An Introduction to the Soar Cognitive Architecture

Frank Ritter
Penn State, USA

Tony Kalus
University of Portsmouth, UK

15 sept 2011
Version control
(not an OHP slide)

These OHPs currently “belong” to FER.
Version 15 updated (twice) for PSU class.
Version 14 for Bamberg, updated code by Kalus
Version 12: Lightly modified by RMY in January ’98, for use at UH. Coupled with extensive revision of HT exercises & code, and use with TSI.
Version 11: Modified by RMY in March ’97, based on Version 10. Used at AISB/ECCS tutorial, Manchester, April ’97. Brought more closely into line with the WWW version of the tutorial, particularly as regards sequencing (which was in bad shape) and NNPSCM content.
Version 10: Used by FER at Berlin, November ’96. Many glitches accumulated by now (sequencing and NNPSCM content), and out of synch with WWW version.
Timetable

09.30  Session 1: Basics, hungry-thirsty example
11.00  coffee
11.15  Session 2: More on hungry-thirsty
12.30  Session 4: Working with Soar, Discussion
13.00  lunch

(not offered at Bamberg, but available online)

14.00  Session 3: Analogy model
16.00  End
Who Are We?

- Frank Ritter
  - Penn State, doing Cognitive modeling
  - Behavioural moderators
  - Tying models plausibly to interfaces
  - Usability (soar-faq, TSI, acs.ist.psu.edu/papers)

- Tony Kalus
  - University of Portsmouth, seconded to DERA to work on intelligent agents.
  - Spent some months with John Laird and the Soar group at the University of Michigan.
What is Soar?

- A production system, albeit a ‘non-standard’ one.
- Can be regarded as:
  - a candidate unified theory of cognition (UTC, Newell, 1990),
  - an architecture for intelligent behaviour.
- Hands-on tutorial, multi-pass
So What Does That Mean?

- It provides a set of principles and constraints based on a theory of cognitive processing.
- Thus provides an integration of:
  - knowledge, planning, reaction, search, and learning
  - within an efficient cognitive architecture.
  (also see Newell, 1990, or Anderson & Lebiere, 1998)
A Brief History of Soar

- First version operational in 1982
  - Written by Allen Newell, John Laird, and Paul Rosenbloom at CMU.
- Versions 1-5 written in Lisp.
- Version 6 written in C.
- Version 7 written in C with Tcl/Tk.
- Version 7.0.4 most commonly used.
- Version 7.2 true multi-platform version.
- Version 8.3/4 latest multi-platform versions.
- Soar 9, 9.1 has improved interfaces and more types of memories
Why Should We Be Interested in Soar?

- Interested in strong equivalence of, cognition, i.e. Outcome and Process
- Fairly interesting Learning mech.
- Some BIG successes in ‘real-life’ applications;
- Proven success for creating intelligent agents;
- Some interesting ‘add-ons’
  - Teamwork, Debrief, Intentions
The Designers’ Aims for Soar

- Work on a full range of tasks;
- Represent and use appropriate forms of knowledge;
- Employ a full range of problem-solving methods;
- Interact with the outside world;
- Learn about the task and its performance.

✓ Support all capabilities of an intelligent agent / serve as a UTC
Levels of the Architecture - Knowledge Level

- An approximation to a knowledge level architecture: Soar can be thought of as an engine for applying *knowledge* to *situations* to yield *behaviour*.

- Soar takes the form of a programmable architecture with theory embedded within it.
Practical 1 - Starting and Running Soar

- Turn to **Practical-1** and follow the directions there.
Levels of the Architecture - Problem Space Level

- All behaviour is seen as occurring in a problem spaces, made up of States (S) and Operators (O or Op).
  (The implementation, however, has changed somewhat from Newell’s 1990 book.)

- Fluent behaviour is a cycle in which an operator is selected, and is applied to the current state to give a new (i.e. modified) current state.

\[ S \xrightarrow{\text{Op}_1} S' \xrightarrow{\text{Op}_2} S'' \]
Problem Space Level (cont)

- Once the situation is set up and running, the main activity is the repeated *selection* and then *application* of one operator after another.

- If something prevents that process from continuing? EG, Soar knows of no operators to apply to *that* state. Or it knows of several, but has no knowledge of how to choose? In such cases, an *impasse occurs*, explained soon.
Some Definitions

- a **goal** - is a desired situation.
- a **state** - is a representation of a problem solving situation.
- a **problem space** - is a set of states and operators for the task.
- an **operator** - transforms the state by some action.
Small Model Running

Taken from the Soar video (1994), acs.ist.psu.edu/papers/soar-mov.mpg
How Does the PS Work?

- Soar is based upon a theory of human problem solving...
  ...in which...
- ...all problem solving activity is formulated as the selection and application of operators to a state, to achieve some goal.
Simple Example: Hungry-thirsty I

- A robot that can perform just two actions, Eat and Drink. Initially it is hungry and thirsty, and its goal is to be not hungry.

- The state has two attributes
  - `hungry`, with possible values `yes` and `no`
  - `thirsty`, also with possible values `yes` and `no`.

- In the initial state the robot is both hungry and thirsty, so we have
  \((<s> ^hungry yes ^thirsty yes)\).
Simple Example: Hungry-thirsty II

- The goal (or desired) state is to achieve \(<s> \ ^{\text{hungry no}}\).

- Operators are named Eat and Drink:
  - Eat can apply to any state with \(^{\text{hungry yes}}\), and yields a new state with \(^{\text{hungry no}}\)
  - Drink can apply to any state with \(^{\text{thirsty yes}}\), and yields a new state with \(^{\text{thirsty no}}\).
  - Each operator is knowledge (in the form of production rules) about:
    - when to propose the operator
    - how to apply (or "implement") the operator
    - when to terminate the operator.
Practical-2: Simple Run

- Turn to the practicals and do Practical 2.
Adding Knowledge to the PS

- In order to act, Soar must have knowledge of that domain (either given to it or learned).

- Domain knowledge can be divided into two categories:
  - (a) basic problem space knowledge: definitions of the state representation, the “legal move” operators, their applicability conditions, and their effects.
  - (b) control knowledge, which gives guidance on choosing what to do, such as heuristics for solving problems in the domain.
Adding Knowledge (cont.)

- Given just the basic knowledge, Soar can proceed to search it. But the search will be "unintelligent" (e.g. random or unguided depth first) — by definition it does not have the extra knowledge needed to do intelligent search.

- Important basic knowledge centres round the operators:
  - when an operator is applicable
  - how to apply it
  - how to tell when it is done.
The Symbol (Programming) Level

- Although we think of Soar with the PSCM, it is realised by encoding knowledge at the more concrete symbol level. (Most of this tutorial is concerned with how to realise the problem space level at the symbol, or programming, level.)

- At the PSCM, each state (or “context”) has one other context slot: the operator.
Levels of the Architecture - Summary

- Thus, Soar is *not* usefully regarded as a “programming language”, eg for “implementing” your favourite psychology theory.

- Soar should not be “programmed”. Rather, “knowledge analysis” of a task, leads to expressing knowledge in terms of PSCM objects through rules, ... and seeing what behaviour emerges.

- Each Soar rule “translated into English” should make sense as a bit of task knowledge.
Soar on a Slide (simple)

Production Memory (LTM)

A & B ⇒ X
C & D ⇒ Y
... ... ⇒ ...

new "results"
give rise to
new productions

conditions
test SDM

actions write
into SDM

Soar's Working Memory (WM)

S1
block
green
impasse
operator: paint

S2
operator
no-change

24
Soar on a Slide (more complex)
The Context Stack

- There can be several problem spaces (i.e. contexts) active at once. Each may lack knowledge needed to continue. Then, Soar sees an impasse and automatically sets up a new context (or substate), whose purpose is to find the missing information and supply it to the context above.

- Each decision cycle ends with some kind of change to the context stack. If the knowledge available (i.e. in the productions) specifies a unique next operator, then that change is made. Otherwise, an impasse arises because the immediately applicable knowledge is insufficient to specify the change.
Operators

- How Soar accomplishes problem solving.
  - Operator selection:
    - operator proposal
    - operator comparison
    - operator selection
  - Operator application
  - (In earlier versions) Operator termination
Operator Selection (1)

- Several cases can arise:
  - a unique choice
    - the operator is selected
  - several operators proposed that are all indifferent
    - a choice is made at random
Operator Selection (2)

- several operators proposed that are *not* all indifferent
- no operators suggested
- Soar cannot proceed and has reached an *impasse*. This is the heart of learning for Soar.
Practical 3: Tracing and Printing

The main preference we’ll meet:

- **acceptable**: an object must have an acceptable to be considered.
- **reject**: the object cannot be considered.
- **better, best**: one object is better than another, or (unary) the object is best.
- **worse, worst**: similarly.
- **indifferent**: one object is indifferent to another, or (unary, and more usually) the object is indifferent.
- **parallel**: the object can be one of multiple values.

The semantics of preferences is — roughly, and in most cases — what you would expect.

You should turn to (Practical-3) and do it now.
Production Rules: Form of Knowledge

- Knowledge is encoded in *production rules*. A rule has *conditions* on its LHS, and *actions* on the RHS: \( C \rightarrow A \).

- Two memories are relevant here:
  - the *production memory* (PM), permanent knowledge in the form of production rules
  - *working memory* (WM), temporary information about the situation being dealt with, as a collection of *elements* (WMEs).
Production Rules: Form of Knowledge II

- The LHSs of rules test WM for patterns of WMEs.

- Soar has no syntactic conflict resolution to decide on a single rule to fire at each cycle. Instead, all productions satisfied fire in parallel.

- The rule firings have the effect of changing WM (as we shall see), so that yet more productions may now have their conditions satisfied. So they fire next, again in parallel. This process of elaboration cycles continues until there are no more productions ready to fire, i.e. quiescence.
What Do Rules Look Like (1) ?

## Propose drink.

\[
\text{sp} \{ \text{ht*propose-op*drink} \\
\text{(state <s> ^problem-space <p> ^thirsty yes)} \\
\text{(p> ^name hungry-thirsty)} \\
\text{-->} \\
\text{(s> ^operator <o> +)} \\
\text{(o> ^name drink +)} \}
\]

\[
\text{sp} \{ \text{ht*propose-op*eat} \\
\text{(state <s> ^problem-space <p> ^hungry-thirsty ^hungry yea)} \\
\text{-->} \\
\text{(s> ^operator <o> )} \\
\text{(o> ^name eat )} \}
\]

IF we are in the hungry-thirsty problem space, AND in the current state we are thirsty

THEN propose an operator to apply to the current state and call this operator “drink”.

```plaintext
sp {ht*propose-op*drink
  (state <s> ^problem-space <p>
    ^thirsty yes)
  (<p> ^name hungry-thirsty)
  -->
  (<s> ^operator <o> +)
  (<o> ^name drink +)}
```

```plaintext
sp {ht*propose-op*eat
  (state <s> ^problem-space <p>
    ^hungry-thirsty
    ^hungry yea)
  -->
  (<s> ^operator <o> )
  (<o> ^name eat )}
```
What Do Rules Look Like (2)

sp { establish-jpg *receiver* all-received *signal
  (state < s > ^operator < o > ^top-state < ts > ^io-input-link < in >
    ^super-state-operator < o 2 > A problem-space-name establish-jpg )
  (o > ^name receiver-confirms ^team-to-coordinate < pg >
    ^self-established-commitment *yes*
    ^object-of-commitment < ooc >) ; if this is what we're committed to
  (ooc > ^name < name >)
  (pg > ^preferred-communication-channel < r > ^communicated < c >)
  (ts > ^unique-name < csa > ^self < self >)
  (r > ^type < ccn > ^value < radio > ^communication-command < command >)
  (pg > ^member-list-member < mem >)
    (mem > ^team-type *yes*)
  (ts > ^self < mem >)
  (pg > ^member-list-leader < mem >)
  (o > ^cannot-obtain-commitment obtained-commitment > < mem >))
...

(write {crlf} |Established JPG |)

(<o2> ^achieved *yes*)
Preferences and Decisions

WM (non-operator slots)

- How rule firings change WM:
  - With parallel rule firings there could be inconsistencies in WM, and faulty knowledge could pre-empt correct knowledge.
  - Rules vote for changes to WM through preferences. Thus, one rule might say that WME-1 is acceptable, another that it is better than WME-2, and a third that it should be rejected.
Preferences and Decisions2 - WM (non-operator slots)

- After each cycle, the preferences are examined by a decision procedure, which makes the actual changes to WM.

- So we have the idea of an *elaboration cycle*, a single round of parallel rule firings, followed by changes to the (non-context) WM:

```
preferences
(Phase)

WM
```

- elaboration cycle
- preference phase (proposal)
- WM phase (insertion)
Memories (1)

- Soar has three memories:
  - Working memory -
    - holds knowledge about the current situation
  - Production memory -
    - holds long-term knowledge in the form of rules
  - Preference memory
Memories (2)

- Preference memory -
  - stores suggestions about changes to working memory.
  - Allows Soar to reason about what it does.
  - If it cannot, then Soar invokes a subgoal and learns about the result.
Preferences and Decision 3 - Operator Slots

- Operator slots are important. To gather the available knowledge, Soar runs a sequence of elaboration cycles, firing rules and making changes to WM (which may trigger further rules) until there are no more rules to fire, i.e. until quiescence is reached. Only then does it look at the preferences for the operator slots.

- A *decision cycle*, consists of elaboration cycles, followed by quiescence, followed by a change to some operator slot (or by the creation of a substate if the preferences don’t uniquely specify a change to the operator slots):
Preferences and Decision 4 - Operator Slots

decision cycle

preference phase  non-context changes  quiescence  context changes, substrstate creation/removal

elaboration cycle
Concrete Representation

- Soar uses attribute-values to represent information. For example,
  - (x35 ^isa block ^colour red ^size large)
- Attributes commonly have a single value. (Theoretical & practical reasons for avoiding multiple values.)
- Operators are state attributes, at most one value:
  - (s13 ^operator o51)
Concrete Representation II

- Symbols like s13, o51, and x35 are generated by Soar, not written by hand. They are *identifiers*, and can be used to link objects in WM together:
  - (x35 ^isa block ^colour red ^size large ^above x47)
  - (x47 ^isa cylinder ^colour green ^below x35)

- The names of rules and objects and almost all attributes are chosen by the programmer. There are very few “reserved words” in Soar: *state*, ^operator, and a few other attribute names we’ll meet later: ^superstate, ^attribute, ^impasse, ^item, and one or two others.
Simple diagram of representation

- For informal discussion, it can be helpful to use a simple diagrammatic depiction of Soar objects in WM.
- The coloured blocks of the previous slide might be shown as:
Practical 4: Eating

- You should turn to **Practical-4** and do it now.
Practical 5: Scratch

- You should turn to *Practical-5* and do it now.
Learning

- Resolving an impasse leads to learning.
- The sole learning mechanism is called *chunking*.
- A chunk is a new production that summarises the processing that was needed to resolve the impasse.
- The chunk will be used in the future to avoid a similar impasse situation.
How It Works (1)
How It Works (2)

Initial state

Selection Problem Space

C
B
A

C
B
A

A
C

B
C

A
B

B
A
Soar, Learning in Action

Add 1+2

Result=3

Op No
Change

Count problem space

New rule:
If op is add 1+2
then result = 3

49
Impasses and Substates

- When Soar encounters an impasse at level-1, it sets up a substate at level-2, which has associated with it its own current state and operators.

- The goal of the 2nd level is to find knowledge sufficient to resolve the higher impasse, allowing processing to resume there.
Impasses and Substates

- The processing at level-2 might itself encounter an impasse, set up a substate at level-3, and so on. So in general we have a stack of such levels, each generated by an impasse in the level above. Each level is referred to as a context (or state), and each context has its own current S and Os.

Example:

Notice that the architecture’s problem solving approach is applied recursively at each level.
Kinds of Impasses

- Soar automatically creates substates to resolve impasses.
  - This is the only way that substates get created.

- Types of impasses: Roughly, for each kind of context slot (O), there can be
  - no candidates for slot $\implies$ a \textit{no change} impasse
  - too many, undifferentiable candidates $\implies$ a \textit{tie} impasse
  - inconsistency among the preferences (A > B and B > A) $\implies$ a \textit{conflict} impasse.
Kinds of Impasses

- The most common (you’re unlikely to meet any others) are concerned with operators:
  - no operators ==> a state no change impasse (SNC) [perhaps better thought of as an operators zero impasse (OZ)]
  - too many operators ==> an operator tie impasse (OT)
  - insufficient knowledge about what to do with the operator ==> an operator no change impasse (ONC)
Resolving Impasses

- Each kind, for straightforward resolution, requires a particular kind of knowledge:
  - An SNC/OZ needs *operator proposal* knowledge.
  - An OT needs *control* knowledge.
  - Interestingly, there are three possible reasons for an ONC
    - (a) Knowledge of how to implement the operator may be lacking, in which case that’s what’s needed.
    - (b) The preconditions of the operator may not be satisfied, which case requires *operator subgoal*ing.
    - (c) The operator may be incompletely specified, and need to be augmented.
  - Note, there are other ways to deal with an impasse, such as rejecting one of the context items that gives rise to it.
Practical 6: Watch an Impasse

- In preparation for creating your first chunk now do **Practical-6**: watch an impasse.
  - Follow up on Exercise 6C.
Soar includes a simple, uniform learning mechanism, called *chunking*. (It is sometimes regarded as an arcane and difficult topic, but it isn’t really.)

Whenever a result is returned from an impasse, a new rule is learned connecting the relevant parts of the pre-impasse situation with the result — next time a sufficiently similar situation occurs, the impasse is avoided.
Chunking 2: The backtrace

IMPASSE

Chunk: A & B & D  ⇒  R

Circles are WMEs or sets of WMEs
Bold circles indicate nodes essential to the resolution
Arrow sets going into a node are rules that fire to add it
Numbered nodes are WMEs in the impasse
Chunking 2: Backtrace described

- Chunks are formed when Soar returns a result to a higher context. The RHS is the result. The LHS are things that have been tested by the linked chain of rule firings leading to the result, the set of things that exist in the higher context ("pre-impasse") on which the result depends.
  - Identifiers are replaced by corresponding variables
  - and certain other changes are made.
Chunking 3: Impasses and Chunks

- Just as each impasse type requires a particular kind of knowledge, so it also gives rise to a characteristic kind of learned rule:
  - SNC/OZ needs *operator proposal* knowledge (and gives rise to an operator proposal chunk).
  - An OT needs *control* knowledge (and gives rise to control chunks).
  - The three kinds of ONC
    - (a) Knowledge of how to implement the operator may be lacking (yielding *operator application* chunks).
    - (b) The preconditions of the operator may not be satisfied, which case requires *operator subgoaling*. (NB Don’t ask about chunking for this case!)
    - (c) The operator may be incompletely specified (yielding *operator modification* chunks).
Practical 7: Watch a Chunk

- Now open and do **Practical-7**: Watch a chunk, where Hungry-Thirsty learns an operator selection chunk.
Problem solving and chunking mechanisms are thus tightly intertwined: chunking depends on the problem solving, and most problem solving would not work without chunking.

Now we are at the point where, if we can model performance on a task, we expect to be able to model learning:
- cf. position in Cognitive Science until just recently

Even when no chunk is actually built (because learning off, or bottom up, or duplicate, or whatever), an internal chunk called a justification is formed:
- It’s needed to get persistence right, e.g. to provide i- or o-support (which we will mention later).
Rule Templates

- Writing models in Soar typically does not proceed from scratch. Typically, new models are built by copying old models and modifying them.
  - The same applies to individual rules.

- There are templates for the common actions in a problem space.
  - state initialisation / impasse recognition
  - state augmentation and problem space proposal
  - for each operator:
    - proposal
    - implementation
    - termination
  - goal recognition / return result
Practical 8: Create a Problem Space

- Do **Practical-8**, which involves creating an operator implementation space for Hungry-Thirsty.
  - Talk through the construction of an operator implementation space.
Basics of Persistence

- WMEs are supported by a kind of TMS (Truth Maintenance System). When a rule fires, it produces preferences. When the conditions become untrue, the rule (instance) is retracted, and the preferences may be retracted too. WMEs supported by those preferences may disappear from WM.

- This issue of persistence is potentially complicated and confusing (and a source of subtle bugs). For now, we just take a simple view.
Basics of Persistence II

Rules that modify the state can be divided into two categories:

(a) *Elaboration* rules that make explicit information that is already implicit in the state. E.g. whenever a block has any ^colour, we would like it also to have (^tinted yes). If it has no ^colour, then we would like it to have (^tinted no). Notice that such rules are *monotonic*: they add to the state, but they do not modify or destroy information already there.

(b) *Operator application* rules that change the state. E.g. when we apply the repaint operator, we change the block from its old colour to the new.
Basics of Persistence III

- Information put into WM by op application rules is *sticky*. We want the information to remain even after the conditions that fired the application rules cease to hold. Preferences for such information are said to have $o$-support ($o$ for *operator*).

- Information put into WM by elaboration rules is *non-sticky*. We want the information to disappear from WM as soon as the conditions it depends on no longer hold. Preferences for such information are said to have $i$-support ($i$ for rule *instantiation*).
Summary - Major Design Principles (1)

- A single framework for all tasks and subtasks (problem spaces);
- A single representation of permanent knowledge (productions);
- A single representation of temporary knowledge (attribute/value pairs);
- A single mechanism for generating goals (subgoaling);
- A single learning mechanism (chunking).
Summary - Major Design Principles (2)

- All decisions are made at run-time, based upon a combination of the current sensory data and any relevant long-term knowledge.
- Decisions are *never* pre-compiled.
Part 3: Discussion and Implications
Psychological Work

- Natural Language Comprehension
- Syllogistic Reasoning
- Concept Acquisition
- Learning and use of Episodic Memory
- Various HCI tasks
- Covert Visual Search
- Abductive Reasoning
- Driving a Car
- ‘Joining up’ Soar and Epic
Soar, Alternative Control Structure

Maintain balance

Maintain height

Op No Change
Soar and Cognition

Mapping between Soar and Cognition

One of the intentions of Soar is that the correspondence between model and psychology is pinned down, not free floating.

Mainly in terms of timescales,

eg elaboration cycle ~ 10 ms
decision cycle ~ 100 ms
per-operator time 50-200 ms
CONSTRAINTS ON A UTC

- behave flexibly
- adaptive (rational, goal-oriented) behaviour
- operate in real time
- rich, complex, detailed environment ...
- symbols and abstractions
- language, both natural and artificial
- learn from environment and experience
- acquire capabilities through development
- live autonomously within a social environment
- self-awareness and a sense of self
- be realisable as a neural system
- arise through evolution
Intelligent Agents

Reflective Level
Impasses and Subgoals

Deliberate Level
Decision Procedure

Reactive Level
Production Memory

Working memory

Input
Output

Faster processing time

More flexible processing
TacAir-Soar (1)

- Used for RWA and FWA (TacAir).
- TacAir has approx 5,500 rules.
- For STOW-97:
  - All US planes flown using TacAir-Soar,
  - Flew all mission types, all plane types,
  - 722 flights over the 48 hour exercise,
  - 99% of missions launched,
  - 94-96% of missions flew without errors.
TacAir-Soar (2)

- Dynamically controlled by humans using speech input.
- Minimal human oversight - agents used autonomous decision-making.
- Ran 4-6 planes per 200MHz Pentium Pro.
- Up to 100 planes ‘in the air’ at once on 20-30 Pentiums.
Current Work (Mainly US)

- Computer Games - UoM
- Modeling social processes
- Learning by Observation
- Pedagogic Agents - Immersive Training
- Adding Emotions
- Electronic Elves Project - USC
- Development Tools
- Making Soar articulate/High level language
Practicalities of Working with Soar (1)

- Expressing a problem in Soar
  - Problem space computational model
- Centrality of learning
  - Provides additional constraint
  - Where the stuff in the problem spaces comes from
- Approximation via successive levels of chunking
  - Create top level behavior
  - What behavior and learning could give rise to this?
  - Recurse
- Listening to the architecture
Practicalities (2)

- Soar comes with access to Java
  - makes adding user-written functions easier;
  - makes debugging a bit easier;
  - makes for rapid prototyping of GUIs.

**BUT**

- Let’s you do things you shouldn’t!
Practicalities (3)

- **Learning Soar**
  - Now has two tutorials and re-written manual.
  - Soar workshop in early Summer

- **Tools are available** – VisualSoar (editor), Debugger, etc.

- **Mailing lists for announcements, help, etc.**, available via UoM.

- **Soar-FAQ** acs.ist.psu.edu/soar-faq/

- acs.ist.psu.edu/papers , passwd is: xxx
Coal

Learning has some problems:

- overgeneralisation of chunks,
- expensive chunks.

- Multiple goals not naturally supported.
- Non-symbolic I/O not naturally supported.
Nuggets

- It’s Free!
- Soar works in ‘real’ world
  - TacAir, IDSS, NavySoar
- Soar is still proving ‘reliable’ on the psychological front.
- Soar remains useful and still a front runner for a unified theory of cognition, certainly the exemplar.
The End
Further Readings


[See also the more extensive bibliography, slanted towards Soar-and-psych, at http://phoenix.herts.ac.uk/~rmy/cogarch.seminar/soar.html]