Ritter, F. E., Jones, R. M., & Baxter, G. D. (1998). Reusable models and graphical interfaces: Realising the potential of a unified theory of cognition. In U. Schmid, J. Krems, & F. Wysotzki (Eds.), *Mind modeling - A cognitive science approach to reasoning, learning and discovery.* 83-109. Lengerich (Germany): Pabst Scientific Publishing.

Reusable models and graphical interfaces: Realising the potential of a unified theory of cognition

Frank E. Ritter[†] (**frank.ritter@nottingham.ac.uk**)
Randolph M. Jones[‡] (**rjones@eecs.umich.edu**)
Gordon D. Baxter[†] (**gdb@psychology.nottingham.ac.uk**)

[†] Department of Psychology, University of Nottingham, Nottingham NG7 2RD, UK

[‡] Artificial Intelligence Laboratory, University of Michigan,
1101 Beal Avenue, Ann Arbor, MI 48109-2110, USA

Abstract

Many results and techniques applicable to human-computer interaction (HCI) have been discovered by using cognitive modelling. However, few of these lessons have been applied to improve the explanation and illustration of cognitive models themselves. We have started to redress this imbalance by developing for a well-known cognitive architecture (Soar) a graphical user interface and reusable models. The general displays in the interface facilitate an understanding by showing the models' behaviour in architectural terms; model-specific displays help explicate the concepts and results of the models at the task and knowledge level. We also illustrate how external displays can help model interactive behaviours. The models demonstrate that existing models can be replicated and integrated within a unified theory of cognition, and, where appropriate, be made into a high level modelling language that includes learning. Their use in cognitive science courses suggests that the understanding of cognitive models and cognitive modelling could be greatly improved if similar displays and reusable models were more widely available. (165 words)

Introduction

Cognitive models suffer from usability problems. Few lessons from the field of human-computer interaction (HCI) have been re-applied to increase the understanding of the models themselves, even though many results and techniques in HCI have been discovered using cognitive modelling (John, this volume).

There are also serious problems restricting the reuse of cognitive models. It is probably fair to say that cognitive models are not generally reused, even when they have been created in a cognitive architecture designed to facilitate their reuse. It is also probably fair to say that cognitive models can often be difficult to explain and understand. There are exceptions,[1] but overall cognitive modelling does not have the level of system reuse and visual displays that the AI and expert systems communities now take for granted.

We have started to address these related limitations by developing a graphical user interface for a well-known cognitive architecture, Soar (Newell, 1990). We also describe here the

[1]Pearson's Version 2 of the Symbolic Concept Acquisition model and its explanatory displays is an exception that helped inspire this work (available at ai.eecs.umich.edu/soar/soar-group.html). Other exceptions include PDP toolkits such as O'Reilly's PDP++ (http://www.cs.cmu.edu/Web/Groups/CNBC/PDP++/PDP++.html).

replication and extension of several earlier cognitive models, which have been modified to make them easier to understand and to reuse as a high level modelling language. We will further motivate each of these issues and then present how they are achieved with the graphic user interface and the models themselves.

Visual interfaces make behaviour visible

Graphic displays are often useful aids when solving problems (Larkin & Simon, 1987). This is also true in cognitive modelling. When a graphic interface was previously available for Soar (Ritter & Larkin, 1994), it led to some new understandings about Soar models. We found, for example, that few extant models did extensive search *in* problem spaces, but rather did search *through* problem spaces, using relatively few operators in each space. When Soar was reimplemented in C, the graphic interface was lost. However, the recent inclusion in Soar of the Tcl/Tk scripting and graphics language (Ousterhout, 1994) made it possible to create a new graphical user interface.

General architectural displays and model-specific displays are both necessary. The general displays are useful for developing, understanding, and explaining any model by showing the behaviour in architectural terms. They also facilitate an understanding of the architecture itself. Model-specific displays may be necessary for higher level behavior, such as specific knowledge level or task level information.

Explicit interfaces support building models of interaction

Support for graphical interfaces can also encourage model designers to make explicit the model's task and its representation and interaction with the external environment. An excellent example of this is provided by our reimplementation of Brown and VanLehn's (1980) subtraction model. In the original model, there was a uniform representation of the concepts and knowledge used in the task. This representation included symbols that presumably come from a perceptual process (e.g. the digits and structure of a subtraction problem), internal goals and operators used to solve the problem, and symbols representing intentions that could map onto physical motor commands (e.g. writing down a digit). All of these symbolic representations were lumped together in a way that made the actual inputs and outputs of the task somewhat unclear.

In our reimplementation of the subtraction model, we made a clean division of perception, reasoning, and external action. Although it is not a high-fidelity model of human perception or motor control, it makes explicit the inputs and outputs of the task, and it precisely defines when information goes into the reasoning system and what the reasoning system must do with it. For the subtraction task, this means there is a simulated world with a blackboard containing a subtraction problem. There is an explicit focus of attention for the reasoner, so it can only perceive the symbols representing a single column at a time (i.e. the simulator only sends the reasoner information about the current focus). There is a precise set of actions the reasoner can use to effect change in the simulated world (scratching out and writing down numbers). In addition, the various parts of the world simulation are represented graphically, so an observer can watch the model solve a problem, see the model's focus of attention and its marking on the blackboard, and generally follow the model's behaviour as if it were a human solving subtraction problems. We have found making the model's behaviour visible useful for other cognitive models as well (Bass, Baxter, & Ritter, 1995; Jones & Ritter, 1997).

Reusable cognitive models

There are several goals to consider when developing models for reuse. The models should be clear, easy to run, and easy to integrate with other models. Many of these desires are consonant with the call for unified theories of cognition (Newell, 1990).

Competitive argumentation (VanLehn, Brown, & Greeno, 1984) proposes that modellers should know which mechanisms in their model provide the power—which mechanisms lead the model to fit the data. The reuse of cognitive models allows us to extend competitive argumentation and the simple reuse of existing code, by packaging mechanisms as a general explanation of particular patterns of behaviour and for application to other tasks.

Computational modelling of psychological phenomena starts with developing a model that performs the task. This model is then analysed empirically or theoretically in an attempt to explain experimental data relating to the phenomena of interest. The need to develop a computer program that actually performs the task often forces the designer to make assumptions that do not necessarily contribute to the psychological theory. When these models are reimplemented, however, different design assumptions may be made, and it (hopefully) becomes clearer which parts of the implementation are theoretically important, and which are simply scaffolding. Building reusable models further clarifies these distinctions so that future users of the model can readily see which aspects of the model are meant to be explored. In addition, once the important parts of a model are explicitly identified, they should be more easily reused in further research on cognition.

How Soar supports building reusable models

In order to construct a library of reusable, cognitive models, it is necessary to develop the models within a uniform, integrated cognitive architecture. For our current efforts, we used Soar (Newell, 1990), which was explicitly designed as an architecture to model cognition. It models behaviour down to the level of deliberate actions that take on the order of hundreds of milliseconds for humans to perform. Because the vast majority of existing models and data are at this time-scale or above, Soar is appropriate. In addition, Soar dictates that all cognitive tasks be viewed as search through problem spaces, stressing the importance of perceptual and internal symbols, operators, and goals.

Soar supports the development of a wide variety of models within a single representation scheme and performance paradigm, so we should also expect it to be of use in reimplementing existing cognitive models. This expectation was realised when the reimplementations of the subtraction model and Able, which serve as example models here, proved to be relatively straightforward.

Some cognitive models may map more easily onto Soar's assumptions than others. For example, the reasoning paradigms used in the repair theory system were slightly awkward to implement within Soar. However, rather than viewing such difficulties as a reason not to use Soar, we view them as opportunities to learn more about both the explicit claims of the cognitive model and the power and characteristics of the Soar architecture. If something seems difficult to implement within a uniform architecture, we may question whether it is an important theoretical aspect of the model. If it does seem important, it is worth exploring how Soar might be made to incorporate similar cognitive constraints, or whether such constraints might arise out of Soar's existing principles. Behaviour faster than a few hundred milliseconds (mostly perceptual and motor processes), for example, would most likely be better modelled in some other architecture or by extending Soar to that level.

Reusing the models presented here

If a unified theory of cognition (such as Soar) is to provide an approach that supports the integration of models (Newell, 1990), then the models have to be reusable. The first step to support this is to make the model and its associated documentation publicly available, which we do.[2] The next step is to implement models in a way that fosters reuse.

Brown and VanLehn's (1980) theory of systematic errors in subtraction problem solving makes it clear that errors may arise from the combination of poorly learning a particular problem-solving strategy together with using a fixed set of mechanisms for repairing knowledge gaps. Reimplementing Brown and VanLehn's model allowed us to make an explicit distinction between the general problem-solving regime and the repair theory. In addition, we built a number of different specific subtraction procedures, which further emphasised the distinction between theory and parameters (in this case, knowledge of a specific subtraction procedure).

The reimplemented subtraction models are useful for understanding subtraction and exploring buggy behaviour. They are particularly useful for examining extensions to Brown and VanLehn's work by studying different subtraction procedures or different repair methods in order to explain the variety of bugs that humans exhibit.

Our first reimplementation of Able (Bass et al., 1995), from Soar 4 to Soar 6, was not difficult, but it was not as straightforward as we would have liked. There we reused Able's principle application mechanism to develop a simple reasoning component that learned in a model that solved a simple air traffic control-like task (Bass et al., 1995). The bulk of the effort creating this version, Able III, went into organising Able's structure into a general mechanism that could be routinely reused in other models of formal domains by adding domain principles. We demonstrate below how we managed to reuse it very quickly to model a simple task.

Both the subtraction models and Able serve as educational examples of how the Soar architecture works, and how it can be used to model different types of human behaviour. The displays are also reusable. The general graphic interface are directly reusable as they will be

[2]The models described here and the graphical user interface are available from http://www.ccc.nottingham.ac.uk/pub/soar/nottingham/.

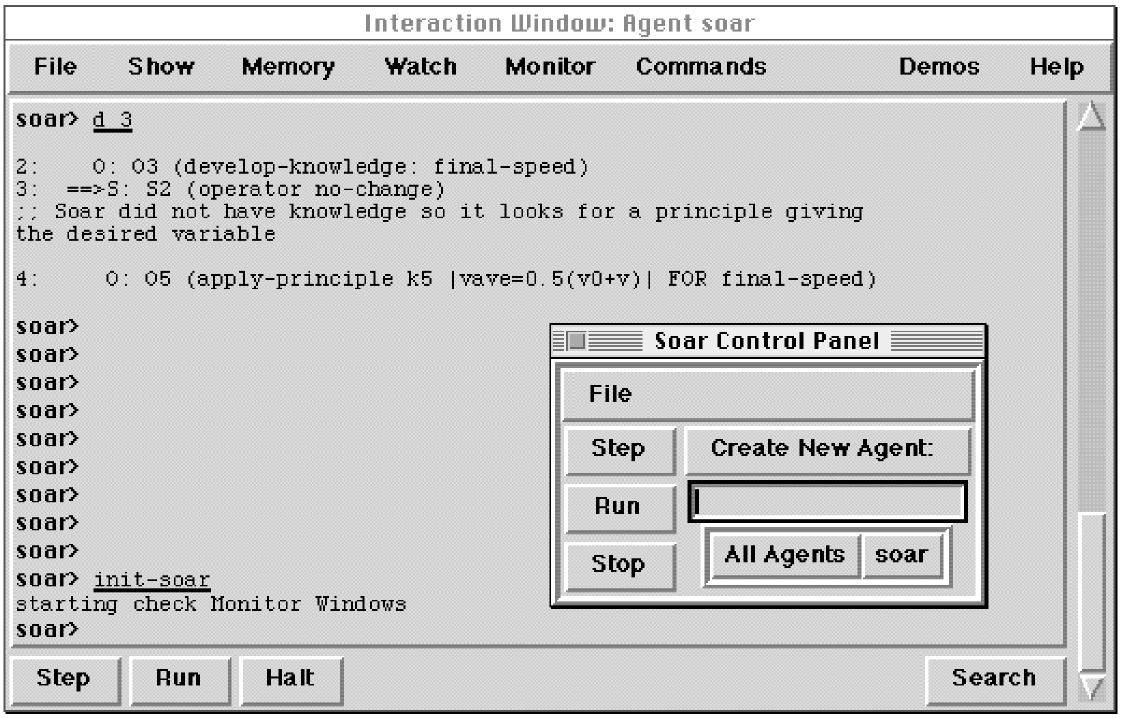
useful to anyone wanting to increase their understanding of Soar or particular Soar models. The specific displays serve as examples and can be modified for use with other models. We now describe in detail the interface and then the models.

General Architecture Displays

We have built a graphical user interface for Soar, called the Tcl/Tk Soar Interface (TSI). The TSI consists of a set of general displays that are available as an extension to Soar. The graphical interface currently runs with the latest Soar release (7.0.4) under Unix and MacOS. Rather than only providing a command line, an interaction console with menus is available, shown in Figure 1. The current version of Soar (7), allows multiple models (agents) to be run to be run in the same process. The Soar Control Panel (inset) is used for selecting, creating, and running these agents. The ten most commonly used Soar commands (Nichols & Ritter, 1995) are either directly supported by the displays, bound to keystrokes, or on menus. The set of available commands can also be restricted for novices.

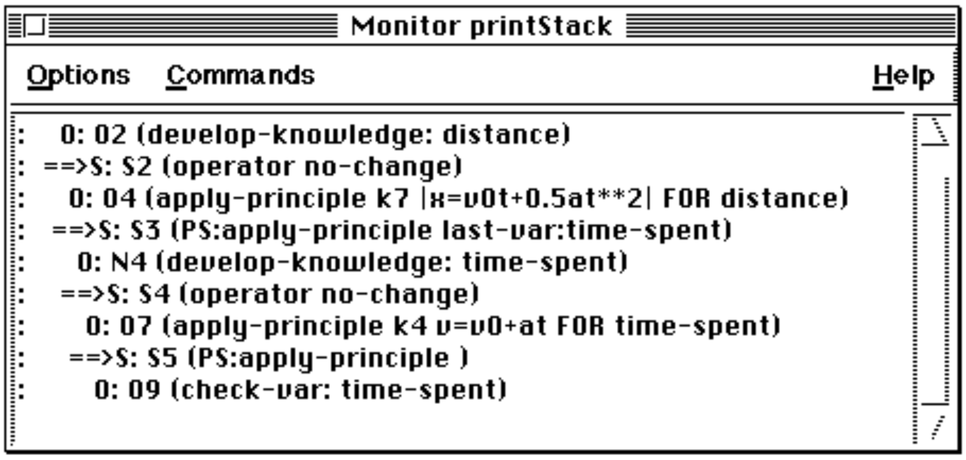
The interface includes context-sensitive pop-up menus for inspecting the various memories in Soar. When the mouse is clicked over a string in the console window, the interface determines which type of object is being chosen (e.g. long-term-memory production rule, short-term-memory attribute, short-term-memory object) and pops up a menu of appropriate methods for viewing that object. This interface helps a novice understand the different types of memories in Soar, and allows the experienced user to search for useful information quite rapidly.

A complete Tcl/Tk interpreter is included in the interface, together with methods for the command interpreter to communicate with a Soar agent. Integration with a Tcl/Tk interpreter

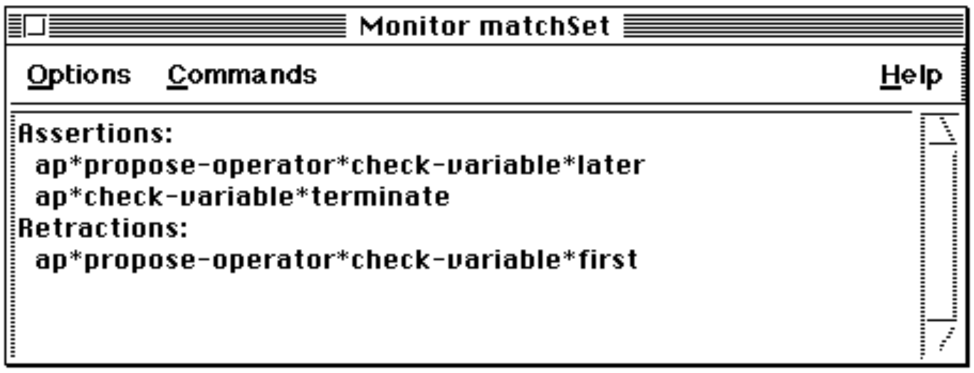


```
soar> d 3

2:    O: O3 (develop-knowledge: final-speed)
3:  ==>S: S2 (operator no-change)
;; Soar did not have knowledge so it looks for a principle giving
the desired variable

4:     O: O5 (apply-principle k5 |vave=0.5(v0+v)| FOR final-speed)

soar>
soar>
soar>
soar>
soar>
soar>
soar>
soar>
soar>
soar> init-soar
starting check Monitor Windows
soar>
```

Figure 1. The TSI console window that replaces the simple command line interface. The agent control window ("Soar control Panel") is overlaid.



```
:   0: 02 (develop-knowledge: distance)
:  ==>S: S2 (operator no-change)
:    0: 04 (apply-principle k7 |x=v0t+0.5at**2| FOR distance)
:   ==>S: S3 (PS:apply-principle last-var:time-spent)
:     0: N4 (develop-knowledge: time-spent)
:    ==>S: S4 (operator no-change)
:      0: 07 (apply-principle k4 v=v0+at FOR time-spent)
:     ==>S: S5 (PS:apply-principle )
:       0: 09 (check-var: time-spent)
```



```
Assertions:
  ap*propose-operator*check-variable*later
  ap*check-variable*terminate
Retractions:
  ap*propose-operator*check-variable*first
```

Figure 2. Selectable TSI display windows updated every cycle when they are open, showing the current goal stack (top window) and the rules that will fire next (bottom).

allows for the fairly rapid prototyping and development of external simulation environments in which to situate a cognitive agent. This assists in clarifying the distinction between cognitive models and their external environments.

There are three optional displays that are continuously updated to display information on the current goal stack, the rules are about to fire, and the details on how the next operator will be selected. In each of these displays the user can also bring up a help menu and directly run the model. Figure 2 shows two of these displays. The continuous goal stack display, shown at the top of Figure 2, indicates the order of operator applications and the current goal stack. Users can examine the substructure of objects in the stack using a separate window (not shown). Users can also select how much detail is displayed, choosing to print several layers of substructure by default or continue to examine substructures one level at a time. The continuous match set display, at the bottom of Figure 2, provides a display of the rules that have matched the current working memory and will fire in the next architectural cycle. Users can display the structure of the matching rules in a separate window.

Only anecdotal evidence about the usefulness of these displays is available so far. When an early version was introduced to a psychology class on programming cognitive models, all of the students elected to use the displays rather than the original command line interface. Subsequent classes that have also used the displays have not reported any problems and appear to have learned Soar more readily than previous classes.

A teaching tool: The subtraction models

The revised subtraction model was initially developed for a graduate course on symbolic cognitive modelling at the University of Michigan, and has subsequently been used for undergraduate courses on AI and cognitive science at Bowdoin College and the University of Nottingham. The subtraction systems supplemented theoretical discussion and analysis with hands-on design, implementation, and experimentation with a specific computational model.[3]

The revised subtraction model was derived from Brown and VanLehn's (1980) paper on repair theory. Brown and VanLehn gathered data from a large number of children learning to do multi-column subtraction. From this data they created a catalogue of the different types of systematic errors (or bugs) that children make when learning to do subtraction and proposed a theory for the source of some of those bugs.

The basic idea of Brown and VanLehn's theory is that children initially acquire a correct or nearly correct procedure for solving subtraction problems. Children that have a nearly correct procedure can still solve some problems, but other problems reveal gaps in their knowledge, and so they encounter impasses. In the face of these impasses, there are a number of courses of action the children might take to find a solution to the problem. If a child did not know how to borrow, for example, and came across the task of subtracting 9 from 7 in a particular column, they might decide to subtract 7 from 9 instead. Subtracting 7 from 9 is something the child knows how to do, and it allows them to continue working on the problem. However, it also leads to an incorrect answer. If the child exhibits this particular behaviour systematically, it is a "bug" that can be used to predict errors that the child would generate on specific problems. A collection of potential repair mechanisms that result in observed bugs are at the heart of Brown and VanLehn's repair theory.

In repair theory, children or others can exhibit systematic errors (errors that persist due to the same incorrect knowledge or procedure) either because they have an incorrect procedure that still allows them to compute answers or because they consistently use particular repairs to overcome their incorrect knowledge when it leads them into impasses. Later work by VanLehn (1983; 1989) focused on learning and how the incorrect subtraction procedures could be acquired in the first place. In contrast to this work (and the Able model described below), the subtraction model we developed did not learn.

The revised subtraction models

The interface to the revised model (developed using Tcl/Tk) is shown in Figure 3. This interface allows students to select for analysis three different procedures for subtraction. The reason for developing three different subtraction procedures was to allow cognitive modelling students to explore how different procedures might lead to different types of systematic errors, even given the same set of repair strategies for each procedure. In addition, however, each of the three subtraction procedures uses the same production rules for most of their representation

[3] This model is available from http://ai.eecs.umich.edu/people/rjones/subtract/index.html

and differs only in some of their operator preconditions. This provides one example of the benefits of reusable cognitive models: they can be used to highlight the general aspects of the models and to explore various alternatives.

The interface also allows students to experiment with the models by deleting operator proposals. When impasses arise, they can select among tied repair operators. The models then give (usually) incorrect answers corresponding to one of the subtraction bugs that Brown and VanLehn identified in children's subtraction procedures.

To implement Brown and VanLehn's subtraction model within Soar, we divided the model into three major components. The first component consists of the general regime for reasoning assumed by Brown and VanLehn: reasoning involves the (possibly conditional) execution of steps in a solution strategy in a forward manner (i.e. there is no internal search and no explicit backward chaining from goals to actions). Some execution steps involve the creation of new goals, which in essence invoke a new procedure that constitutes part of the overall problem-solving strategy. Other execution steps involve explicit intentions in solving the problem, such as writing down a number, scratching out a number, or explicitly shifting focus of attention from one column to another. Our version of the subtraction model interacts with the external blackboard simulator, so the explicit intentions translate directly into output commands sent to the simulator. In response, the simulator would changes state and sends a representation of the new state (subject to the model's current focus of attention) back to the reasoning process via a symbolic, perceptual input link.

The second component of the Soar implementation includes the specific set of operators that constitute each particular procedure for solving subtraction problems. This includes knowledge of particular subtraction goals, like "borrow from a column", and knowledge of "primitive" reasoning steps, like "write the difference of two numbers" or "shift focus to the left". The third component consists of a set of operators that implement the various types of repair strategies Brown and VanLehn hypothesise in repair theory. In the Soar implementation, the repair operators become applicable when execution of the subtraction strategy encounters an impasse (execution cannot proceed) during a problem. Each repair operator makes some change to the internal representation of the problem that (usually) allows the system to continue executing the subtraction procedure. Repairs involve things like skipping a column, swapping numbers in a column, or incrementing a number in a column.

Brown and VanLehn's general reasoning paradigm maps quite nicely onto the existing Soar architecture. In general, reasoning steps in Brown and VanLehn's subtraction model consist of operator preconditions, operator applications, subgoals, and subgoal satisfaction conditions, each of which map directly onto the representation of knowledge and reasoning in Soar. The primary difficulty in implementing Brown and VanLehn's model arose from the fact that their subtraction procedure has script-like portions, which are basically an ordered set of steps to follow, without dependencies between steps. With this reasoning formulation, one can delete an intermediate step without disturbing the flow of the processing. Implementing such a script-like mechanism in Soar is not difficult, but there are simpler ways of achieving the same effect of implementing a problem-solving procedure. In addition, we could not simply ignore this aspect of Brown and VanLehn's reasoning paradigm, because it leads directly to the model's explanation of where systematic problem-solving errors arise.

Brown and VanLehn's subtraction procedure assumes a strictly internal representation for subtraction problems. Thus, even though their procedure can only use information from one column at a time, the whole of the problem's representation is always accessible in memory. The main impact of this arises when the computer simulation traverses columns during a borrow. In Brown and VanLehn's procedure, this occurs by pushing and popping columns on an internal stack. A pop can shift the focus of attention directly back to the appropriate column, and the procedure knows it is done with a traversal when the stack is empty.

We built a simulated external environment (the blackboard) for the subtraction model to use. We developed this environment because it is a more realistic match to how children solve subtraction problems on a worksheet. The blackboard environment used by the model appears at the top of Figure 3. Using the simulated blackboard also forced us to make internal intentions and external actions explicit in the model. Finally, the blackboard is useful because it gives student modellers something familiar to watch and allows them to directly observe the task level behaviour of the model.

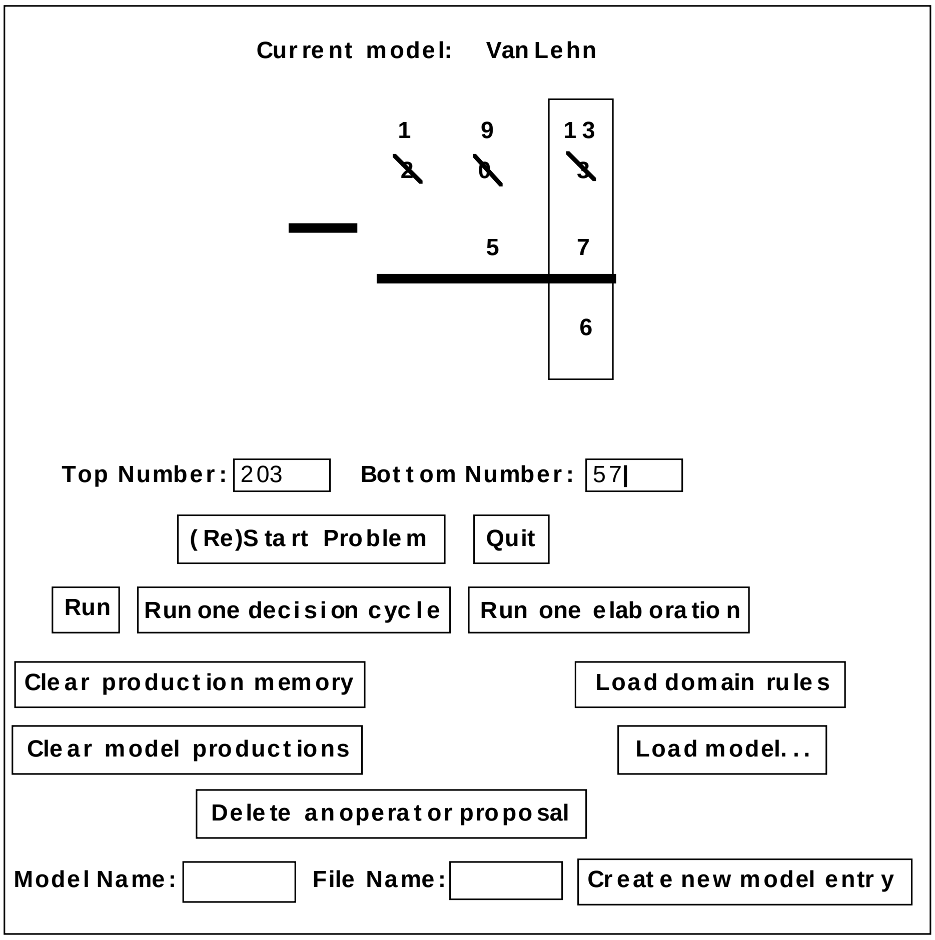


Figure 3. The blackboard display and interface for running the subtraction models.

Our external simulation, however, restricts the model's access to the information in the current focus column. In order to switch columns, a physical action operator of MOVE-LEFT or MOVE-RIGHT must be executed. In the external environment there is no analogue to an internal stack. Although the computer simulation can still use a goal stack to keep track of how many columns have been shifted, the actual shifting has to occur as a physical, external action, with no backtracking possible. This means our implementation of Brown and VanLehn's subtraction procedure had to be altered slightly. A pseudocode description (in the style used by Brown and VanLehn) appears in Table 1 with the primitive operations (i.e. operations that involve output commands to the blackboard interface) listed in italics. All other operations are internal constructs that effectively just create subgoals. The interested reader should compare this algorithm with that of Brown and VanLehn (1980).

Table 2 presents an example trace of the VanLehn procedure on a particular subtraction problem and a buggy trace of the same problem when the step proposing a BORROW-FROM-ZERO is missing from the BORROW-FROM procedure.

Table 1. Brown and VanLehn's subtraction procedure with external focus operations included. Possible application conditions for each step are surrounded by {..}. Thus, {} means "always execute this step". A satisfaction condition of TRUE means to apply only one step of the procedure. FALSE means apply *every* applicable step of the procedure in sequence. Otherwise, the procedure terminates when the satisfaction condition is satisfied.

```
SUBTRACT()   Satisfaction condition: TRUE
  {} -->                       COLUMN-SEQUENCE

COLUMN-SEQUENCE()   Satisfaction condition: top is blank
  {} -->                       SUBTRACT-COLUMN
  {} -->                       MOVE-LEFT
  {} -->                       COLUMN-SEQUENCE

SUBTRACT-COLUMN()   Satisfaction condition: answer is not blank
  {BOTTOM IS BLANK} -->        WRITE-ANSWER
  {TOP < BOTTOM} -->           BORROW
  {} -->                       DIFFERENCE

BORROW()   Satisfaction condition: FALSE
  {} -->                       MOVE-LEFT
  {} -->                       BORROW-FROM
  {} -->                       MOVE-RIGHT
  {} -->                       ADD-10

BORROW-FROM()   Satisfaction condition: TRUE
  {TOP IS ZERO} -->            BORROW-FROM-ZERO
  {} -->                       DECREMENT

BORROW-FROM-ZERO()   Satisfaction condition: FALSE
  {} -->                       WRITE-9
  {} -->                       MOVE-LEFT
  {} -->                       BORROW-FROM
  {} -->                       MOVE-RIGHT
```

Table 2. A sample trace on the problem 205 - 47 for the VanLehn procedure and a buggy version of it with the BORROW-FROM-ZERO step missing from the BORROW-FROM procedure. Operations in italics represent external actions.

Original VanLehn Procedure

```
Subtract
   Column-Sequence
      Subtract-Column
         Borrow
            Move-Left
            Borrow-From
               Borrow-From-Zero
                  Write-9
                  Move-Left
                  Borrow-From
                     Decrement
                  Move-Right
            Move-Right
            Add-10
         Difference
      Move-Left
      Column-Sequence
         Subtract-Column
            Difference
         Move-Left
         Column-Sequence
            Subtract-Column
               Write-Answer
            Move-Left

Answer: 158
```

Buggy VanLehn Procedure

```
Subtract
   Column-Sequence
      Subtract-Column
         Borrow
            Move-Left
            Borrow-From
               Decrement
                  IMPASSE:
                  Please choose a repair
                    1. Dememoize
                    2. Increment
                    3. Write-9
                    4. Add-10
                    5. Repair-Quit
                    6. Repair-Skip
                    7. Repair-Swap
                  Choice: 3
                  Write-9
            Move-Right
            Add-10
         Difference
      Move-Left
      Column-Sequence
         Subtract-Column
            Difference
         Move-Left
         Column-Sequence
            Subtract-Column
               Write-Answer
            Move-Left

Answer: 258
```

For the project as it was presented in the cognitive modelling classes, the basic model was referred to as the *VanLehn* model. We also provided the students with two alternative subtraction procedures. The first, which we call *Neo-VanLehn*, is the same as the *VanLehn* procedure, except that it eliminates the special BORROW-FROM-ZERO operation. Instead, if the top digit is zero when doing a BORROW-FROM operation, the procedure does a BORROW followed by a DECREMENT. The second alternative procedure has a flatter structure, eliminating both the BORROW-FROM-ZERO and SUBTRACT-COLUMN operations from the *VanLehn* model and collapsing their primitive actions into other operators. This procedure was called *Neches*, because it is loosely based on a subtraction procedure presented by Neches, Langley, and Klahr (1987).

How student modellers used and extended these models

For the cognitive modelling course at the University of Michigan, the students read Brown and VanLehn's (1980) paper and a paper on Soar, and attended an hour-long tutorial on Soar. They were then shown how to use the subtraction interface. In addition they were given descriptions of the three subtraction procedures and the repair strategies we implemented using

Table 3. Task given to students working with the subtraction models.

1. Examine two of the provided subtraction procedures and try to identify about ten of Brown and VanLehn's bugs that could arise from each procedure. Make a list of the predicted bugs, then run the system and try to generate the bugs with appropriate deletions of operator proposals and use of repair strategies.
2. Find a way to generate a few bugs that the models presented by Brown and VanLehn (1980) could not generate. This could be done either by implementing a new procedure for doing subtraction, or by developing and implementing a new repair strategy (or a combination of the two). Demonstrate each of the new bugs with the newly implemented procedure and/or repair.
3. Write a report on the findings in steps 1 and 2, and give us feedback on the overall project.

Soar. The students divided into groups of two or three (each including one person already familiar with Soar), and were allowed two weeks to complete the assignment shown in Table 3.

The results of the project were an unqualified success. Every group completed the first part of the project, with no two groups generating the same set of bugs. For the second part, each group came up with a unique solution. Three groups created new subtraction procedures. One of the new procedures was the subtraction procedure taught to a group member who grew up in India. In this procedure, instead of decrementing the top digit of a column for a borrow, one increments the bottom digit. The other new procedure was designed to be an "anytime algorithm" for subtraction, basically working the problem from left to right instead of right to left. The third group modified the VanLehn procedure to treat borrowing into zero as a special case, allowing this to become a point at which bugs could occur. Another group focused on what new bugs could be generated if some of the operators were actually changed rather than simply deleted. Finally, one group explored the use of some new, rather specific repair strategies, which seemed necessary for generating some of the trickier bugs. All of the groups were able to generate bugs that were found in Brown and VanLehn's catalogue of bugs, but that were not generated by Brown and VanLehn's model.

All of the students clearly improved their understanding of Brown and VanLehn's work, and gained some insight into the internal workings of Soar. They also appeared to gain an increased appreciation for the design and analysis of computational models of human processing. The project provided an excellent complement to the theoretical frameworks being discussed in class. Finally, everyone was excited about the ability to combine different procedures and techniques in order to increase the coverage of bugs discovered by Brown and VanLehn. The exercise suggested that a similar interface could be used to analyse a number of different subtraction procedures, which taken together may be able to generate a large number of the bugs.

A rule learning utility: Able III

Able III is a substantially revised version of Levy's Able-Soar, Jr. (Levy, 1991). As part of the revision, the code was substantially updated and documented. The model was extended to cover more problems, and we created graphic displays to illustrate its specific behaviours.

Most importantly, we modified Able's principle application mechanism so that it can be used as a building block for other models.

Able is a model of physics problem solving (Larkin, 1981; Larkin, McDermott, Simon, & Simon, 1980b). Able's predictions at the level of applying physics principles have been matched to extensive amounts of protocol data of subjects solving kinematics problems. Able initially works backward from the target variable(s), using means-ends analysis to find which principles to apply; after learning, it ends up with a more expert behaviour, working forwards from the known variables without search.

Able does not provide as detailed a model of physics problem solving as some more recent models (Elio & Scharf, 1990; Ploetzner, 1995; VanLehn, Jones, and Chi, 1992), in that it does not model as much of the complete process, such as learning the principles, setting up the problem, or performing the algebraic manipulations. The novice to expert transition in formal domains like physics is modelled fairly well by Able, albeit on the high level of principle application. The model emphasises how the order of principle application in formal domains changes with practice, even though novices may fully apply a principle to solve for mass while experts may carry mass forward through the application of a principle, knowing that mass will cancel out later.

Able was initially written as two related models: ME to simulate novice physics problem solvers (barely able); and KD to simulate expert problem solvers (more able) in kinematics (Larkin et al., 1980b) and fluid statics (Larkin & Simon, 1981). These models matched problem solving protocols very well. The models were later unified by a chunking mechanism that allowed the model to learn while solving problems, thereby showing how the novice model could become an expert model through practice (Larkin, 1981). This unified Able model was translated by Levy (1991) into the Able-Soar, Jr. model that ran in Soar 4. His translation suggests that Able's learning mechanism was essentially the chunking mechanism in Soar (Newell, 1990). Levy's work remains an interesting example of how quickly someone can learn and model in Soar, for he wrote it in two weeks. His model is where we started.

Able's novice/expert performance characterisation is similar in some ways to Klein's (1989) widely applied theory of recognition-primed decision making (which might more correctly be called "recognition-guided problem solving in dynamic tasks" because experts typically do not simply make a single decision but a series of decisions based on interaction with the environment). Like Klein's theory, expert Able works forward from known information; its behaviour is based on previous problem solving, and Able does not consider alternative actions. Able is different in that it is spelled out in enough detail to implement some of the structural details of behaviour in a limited area, whereas Klein's theory remains descriptive.

So far, Able has only been applied to formal domains: those "involving a considerable amount of rich semantic knowledge but characterized by a set of principles logically sufficient to solve problems in the domain" (Larkin, 1981, p. 311). So, mathematics, physics and sophisticated games (e.g. chess) are formal, whereas biology and English literature are much less so. Whether Klein's domains (e.g. fire fighting) are formal or can be formalised is unclear. The field of cognitive science assumes that they could be, and the attempts to build expert systems in these areas are consistent with that belief. Able suggests that it may be possible to create a wide range of cognitive models that start to explain the novice-expert differences that Klein reports by the way they improve through performing tasks.

The revised Able model

In Able III we have updated Able-Soar, Jr. to run under Soar 7 (Congdon & Laird, 1995). There have been several changes to the Soar architecture and its implementation since Levy's model, including allowing the reuse of state representations and making the representation of problem spaces more implicit. Some of the rule syntax, firing, and support mechanisms have changed slightly as well.

At the start of a problem, unlike the subtraction models, Able has all the known and unknown variables in its working memory (its top problem solving state). Its problem solving ends when the target variable(s) are known. Also on the top state are the physics principles. Able has eight principles, such as $F = m\ a$ and $X = V_0\ T + 0.5\ a\ T^2$. These equations are represented in a simple way as sets of variables (e.g. F m and a) because Able only models results at a relatively abstract level, such as if F and M are known, then A would be known as well.

Figure 4 shows the operators and their relationships. After a problem has been retrieved with FETCH-PROBLEM, problem solving proceeds with a top-level operator proposing to solve the problem. DEVELOP-KNOWLEDGE will later implement single inference steps that directly solve the problem, but initially, nothing can be done, and an impasse is noted by the architecture. In this impasse, the target variable is selected as the variable to solve. APPLY-PRINCIPLE operators are proposed to apply each of the principles on the state. There is some fairly powerful heuristic knowledge used to select the APPLY-PRINCIPLE operator to use first. Not all problem solvers necessarily have such heuristics, but all of Larkin's subjects appear to have had them. Operators that apply principles with mostly known variables are preferred, but more importantly, operators that propose principles including the target variable are highly preferred. Operators that apply principles with the same number of unknowns and relationship to the target are made equivalent. If the system has available additional domain knowledge about which principles to apply first, it could be used here as well.

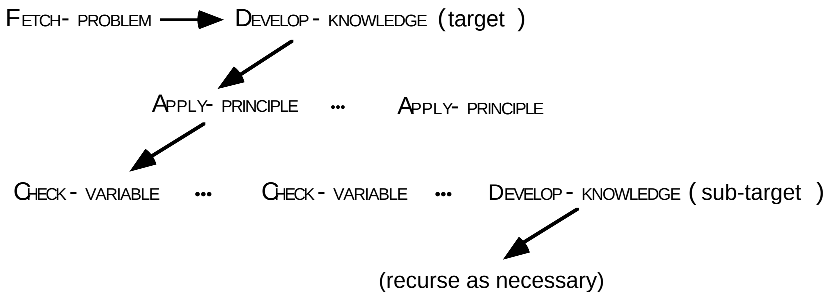


Figure 4. The structure of the operators in Able. Arrows indicate order of application and relationships in the hierarchy. Ellipses (...) indicate that multiple applications of the previous operator may occur.

Although Able may know that it needs to apply a particular principle (using Apply-principle), it may not know initially *how* to apply that principle. However, Able learns how to apply each principle with experience. Another impasse occurs, and lower level CHECK-VARIABLE operators check each of the variables in the principle. If all variables except the target are known, the target can be derived, and this result is passed up to the higher APPLY-PRINCIPLE operator. If variables other than the target are unknown, DEVELOP-KNOWLEDGE is applied recursively, with the unknown variable as a target. This leads to behaviour that is typical of novices in this domain, working backwards from target variables (Larkin, McDermott, Simon, & Simon, 1980a; Larkin et al., 1980b).

During problem solving, new, learned productions (chunks) are created that encapsulate the essential aspects of the impasse and the result that was used to resolve the impasse. These new rules allow APPLY-PRINCIPLE to be applied atomically when similar circumstances occur. With additional problem solving, because the bottom-most operators must be learned first, the derivation of unknown variables from known variables eventually occurs directly with the DEVELOP-KNOWLEDGE operator.

Learning changes how Able solves problems. With enough practice, fully learned behaviour occurs with the DEVELOP-KNOWLEDGE operator solving problems directly through application of the learned rules, working forward, using the known variables to derive additional known variables. The model changes from being driven in a goal-directed way, applying principles to derive the target variable, to being data-driven, where the known variables are used to directly derive additional known variables.

Practice also drastically decreases Able’s problem-solving time. Able III initially takes 27 architectural (decision) cycles to solve a typical problem (number 5) on the first attempt. This time includes time to find the principles to apply, to check each of the variables, and recursively solve for variables where necessary. After practice over 7 trials with the same problem, Able takes just 2 cycles to solve the problem and no longer improves. The learning curve that is generated for a single problem does not even approximately fit the power law of learning (Rosenbloom & Newell, 1987), but it is difficult to comment further, because there are multiple aspects of the task not yet included in the model, the learning curve when computed across multiple problems looks more like typical subject data, and solution times were not reported for the original subjects.

The relative ease with which Able-Soar, Jr. was translated shows that Able was not fundamentally affected by the changes in the Soar architecture in the last five years. While Able's functionality has basically stayed the same (Able-Soar, Jr. solved 13 unique dynamics problems, Able III solves 16), the number of rules has slightly decreased from 51 basic rules (excluding monitoring and problem generating rules) to 48 rules. The rules have not become more complex, however, since the number of clauses and their complexity have both decreased dramatically from 371 to 223 clauses and from 684 to 477 patterns. The differences in these rule sets suggest that the syntax for specifying models in Soar has become simpler without substantially changing the architecture, which is indeed what its architects endeavoured to do (Laird, Huffman, & Portelli, 1990).

One of the valid criticisms suggested by Cooper and Shallice (1995) was that as the Soar architecture was modified, older models must be carried forward for their results to remain valid. This has not typically happened each time the architecture has been released as new software. The Soar community has not been convinced of the need because they understood the changes, and theoretically the changes have nearly always been small with limited impact on existing models. Able is a relatively straightforward model, but the absence of difficulties suggests that the approach Cooper and Shallice put forward to classify changes to an architecture did not correctly classify changes. Many of the changes they noted were changes in the implementation and interface rather then changes in the theory. More complicated models, however, have a greater chance of suffering from changes to the architecture.

There are two model-specific displays included with Able III to help explain how it its behaviour. The first display, shown in Figure 5, describes the problem Able III is working on, including the text of the physics story problem, the target variables, and the current status of all variables (known or unknown). This display currently only works with the physics variables in Able.

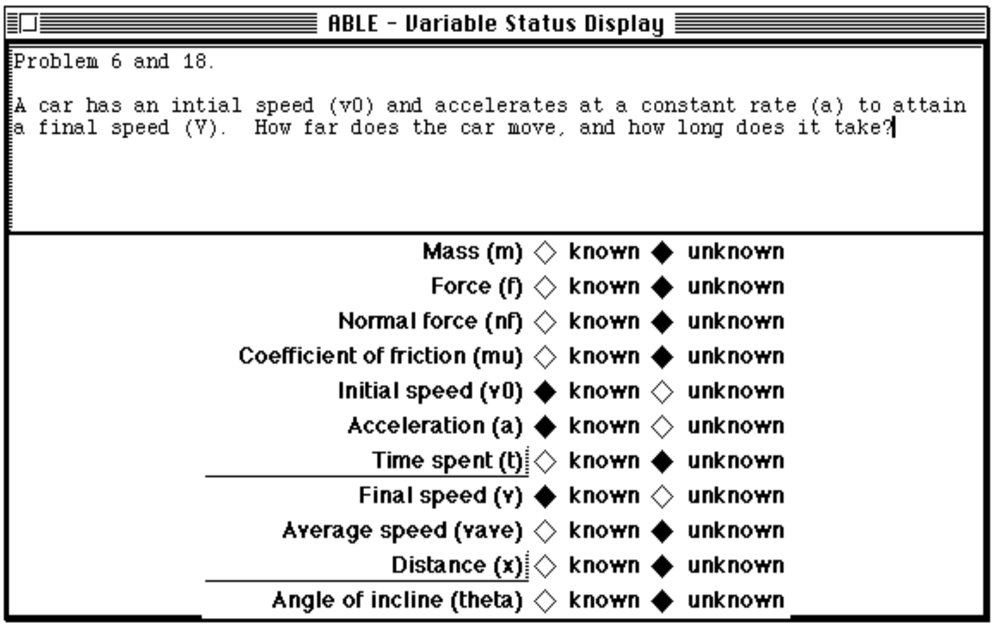
The second display, shown in Figure 6, indicates the order of principle application. It shows that Able III when it is a novice (really an apprentice, since it knows something) works backward from the target variable. The more expert Able III, after it has solved problems and has nearly doubled its number of rules, does not appear to apply principles at all, but works forward, immediately deriving what is known. Because the display is based on the application of principles, it works with any set of principles loaded into Able.

Using principles as high level language

Previous work on Able did not treat its principles and their application mechanism as a high level programming language for cognitive models, but they can be used that way. Such reasoning occurs often enough that the principle application mechanism should be available as a general utility. Based on our work it is now straightforward to add new principles to model another domain.

New problems can be included by representing their features on the top state. New principles are represented one per production rule. The principle application mechanism in Able can then demonstrate how unknown problem variables would be derived through principle application. Additional knowledge ordering principle application can be added, but the weak methods of search in Soar and Able's existing knowledge will otherwise solve the problem if it is solvable.

This principle application mechanism could be used to model novice-expert transitions in other domains, and it provides a way to include routine learning in models. With any set of domain principles, choosing and applying the principles will initially be effortful. With practice, the model's performance will become situation driven and faster. This approach may make it easier to create Soar models by providing a mechanism that more closely resembles the highest conceptual level, the knowledge level (Newell, 1982), and it provides a mechanism for moving from declarative to procedural knowledge.



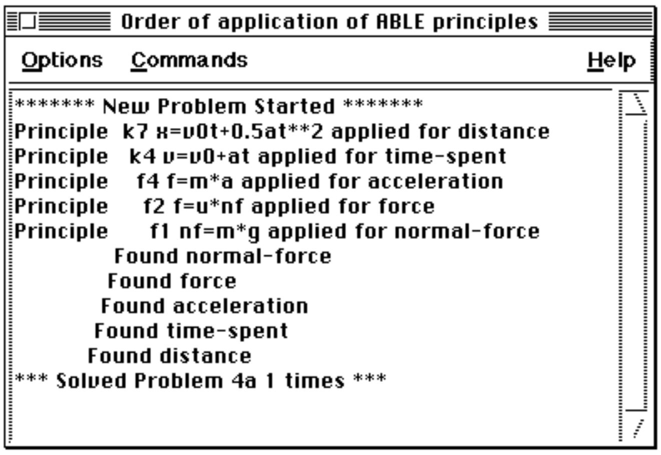
```
Problem 6 and 18.

A car has an intial speed (v0) and accelerates at a constant rate (a) to attain
a final speed (V).  How far does the car move, and how long does it take?
```

Mass (m)	◇	known	◆	unknown
Force (f)	◇	known	◆	unknown
Normal force (nf)	◇	known	◆	unknown
Coefficient of friction (mu)	◇	known	◆	unknown
Initial speed (v0)	◆	known	◇	unknown
Acceleration (a)	◆	known	◇	unknown
Time spent (t)	◇	known	◆	unknown
Final speed (v)	◆	known	◇	unknown
Average speed (vave)	◇	known	◆	unknown
Distance (x)	◇	known	◆	unknown
Angle of incline (theta)	◇	known	◆	unknown

Figure 5. The problem display in Able III, showing the problem (as text in the top pane) and the current status (known/unknown) of the variables. The target variables, Time spent (t) and Distance (x), are in raised text on the screen, which appear here as underlined text.

To test how easy it would be to create a new model, we created and tested in 30 minutes a model that solved a gas physics problem noted as one that should show novice/expert differences (vanSomeren, Barnard, & Sandberg, 1994, p. 14-15). The model consists of three production rules to be added to the existing Able mechanism—a simple model for a simple problem, but it demonstrates that models can be created quickly. Students in a cognitive modelling class at Nottingham as a single week's homework assignment were also able to create models of problem solving in new domains such as electronics and rotational motion.

Difficulties remain with using Able as a utility, however. It was developed to model behaviour in formal domains. Few domains are as formal as physics. The novice-expert transition, which takes well under 100 trials on our problem set, normally takes years of practice. The



```
******* New Problem Started *******
Principle  k7 x=v0t+0.5at**2 applied for distance
Principle   k4 v=v0+at applied for time-spent
Principle    f4 f=m*a applied for acceleration
Principle     f2 f=u*nf applied for force
Principle      f1 nf=m*g applied for normal-force
              Found normal-force
             Found force
            Found acceleration
           Found time-spent
          Found distance
*** Solved Problem 4a 1 times ***
```

Figure 6. The principle application display in Able III shows the order that principles are applied. With practice on this problem, explicit reference to principles disappears.

transition that is modelled—the order in which to apply principles—may be learned this quickly, but the model does not include the gamut of knowledge that makes up an expert. The principle application mechanism is also unrealistic in the way it uses working memory. It keeps the problem and all the principles on the top state, which is not appropriate. These flaws should not be seen as reasons to reject it, but rather clear indications about where it can be improved (Grant, 1962).

Summary

The exemplar models presented here suggest two useful and eventually necessary additions to cognitive modelling. The first addition is that models and architectures should routinely include graphic displays. The general displays presented here will be useful when developing any Soar model because they make the architectural behaviour visibly explicit. The utility of the specific, knowledge level displays suggests that similar displays should be provided for other models. Both types of displays may also provide suggestions of useful displays for other architectures.

The displays have, again, also told us something about Soar. Soar proposes that there are three interesting levels of theoretical interest: the knowledge level, the problem space level, and the symbolic or implementation level (Newell, 1990). Implementing the TSI has emphasised that the problem-space level does not explicitly exist in the code that makes up Soar models—it is an emergent feature arising from production firings. Creating a new interface has emphasised this. There are numerous commands to manipulate productions but far fewer commands to manipulate objects on the problem space level. Creating a graphical interface helps visualise these levels and encourages us to support the higher levels more directly. We have already included a facility to allow users to apply a specified operator. We need to extend the TSI to list objects on the problem space level and how often they have been used, like what is already provided for productions.

The second addition we offer is that these models provide exemplars of the ability to abstract and export fundamental mechanisms for inclusion in other models by working within a cognitive architecture. Here, the principle application mechanism in Able III becomes a utility as a new programming language. This is an important exemplar, for cognitive models as sets of knowledge should be reusable, including their knowledge based mechanisms. The subtraction blackboard and operator implementations provide a similar framework for models of arithmetic. This extends the view of competitive argumentation (VanLehn, Brown, & Greeno, 1984), encouraging one not only to know what gives the model its power, but also to package the mechanism for explanation and reuse. With both models, by providing general mechanisms we have seen that cognitive models can be created more easily and that they can also be extended in interesting ways by relative novices.

We believe that for cognitive modelling to thrive, rather than just survive, more models will have to be developed in the ways we have described. Models must be made easier to understand, easier to extend, and easier to reuse. Packaging models as utilities of high level programming languages with displays within a cognitive architecture provides one way of facilitating this.

Acknowledgements

Initial versions of some of the graphic displays were created by Karl Schwamb. Moritz Baumann, Pete Bibby, Fernand Gobet, and Rolf Ploetzner have provided useful comments. The domain-specific interfaces for the subtraction model were developed with the aid of Peter Wiemer-Hastings. John Laird and Clare Congdon helped design the course assignments for the subtraction model. Support for this work has been provided by the DRA, contract 2024/004, and by the ESRC Centre for Research in Development, Instruction and Training. Any errors remain the fault of the authors, and an acknowledgement here does not constitute an endorsement of this article from those listed. This paper is a revised and extended version of a paper presented at the First European Cognitive Modeling Workshop.

References

Bass, E. J., Baxter, G. D., & Ritter, F. E. (1995). Using cognitive models to control simulations of complex systems. *AISB Quarterly, 93,* 18-25.

Brown, J. S., & VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4,* 379-426.

Congdon, C. B., & Laird, J. E. (1995). *The Soar user's manual, Version 7.* Ann Arbor, MI: Electrical Engineering and Computer Science Department, U. of Michigan.

Cooper, R., & Shallice, T. (1995). Soar and the case for unified theories of cognition. *Cognition, 55,* 115-149.

Elio, R., & Scharf, P. B. (1990). Modeling novice-to-expert shifts in problem-solving strategy and knowledge organization. *Cognitive Science, 14*(4), 579-639.

Grant, D. A. (1962). Testing the null hypothesis and the strategy and tactics of investigating theoretical models. *Psychological Review, 69*(1), 54-61.

John, B. E. (this volume). Cognitive modeling and human computer interaction.

Jones, G., & Ritter, F. E. (1997). Modelling transitions in childrens' development by starting with adults. In *Proceedings of the European Conference on Cognitive Science (ECCS '97).* 62-67. Manchester, UK.

Klein, G. A. (1989). Recognition-primed decisions. In W. B. Rouse (Ed.), *Advances in man-machine systems research (vol. 5).* Greenwich, CT: JAI.

Laird, J., Huffman, S., & Portelli, M. (1990). Status of NNPSCM and S-support. In T. Johnson (Ed.), *Thirteenth Soar Workshop.* 49-51. THE Ohio State University: The Soar Group.

Larkin, J. H. (1981). Enriching formal knowledge: A model for learning to solve textbook physics problems. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Larkin, J. H., McDermott, J., Simon, D. P., & Simon, H. A. (1980a). Expert and novice performance in solving physics problems. Science, 208, 1335-1342.

Larkin, J. H., McDermott, J., Simon, D. P., & Simon, H. A. (1980b). Models of competence in solving physics problems. *Cognitive Science, 4,* 317-345.

Larkin, J. H., & Simon, H. A. (1981). Learning through growth of skill in mental modeling. In H. A. Simon (Ed.), *Models of thought II.* New Haven, CT: Yale University Press.

Larkin, J. H., & Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science, 11*(1), 65-99.

Levy, B. (1991). Able Soar, Jr: A model for learning to solve kinematic problems. Unpublished.

Neches, R., Langley, P., & Klahr, D. (1987). Learning, development, and production systems. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development*.

Newell, A. (1982). The knowledge level. Artificial Intelligence, 18, 87-127.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Nichols, S., & Ritter, F. E. (1995). A theoretically motivated tool for automatically generating command aliases. In *Proceedings of the CHI '95 Conference on Human Factors in Computer Systems*. 393-400. New York, NY: ACM.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Reading, MA: Addison-Wesley.

Ploetzner, R. (1995). The construction of coordination of complementary problem representations in physics. *J. of Artificial Intelligence in Education, 6*(2/3), 203-238.

Ritter, F. E., & Larkin, J. H. (1994). Using process models to summarize sequences of human actions. *Human-Computer Interaction, 9*(3&4), 345-383.

Rosenbloom, P. S., & Newell, A. (1987). Learning by chunking, a production system model of practice. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development.* 221-286. Cambridge, MA: MIT Press.

VanLehn, K. (1983). Human skill acquisition: Theory, model and psychological validation. *Proceedings of the Third National Conference on Artificial Intelligence.* Cambridge, MA: MIT Press.

VanLehn, K. (1989). *Mind bugs: The origins of procedural misconceptions.* Cambridge, MA: MIT Press.

VanLehn, K., Brown, J. S., & Greeno, J. (1984). Competitive argumentation in computational theories of cognition. In W. Kintsch, J. R. Miller, & P. G. Polson (Eds.), *Methods and tactics in cognitive science*. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

VanLehn, K., Jones, R. M., & Chi, M. T. H. (1992). A model of the self-explanation effect. *Journal of the Learning Sciences, 2*, 1-59.

vanSomeren, M. W., Barnard, Y. F., & Sandberg, J. A. C. (1994). *The Think Aloud Method: A practical guide to modelling cognitive processes*. London/San Diego: Academic Press.