		
	daily	daily	special		
	weekly	weekly	special		
	weekly	weekly	weekly	tried	
TAQL-MODE	daily	tried		daily	1
SOAR- & TAQL-MODE	daily	daily		daily	6
	daily	daily	tried	daily	
	daily	daily		daily	
	daily	daily		daily	
	weekly	weekly		weekly	
	monthly	monthly		monthly	
SOAR-MODE	daily	weekly	tried		17
	daily	daily		tried	
	daily	daily			
	daily	daily			
	daily	daily			
	daily	daily			
	daily	daily			
	daily	daily			
	daily	daily			
	weekly	weekly		tried	
	weekly	weekly			
	weekly	weekly			
	weekly	weekly			
	weekly	weekly			
	weekly	weekly			
	monthly	monthly			
	monthly	monthly	tried	tried	
NOTHING	daily	tried		tried	21
	daily	tried			
	daily				
	daily				
	daily				
	daily				
	daily				
	daily				
	daily				
	daily				
	daily				
	daily				
	weekly				
	weekly				
	weekly				
	weekly				
	monthly	tried		tried	
	monthly				
	quarterly				
	na				
	na				
Totals	60	38	15	14	60

they were satisfied with it. There were no underlying problems reported with the metaphor, representations, and manipulation of the problem space level objects.

Other users had problems with the underlying systems that the tools were built on. Several users (4) reported that they did not have a machine that could run the X window system, and some users (2) did not know or want to learn Emacs. A few users, perhaps four or five, use a Macintosh exclusively, or nearly exclusively, and the current environment is unavailable to them.

While only two respondents had not heard of all the software, a few were misinformed. One user did not know that they were using Soar-mode (but loaded it in their startup files), and one did not know that they were using Taql-mode (but when reporting useful Soar-mode features included a feature only in Taql-mode).

Use in video productions. The SX graphic display has been used to make three videos of Soar and Soar models that have been shown outside of CMU. A 20 minute tape of NTD-Soar was shown at a NASA contractors' meeting and as part of a research talk at Queen Mary & Westerfield College, both in the Spring of 1992. A 2 minute video showing the basic interaction method with the DSI and how Soar uses the Garnet toolkit has been shown four times: at the CHI '91 Garnet Special interest group meeting, at the CHI '92 Doctoral Consortium, May 1992, and as part of research talks at the Applied Psychology Unit in Cambridge, England and at Queen Mary & Westerfield College in the Spring of 1992.

Work is underway to create an introductory video explaining Soar (Newell, P., et al., forthcoming). This video is a demonstration of what will be a general capability to take a graphic description of Soar models and create high quality graphic output suitable for commercial broadcast. The initial depictions of the Soar model were created with the SX graphic display and then sent to a commercial computer graphics company for visual enhancement. The project is expected to be completed in the Spring of 1993.

Impact of the DSI on the next release of Soar software: Soar6. In the next release of the Soar software, called Soar 6, several of the features of the DSI have been incorporated or have encouraged the Soar 6 developers to include similar features. These include a very customizable trace, hooks for interacting with Soar-mode, and a better command line interpreter. Soar 6 is still under development; given time, we hope to migrate additional features to Soar 6, such as the ability to display the match set continuously, and the ability to provide a display of which productions will fire on the next decision cycle.

8.2 Usage of S-mode to create functions in S

S-mode has been distributed through three sources that make its total usage hard to compute. It appears, however, to be one of the dominant ways of interacting with S. It was first placed in 1991 in the GNU-Emacs archives at The Ohio State University. This makes it available via anonymous FTP. S-mode has also been distributed via anonymous FTP from the authors' machines. The number of users who picked it up in these two ways cannot be known.

The second mode of distribution, through a statistics software mail server, allows an approximation of a lower bound. Statlib, run by Dr. Michael Meyer at CMU, is a system for distributing statistical software and datasets by electronic mail. The system keeps track of the mail requests for each piece of software and can provide a listing of who requested each piece. Since S-mode was first placed in the Statlib server, there has been 1,043 requests for it, including requests for updated versions (personal communication, M. Meyer, October, 1992).

The exact size of its distribution is confounded further by the nature of GNU-Emacs' copy protection and the nature of S-mode's installation. GNU-Emacs and S-mode are copylefted, which means that users are entitled to (and indeed legally obligated to) provide others with copies upon request, although a copying fee can be charged. How many sites have passed S-mode on would be impossible to

compute. GNU-Emacs and its extensions are installed primarily on multi-user machines and distributed file systems. Once installed, many users can use the same piece of software although on different machines. For example, S-mode has been installed with GNU-Emacs on the Andrew system at CMU (and I don't even know who installed it). Any of the approximately 5,000 Andrew users at CMU can use S-mode.

Appendix to Chapter 8: Survey distributed to Soar users

```
                Survey on the Developmental Soar Interface
                              Frank Ritter
                               12-Oct-92

I'm writing up my thesis and would like to get a better headcount of
how many people use the DSI, and how they use it.  Your comments will
also be used to improve the current interface and serve as background
for future versions.

* How often do you use Soar?
  Daily     Weekly     Monthly     Quarterly     Other (describe)
----------------
* Which of the following have you heard of and which have you used?
                                        Heard of       Have used
  SX graphic display (triangle thingy)  Y  N           Y  N
  Soar-mode                             Y  N           Y  N
  Taql-mode                             Y  N           Y  N

* For items you've heard of, but never used, have you considered using any?
  Any specific reasons why you have not used them?




* Are there any features that you would like to see added to the Soar
  interface for programming, editing, or understanding Soar models ?




If you have not used any items, you can quit here.  Thank you.
```

```
-------------------------------------
SX graphic display (triangle thingy)
-------------------------------------
* How often you use it ?   (tick one and/or write in a modifying number)

  Daily                     Weekly          Monthly               Quarterly
  Tried once or twice       Never
  Special purpose (e.g., demos, making figures; please explain)

* If you don't use the SX graphic display, why don't you use it?



* How do you use it?  (you may tick more than one)

    I've only tried it.
    I use it for special debugging.     I use it for routine development.
    I use it for demos.                 I use it to make presentation diagrams

* How long have you used it (e.g., 3/91 to present) ?

* What are the most valuable features ?



* What are the worst problems/bugs/factors stopping you from using the SX
  graphic display more often?
```

```
----------------
Soar-mode, extensions to gnu-emacs for editing productions.
----------------
* How often you use Soar-mode ? (tick one and/or write in a modifying number)

  Daily                      Weekly          Monthly            Quarterly
  Tried once or twice        Never
  Special purpose (e.g., demos, making figures; please explain)

* How do you use Soar-mode?  (you may tick more than one)

    I use it for special debugging.     I use it for routine development.
    I use it for demos.                 I've only tried it.

* How long have you used soar-mode (e.g., 3/91 to present) ?


* What are the most valuable features of Soar-mode?




* What are the worst problems/bugs/factors stopping you from using
  Soar-mode more often?




* If you don't use Soar-mode, why don't you use it?
```

```
----------------
Taql-mode (extensions to Emacs for editing TAQL constucts)
----------------
* How often do you use taql-mode ? (tick one and/or write in a modifying number)

  Daily                       Weekly         Monthly            Quarterly
  Tried once or twice         Never
  Special purpose (e.g., demos, making figures; please explain)

* How do you use taql-mode?  (you may tick more than one)

    I've only tried it.                  I use it for demos.
    I use it for special debugging.      I use it for routine development.

* How long have you used taql-mode (e.g., 3/91 to present) ?


* What are the most valuable features of taql-mode?




* What are the worst problems/bugs/factors stopping you from using
  taql-mode more often?




* If you don't use taql-mode, why don't you use it?


----------------
Additional on-line & hardcopy copies available from Frank Ritter@cs.cmu.edu

    Please return surveys by email or hardcopy to Frank Ritter@cs.cmu.edu
```

Chapter 9

Contributions and steps toward the vision of routine automatic model testing

Compared with Chapter 1, we are not in the same place in many ways, and we are considerably further along toward the capacity to perform routine process model testing. Progress has been made on defining a methodology for testing the sequential predictions of process models. A computer environment has been implemented to support this methodology, and this environment has been used to test an actual model with actual data. Portions of the environment are used by researchers around the world. The environment was used to test and extend the sequentiality assumption of Ericsson and Simon's (1984) theory of verbal protocol production. The path to an intelligent automatic modeling system based on agent tracking is clearer. Only model testing (open analysis) has been considered in this work, but the methodology and environment should largely be applicable to using models to classify sequential behavior (closed analysis) for such things as cognitive-based testing (Ohlsson, 1990).

The central problem: dealing with large amounts of information. Within the methodology of TBPA the essential problem in testing process models still appears to be one of manipulating and understanding the large amounts of information involved: the model, its predictions, and the data used to test it. Scientists do not decry the difficulty of model creation and manipulation as often as they have the amount of bookkeeping required for testing the sequential predictions. The size of the data sets prove a real problem; the amount of qualitative information used in this task is relatively large given the analyst's limited processing capabilities.

Each of the steps in TBPA requires manipulating large amounts of information. This is a central problem that runs through this work, and it is fought in every tool in the Soar/MT environment. Two approaches have been developed for dealing with it. The first is to automate as many tasks as possible, and to support the analyst for the remainder. The second is to design and use visual displays of information.

Secret weapon #1: Automate and support. Automating aspects of each step reduces the work load required of the analyst. Soar/MT assists the analyst by automatically aligning unambiguous parts of protocols, creating model-based summary displays of the comparison, and providing many aids for displaying and manipulating the model. Although the automatic processes fall short of the ideal speed, and still must be speeded up through better algorithm and data structure design, they have proved useful in their current state. The process is not so inherently large or computationally intensive that so-called super-computing will be required.

The data set presented with Browser-Soar (Peck & John, 1992) is not the largest data set ever used to test a model (although it is fairly large, see Table 2-2), but Soar/MT has substantially speeded up the analysis of this data set. We can now imagine analyzing enough protocol data to achieve Ericsson and Simon's (1984) vision of verbal reports as data.

Supporting the analyst in performing the tasks that are not yet automated has required careful design of the displays and manipulation tools for the large amount of information. The current maximum size of the predictions and data, not including the model, is about 330 Kb. The analyst cannot directly visualize and manipulate information sets the size of a small phone book (5,000 names at 60 bits per name, or 300 Kb total). Special displays have been created to show the important trends in the data, which is the next secret weapon.

Secret weapon #2: Scientific visualization of qualitative information. Appropriate visual displays can support faster processing rates and provide new insights (Larkin & Simon, 1981). Visual displays of qualitative information have become central to quantitative data analysis in many domains and they have lead to the major methodology of scientific visualization.

Visual displays should now be considered essential for performing each step of protocol analysis and process model testing. Visual displays help the analyst understand the model's structure and performance, relating them to each other in a single display, the SX graphic display. Tabular displays of the model's predictions, the data, and their correspondences show simple and directly where the model's predictions do and do not match the data. Other displays aggregate the correspondences in terms of the model components and in terms of relative processing rates. These displays summarize where the model performs well and where it performs poorly, providing clues about where and how to improve the model's fit to the data.

9.1 A methodology for testing the sequential predictions of process models

Trace Based Protocol Analysis (TBPA), a methodology for testing the sequential predictions of process models with protocol data has been defined through listing its inputs, processing steps, and their requirements. TBPA tests a model by running it to generate a trace of how the model performs the task. This trace provides a set of theoretical predictions of what will be found in a subject's verbal and non-verbal protocol, and it is used to interpret the data. TBPA is designed to be an integrated and iterative process, so a summary of where the predictions are unmatched in the protocol is then used to modify the model, and the model is run again. The necessary inputs to TBPA, its steps, and the processing requirements for each step to perform the testing routinely, were specified in enough detail to create a computer environment to support this methodology.

Clarification of the testing process. What it means to test the model became clearer from specifying each step in the process. What are tested in any given episode are the model's predictions. The comparison of the predictions with the data is not just one of alignment. The model's predictions are used to interpret the data. With unambiguous data, such as mouse clicks on menu items, the process appears to be one of simple alignment and it can be treated that way. When the data are verbal protocols, then the items in the trace may provide substantial guidance for interpreting the meaning and function of the information described verbally.

Some theories require every prediction to be matched, but the theory of verbal protocol used to interpret the utterances (Ericsson & Simon, 1984) states that not every possible prediction will be found. The model's predictions are predictions of what could be found in the subject's verbal protocol.

The need for declarative versions of models. It is necessary for model based analysis to refer to structures of the model and to note which parts of the model did and did not apply, or were and were not supported. It is necessary to have declarative representations of process models representing procedural knowledge. Running the model to create the structures upon demand is not enough. There is the simple problem that the structures will be created and then disappear as the context changes. There is also a more complicated problem of coverage, on any given run not all the possible structures will be created. Examining the initial implementation of the model is not adequate either, the model might learn from its environment, and computing all the model's structures is equivalent to running it.

At a minimum, it is necessary to create a description of the model computed by observing the model's performance over time, although combinations of the other methods, such as derivation from the static structure, are a useful adjunct. Although this method is the best way to build the model, even this model is not guaranteed to be complete.

The DSI creates a declarative representation of Soar models. While the Soar model runs, the DSI displays and remembers which and how often the problem spaces, states, and operators have been applied. By watching the model as it runs the DSI builds up as complete a view of the model as is possible. The resulting description can be used by other components in the environment. The interpretation environment can use it to initially code the data. The saved model can be used to summarize the correspondences created though interpreting and aligning the data with respect to the predictions.

9.2 Each step in the methodology was supported in a software environment

An environment to support an analyst performing TBPA has been created based on its requirements. The environment directly supports the main tasks of model tracing; interpreting and aligning the model's predictions with the data, both automatically and semi-automatically; aggregating the comparison data in a variety of displays designed to show how to improve the model; understanding and modifying the model based on how it does not fit.

The steps were specified and broken down to a level that they could be performed automatically, or semi-automatically. Building, loading, and running models was supported in a semi-automatic way. Many small tasks are supported through keystroke macros in the structured editors and smarter interfaces. Finding the emergent properties of Soar models (listing the problem spaces and their operators) is supported, as is counting how often they are instantiated. Unambiguous portions of the subject data are now matched automatically. The same algorithm can be used to interpret and align the data in an incomplete and heuristic fashion, requiring the analyst only to check and clean up the approximate interpretation. Finally, the analytic displays can be automatically created from the comparison data.

The environment also supports the requirements of integrating the steps, automating the tasks where possible, and supporting the analyst for the rest. The environment and the methodology it supports were tested by testing a process model, and in the process learning new things about the model and its fit to the data. The tasks in TBPA that the environment support overlap with other tasks often performed in cognitive model building and modification, data manipulation with a tabular display, and exploratory data analysis.

Sub-portions of the environment supported other users doing the sub-tasks for different reasons, the DSI for AI modeling, Dismal for spreadsheets, and S-mode for statistics and graphing. A survey of users of the DSI found that over half the Soar community uses some portion of the DSI whenever they use Soar. It would be safe to say that pieces of the environment supporting these tasks are in use by over 500 researchers around the world.

The analyses are fast enough to be considered routine. A minute long episode of subject data (approximately 20 verbal segments and 30 motor actions in the browsing task) can now be compared with the model's predictions in 2.5 hours given sufficient inputs, the process model and transcribed data. This is almost within automating range; when it took 60 hours to perform (estimate derived from Ohlsson, 1980), too many under specified processes were required, and automating this task was not conceivable.

Example testing of Browser-Soar using TBPA. The methodology was demonstrated on the Browser-Soar (Peck & John, 1992) model. A set of suggestions for improving Browser-Soar was generated, and one of them was implemented. This lead to a slightly better fit, but more importantly, to a much more parsimonious model. Browser-Soar and its data set did not push this methodology in all directions, but this was good. It allowed making headway on some problems by avoiding others.

9.2.1 Interpreting and aligning the model's predictions and the data

This thesis explored the automatic alignment of unambiguous data to model predictions. The Card algorithm for doing this was slightly improved, and its behavior characterized more clearly.

A spreadsheet approach to the comparison process was demonstrated, and it appears to visually support many of the necessary operations on the data that would otherwise require extensive computation by hand. For example, areas where the predictions match the data in a denser manner is clearly presented. The spreadsheet was also effective in supporting the analyst in easily adjusting the alignment manually when necessary.

9.2.2 Analyzing the results of the testing process

A lack of clarity about what measures are necessary or desirable for measuring predictions fit to the data may have contributed to the lack of progress. The review in Chapter 2 outlined the uses and abuses of several of these measures, and championed Grant's (1962) approach of analytic testing, of finding out where the model can be improved.

A display for showing the support of operators in the model was automated, and an additional family of displays were produced for presenting and analyzing the relative processing rate of the subject with respect to the model. These two sets of displays can be created automatically from the comparison data. They have shown the periodicity of human browsing behavior, the types of mismatches between model and data, and ways to improve the fit of the model. There are many ways for data to not match the model. Additional graphs will be necessary, so an environment is provided to assist in editing and designing these graphs.

9.2.3 Steps related to manipulating the model: Prediction generation and modification

While the model's components are used throughout the analyses, the process model itself is directly involved in two steps, that of generating the sequential predictions, and the final step of revising the model based on the testing process.

Generating the predictions. Generating the model's predictions in a way that they can be used for automatic alignment has required extending infrastructure from the model (in this case, a Soar model) out further toward the data. This has resulted in a better trace — one that is less ambiguous and more readable by humans. Based on the example analysis, we also found that a problem space model must provide state traces in addition to operator traces.

The improved trace lead to an unexpected benefit. We found that deriving aggregate measures in the trace was useful for comparing models and describing their behavior in general terms.

Manipulating and creating models. The Developmental Soar Interface demonstrates the feasibility and utility of several design principles. Across the environment it was possible to meet the design shown in Table 9-34.

Table 9-34: The ease of use and learnability design features met by each tool in the environment.

- Provide a path to expertise through:
    - Menus to drive the interface.
    - Keystroke accelerators available and automatically placed on menus for users to learn.
    - Help provided for each command on request.
    - Hardcopy manuals also available on-line through the menu.
- Treat structures on the theoretical level as first class objects.
- Provide a general tool with macro facilities.

These features make the task of inserting the model's knowledge into Soar easier. Keystroke level models can be presented as evidence for this, as well as the fact that approximately two-thirds of the Soar community now use some portion of the DSI in their daily work.

Node based graph display. Many structure display algorithms draw the complete structure, forcing the user to scroll a window pane across it. Presenting Soar's working memory contents is such a structure

display task. The set of tasks users need to perform when examining the structures within working memory have been identified, and a display meeting these requirements has been designed and implemented. The task analysis lead to a different design than a big scrollable window — a node-based design that allows users to open up individually selected nodes in working memory, close their parents, and so on. The users seem pleased, and it provides a much faster display.

General results about Soar. The visual and structural representations in the Developmental Soar Interface highlighted several features of Soar models and the TAQL macro language. For TAQL, the templates within the structured editor provided a measure of the cumbersome size of the TAQL syntax.

For several specific models we were able to display how their behavior is not best characterized as just search in problem spaces. Behavior within many models now includes routine behavior, search through problem spaces, migration of knowledge between problem spaces, and composition of knowledge.

Within Soar models in general, displaying their behavior graphically pointed out how ephemeral problem spaces and their structures are. In many ways the application and interactions of objects on the problem space level should be considered as emergent behavior. The structure of the model is only available from repeated viewing; the model itself has no representation of itself, and cannot conjure up all the problem spaces and operators that are possible.

9.2.4 The synergy from integration

The environment receives much of its power from integration. The model, its behavior, the subject data, and the comparison of the model and the data all exist in the same environment. This supports several analyses that would be difficult without the integration and it allows them to be much more fluid. Integration allows: (a) direct, preliminary coding of the protocols based on the model's components; (b) appropriate mixed (text and symbolic graphics) presentation of data in the DSI; (c) appropriate mixed (text and symbolic graphics) presentation of data in the analyses; and (d) the portions of the trace that were well aligned and not well aligned could be directly compared with the model's structures.

9.3 Validated and extended the sequentiality assumption of protocol generation theory

Using the TBPA methodology and the Soar/MT environment, the Browser-Soar model and data of Peck & John (1992) were re-examined. Besides providing a test-bed for the methodology and environment, this effort yielded the following new scientific result.

The verbal protocol production theory of Ericsson and Simon (1984) assumes that working memory structures are reported in the order that they enter working memory. This assumption can be tested with a model that predicts when objects enter working memory. The Soar/MT display of the relative processing rates of the Browser-Soar model and the subject provided a direct visual test of this assumption. The underlying data structures were then directly queried to confirm and count the number of sequential and non-sequential pairs of events there were. In every episode of the Browser-Soar, the sequentiality assumption was found to hold for the verbal protocol. An examination of the non-verbal protocol segments found that they too were always performed in the same order as the model, both for overt task actions, and for actions that were not directly related to the task, such as moving the mouse pointer over words being read on the screen.

The two data streams appeared to be presented in a non-sequential order. Verbal utterances typically lagged 10 to 30 simulation cycles (approximately 1 to 3 s) behind the overt actions; and rarely (3/300) they lagged up to 400 simulation cycles (approximately 40 s).

The shorter lags were probably reports of working memory delayed by workload associated with the task, and minor inconsistencies in the model. Examination of the correspondences showed that the

primary cause of the long lags was probably an artifact of the interpretation process. The verbal utterances in the analysis were matched to operators rather than to the state information created by the operators. This approximation simplified the analysis considerably, and it should remain available — it is a valuable technique. But it must be seen as only an approximation; one that will sometimes lead to inconsistencies in the comparison. Any operator that sets up long lasting state information can cause this problem.

As a result of these analyses it is proposed that the sequentiality assumption holds for both verbal utteranances and task actions. Including motor task actions as part of the protocol provides reference points for fixing the correspondences between the predictions and subject's actions, and allows the lag of the verbal utterances to be measured.

9.4 Progress toward the vision of routine applied theoretically guided protocol analysis

This work has made appreciable progress toward the vision of automatic modeling. All the parts of Soar/MT are part of a grand vision of what an integrated modeling and data analysis system would need to do, and could do. The major steps and inputs have been identified as the parts of TBPA, and a prototype environment has been created that an automatic modeling system would need. The next steps will be to create initial models, and to provide a more intelligent process for interpreting ambiguous data with respect to the model's predictions.

Because this environment is based on an architecture for general intelligence, it is conceptually possible to add knowledge to the architecture of how to perform parts or all of the analysis. To do this completely would require incorporating a complete model of the analyst. However, the architecture used in this environment, Soar, also learns. So perhaps an easier, but less direct way to automate this task might be through having a Soar-based agent learn to perform the analyses by watching a series of analyses. As it watched a series of routine analyses over similar episodes be performed, it could follow along, learning how to run the analyses, and then driving the analyses programs itself.

Not that we are there, but we can now see further down the path toward completely automatic modeling. If NL-Soar (a Soar system for interpreting natural language) were to be incorporated, then Soar/MT might take in instructions for different experiments, and use the models that NL-Soar creates from reading the instructions as initial models to predict the behavior of subjects for each experiment (Lewis, Newell & Polk, 1989; Newell, 1991). The alignment also could be automated. The non-verbal overt actions can be compared directly; the verbal utterances would have data structures, the predictions, laying around that are designed to be sufficient to parse them. NL-Soar (Lehman et al., 1991) is available as a potential parser designed to use these predictions.

This style of protocol analysis requires further computer science and AI work: performing the alignment of predictions to natural language, running the models more quickly, and gathering better statistics. But it remains a task within psychology: the real use is for comparing protocols against models' predictions.

Remaining problems. Many problems remained in this methodology and environment. I would like to note a few here to admit its deficiencies, to warn potential users of the current specificity of the tasks Soar/MT can address, and to suggest directions for future work.

How to aggregate support from the predictions to the model structures is not always as straightforward as it appeared in the sample analysis of Browser-Soar. There is a problem of specifying how the predictions are used to interpret the data. There is also a problem in specifying how to aggregate support for model components. Across episodes, the structures in the model that generated the predictions remain and summarize the behavior over time. The current model implemented its operators rather directly and in the same manner each time. This need not be the case. Consider an *Add* operator such as Siegler uses in his work modeling children's arithmetic knowledge (Siegler,

1988; Siegler & Shrager, 1984). Different operands result in different reaction times and error patterns. Assigning support to an operator in this case must be differentiated by the operator's arguments, and a representation for this must be developed. So there is an additional step to TBPA, not yet made explicit, of translating the support that individual predictions receive from the data back to the structures in the model that generated them.

The analyst is currently left with an abduction task of improving the fit with indications of where the model does not fit and with tools for understanding and modifying the model. There are some simple rules that would apply in specific circumstances, and these were noted in the chapter describing the graphical measures of model fit. The possibility of finding a more complete and algorithmic description, like Heise (1987, 1989; Corsaro & Heise, 1990; Heise & Lewis, 1991) provides for his models, should be explored.

Speed, always and everywhere — the analyst always desires a faster system that performs more complicated analyses automatically. Partial views of the data and model are included in this. The recent translation of Soar to the C language offers a speedup in the basic architecture. Taking advantage of this may require translating the DSI into C.

Directions for future work. The way to improve this methodology is the same way to improve a model, by testing and using it on additional models and data sets. Some preliminary discussions have taken place with other researchers about using Soar/MT to test their process models, usually models implemented in Soar.

The software environment could be automated further, and as noted in Chapter 3, the next direct step toward automatic agent modeling would be to represent the knowledge to perform a single step as a Soar model. This would provide further automation. One of the potential places for doing this would be to have NL-Soar parse the verbal utterances, another would be to further automate the generation of the analytical diagrams.

9.5 Concluding remarks

We build our theories, test them, then modify them, iterating through a loop. This loop was described briefly and perhaps for the first time with respect to process models and protocol analysis by Feldman (1962, p. 342). But not surprisingly, it is like all theory testing in science. Models are not primarily tested to be rejected (as the popularization of Popper's (1959) views goes), or tested simply with a significance test to determine their value, but models are tested in order to improve them (Grant, 1962; Newell, 1990, p. 14). By using protocols to test these models, we are not attempting to *code* a segment so that it is *coded*, but we are using the data to build a model (e.g., a simulation process model). That is, to test whether subjects perform the same actions in the same order as the model predicts.

Because they will allow us to see new things, new analyses and tools are also science (Hall, 1992; Laird & Rosenbloom, 1992; Newell, 1991; Ohlsson, 1990; Simon, 1991). New scientific problems are found this way (Toulmin, 1972). Indeed, much of what science consists of — what is passed on from generation to generation of scientists — is just technique (Ohlsson, 1990; Toulmin, 1972).

Because of the difficulties associated with creating process models and of manipulating protocol data, sometimes analysts have lost sight of this fundamental nature of protocol analysis. The technique of testing process models' predictions of sequential behavior has been nudged forward just a bit.

References

Afifi, A., & Clark, V. (1984). *Computer-aided Multivariate Analysis.* New York: Van Nostrand Reinhold Company.

Altmann, E. (February, 1992). Toward Human-scale task performance: Preliminaries. Talk presented at the Soar X workshop.

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, Massachusetts: Harvard University Press.

Anderson, J. R. (in press). *Rules of the Mind.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, J. R., & Bower, G. H. (1973). *Human associative memory.* Hillsdale, NJ: Lawrence Erlbaum Associates. Third revised printing 1979.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science, 13*(4), 467-505.

Anderson, J. R., Farrell, R., & Sauers, R. (1981). Learning to program in Lisp. *Cognitive Science, 8*, 87-129.

Anderson, J. R., Greeno, J. G., Kline, P. J., & Neves, D. M. (1981). Acquisition of problem-solving skill. In Anderson, J. R. (Ed.), *Cognitive skills and their acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Anjewierden, A., Wielemaker, J., & Toussaint, C. (1990). Shelley — Computer aided knowledge engineering. In Wielinga, B., Boose, J. H., Gaines, B. R., Schreiber, G, & van Someren, M. (Eds.), *Current trends in Knowledge acquisition (Proceedings of the 4rth European workshop on knowledge acquisition, EKAW-90, Amsterdam.* Amsterdam: IOS Press.

Anzai, Y., & Simon, H. A. (1979). The theory of learning by doing. *Psychological Review, 86*, 124-140.

Ardis, M. A. (1987). Template-Mode for GNU Emacs. Available from The Ohio State University elisp archives on archive.cis.ohio-state.edu as file pub/gnu/emacs/elisp-archive/modes/templatemode.tar.Z.

Atwood, M. E., & Poulson, P. G. (1976). A process model for water jug problems. *Cognitive Psychology, 8*, 191-216.

Bates, D., Kademan, E., & Ritter, F. E. (Fall 1990, revised Fall 1991). *S-mode for GNU Emacs.* Available from the Statlib software archive (S is a statistics package, Statlib is statlib@lib.stat.cmu.edu).

Becker, R.A., Chambers, J.M., & Wilks, A.R. (1988). *The New S Language.* Pacific Grove, CA: Wadsworth and Brooks/Cole.

Bentler, P. M. (1980). Multivariate analysis with latent variables: Causal modeling. *Annual Review of Psychology, 31*, 419-456.

Bhaskar, R. (1978). *Problem solving in semantically rich domains.* Doctoral dissertation, Carnegie-Mellon University.

Bhaskar, R., & Simon, H. A. (1977). Problem solving in semantically rich domains: An example from engineering thermodynamics. *Cognitive Science, 1*, 193-215.

Bree, D. S. (1968). *The understanding process as seen in geometry theorems.* Doctoral dissertation, Carnegie Mellon University.

Brooks, F. P. (1975). *The mythical man-month: Essays on software engineering.* Reading, MA: Addison-Wesley Pub. Co.

Brown, C. R. (1986). The verbal protocol analysis tool (VPA): Some formal methods for describing expert behavior. In *Proceedings 2nd Symposium on Human Interface, Oct. 29-30, Tokyo, Japan,* 561-567.

Brown, J. S., & Burton, R. B. (1980). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science, 2,* 155-192.

Brueker, J., & Wielinga, B. (1989). Models of expertise in knowledge acquisition. In Guida, G., & Tasso, C. (Eds.), *Topics in expert system design.* North Holland: Elsevier Science Publishers B.V.

Card, S. K., Moran, T. P., & Newell, A. (1980). The keystroke-level model for user performance time with interactive systems. *Communications of the ACM,* Vol. *23*(7).

Card, S., Moran, T., & Newell, A. (1983). *The psychology of human-computer interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Carley, K. (1988). Formalizing the social expert's knowledge. *Sociological methods and research, 17*(2), 165-232.

Carpenter, P. A., Just, M. A., & Shell, P. (1990). Cognitive coordinate systems: Accounts of mental rotation and individual differences in spatial ability. *Psychological Review, 92,* 137-172.

Chambers, J.M., & Hastie, T.J., eds. (1992). *Statistical Models in S.* Pacific Grove, CA: Wadsworth and Brooks/Cole.

Cohen, M. S., Payne, D. G., & Pastore, R. E. (1991). Computerized task analysis. *SIGCHI Bulletin, 23*(4), 57-58.

Corsaro, W. A., & Heise, D. R. (1990). Event structure models from ethnographic data. In Clogg, C. (Ed.), *Sociological methodology: 1990.* Cambridge, MA: Basil Blackwell.

Dansereau, D. (1969). *An information processing model of mental multiplication.* Doctoral dissertation, Department of Psychology, Carnegie-Mellon University.

Diederich, J., Ruhmann, I, & May, M. (1987). KRITON: A knowledge-acquisition tool for expert systems. *International Journal of Man-Machine Studies, 26,* 29-40.

Dillard, J. F., Bhaskar, R., & Stephens, R. G. (1982). Using first-order cognitive analysis to understand problem solving behavior: An example from accounting. *Instructional Science, 11*(1), 71-92.

Doorenbos, R., Tambe, M., & Newell, A. (1992). Learning 10,000 chunks: What's it like out there? *Proceedings of the Tenth National Conference on Artificial Intelligence.* AAAI.

Dukes, N. F. (1968). N=1. *Psychological Bulletin, 64*(1), 74-79.

Embretson, S. E. (1992). Computerized adaptive testing: Its potential substantive contributions to psychological research and assessment. *Current directions in psychological science, 1*(4), 129-131.

Ericsson, K. A., & Simon, H. A. (1980). Protocol analysis: Verbal reports as data. *Psychological Review, 87,* 215-251.

Ericsson, K. A., & Simon, H. A. (1984). *Protocol analysis: Verbal reports as data.* Cambridge, MA: The MIT Press.

Feigenbaum, E. A., & Simon, H. S. (1984). EPAM-like models of recognition and learning. *Cognitive Science, 8,* 305-336.

Feldman, J. (1962). Computer simulation of cognitive processes. In Borko, H. (Ed.), *Computer applications in the behavioral sciences.* Englewood cliffs, NJ: Prentice-Hall.

Feldman, J., Tonge, F. M., & Kanter, H. (1991). Empirical explorations of a hypothesis-testing model of binary choice behavior. Hoggatt, A. C., & Balderston, F. E. (Eds.), *Symposium on simulation models.* Cincinnati, OH, South-Western Publishing Company.

Fielding, N. G., & Lee, R. M. (Eds.). (1991). *Using computers in qualitative research.* London & Beverly Hills, CA: Sage.

Finlay, J., & Harrison, M. (1990). Pattern recognition and interaction models. Diaper, D., et al. (Eds.), *Human-computer interaction — INTERACT '90.* IFIP, Elsevier Science Publishers B. V.

Fisher, C. (1987). Advancing the study of programming with computer-aided protocol analysis. In Olson, G., Soloway, E., & Sheppard, S. (Eds.), *Empirical studies of programmers: Second workshop.* Norwood, NJ: Ablex.

Fisher, C. (1991). *Protocol Analyst's Workbench: Design and evaluation of computer-aided protocol analysis.* Doctoral dissertation, Department of Psychology, Carnegie-Mellon University.

Forgy, C. L. (1981). *OPS5 user's manual* (Tech. Rep.) CMU-CS-81-135. Department of Computer Science, Carnegie-Mellon University.

Free Software Foundation. (1988). *GNU Emacs.* Boston: Free Software Foundation. Directions for obtaining GNU-Emacs are available by FTPing file /pub/gnu/GNUinfo/FTP on prep.ai.mit.edu, using the anonymous FTP protocol.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-Completeness.* New York, New York: W. H. Freeman and Company.

Garlick, S. & VanLehn, K. (1987). *CIRRUS: An automated protocol analysis tool* (Tech. Rep.) 6. Department of Psychology, Carnegie-Mellon University.

Gascon, J. (1976). *Computerized protocol analysis of the behavior of children on a weight seriation task.* Doctoral dissertation, Departement de Psychologie, Universitè de Montrèal.

Gottman, J. M., & Roy, A. K. (1990). *Sequential analysis: A guide for behavioral researchers.* Cambridge, UK: Cambridge University Press.

Grant, D. A. (1962). Testing the null hypothesis and the strategy and tactics of investigating theoretical models. *Psychological Review, 69,* 54-61.

Greenblatt, R. D., Knight, T. F. Jr., Holloway, J., Moon, D. A., & Weinreb, D. L. (1984). The LISP machine. In Barstow, D. R., Shrobe, H. E., & Sandewall, E. (Eds.), *Interactive programming environments.* New York, NY: McGraw-Hill.

Greeno, J. G., and Simon, H. A. (1984). Problem solving and reasoning. In Atkinson, R. C., Herrnstein, G., Lindzey, G., and Luce, R. D. (Eds.), *Stevens' handbook of experimental psychology, 2nd edition, Volume II.* New York, NY: John Wiley & Sons. Also available as tech report UPITT/LRDC/ONR/APS-14.

Gregg, L. W., & Simon, H. S. (1967). An information-processing explanation of one-trial and

incremental learning. *Journal of Verbal Learning and Verbal Behavior, 6*, 780-787.

Hall, S. (1992). How technique is changing science. *Science, 257*, 344-349.

Hansen, J. P. (1991). The use of eye mark recordings to support verbal retrospection in software testing. *Acta Psychologica, 76*, 31-49.

Hegarty, M. (1988). *Comprehension of diagrams accompanied by text.* Doctoral dissertation, Department of Psychology, Carnegie-Mellon University.

Heise, D. R. (August 1987). Computer assisted analysis of qualitative field data, Didactic seminar, Session 176, American Sociological Association, Chicago.

Heise, D. R. (1989). Modeling event structures. *Journal of Mathematical Sociology, 14*(2-3), 139-169.

Heise, D. R. (1991). Event structure analysis: A qualitative model of quantitative research. In Fielding, N. G., & Lee, R. M. (Eds.), *Using computers in qualitative research.* London: Sage.

Heise, D., & Lewis, E. (1991). *Introduction to ETHNO.* Dubuque, Iowa: Wm. C. Brown Publishers.

Hinton, G. E., & Sejnowski, T. J. (1986). Learning and relearning in Boltzmann Machines. In *Parallel distributed processing: Explorations in the microstructure of cognition. Volume 1: Foundations.* Cambridge, Massachusetts: The MIT Press.

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM, 18*(6), 341-343.

James, J. M., Sanderson, P. M., & Seidler, K. S. (1990). *SHAPA Version 2.0: Instruction manual and reference* (Tech. Rep.) EPRL-90-16/M. Engineering Psychology Research Laboratory.

Jefferys, W. H., & Berger, J. O. (1992). Ockham's Razor and Bayesian Analysis. *American Scientist, 80*(January-February), 64-72.

John, B. E. (1988). *Contributions to engineering models human-computer interactions, Volume 1.* Doctoral dissertation, Department of Psychology, Carnegie-Mellon University.

John, B. E. (1990). Applying cognitive theory to the evaluation and design of human-computer interfaces. Final report to US West sponsored research program.

John, B. E., & Vera, A. H. (May 1992). A GOMS analysis of a graphic, machine-paced, highly interactive task. *CHI'92 Proceedings of the Conference on Human Factors and Computing Systems.* NewYork: SIGCHI, ACM Press.

John, B. E., Vera, A. H., & Newell, A. (December 1990). *Toward Real-Time GOMS* (Tech. Rep.) CMU-CS-90-195. School of Computer Science, Carnegie-Mellon University.

John, B.E., Remington, R.W. & Steier, D.M. (May 1991). *An Analysis of Space Shuttle Countdown Activities: Preliminaries to a Computational Model of the NASA Test Director* (Tech. Rep.) CMU-CS-91-138. School of Computer Science, Carnegie-Mellon University.

Johnson, T. R., & Smith, J. W. (1991). A Framework for Opportunistic Abductive Strategies. *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society.* Hillsdale, New Jersey: Cognitive Science Society, Lawrence Erlbaum Associates.

Johnson, P. E., Dura'n, A. S., Hassebrock, F., Moller, J., Prietula, M., Feltovich, P. J., Swanson, D. B. (1981). Expertise and error in diagnostic reasoning. *Cognitive Science, 5*, 235-283.

Just, M. A., & Carpenter, P. A. (1980). A theory of reading: From eye fixations to comprehension. *Psychological Review, 87*(4), 329-354.

Just, M. A., & Carpenter, P. A. (1985). Cognitive coordinate systems: Accounts of mental rotation and individual differences in spatial ability. *Psychological Review, 92*(2), 137-172.

Just, M. A., & Carpenter, P. A. (1987). *The psychology of reading and language comprehension.* Newton, MA: Allyn & Bacon.

Just, M. A., & Carpenter, P. A. (1992). A capacity theory of comprehension: Individual differences in working memory. *Psychological Review, 99*, 122-149.

Just, M. A., & Thibadeau, R. A. (1984). Developing a computer model of reading times. In Kieras, D. E., & Just, M. A. (Eds.), *New methods in reading comprehension research.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Kadane, J. B., Larkin, J. H., & Mayer, R. H. (1981). A moving average model for sequenced reaction-time data. *Journal of Mathematical Psychology, 23*(2), 115-133.

Kaplan, C. (1987). Computer simulation: Separating fact from fiction. Published as Technical report #498 in the C. I. P. Series, Department of Psychology, Carnegie-Mellon University.

Karat, J. (1968). A model of problem solving with incomplete constraint knowledge. *Cognitive Psychology, 14*, 538-559.

Kennedy, S. (1989). Using video in the BNR usability lab. *SIGCHI Bulletin, 21*(2), 92-95.

Kiearas, D. (May 1992). Personal communication.

Klahr, D., & Dunbar, K. (1988). Dual space search during scientific reasoning. *Cognitive Science, 12*(1), 1-48.

Koedinger, K. R., & Anderson, J. R. (1990). Abstract planning and perceptual chunks: Elements of expertise in geometry. *Cognitive Science, 14*(4), 511-550.

Kolen, J. F., & Pollack, J. B. (1988). Scenes from exclusive-or: Back propagation is sensitive to initial conditions. *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society.* Cognitive Science, LEA.

Kowalski, B., & VanLehn, K. (1988). Cirrus: Inducing subject models from protocol data. *Proceedings of the Tenth Annual Conference of the Cognitive Science Society.* Cognitive Science, LEA.

Krishnan, R., Li, X., & Steier, D. M. (September 1992). Development of a knowledge-based mathematical model formulation system. *Communications of the ACM, 35*(9), 138-146.

Kulkarni, D., & Simon, H. A. (1988). The process of scientific discovery: The strategy of experimentation. *Cognitive Science, 12*, 139-176.

Laird, J. E., & Rosenbloom, P. S. (1992). In pursuit of mind: The research of Allen Newell. To appear in *AI Magazine.*

Laird, J.E., Congdon, C.B., Altmann, E. & Swedlow, K. (October 1990). *Soar User's Manual: Version 5.2* (Tech. Rep.) CSE-TR-72-90. Electrical Engineering and Computer Science Department, University of Michigan. Also available from The Soar Project, School of Computer Science, Carnegie-Mellon University, as technical report CMU-CS-90-179.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33*(1), 1-64.

Langley, P., & Ohlsson, S. (1989). Automated cognitive modeling. *Proceedings of AAAI-84.* Los Altos, CA, Morgan Kaufman.

Langley, P., Bradshaw, G. L., & Simon, H. A. (1983). Rediscovering chemistry with the Bacon system. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.), *Machine learning, an artificial intelligence approach.* Palo Alto, CA: Tioga.

Larkin, J. H. (1981). Enriching formal knowledge: A model for learning to solve textbook physics problems. In Anderson, J. R. (Ed.), *Cognitive skills and their acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Larkin, J. H., & Simon, H. A. (1981). Learning through growth of skill in mental modeling. *In Proceedings of the Third annual conference of the Cognitive Science Society.* Cognitive Science Society, Lawrence Erlbaum Associates.

Larkin, J. H., & Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science, 11,* 65-99.

Larkin, J. H., Mayer, R. H., & Kadane, J. B. (1986). An information-processing model based on reaction times in solving linear equations. *Journal of Mathematical Psychology, 23*(2), 115-133.

Larkin, J. H., McDermott, J., Simon, D. P., & Simon, H. A. (1980). Models of competence in solving physics problems. *Cognitive Science, 4,* 317-345.

Lehman, J. F., Lewis, R. L., & Newell, A. (1991). Integrating knowledge sources in language comprehension. *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society.* .

Levy, B. (Fall 1991). Able Soar Jr: A model for learning to solve kinematic problems. Final project for PSY 85-711: Cognitive processes and problem solving.

Lewis, R. L., Huffman, S. B., John, B. E., Laird, J. E., Lehman, J. F., Newell, A., Rosenbloom, P. S., Simon, T., & Tessler, S. G. (July 1990). Soar as a Unified Theory of Cognition: Spring 1990. *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society.* Cambridge, MA.

Lewis, R. L., Newell, A., & Polk, T. A. (1989). Toward a Soar theory of taking instructions for immediate reasoning tasks. *Proceedings of the Annual Conference of the Cognitive Science Society.* Hillsdale, New Jersey: Cognitive Science Society, Lawrence Erlbaum Associates.

Lueke, E., Pagerey, P. D., & Brown, C. R. (1987). User requirements gathering through verbal protocol analysis. In Salvendy, G. (Ed.), *Cognitive Engineering in the Design of Human-Computer Interaction and Expert Systems.* Amsterdam: Elsevier Science Publishers.

Luger, G. F. (1981). Mathematical model building in the solution of mechanics problems: Human protocols and the MECHO trace. *Cognitive Science, 5,* 55-77.

Mackay, W. (1989). EVA: An experimental video annotator for symbolic analyses of video data. *SIGCHI Bulletin, 21*(2), 68-71.

MacWhinney, B. (1991). *The CHILDES project: Tools for analyzing talk.* Hillsdale, NJ: Lawrence Erlbaum Associates.

MacWhinney, B., & Snow, C. (1990). The Child Language Data Exchange System: An update. *Journal of Child Language, 17,* 457-472.

McClelland, J. L., & Rumelhart, D. E. (1988). *Explorations in parallel distributed processing: A handbook of models, programs, and exercises.* Cambridge, Massachusetts: The MIT Press.

McClelland, J. L., Rumelhart, D. E., & the PDP research group. (1986). *Parallel distributed processing: Explorations in the microstructure of cognition. Volume 2: Psychological and biological models.* Cambridge, Massachusetts: The MIT Press.

McConnell, C. (Spring 1992). *Ilisp: Fancy LISP interface for GNU Emacs that supports multiple dialects* (Version 4.12). Available from The Ohio State University elisp archives on archive.cis.ohio-state.edu as file pub/gnu/emacs/elisp-archive/packages/ilisp.tar.Z, and from katmandu.mt.cs.cmu.edu:/pub/ilisp/ilisp.tar.Z.

Miller, C. S., & Laird, J. E. (1991). A Constraint-Motivated Model of Lexical Acquisition. *Proceedings of the thirteenth annual conference of the Cognitive Science Society.* Cognitive Science, Lawrence Erlbaum Associates.

Milnes, B. G. (1988). The Soar Graphic Interface. Talk and demo presented at the Soar V Workshop.

Miwa, K., & Simon, H. A. (1992). Measuring individual differences by modifying production systems. Submitted for publication.

Motta, E., Eisenstadt M., Pitman, K., & West, M. (1988). Support for knowledge acquisition in the Knowledge Engineer's Assistant (KEATS). *Expert Systems, 5*, 6-28.

Myers, B. A., & Rossen, M. B. (May 1992). Survey on user interface programming. *CHI'92 Proceedings of the Conference on Human Factors and Computing Systems.* NewYork, ACM Press, Also available as Carnegie-Mellon School of Computer Science technical report CMU-CS-92-113.

Myers, B. A., Giuse, D. A., Dannenberg, R. B., Vander Zanden, V., Kosbie, D. S., Pervin, E., Mickish, A., & Marchal, P. (November 1990). Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer, 23*(11), 71-85.

Myers, B. A., Guise, D., Dannenberg, R. B., Vander Zanden, B., Kosbie, D., Marchal, P., Pervin, E., Mickish, A., Kolojejchick, J. A. (1991). *The Garnet toolkit reference manuals, revised for Version 1.4* (Tech. Rep.) CMU-CS-90-117-R. School of Computer Science, Carnegie-Mellon University.

Neches, R. (1982). A process model for water jug problems. *Behavior research methods & instrumentation, 14*(2), 77-91.

Neches, R., Langley, P., & Klahr, D. (1987). Learning, development, and production systems. In Klahr, D., Langley, P., & Neches, R. (Eds.), *Production system models of learning and development.* Cambridge, MA: Massachusetts Institute of Technology.

Nelson, T. O. (1984). A comparison of current measures of the accuracy of feeling-of-knowing predictions. *Psychological Bulletin, 95*(1), 109-133.

Nerb, J. & Krems, J. (1992). Kompetenzerwerb beim Loesen von Planungsproblemen: experimentelle Befunde und ein SOAR-Model (Skill acquisition in solving scheduling problems: Experimental results and a Soar model.). FORWISS-Report FR-1992-001, Muenchen (Germany).

Neter, J., Wasserman, W., & Kutner, M. H. (1985). *Applied linear statistical models.* Homewood, IL: Irwin.

Neuwirth, C. M., Kaufer, D. S., Chandhok, R., & Morris, J. H. (1990). Issues in the design of computer support for co-authoring and commenting. *In Proceedings of the Third Conference on Computer Supported Cooperative Work (CSCW'90).* Computer supported cooperative work, Association for Computing Machinery.

Newell, A. (1968). On the analysis of human problem solving protocols. In Gardin, J. C., & Jaulin, B. (Eds.), *Calcul et formalisation dans les sciences de l'homme*. Paris: Centre National de la Recherche Scientifique. Excerpt published in Johnson-Laird, P. J., & Wason, P. C. (1977) (Eds.), "On the analysis of human problem solving protocols", *Thinking: Readings in cognitive science*, 46-61, Bath (UK): The Pitman Press.

Newell, A. (1972). A theoretical exploration of mechanisms for coding the stimulus. In Melton, A. W., & Martin, E. (Eds.), *Coding processes in human memory*. Washington, DC: V. H. Winstor.

Newell, A. (1973). You can't play 20 questions with nature and win. In Chase, W. G. (Ed.), *Visual information processing*. New York, NY: Academic Press.

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, *18*, 87-127.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, Massachusetts: Harvard University Press.

Newell, A. (1991). Desires and diversions. School of Computer Science Distinguished Lecture, Carnegie-Mellon University. December 4rth.

Newell, A. (1992). Unified theories of cognition and the role of Soar. In Michon, J. A., & Akyurek, A. (Eds.), *Soar: A cognitive architecture in perspective*. Dordrecht (the Netherlands): Kluwer Academic Publishers.

Newell, A. (1980a). Perception and production of fluent speech. In Cole, R. (Ed.), *Harpy, production systems, and human cognition*. Hillsdale, NJ: Lawrence Erlbaum Associates. Also available as CMU tech. report CMU-CS-78-140.

Newell, A. (1980b). Reasoning, problem solving and decision processes: The problem space as a fundamental category. In Nickerson, R. (Ed.), *Attention and performance VIII*. Hillsdale, NJ: Lawrence Erlbaum Associates. Also available as a Department of Computer Science, Carnegie-Mellon University tech report.

Newell, A., & Rosenbloom, P. S. (1981). Mechanisms of skill acquisition and the law of practice. In Anderson, J. R. (Ed.), *Cognitive skills and their acquisition*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Newell, A., & Simon, H. A. (1961). The simulation of human thought. In *Current trends in psychological theory*. Pittsburgh, PA: University of Pittsburgh Press. Also available as RAND tech. reports P-1734 and RM-2506.

Newell, A., & Simon, H. A. (1972). *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Newell, A., & Steier, D. (1992). Intelligent Control of External Software Systems. *AI in Engineering*, Vol. *in press*. Also available as Technical Report EDRC 05-55-91, Engineering Design Research Center, Carnegie Mellon University, April, 1991.

Newell, A., Shaw, J. C., & Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review*, *65*, 151-166.

Newell, A., Shaw, J. C., & Simon, H. A. (1960). Report on a general problem-solving program for a computer. *Information processing: Proceedings of the international conference on information processing*. Paris, UNESCO, Also available as Rand tech. report P-1584; reprinted in *Computers and Automation*, July 1959..

Newell, A., Yost, G. R., Laird, J. E., Rosenbloom, P. S. & Altmann, E. (1991). Formulating the

problem space computational model. In Rashid, R.F. (Ed.), *Carnegie Mellon Computer Science: A 25-Year Commemorative*. Reading, PA: ACM-Press: Addison-Wesley.

Newell, P., Lehman, J., Altmann, E., Ritter, F., & McGinnis, T. (1992). The Soar video. forthcoming.

Norman, D. A. (1990). Approaches to the study of intelligence. *Artificial Intelligence*, Vol. *26*. Also to be published in Kirsh, D. (Ed.) (in preparation). *Foundations of artificial intelligence*. Cambridge, MA: MIT Press.

O'Reilly, R. C. (1991). *X3DNet: An X-Based Neural Network Simulation Environment*. Available from oreilly@cmu.edu, or via anonymous FTP from hydra.psy.cmu.edu as file pub/x3dnet/x3dnet.tar.Z.

Ohlsson, S. (1980). *Competence and strategy in reasoning with common spatial concepts: A study of problem solving in a semantically rich domain.* Doctoral dissertation, U. of Stockholm. Also published as #6 in the Working papers from the Cognitive seminar, Department of Psychology, U. of Stockholm.

Ohlsson, S. (1990). Trace analysis and spatial reasoning: An example of intensive cognitive diagnosis and its implications for testing. In Frederiksen, N., Glaser, R., Lesgold, A., & Shafto, M. G. (Eds.), *Diagnostic monitoring of skill and knowledge acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Ohlsson, S. (October 1992). Personal communication.

Olson, J. S., Olson, G. M., Storrosten, M., & Carter, M. (1992). How a group-editor changes the character of a design meeting as well as its outcome. Paper presented at the HCI Consortium meeting, February 1992.

Peck, V. A. (November 1992). Personal communication.

Peck, V. A., & John, B. E. (May 1992). Browser-Soar: A computational model of a highly interactive task. *CHI'92 Proceedings of the Conference on Human Factors and Computing Systems*. NewYork, ACM Press.

Pitman, K. M. (1985). *Cref: An editing facility for managing structured text* (Tech. Rep.) A.I. Memo No. 829. Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Platt, J. R. (1964). Strong Inference. *Science*, *146*(3642), 347-353.

Polk, T. A. (August 1992). *Verbal reasoning.* Doctoral dissertation, School of Computer Science, Carnegie-Mellon University.

Poltrock, S. E., & Nasr, M. G. (1989). *Protocol analysis: A tool for analyzing human-computer interactions* (Tech. Rep.) ACT-HI-186-89. Microelectronics and Computer Technology Corporation.

Popper, K. R. (1959). *The logic of scientific discovery.* New York, NY: Basic Books.

Priest, A. G., & Young, R. M. (1988). Methods for evaluating micro-theory systems. In Self, J. (Ed.), *Artificial Intelligence and Human Learning: Intelligent Computer-Aided Instruction.* London: Chapman and Hall.

Qin, Y., & Simon, H. A. (1990). Laboratory replication of scientific discovery processes. *Cognitive Science*, *14*(2), 281-312.

Quinlan, R. (1983). Learning efficient classification procedures and their application to chess end games. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.), *Machine learning:*

An artificial intelligence approach. Palo Alto, CA: Tioga.

Reiser, B. J., Anderson, J. R., & Farrell, R. G. (1985). Dynamic student modeling in an intelligent tutor for LISP programming. *Proceedings of the International Joint Conference on Artificial Intelligence - 85.* Los Angeles: International Joint Conference on Artificial Intelligence.

Reynolds, H. T. (1984). *Analysis of nominal data* (Tech. Rep.) 07-001. London & Beverly Hills, CA: Sage university paper series on quantitative application in the social sciences.

Ritter, F. E. (September, 1988). "Extending the Seibel-Soar Model". Presented at the Soar V Workshop held at CMU.

Ritter, F. E. (1989). Transparencies from Soar Meeting, May, 1989. FOKIBOFIT-Soar: A Soar model of the effect of problem-part frequency on feeling-of-knowing, Department of Psychology, Carnegie-Mellon University, Unpublished. Also presented as part of the Understand Seminar, PSY 85-811.

Ritter, F. E. (1991). *TAQL-mode Manual.* The Soar Project, School of Computer Science, Carnegie-Mellon University.

Ritter, F. E. (February, 1992). "Bruno Levy's Able-Soar, Jr. model". Presented at the Soar X Workshop held at The University of Michigan.

Ritter, F. E., & Fox, D. (1992). *Dismal: A spreadsheet for GNU-Emacs.* The Soar Project, School of Computer Science, Carnegie-Mellon University.

Ritter, F. E., & McGinnis, T. F. (1992). Manual for *SX: A graphical display and interface for Soar in X windows.* The Soar Project, School of Computer Science, Carnegie-Mellon University.

Ritter, F. E., Hucka, M., & McGinnis, T. F. (1992). *Soar-mode Manual* (Tech. Rep.) CMU-CS-92-205. School of Computer Science, Carnegie-Mellon University.

Rosenbloom, P. S. & Lee, S. (1989). Soar arithmetic and functional capability. Software provided with the Soar 5 distribution.

Rosenbloom, P. S., & Newell, A. (September 1982). *Learning by chunking, a production system model of practice* (Tech. Rep.) CMU-CS-82-135. Department of Computer Science, Carnegie-Mellon University.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (1987). Meta-levels in Soar. In Maes, P., & Nardi, D. (Eds.), *Meta-Level Architectures and Reflection.* Amsterdam: North Holland Publishing Company.

Rumelhart, D. E., McClelland, J. L., & the PDP research group. (1986). *Parallel distributed processing: Explorations in the microstructure of cognition. Volume 1: Foundations.* Cambridge, MA: The MIT Press.

Sakoe, H., & Chiba, S. (1978). Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on acoustics, speech, and signal processing*, *26*(1), 43-49.

Samual, A. L. (July 1959). Some studies in machine learning using the game of checkers. *IBM J. of Research and Development*, *3*, 210-229.

Sanderson, P. M. (1990). Verbal protocol analysis in three experimental domains using SHAPA. *Proceedings of the Human Factors Society 34th Annual Meeting.* Human Factors Society.

Sanderson, P., James, J., Watanabe, L., & Holden, J. (1990). Human operator behavior in

complex worlds: Rendering sequential records analytically tractable. *Proceedings of the 9th Annual Conference on Human Decision Making and Manual Control.* Varese, Italy, Also available from the Engineering Psychology Research Laboratory, Department of Mechanical and Industrial Engineering, U. of Illinois, as technical report EPRL-90-12.

Sanderson, P. M., Verhage, A. G., & Fuld, R. B. (1989). State-space and verbal protocol methods for studying the human operator in process control. *Ergonomics, 32*(11), 1343-1372.

Schank, R. C. (1982). *Dynamic memory.* Cambridge, UK: Cambridge University Press.

Schroeder, D. (November 1992). Personal communication.

Shadbolt, N. R., & Wielinga, B. (1990). Knowledge-based knowledge acquisition: the next generation of support tools. In Wielinga, B., Boose, J. H., Gaines, B. R., Schreiber, G., & van Someren, M. (Eds.), *Current trends in Knowledge acquisition (Proceedings of the 4th European workshop on knowledge acquisition, EKAW-90, Amsterdam.* Amsterdam: IOS Press.

Sherwood, B. A., & Sherwood, J. N. (1984). The cT language and its uses: A modern programming tool. In Redish, E. F., & Risley, J. S. (Eds.), *The Conference on Computers in Physics Instruction Proceedings.* Redwood City, CA: Addison-Wesley. Also available as tech report UPITT/LRDC/ONR/APS-14.

Sherwood, B. A., & Sherwood, J. N. (1992). *The cT Language Manual.* Wentworth, NH: Falcon Software. The cT programming language is distributed by Falcon Software, Inc., P.O. Box 200, Wentworth, NH 03282; phone 603-764-5788, fax 603-764-9051. A site license is available for users at CMU.

Shrager, J., Hogg, T., & Huberman, B. A. (1988). A dynamical theory of the power-law of learning in problem-solving. Draft paper submitted to AAAI-88.

Siegler, R. S. (1988). Strategy choice procedures and the development of multiplication skill. *Journal of Experimental Psychology: General, 117*(3), 258-275.

Siegler, R. S., & Shrager, J. (1984). Strategy choices in addition and subtraction: How do children know what to do? In Sophian, C. (Ed.), *Origins of cognitive skills.* Hillsdale, NJ: Lawrence Erlbaum Associates.

ACM Special interest group on Artificial Intelligence. (April 1989). Special issue on: Knowledge Acquisition, *Sigart Newsletter (108).*

Special interest group on Artificial Intelligence. (1991). Special section on integrated cognitive architectures *Sigart Bulletin, 2(4),*

Special interest group on Computer-Human Interaction. (1989). Special issue on protocol analysis tools and methods, *SigChi Bulletin, 21(2),*

Simon, H. A. (1979). *Models of Thought.* New Haven, CT: Yale University Press.

Simon, H. A. (1989). *Models of Thought, Volume II.* New Haven, CT: Yale University Press.

Simon, H. S. (1990). Invariants of human behavior. *Annual Review of Psychology, 41,* 1-19.

Simon, H. A. (October 1991). Setting up research programs. Talk presented as part of the Graduate student professional seminar series: Interfacing the science and the profession, Department of Psychology, Carnegie-Mellon University.

Simon, H. A., & Newell, A. (1956). Models: Their uses and limitations. In White, L. D. (Ed.), *The state of the social sciences.* Chicago: University of Chicago Press.

Simon, H. A., & Reed, S. K. (1976). Modeling strategy shifts in a problem-solving task. *Cognitive Psychology, 8*, 86-97.

Simon, D. P., & Simon, H. A. (1978). Individual differences in solving physics problems. In Siegler, R. S. (Ed.), *Children's thinking: What develops?*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Simon, T., Newell, A., & Klahr, D. (1991). A computational account of children's learning about number conservation. In Fisher, D., & Pazzani, M. (Eds.), *Working Models of Human Perception*. Los Altos, California: Morgan Kaufman.

Singley, M. K. (1987). *Developing models of skill acquisition in the context of intelligent tutoring systems.* Doctoral dissertation, Department of Psychology, Carnegie-Mellon University.

Singley, M. K., & Anderson, J. R. (1989). *The transfer of cognitive skill.* Cambridge, MA: Harvard University Press.

Siochi, A. C., & Hix, D. (1991). A study of computer-supported user interface evaluation using maximal repeating pattern analysis. *Proceedings of the Chi'91*. SIGCHI.

Sleeman, D., Hirsh, H., Ellery, I., & Kim, I. (1990). Extending domain theories: Two cases studies in student modeling. *Machine Learning, 5*, 11-37.

Smith, E. E. (1967). Effects of familiarity on stimulus recognition and categorization. *Journal of Experimental Psychology, 74*(3), 324-332.

Smith, D. (September, 1992a). S-mode version 3.1 (program documentation). Distributed with S-mode.

Smith, D. (September, 1992b). S-mode reference card. Distributed with S-mode.

Stallman, R. M. (1984). EMACS: The extensible, customizable, self-documenting display editor. In Barstow, D. R., Shrobe, H. E., & Sandewall, E. (Eds.), *Interactive programming environments*. New York, NY: McGraw-Hill. Extended version of a paper that was published in *Proceedings of the ACM SIGPLAN SIGOA Symposium on text manipulation*, June 1981, Portland, OR, pp. 147-156.

Stobie, I., Tambe, M., & Rosenbloom, P. (November 1992). Flexible integration of path-planning capabilities. *Proceedings of the SPIE conference on Mobile Robots*. .

Stone, P. J., Dunphy, D. C, Smith, M. S., & Ogilvie, D. M. with associates. (1966). *The General Inquirer: A computer approach to content analysis*. Cambridge, MA: The MIT Press.

Suchman, L. (October 1983). Office procedures as practical action: Models of work and system design. *ACM Transactions on Office Information Systems, 1*(4), 320-328.

Swets, J. A. (1973). The relative operating characteristic in psychology. *Science, 182*(7 December 1973), 990-1000.

Swets, J. A. (1986). Indices of discrimination or diagnostic accuracy: Their ROCs and implied models. *Psychological Bulletin, 99*(1), 100-117.

Tesler, L. (January 1983). Enlisting user help in software design. *SIGCHI Bulletin, 14*(3), 5-9.

The Soar group. (24 September 1990). *A brief introduction to Soar and selected readings*. The Soar group, Department of Computer Science, Carnegie-Mellon University. 2 pages.

Thibadeau, R., Just, M. A., & Carpenter, P. A. (1982). A model of the time course and content

of reading. *Cognitive Science, 6*, 157-203.

Toulmin, S. E. (1972). *Human understanding*. Princeton, NJ: Princeton University Press.

Touretzky, D. S. (December, 1986). *A distributed connectionist production system* (Tech. Rep.) CMU-CS-86-172. Computer Science Department, Carnegie Mellon University.

Tufte, E. R. (1990). *Envisioning information*. Cheshire, CT: Graphics Press.

Turing, A. M. (1956). Can a machine think? In Newman, J. R. (Ed.), *The world of mathematics (4)*. New York, NY: Simon and Schuster.

Underwood, B. J. (1969). Attributes of memory. *Psychological Review, 76*(6), 559-573.

Unruh, A. (1986). Comprehensive programming project: An interface for Soar. Dept. of Computer Science, Stanford University.

van Gelder, T. (1991). Connectionism and Dynamic Explanation. *Proceedings of the Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Cognitive Science Society, Lawrence Erlbaum Associates.

VanLehn, K. (1989). Human skill acquisition: Theory, model, and psychological validation. *Proceedings of AAAI-83*. Los Altos, CA, Morgan Kaufman.

VanLehn, K., & Garlick, S. (1987). Cirrus: an automated protocol analysis tool. Langley, P. (Ed.), *Proceedings of the Fourth Machine Learning Workshop*. Los Altos, CA, Morgan-Kaufman, also published as Technical Report PCG-6, Departments of Psychology and Computer Science, Carnegie-Mellon University.

VanLehn, K., Brown, J. S., & Greeno, J. (1984). Competitive argumentation in computational theories of cognition. In Kintsch, W., Miller, J. R., & Polson, P. G. (Eds.), *Methods and tactics in cognitive science*. Hillsdale, NJ: Lawrence Erlbaum Associates.

VanLehn, K., Jones, R. M., & Chi, M. T. H. (1991). Modeling the self-explanation effect. *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Cognitive Science Society, Lawrence Erlbaum Associates.

VanLehn, K., & Ball, W. (1987). *Flexible execution of cognitive procedures* (Tech. Rep.) PCG-5. Departments of Psychology and Computer Science, Carnegie-Mellon University.

Wagner, R. A., & Fisher, M. J. (1974). The string-to-string correction problem. *J. of the Association for Computing Machinery, 21*, 168-172.

Wagner, D. A., & Scurrah, M. J. (1971). Some characteristics of human problem-solving in chess. *Cognitive Psychology*, 2, 454-478.

Waldrop, M. M. (July 1988). Soar: A Unified Theory of Cognition? *Science, 241*, 296-298.

Waltz, D. L. (January 1987). Applications of the Connection Machine. *IEEE Computer, 20*(1), 85-97.

Ward, B. (May 1991). *ET-Soar: Toward an ITS for Theory-Based Representations*. Doctoral dissertation, School of Computer Science, Carnegie-Mellon University.

Waterman, D. (1973). Pas-II reference manual, Version 29. Copies are currently available from Herb Simon and Frank Ritter, Department of Psychology, Carnegie-Mellon University.

Waterman, D. A., & Newell, A. (1971). Protocol Analysis as a task for artificial intelligence. *Artificial Intelligence*, 2, 285-318. Shorter version published as Waterman, D.A. and Newell, A., (1971). Protocol Analysis as a Task for Artificial Intelligence, in *Proceedings of the Second*

International Joint Conference on Artificial Intelligence (IJCAI).

Waterman, D. A., & Newell, A. (1973). *Pas-II: An interactive task-free version of an automatic protocol analysis system* (Tech. Rep.). Department of Computer Science, Carnegie-Mellon University. Also included in the preprints for the Third International Joint Conference on Artificial Intelligence.

Wickens, T. D. (1982). *Models for behavior: Stochastic processes in psychology.* San Francisco: W. H. Freeman and Company.

Winikoff, A. (1967). *Eye movements as an aid to protocol analysis of problem solving behavior.* Doctoral dissertation, Carnegie-Mellon University.

Yost, G.R. (March 1992). *TAQL: A Problem Space Tool for Expert System Development.* Doctoral dissertation, School of Computer Science, Carnegie-Mellon University.

Yost, G.R., & Altmann, E. (1991). TAQL 3.1.3: Soar Task Acquisition Language User Manual. School of Computer Science, Carnegie-Mellon University, 19 December, 1991. Unpublished.

Yost, G. R., & Newell, A. (1989). A problem space approach to expert system specification. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* IJCAI.

Young, R. M. (1973). *Children's seriation behavior: a production-system analysis.* Doctoral dissertation, Carnegie-Mellon University. Also published as C.I.P. report #245.

Young, R. M. (1979). Production systems for modeling human cognition. In Michie, D. (Ed.), *Expert systems in the micro-electronic age.* Edinburgh (UK): The University Press.

Young, R. M., & O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science, 5,* 153-177.

I. How to obtain the software described in this thesis

All of the software presented in this thesis is available. Many of the tools described in this thesis are protected under the GNU copy-left agreement. This basically means that if you get a copy, you also get the source code, and take on an obligation to provide it to others upon request. In short: you may use this code any way you like, as long as you don't charge money for it, remove this notice, or hold anyone liable for its results. Most of the remaining tools are in the public domain and can be freely copied. The S statistics system is the only exception. It is available from AT&T and Stat Science.

The DSI

The SX graphic display. Starting in your local directory where you want to install the SX display code (e.g., /tOSU/soar/sx/5.3), open an FTP connection to Centro.soar.cs.cmu.edu [128.2.242.245]. Login as "anonymous", using your address as password (e.g., "user@machine.site.edu").

Change to the directory /afs/cs.cmu.edu/project/soar/5.2/2/public (cd /afs/cs.cmu.edu/project/soar/5.2/2/public). NB: You will not be able to cd to any directory between /afs and .../public.

Set file type to binary (binary).

Retrieve the file all-sx.5.3.1.tar.Z (retrieve all-sx.5.3.1.tar.Z). Close the FTP connection (quit). This file is approximately .5 meg. The final uncompressed, untared, compiled distribution will come to approximately 9 meg plus an image, which is about 17 M on a Dec 3100/5000 machine.

Uncompress it (uncompress all-sx.5.3.1.tar.Z) and untar it (tar xf all-sx.5.3.1.tar). You should find the following five lisp files and four bitmap files: build-sx.lisp, g1.lisp, g2.lisp, g3.lisp, all-sx.lisp, garnet.cursor, hourglass.cursor, garnet.mask, hourglass.mask. You will also get an example .sx-init.lisp file called default.sx-init.lisp, examples of things you can put in your .cshrc in dsi-cshrc-additions, postscript and doc versions of this manual, and a checkout script.

Follow the directions included in the manual to complete the installation.

Soar-mode. The complete source for soar-mode is available from /afs/cs.cmu.edu/project/soar/5.2/2/public/soar-mode.tar.Z. You can copy this file directly if you are at Michigan or ISI, or you can retrieve it via anonymous-FTP to Centro.soar.cs.cmu.edu [128.2.242.245]. Note: CMU's machines do not allow you to access intermediate directories in this path. To access the latest version you may have to do a listing of the files in that directory ("ls"). Follow the directions included in the manual to complete the installation.

Taql-mode. The complete source for Taql-mode is available from "/afs/cs.cmu.edu/project/soar/5.2/2/public/taql-mode.2.2.tar.Z". You can copy this file directly if you are at Michigan or ISI, or you can retrieve it via anonymous-FTP to Centro.soar.cs.cmu.edu [128.2.242.245]. Note: CMU's machines do not allow you to access intermediate directories in this path. To access the latest version you may have to do a listing of the files in that directory ("ls"). Follow the directions included in the manual to complete the installation.

The alignment tool and dismal spreadsheet

Please contact Ritter@cs.cmu.edu to receive these tools. They have not yet been generally released.

The interface to the S programming system

S is a statistics package available from Bell Labs particularly suited for descriptive and exploratory

statistics. S-mode is built on top of comint (the general command interpreter mode written by Olin Shivers), as an interface to S.

The latest version of S-mode is available from the Statlib email statistical software server by sending a blank message with subject "send index from S" to statlib@stat.cmu.edu, and following the directions from there. Comint is probably already available at your site, and already in your load path. If it is not, you can get it from archive.cis.ohio-state.edu (login name is anonymous, password is your real id) in directory /pub/gnu/emacs/elisp-archive/as-is/comint.el.Z. This version has been tested and works with (at least) comint-version 2.03. You probably have copies of comint.el on your system. Copies of comint are also available from ritter@cs.cmu.edu, and shivers@cs.cmu.edu.

S-mode is also available for anonymous FTP from attunga.stats.adelaide.edu.au in the directory pub/S-mode, and from the Emacs-lisp archive on archive.cis.ohio-state.edu.

The simple menu package

Updated versions (if any) of the simple-menu package used to provide the menus in S-mode, Soar-mode, and Taql-mode are available from the author or via FTP: from the elisp archive on archive.cis.ohio-state.edu as file pub/gnu/emacs/elisp-archive/interfaces/simple-menu<version>.el.Z. Iff you post me mail that you use it, I'll post you updates when they come out.