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.

<u>Constraining the model with additional sources of data.</u> 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.

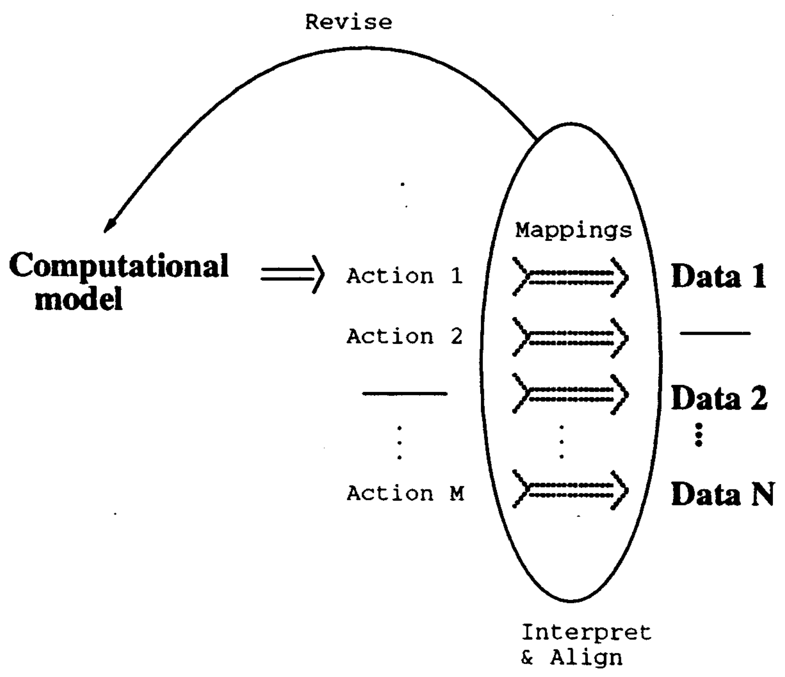


Figure 3-12: Diagram showing the inputs (in bold) to trace based protocol analysis (TBPA): A computational model and transcribed and segmented protocol data.

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

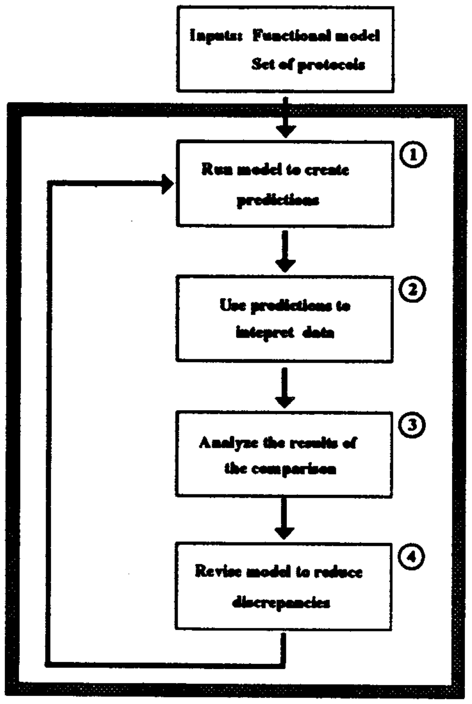

Inputs: Functional model
Set of protocols
Run model to create predictions
Use predictions to intepret data
Analyze the results of the comparison
Revise model to reduce discrepancies
①
②
③
④

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.

<u>(b) Be readable by humans.</u> 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.

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.

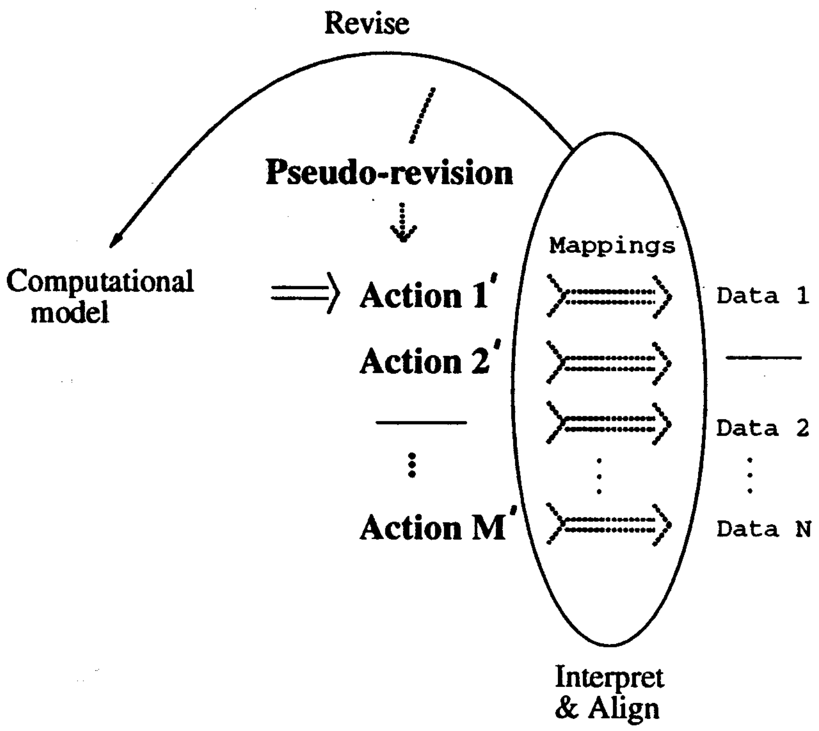


Figure 3-14: Diagram illustrating direct trace modification as a form of pseudo-model revision.

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

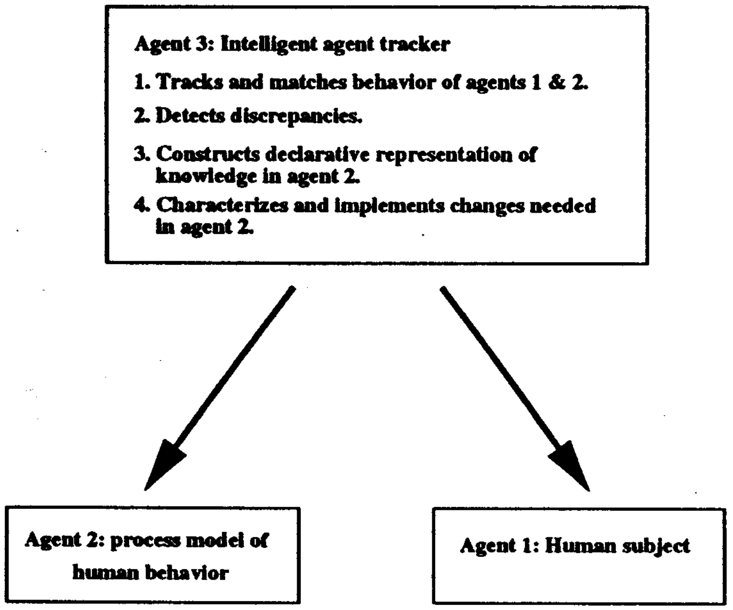

Agent 3: Intelligent agent tracker
1. Tracks and matches behavior of agents 1 & 2.
2. Detects discrepancies.
3. Constructs declarative representation of knowledge in agent 2.
4. Characterizes and implements changes needed in agent 2.
Agent 2: process model of human behavior
Agent 1: Human subject

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.

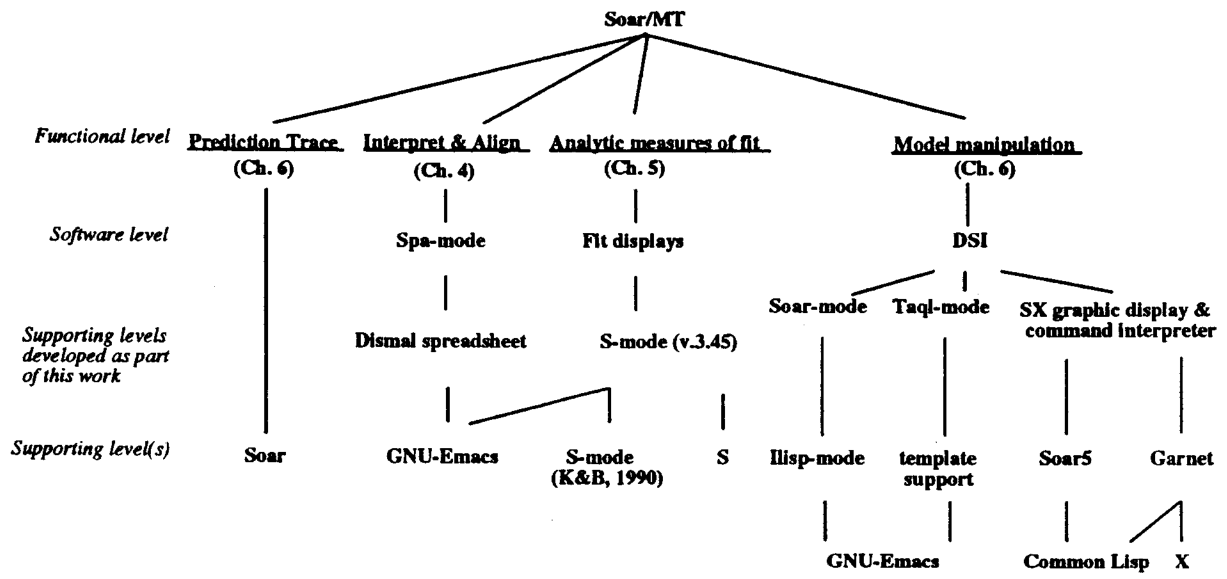


Figure 3-16: Requirements for an environment for testing process models and overview of the Soar/MT environment to support these requirements.

Model tracing and manipulation. 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.

A shared design. 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.

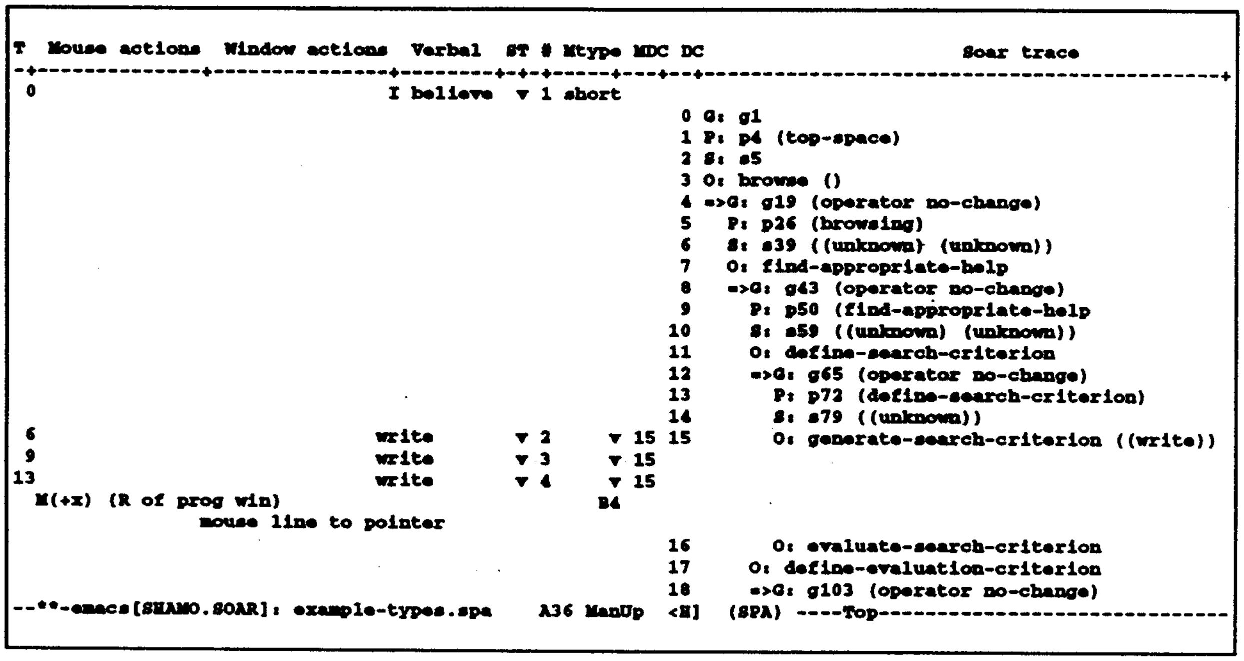
Spa-mode is built on the Dismal spreadsheet (Ritter & Fox, 1992), which is implemented as a set of extensions (a major-mode in Emacs terms) to the GNU-Emacs editor (Free Software Foundation, 1988). Dismal includes most of the major functions that one now expects from a spreadsheet, such as (a) the addition, 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.

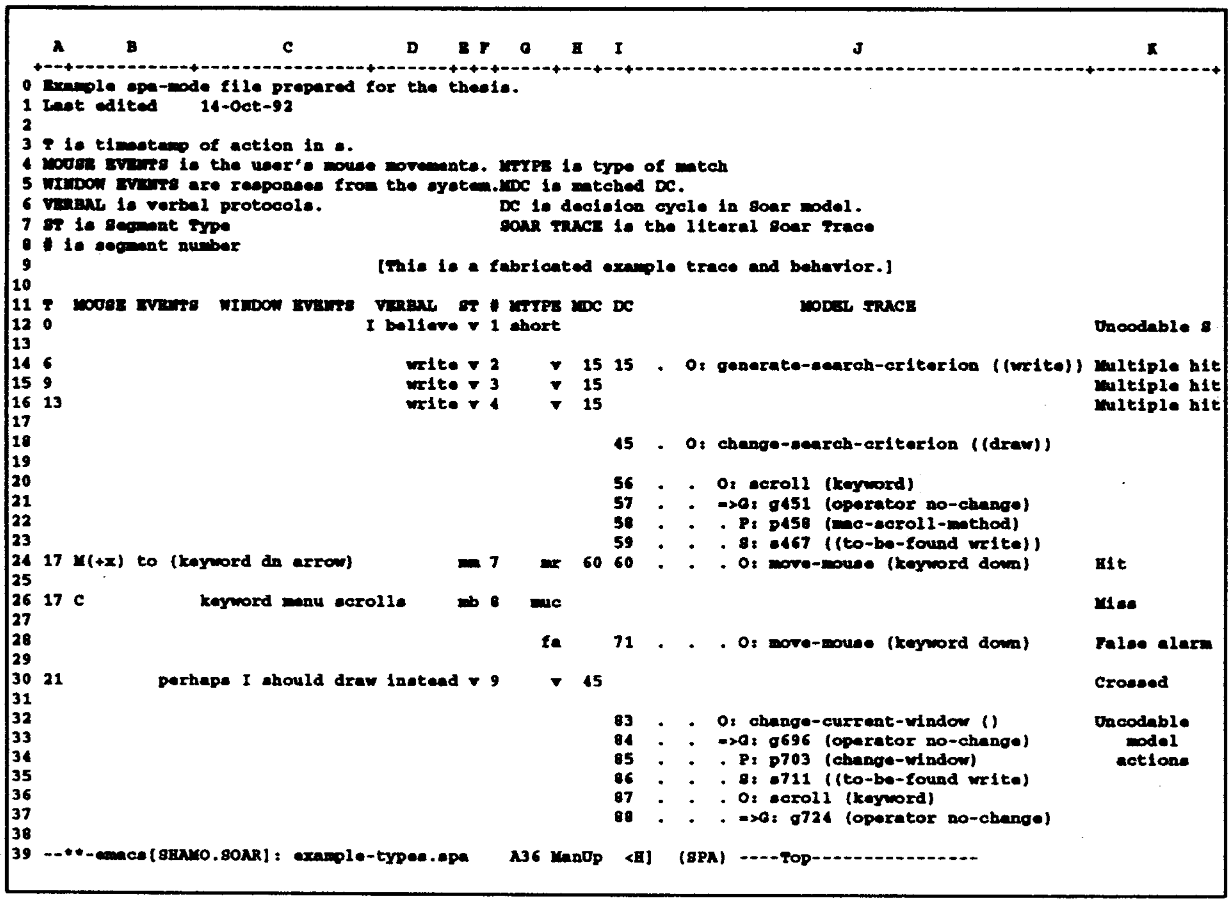
There is one type of correspondence mentioned in Table 2-5 that cannot currently be represented in Spa-mode, a subject action that is matched by multiple model actions. The current representation assumes that what matches a given data segment is a single action of the model. This lack of representation 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 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.

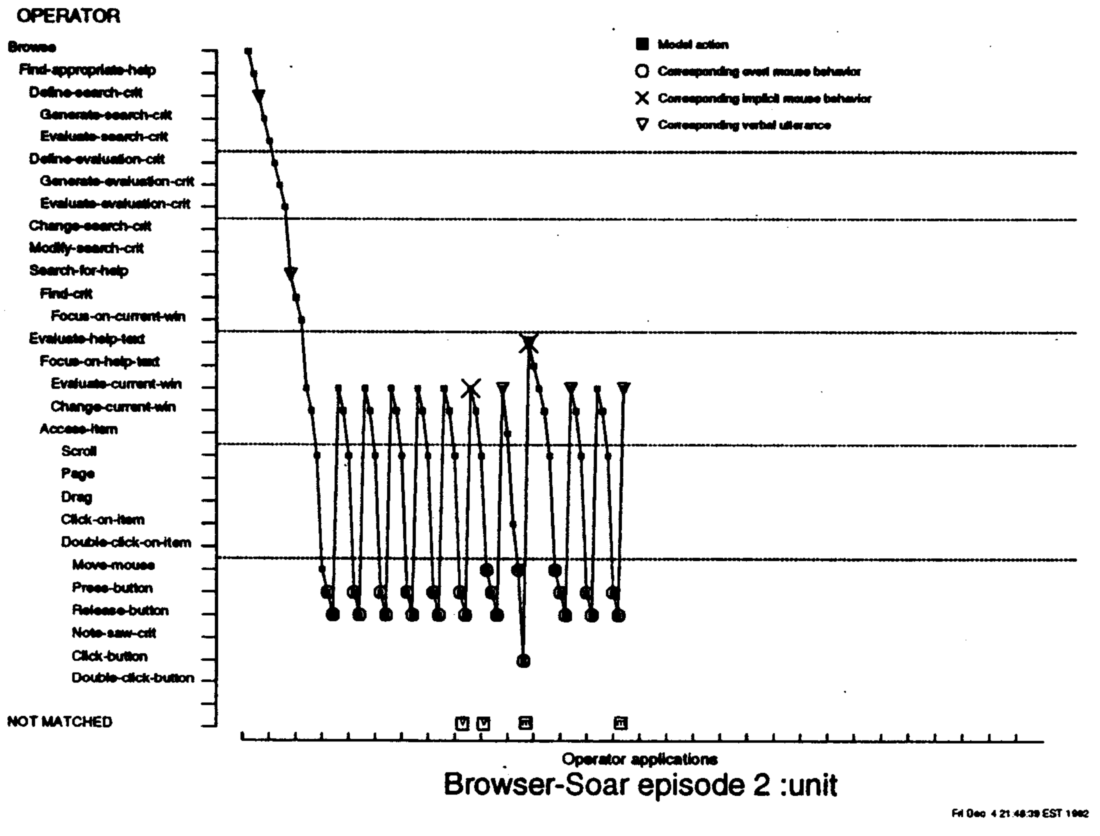


Figure 5-20:
Example operator prediction support display taken from the Unit episode of Browser-Soar. The model's operators are shown on the left-hand side, indented according to their depth in the problem space hierarchy. The connected black squares represent the model's performance. Corresponding data 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

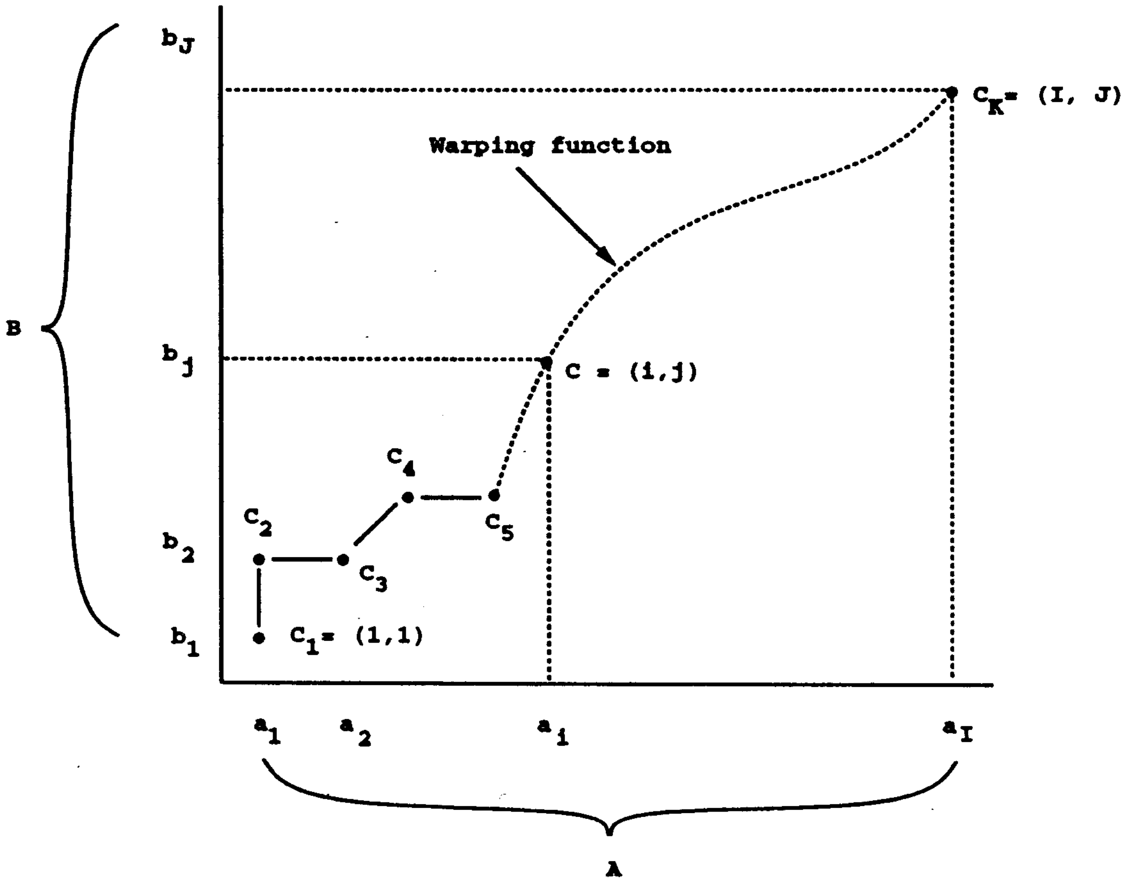


Figure 5-21:
Depiction of Sakoe and Chiba's (1978) correspondence diagram from their speech recognition task. The A axis represents the times of the subjects utterances, and the B axis represents the times of the model's predictions. The places where they correspond are represented by the C terms. The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

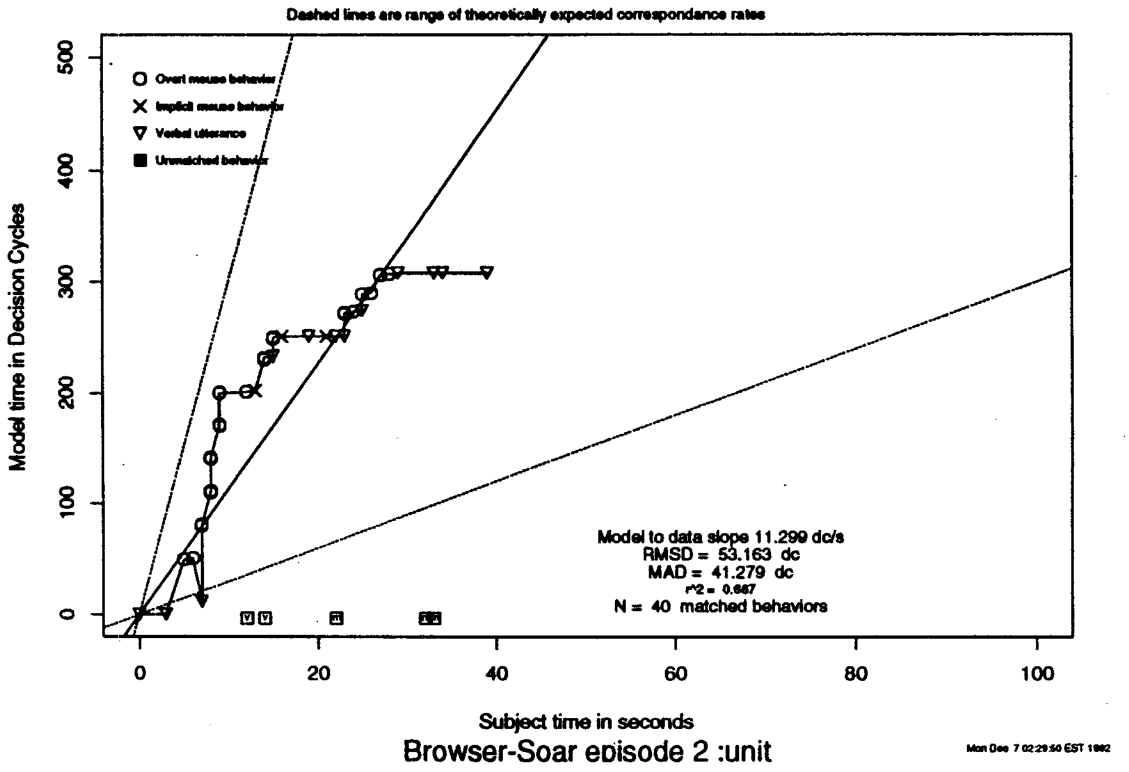


Figure 5-22:
Example relative processing rate display based on decision cycles taken from the Unit episode of Browser-Soar. The straight, solid line is a least-squares regression line through all the correspondences. Its slope is the relative rate between decision cycles and seconds. The dashed lines 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.

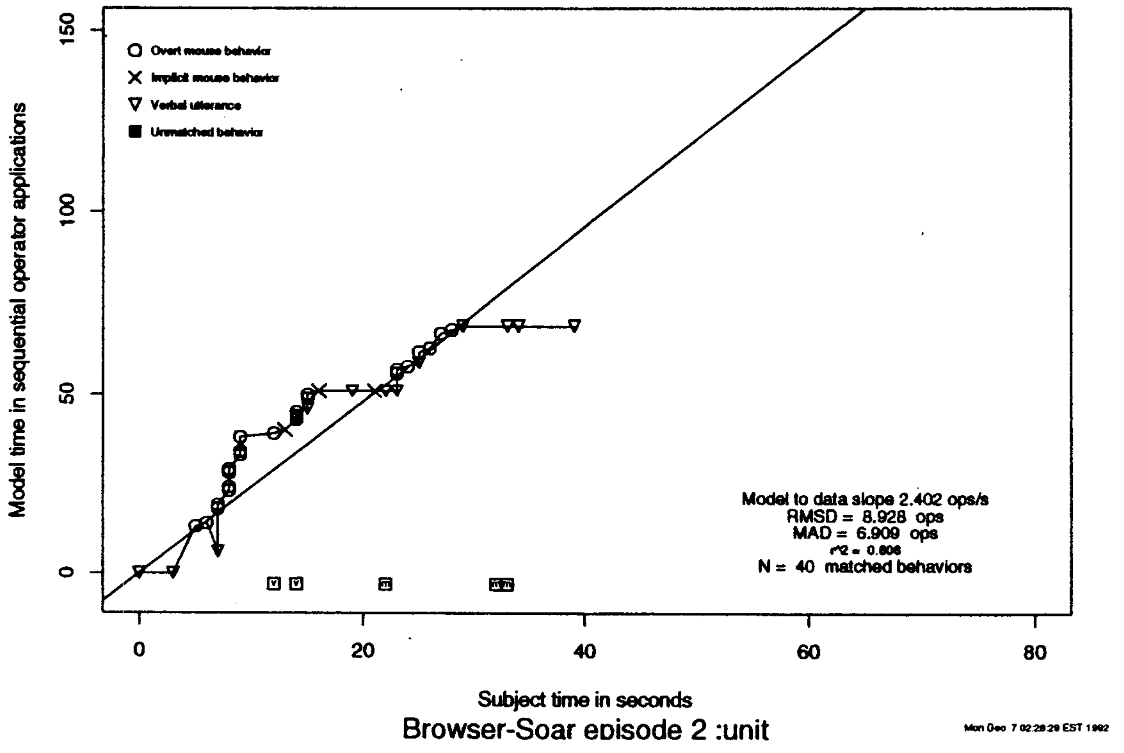


Figure 5-23: Example relative processing rate display based on operator applications taken from the Unit episode of Browser-Soar.

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.

<u>The sequentiality assumption.</u> 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.

<u>The cross-modal sequentiality assumption.</u> 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.

<u>The amount and direction of lag may be an indicator of protocol quality.</u> 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.