Chapter 6
The model manipulation tool -- the Developmental Soar Interface (DSI)

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

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

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

<u>Requirements for the process model's trace</u>
- (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.

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

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

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

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

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

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

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

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

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

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

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

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

6.1.1 Providing predictions for comparison with the data

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

6-24(a) Original Soar 5 trace:

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

6-24(b) Modified Soar 5 trace:

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

Improvements to the Soar trace for use in TBPA

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

Figure 6-24: Original and modified Soar trace.

(a.i) Be unambiguous. The new trace removes several ambiguities and retains the decision cycle number of the original trace. The name and traced attributes of the selected object have fixed positions. The use of the object's ID when a name is not available may not turn out to be the best choice; it may be better to insert "no-name" or some other distinct marker that can be more easily interpreted than the ID as the lack of a name.

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

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

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

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

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

6.1.2 Aggregating the model's performance

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

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

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

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

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

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

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

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

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

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

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

6.2 Displaying the model so that it can be understood

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

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

```
PSCM Level statistics on November  22, 1992

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

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

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

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

The graphic display can be used in two ways, as a normative display of what problem spaces may exist in the model and their relationships to each other, and as a descriptive display of the goal stack contents while the model is running. Both types of information can be displayed simultaneously (the display can get a bit complicated) to see if the model normative behavior is correct.

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

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

6.2.1 Normative displays of the model

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

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

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

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

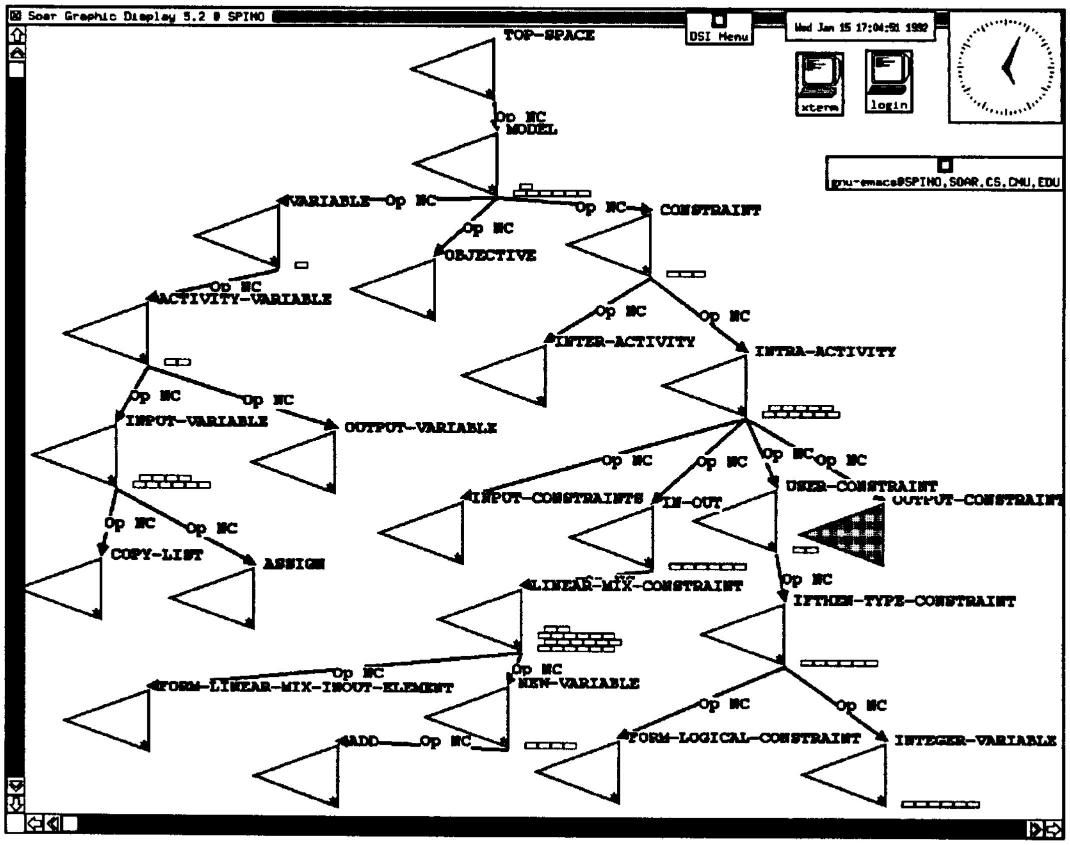


Figure 6-26:
The problem space structure of MFS-Soar (picture taken by David Steier). Learned chunks (small bricks) shown on chunk walls to right of each problem space (triangles). Lines between problem spaces labeled "OP NC" stands for operator no-change impasses in the higher space that are resolved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

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

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

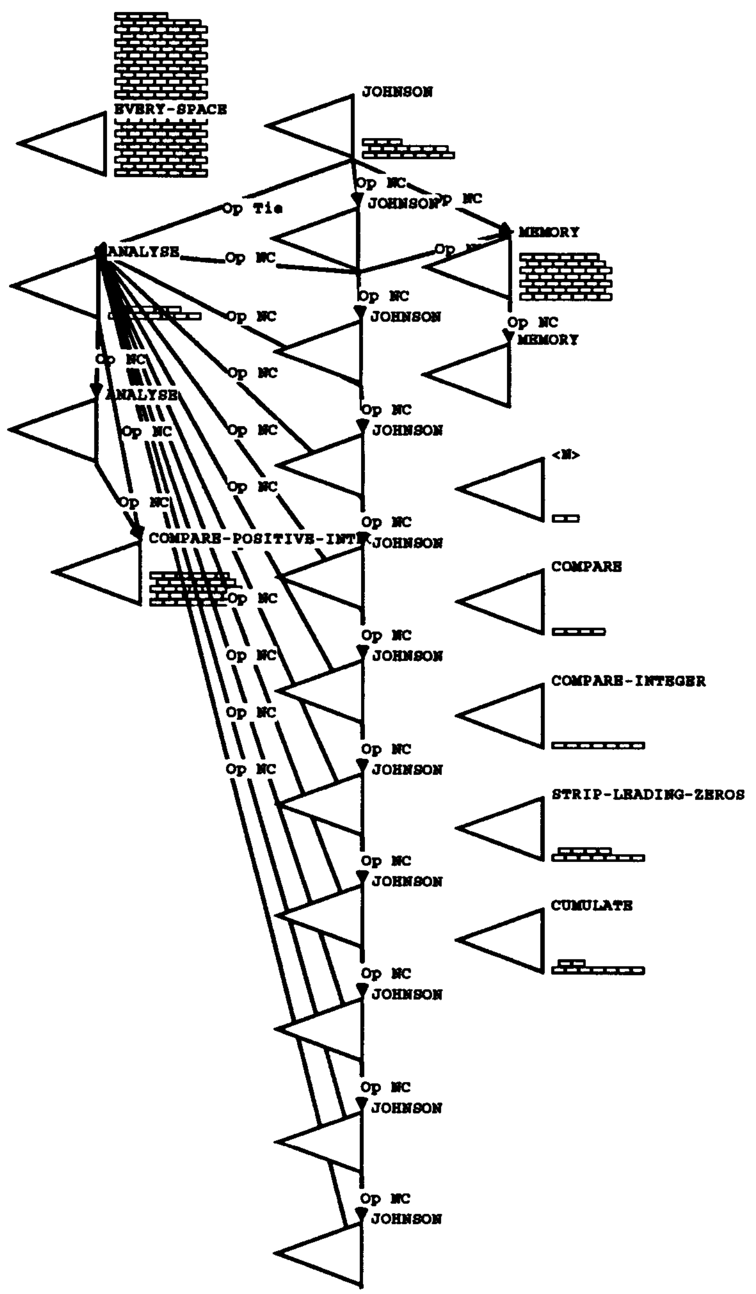


Figure 6-27: Normative display of Sched-Soar showing the productions in each problem space as chunks on the chunk wall to the right of each problem space.

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

The knowledge in each problem space has to be measured in terms of productions. Although this certainly appears to be an imperfect measure, there is no other coherent metric. The generality of the productions might be measurable through the number of clauses, but in the quest for accuracy, even that should be adjusted for the frequency of the features tested in the environment. The number of operators is another possible metric, but they vary even more than productions in size and generality.

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

6.2.2 Descriptive displays of the model's performance

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

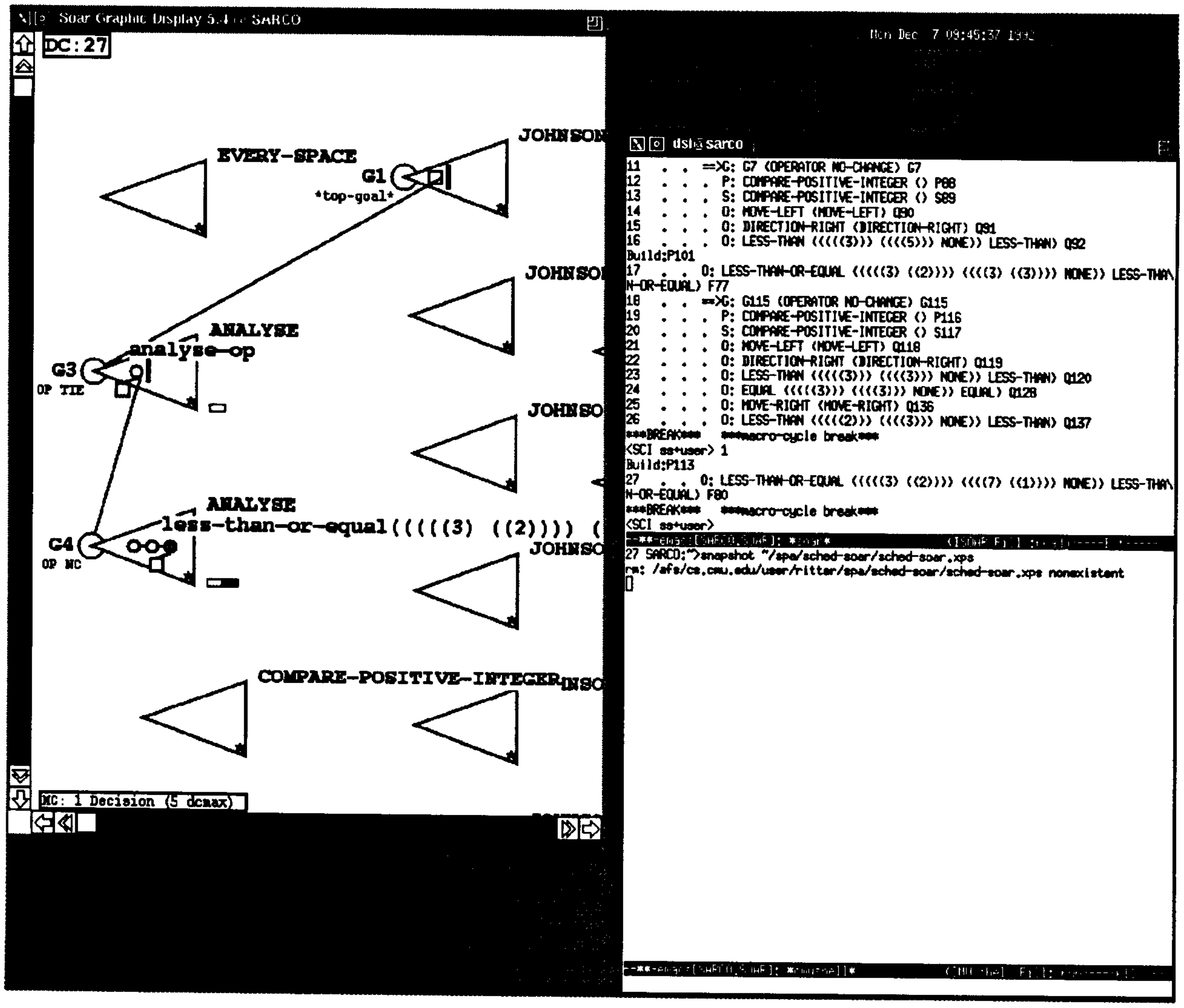
Because the problem space level objects persist over time in the SX graphic display, a declarative model of the structures in the productions is created. This can support simple discoveries about models. Until Soar and then TAQL were run with the same graphic display, a mistake really, the developers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

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

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

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

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



```
11   . .  ==>G: G7 (OPERATOR NO-CHANGE) G7
12   . . .  P: COMPARE-POSITIVE-INTEGER () P88
13   . . .  S: COMPARE-POSITIVE-INTEGER () S89
14   . . .  O: MOVE-LEFT (MOVE-LEFT) Q90
15   . . .  O: DIRECTION-RIGHT (DIRECTION-RIGHT) Q91
16   . . .  O: LESS-THAN ((((3))) ((((5))) NONE)) LESS-THAN) Q92
Build:P101
17   . .  O: LESS-THAN-OR-EQUAL (((((3) ((2)))) ((((3) ((3)))) NONE)) LESS-THA\
N-OR-EQUAL) F77
18   . .  ==>G: G115 (OPERATOR NO-CHANGE) G115
19   . . .  P: COMPARE-POSITIVE-INTEGER () P116
20   . . .  S: COMPARE-POSITIVE-INTEGER () S117
21   . . .  O: MOVE-LEFT (MOVE-LEFT) Q118
22   . . .  O: DIRECTION-RIGHT (DIRECTION-RIGHT) Q119
23   . . .  O: LESS-THAN ((((3))) ((((3))) NONE)) LESS-THAN) Q120
24   . . .  O: EQUAL ((((3))) ((((3))) NONE)) EQUAL) Q128
25   . . .  O: MOVE-RIGHT (MOVE-RIGHT) Q136
26   . . .  O: LESS-THAN ((((2))) ((((3))) NONE)) LESS-THAN) Q137
***BREAK***   ***macro-cycle break***
<SCI ss+user> 1
Build:P113
27   . .  O: LESS-THAN-OR-EQUAL (((((3) ((2)))) ((((7) ((1)))) NONE)) LESS-THA\
N-OR-EQUAL) F80
***BREAK***   ***macro-cycle break***
<SCI ss+user>
27 SARCO:~>snapshot ~/spa/sched-soar/sched-soar.xps
rm: /afs/cs.cmu.edu/user/ritter/spa/sched-soar/sched-soar.xps nonexistent
```

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

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

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

6.2.3 The working memory walker

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

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

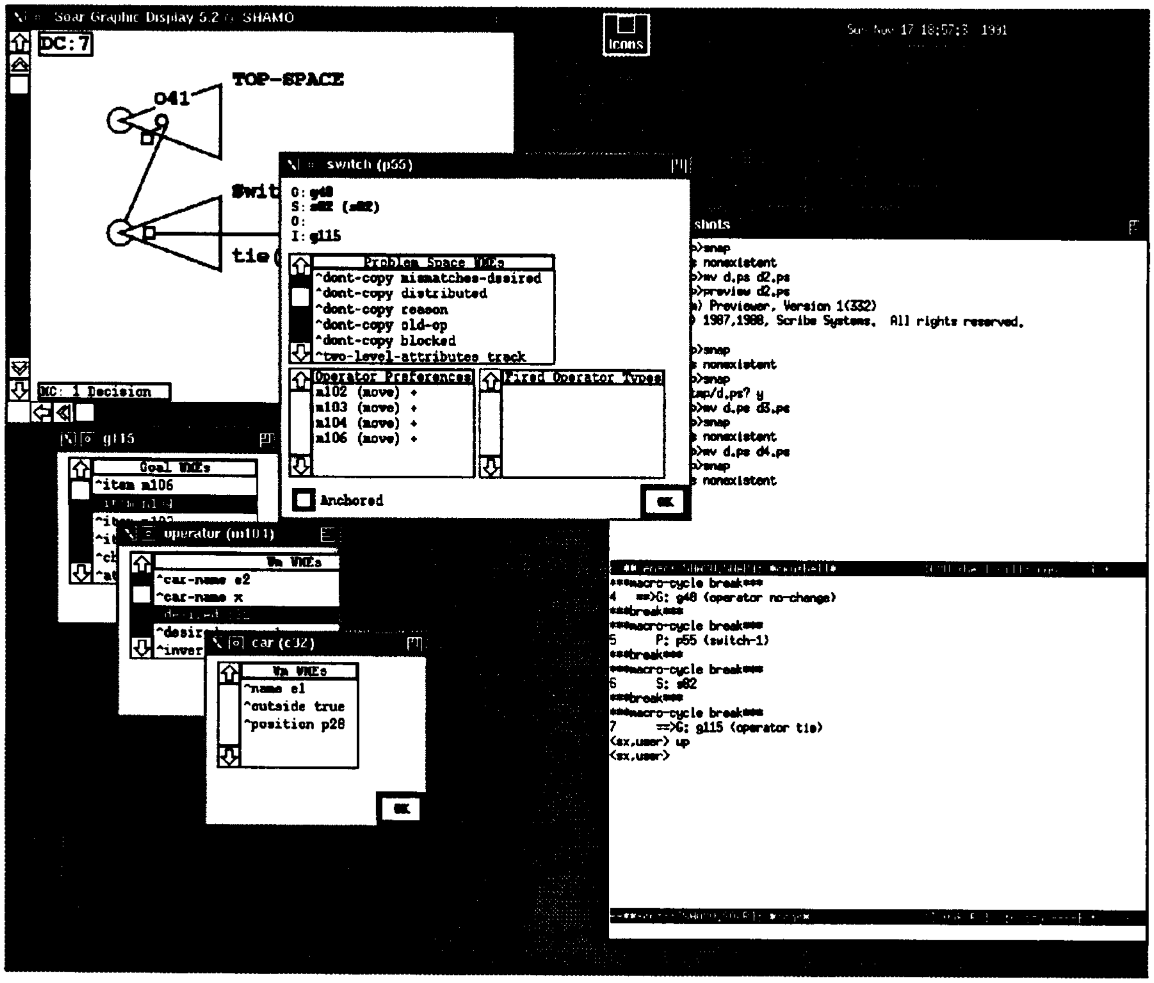
- Click on objects to examine them.
- Hide objects.
- Do not require lots of scrolling.
- Examine memory all the way down.
- Look at multiple objects at once, perhaps from various levels.
- Hide sibling subtrees.
- Hide parent links that are not informative.
- Update structures as Soar runs.
- Run quickly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

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

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

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

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



```
O: g48
S: s82 (s82)
O:
I: g115
```

```
^dont-copy mismatches-desired
^dont-copy distributed
^dont-copy reason
^dont-copy old-op
^dont-copy blocked
^two-level-attributes track
```

```
m102 (move) +
m103 (move) +
m104 (move) +
m106 (move) +
```

```
^item m106
```

```
^car-name e2
^car-name x
^desir
^inver
```

```
^name e1
^outside true
^position p28
```

```
>snap
 nonexistent
>mv d.ps d2.ps
>preview d2.ps
) Previewer, Version 1(332)
 1987,1988, Scribe Systems.  All rights reserved.

>snap
 nonexistent
>snap
mp/d.ps? y
>mv d.ps d3.ps
>snap
 nonexistent
>mv d.ps d4.ps
>snap
 nonexistent
```

```
***macro-cycle break***
4    ==>G: g48 (operator no-change)
***break***
***macro-cycle break***
5       P: p55 (switch-1)
***break***
***macro-cycle break***
6       S: s82
***break***
***macro-cycle break***
7       ==>G: g115 (operator tie)
<sx,user> up
<sx,user>
```

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

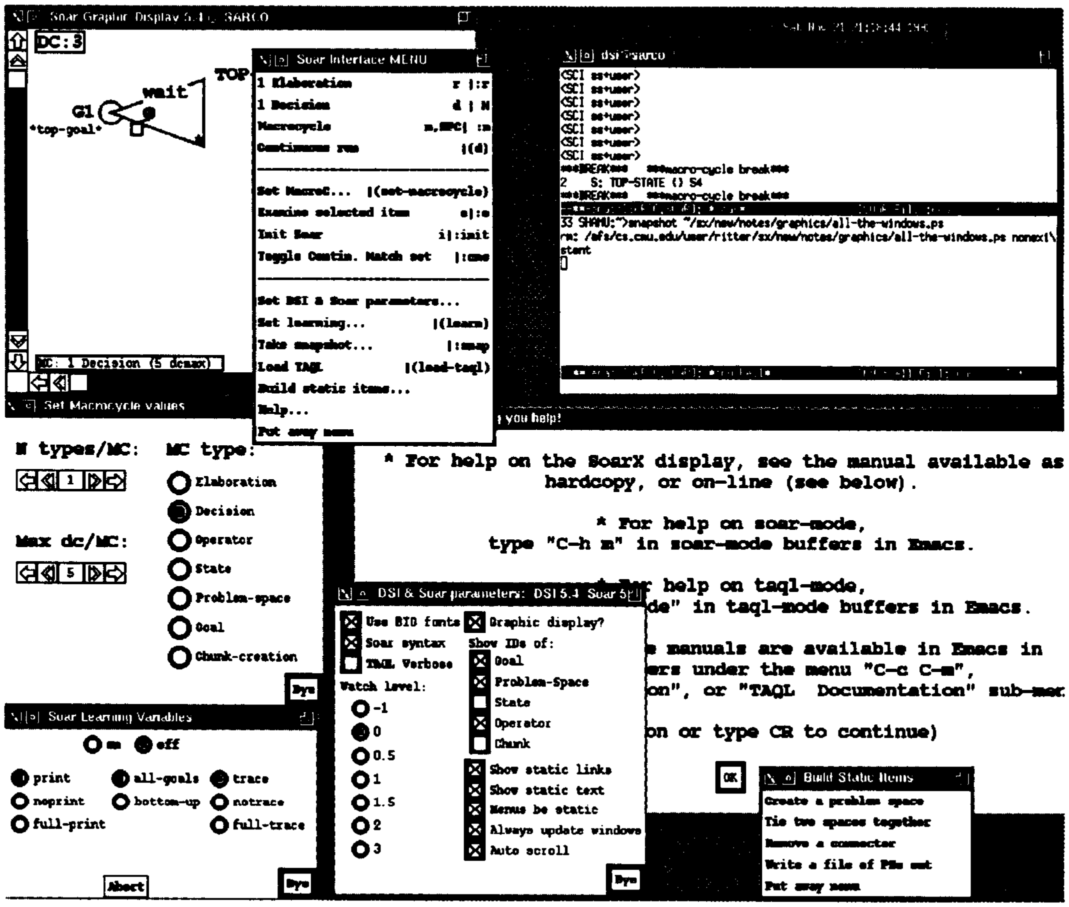
change they can modify their initialization file.

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

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

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

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



```
<SCI ss*user>
<SCI ss*user>
<SCI ss*user>
<SCI ss*user>
<SCI ss*user>
<SCI ss*user>
<SCI ss*user>
***BREAK***   ***macro-cycle break***
2    S: TOP-STATE () S4
***BREAK***   ***macro-cycle break***
33 SHAMU:~>snapshot ~/sx/new/notes/graphics/all-the-windows.ps
rm: /afs/cs.cmu.edu/user/ritter/sx/new/notes/graphics/all-the-windows.ps nonexi\
stent
```

Set Macrocycle values

N types/MC: 1

MC type: ○ Elaboration [✓ ● Decision] ○ Operator ○ State ○ Problem-space ○ Goal ○ Chunk-creation

Max dc/MC: 5

Soar Learning Variables

○ on [✓ ● off]

[✓ ● print] ○ noprint ○ full-print

[✓ ● all-goals] ○ bottom-up

[✓ ● trace] ○ notrace ○ full-trace

DSI & Soar parameters: DSI 5.4 Soar 5

☒ Use BIG fonts
☒ Soar syntax
☐ TAQL Verbose
☒ Graphic display?

Watch level: ○ -1 [✓ ● 0] ○ 0.5 ○ 1 ○ 1.5 ○ 2 ○ 3

Show IDs of:
☒ Goal
☒ Problem-Space
☐ State
☒ Operator
☐ Chunk

☒ Show static links
☒ Show static text
☒ Menus be static
☒ Always update windows
☒ Auto scroll

* For help on the SoarX display, see the manual available as hardcopy, or on-line (see below).

* For help on soar-mode,
type "C-h m" in soar-mode buffers in Emacs.

r help on taql-mode,
de" in taql-mode buffers in Emacs.

manuals are available in Emacs in
ers under the menu "C-c C-m",
on", or "TAQL Documentation" sub-mer

on or type CR to continue)

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

6.3 Creating and modifying the model

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

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

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

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

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

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

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

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

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

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

Figure 6-31: TAQL-mode templates menu.

6.3.3 The Soar Command Interpreter

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

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

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

Figure 6-32: Example TAQL-construct template.

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

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

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

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

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

6.4.1 Providing consistent representations and functionality

In the DSI, while each of the tools can stand alone, they also know about the others, and can interact appropriately with them. For example, commands executed from the menu on the graphic display window can request buffers to appear in Emacs. (In the best of all possible worlds, if the other tool is not present, something appropriate still happens.) Similarly, commands in Soar-mode can run commands in Soar directly. In each tool and across tools, some care has been taken to provide multiple entry points. That is, each command is available in each tool and often in a variety of appropriate and similar ways. For example, there are several ways to run the init-soar command; one can type *(init-soar)*, *:init*, or *init* to the Soar Command Interpreter, choose *Init* on the graphic display menu, or type an "i" on the graphic display window. Help is provided with each tool to facilitate learning the other entry points. For example, the graphic menu item for init-soar includes a listing of the other expressions of this command in the other modules.

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

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

6.4.2 Automating what it can: Keystroke savings

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

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

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

6.4.3 Providing a uniform interface including a path to expertise

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

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

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

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

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

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

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

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

6.4.4 Providing a set of general tools and a macro language

The DSI is designed to support a general activity, inserting knowledge into a Soar model, and is itself general. It can be used to create any Soar model, and is designed to be able to display any Soar model. Macro-languages and an interpreter are available for each component. Common Lisp is available with the graphic display, and GNU-Emacs Lisp is available with the structured editors, Soar- and TAQL-mode. The source code is provided for each component, so what is poorly documented or not documented in sufficient detail can be found in the source code.

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

6.4.5 Displaying and manipulating large amounts of information

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

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

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

6.5 Lessons learned from the DSI

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

6.5.1 The relatively large size of the TAQL grammar

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

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

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

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

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

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

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

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

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

What is search in a problem space? Search in Soar would appear to describe primarily two types of behavior. The first is the application of numerous operators in a single space. Backup, when necessary, would be performed by other operators to modify the state. Soar4 models often used this technique when they performed search. These operators could also use other knowledge sources through impasses to other problem spaces. If the operators were all indifferent, there would not have to be a conflict leading to a tie between operators and the associated impasse. Search in this instance would have a large number of operators applied per problem space, and a large number of states.

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

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

Visual displays of search. With the graphic display having provided dynamic pictures of several model's goal stacks and counts of how many operators the models use and how many operators are used in each problem space, we can now make the argument that search within a single problem space does occur, but it is not the only mode of activity and is too weak of a description of how current models in Soar use knowledge. The graphic display's representation of the goal stack shows that the models are not just performing search in a problem space. Observing the goal stack for Browser-Soar (Peck & John, 1992), Seibel-Soar (Ritter, 1988), Sched-Soar (Nerb & Krems, 1992), MFS-Soar (Krishan et al., 1992), NTD-Soar (John, et al., 1991), NL-Soar (Lehman, Lewis, & Newell, 1991) and Rail-Soar (Altmann, 1992; Newell, P., Lehman, Altmann, Ritter, & McGinnis, forthcoming) indicates that most of the time these models do not apply many operators in a row before subgoaling, and instantiate nearly as many problem spaces as they do operators. After much worry and concern about how what happens when operators walk out the rear of problem spaces, it does not seem to happen all that often. Indeed, only two systems (Red-Soar: (Johnson & Smith, 1991), Able-Soar, Jr.: (Levy, 1991; Ritter, 1992)) have seriously overrun the current limitation of being able to display four or five operators in a problem space before they are no longer graphically in the triangle.

Several models do perform explicit search as part of their behavior. Sched-Soar, Rail-Soar, NL-Soar, and Groundworld, at least, sometimes do it. For example, part of the structure of Sched-Soar's search can be seen in Table 6-25 and Figure 6-27. Other models do not perform any search on the problem space level. If the operator support displays for Browser-Soar are examined (the Appendix to Chapter 7), one can conclude that Browser-Soar's behavior is routine (and this is indeed what Peck and John intended and claim). The operators are applied in a very orderly way. A system that was performing search that depended on the information it found would presumably be less regular.

Table 6-28 presents other possible measures for characterizing behavior as search: the number of operators, the number problem spaces, and instantiations of operators and problem spaces over a typical task episode (as defined by their authors) for several Soar models. In each case, the number of different operator types in each problem space is relatively small (the largest average ratio is approximately 4 operators per problem space in Red-Soar), and the average number of instantiated operators per instantiated problem space is small too.

The proportion of goals that are operator no-changes are shown for each of the programs in Table 6-28. Several of these programs do use lookahead search, but the ratio of operator no-change impasses suggests that these programs are not spending a substantial amount of their effort performing lookahead search.

There are also some unusual, very non-search-line behaviors exhibited by the models in Table 6-28.

Table 6-28: The number of operators, problem spaces, and instantiations of these per run for several Soar models.

	Descriptive			Instantiations					
Model	Ops	Spaces	Ratio (ops/ps)	OP nc goal ratio	Ops	Spaces	Ratio (ops/ps)	Max (ops/ps)	Space
Browser-Soar	31	18	1.72	0.87	238	52	4.57	7.22	Evaluate-items-in-window
Groundworld	33	15	2.20	0.92	531	74	7.17	179.50	Wait-external
Liver-Soar	55	20	2.75	0.72	208	44	4.72	15.66	Check-features
MFS-Soar	69	23	3.00	0.96	347	92	3.77	6.33	Input-variable
NL-Soar	18	8	2.25	0.56	122	52	2.34	6.33	Check-constraints
all learned	18	8	2.25	0.50	38	2	19.00	37.00	Comprehension
NTD-Soar	42	11	3.81	0.93	779	73	10.67	24.40	Sqagr
Rail-Soar	25	13	1.92	0.73	233	48	4.85	8.00	Eval-state
Red-Soar									
plain episode	107	27	3.96	0.94	1258	130	9.67	154.00	Rule-out
"Searchy" episode	109	29	3.75	0.86	923	126	7.32	81.50	Match-hyp-to-antigram
Sched-Soar	11	4	2.75	0.69	866	187	4.63	3.25	Analyze

Red-Soar uses 154 operators in the rule-out space to check constraints when typing blood. The goal stack and pscm-stats in the SX graphic display indicate that Groundworld (108b, 1992) performs one 9-step look ahead search, and then waits for approximately 270 operators. It is a program designed for a continued existence, and can keep running after its initial task is finished. NL-Soar, after it has learned a sentence, performs rather differently from its initial behavior. The "expert" behavior has no search whatsoever, and directly applies a series of 37 operators to understand the 10 words used in the example sentence.

Examination of the visual displays of these models suggests that they can best be characterized as a set of behaviors, including search *through* problem spaces, hierarchical decomposition of problem solving, as migrating and combining knowledge sources, and as search within a single problem space. In a fully learned Soar model, actions just happen automatically in the top space, which is not a search space at all then. The problem spaces used for search have disappeared. Search may remain on other levels. There may be the results of previous searches guiding behavior that can be seen as degenerate search; there may be search going in the external environment; there may be search being performed in the Rete net to find which productions to fire. But in many cases there is not search being performed on the problem space level. These other searches are not wrong, but they must be included in the explanation of behavior of Soar models.

6.5.3 Soar models do not have explicit operators

Problem spaces and their objects, such as operators, do not exist in Soar models in an explicit sense. Within a running Soar model, neither the model nor the modeler can obtain a list of all the problem spaces and operators that exist. They are only available to an observer (including the model itself) by watching the system perform over time, and a history of their appearance and use is not saved automatically (except by the SX graphic display).

The "operators" (or any problem space level object) that are selected for application are not Operators (capital O). A chain of the same operator in the graphic display, all in a row, illustrates that the Operator is not being applied, but instantiations of it are being created and applied. If the same operator was being applied, then a chain would not be an appropriate metaphor, but a moving dot would be. Operator preferences may really be preferences for a given operator, but perhaps they should be seen as operator instantiation preferences. Or how else could you prefer *add(3 4)* over *add(5 6)*? Both appear to be the *add* operator.

What is selected then? The objects selected are instantiations of a semantic object, or object instances generated by an implicit generator. Both the semantic operator and the operator generator are not available for inspection on the symbol level. They are knowledge level objects, and can only be manipulated on that level. The symbol level, which the Soar architecture provides, can only approximate them, and can only obtain them through effort and observation.

This difference between operators and operator instantiations may seem small, but it is necessary to disambiguate these differences for automatic model testing. When aggregating the results of testing the model the objects that are supported must first be identified, and represented across runs of the model. The instantiations are not the theoretical level objects they are often mistaken to be, and cannot aggregate support. Identifying architectural objects is also necessary to display them.

6.6 Summary

The Developmental Soar Interface supports creating models in Soar by treating model building more like an AI programming task. Users can load and run code more directly, manipulating productions as productions, rather than as portions of plain text. By integrating a Soar process within the editor, the textual representation of productions can be quickly augmented with features found only in the process, such as how well a production matches the current goal stack and its contents.

The DSI has added several key ideas to building models in Soar. The first is that the theoretical constructs of Soar models should be displayed. The SX graphic display provides a visual description of the model's structure and behavior over time, and the improved trace provides a better linear description. By aggregating the ephemeral trace over time, the SX graphic display can infer the structure of the problem spaces.

The second is that the user should be able to directly manipulate the theoretical structures. The two structured, integrated editors provide commands for creating, evaluating, and examining models in various ways on the production or TAQL construct level. The SX graphic display provides the ability to examine objects on the PSCM level, but not the ability to create them.

Implementing and using the DSI has provided some lessons on Soar programming languages on the behavior of Soar models. Implementing an aid for TAQL programming pointed out the relatively large size of the TAQL language. Using the graphic display has pointed out two features of Soar models that are more accessible with a graphic display of Soar model's behavior. First, that Soar models include other types of behavior than just search in problem spaces. Second, that within a Soar model, its basic structures, problem spaces level objects, do not exist in an explicit form. Users and systems that want to manipulate Soar models will have to create their own representations of them.

Remaining problems with the DSI. The main components of the DSI represented different levels of support for the user and had different levels of success. The two editors, Soar-mode and TAQL-mode, are well received, and will continue to be used by a large part of the community given normal software maintenance. The current version of the graphic display of the model's behavior and structure has several problems that will have to be fixed for it (or by future systems) to truly useful.

States remain essentially untraced. This is a problem both for testing predictions against protocols and for basic model building. What the necessary information is, how to let the user represent it, and how to provide it succinctly, remains a high priority design issue. Implementing the trace once it has been designed is probably straightforward.

Users have requested several extensions to SX display. These include the ability to remove working memory elements and to show how a single production matches over time, but the largest problem with the SX graphic display has been speed. This is the largest acceptability issue that the graphic display has faced. People who do not use it, do not use it because it unacceptably slows down Soar. It has only been truly acceptable where speed is not an issue, such as for teaching novices and for demos, but the lack of later acceptability has even encouraged some novices to not start to use it. As the Soar architecture gets implemented in C, this system should get duplicated in C, but it is unclear that the relative slow down will not also occur there. Any display, but particularly graphic ones, may always offer at best a two-to-one slow down compared with the underlying application (Myers & Rossen, 1992).

Several graphic design issues remain. The dynamic structures of Soar in the goal stack are all represented fairly well. How to represent several of the static structures remains a problem, for example, how to nicely display the operators in a space; we use a simple way for chunks, can we find a similar one for operators? Representing the states that exist in a problem space suffers from a similar problem.

Finally, can we tie creating and editing productions to the graphic display? The ability to click on chunks and examine them has proved useful in exploring the types of chunks that end being assigned to *Every-Space*. Being able to go between a graphic and textual representation is appealing.