Introduction to SOAR

Amit Prakash Tiwary
SILP Lab

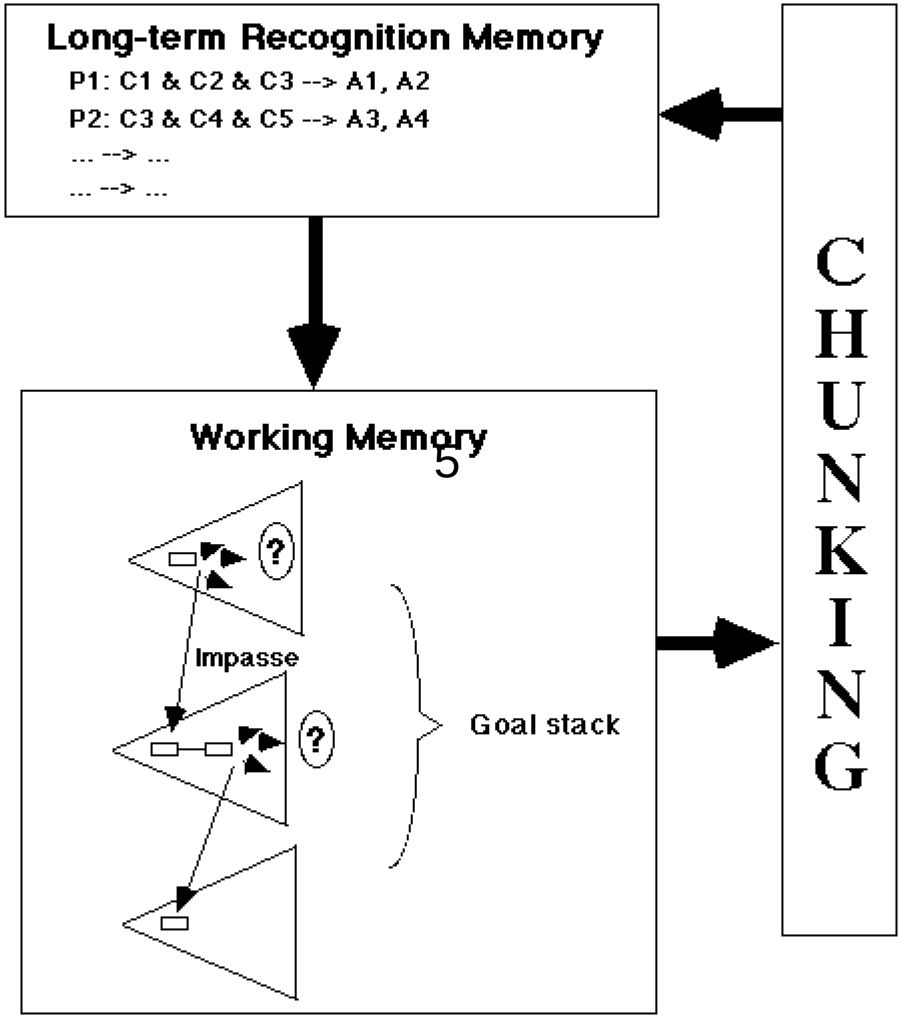
Soar as a candidate Unified Theory of Cognition (UTC)

- For coherence in theorising: "It is one mind that minds it all".
- For bringing to bear multiple constraints from empirical data.
- For reducing the theoretical degrees of freedom.

Soar as a Theoretically Constrained Cognitive Architecture

- As 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.
- Hence, Soar is not usefully regarded as a "programming language", for example, for "implementing" your favourite psychological theory.
- Soar is not "programmed" in the usual way. Rather, we want to think of the modeller as performing a "knowledge analysis" of a task, expressing the knowledge in terms of Soar objects and rules, giving those rules to Soar...and seeing what behaviour emerges.
- Each individual Soar rule, if "translated into English", should make sense as a piece of task knowledge the agent would plausibly have.

The Soar architecture


Long-term Recognition Memory
P1: C1 & C2 & C3 --> A1, A2
P2: C3 & C4 & C5 --> A3, A4
... --> ...
... --> ...
Working Memory
Impasse
Goal stack
CHUNKING

Soar as a problem space architecture

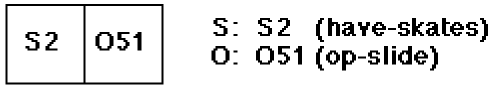
- In Soar, all behaviour is seen as occuring in a problem space, made up of Goals (G), Problem Spaces (P or PS), States (S) and Operators (O or Op). In earlier versions of Soar these were all explicit choices. In the current NNPSCM Soar (Soar6 or Soar7), the Goal and Problem Space are treated as part of the State. Because of this, the State -- especially when thought of as containing the Goal and Problem Space -- is sometimes referred to as a "Context". In this tutorial, we will continue to use the term "Problem Space" sometimes in a deliberately loose way, to mean the Context.
- There can be several problem spaces (i.e., contexts) active at any one time. Each may lack some required knowledge, and be able to provide knowledge to other contexts. The main idea behind splitting knowledge into problem spaces (contexts) is that it reduces the search for information. The idea has also been used as a software design technique because it is a successful way to partition knowledge.
- Within the context there is a current state, that is, the state itself. The current state specifies the position we are currently at. For example, in a blocks world, the state might consist of "block A is on top of block B, and block B is on the table".
- Problem spaces, as such, appear as attributes of a state. This means that various different states may all have the same problem space as an attribute.
- Fluent behaviour consists of a repeated cycle in which an operator is selected, and is applied to the current state to give a new (i.e., modified) current state. This new state will typically be associated with the same problem space as the old state. So in our previous example, we could have applied an operator to move block A to the table, in which case the current state would be that block A and block B are both on the table.
- So, once the situation is set up and running, the main activity of Soar consists of the repeated selection and then application of one operator after another, each application yielding a new state.
- But what happens if something prevents that process from continuing smoothly? For example, perhaps Soar knows of no operators to apply to that state. Or it knows of several, but has no knowledge of how to choose between them? In such cases, Soar encounters an impasse. We will have more to say about impasses later. (If you want to find out about them now, click here and then come back here to continue the tutorial.)

Knowledge in Soar

- No knowledge => No behaviour.
- In order to act in a domain, Soar must have knowledge of that domain (either given to it or learned).
- It's useful to divide domain knowledge into two categories:
- Basic problem space knowledge: definitions of the state representation, the "legal move" operators, their applicability conditions and their effects.
- Control knowledge, which gives guidance on choosing what to do, such as heuristics for solving problems in the domain.
- Given just the basic problem space knowledge, Soar can proceed to search using it. But the search will be "unintelligent" (e.g., random or unguided depth first), since by definition it does not have the extra knowledge needed to do intelligent search.
- Important basic problem space knowledge centres round the operators: when an operator is applicable, how to apply it, and how to tell when it is done.

Soar at the symbol (programming) level

- Although we think of Soar as operating conceptually at the problem space level, its behavior is realised by encoding knowledge at the more concrete symbol level. In this tutorial we will mainly concentrate on how to realise the problem space level at the symbol, or programming, level.
- At the symbol level, each context has two context slots, one for the state and one for the operator. Note that the problem space associated with the state is represented as an attribute of that state, rather than having an explicit context slot of its own (as was the case in earlier versions of Soar).



Knowledge as production rules

- Knowledge in Soar is encoded in production rules. A rule has conditions on its Left Hand Side (LHS), and actions on the Right Hand Side (RHS):
- C --> A.
- Two of Soar's memories are of relevance here: the production memory (PM) or long-term memory, permanent knowledge in the form of production rules; and the working memory (WM), temporary information about the situation being dealt with, as a collection of elements (WMEs).
- The LHSs of productions test WM for particular patterns of WMEs. Unlike most other production systems, Soar has no syntactic conflict resolution to decide on a single rule to fire at each cycle. Instead, all productions whose conditions are satisfied fire in parallel.
- For example, the following rule proposes an operator 'eat' if we are hungry and desire not to be hungry.

```
    ;; Propose eat.
     sp {ht*propose-op*eat
     (state <s> ^problem-space <p> ^desired <d>)
       (<p> ^name hungry-thirsty)
   (<d> ^hungry no)
     (<s> ^hungry yes)
     -->
      (<s> ^operator <o>)
     (<o> ^name eat)}
```

- Translated into 'English', this rule would be:

```
  If   we are in the hungry-thirsty problem space  AND
    we desire to be not hungry  AND
    the current state says we are hungry
then propose an operator to apply to the current state  AND   call this operator 'eat'
```

- The rule firings have the effect of adding elements to 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. That is quiescence.

Representation of working memory elements

- Soar uses an attribute-value scheme to represent information in WM. Thus a conceptual object that stands for a large red block might be represented as:
-     (<x> ^isa block ^colour red ^size large)
- All attributes are marked by a ^ (called an up-arrow or caret) and their values directly follow them. Note that the attribute-values do not have to be in any specific order, so we could have put ^isa block last in the list. Attributes can be added to working memory by the application of rules whose conditions have been satisfied.
- Attributes commonly have just a single value, though they can have multiple values. However, there are both theoretical and practical reasons for avoiding multiple values whenever possible.
- The operator, which is a context slot, is represented as an attribute of a state symbol, though as we shall see (when we come to talk about the decision cycle) the slot gets its value only after quiescence is reached, i.e. when there are no more productions to fire. The context slots (state and operator) always have at most one value.
- Within a production rule, symbols like <x> are variables, and so can be used to link one object to another. For example, in the clauses below, <y> is used to reference one object from another:
-     (<x> ^isa block ^colour red ^size large ^above <y>)
-     (<y> ^isa cylinder ^colour green ^below <x>)
- In Soar, each of these variables will be bound to a unique, internally generated identifier (ID) such as x35, which represents the object, though we need not concern ourselves with this since we primarily reference objects through variables such as <s>. However, the ID's appear in the trace (which we will come to later), and can also be used to print objects for inspection.
- The variables used for the attribute values do not necessarily have to take the initial letter of the attribute name (such as <o> for the ^operator value), but it will make your productions easier to understand, and hence you should take some care in naming them.

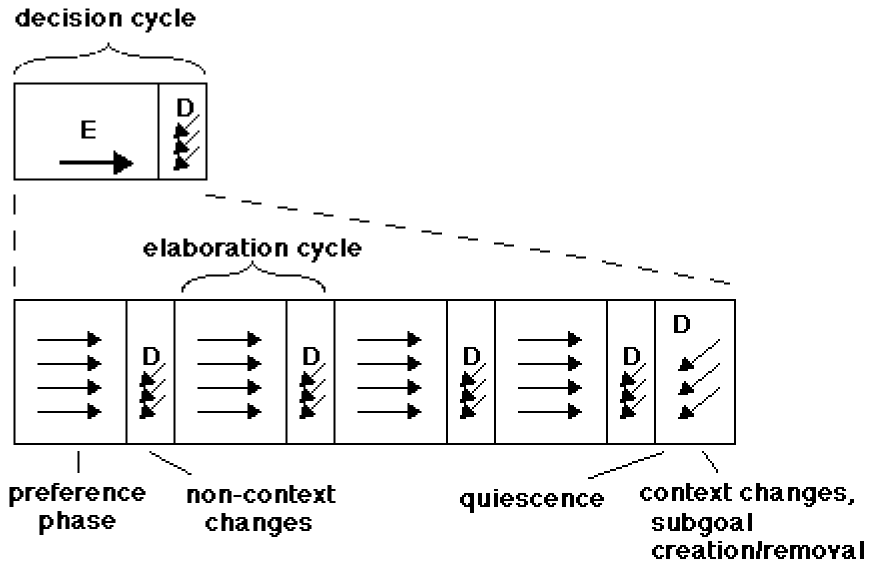
Preferences and decisions

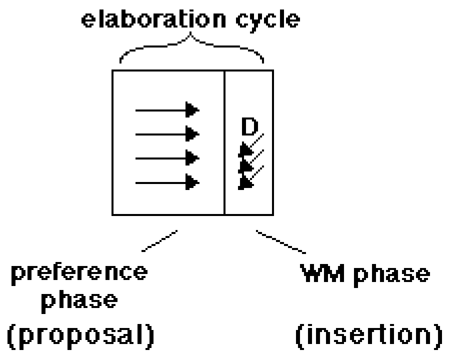
- We need to look a little more closely at the way rule firings change Working Memory (WM).
- First, we look at changes to "ordinary" WM Elements (WME's), that is, ones other than the context slots (state and operator).
- With parallel rule firings, it is important that rules are not able to change WM directly, else there could be inconsistencies in WM and faulty knowledge could pre-empt correct knowledge.
- So, in Soar, the RHS's of rules do not make changes directly to WM. Instead, they vote for changes to WM by producing preferences.
- After each cycle, all the preferences are examined by a decision procedure that makes the actual changes to WM.
- This means we have the idea of an elaboration cycle, which is a single round of parallel rule firings, followed by changes to the (non-context) WM.

Context Slots

The most important changes are those to one of the context slots (i.e. S or O). To gather all 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 context slots.

So we have the idea of a decision cycle (not to be confused with the decision procedure mentioned above), consisting of a number of elaboration cycles, followed by quiescence, followed by a change to some context slot (or by the creation of a subgoal if the preferences don't uniquely specify a change to the context slots):





- Decisions for ordinary WM changes are made at the end of each elaboration cycle.
- Decisions for context slot changes are made only after quiescence, which is at the end of each decision cycle.
- Soar runs by executing a sequence of decision cycles or elaboration cycles (depending on what command it was given) until told to stop or the number of cycles specified in the command.
- We can see how elaboration cycles and decision cycles take place from the example below using run:
-

```
     Soar> run 1
        0: ==>S: S1
  Firing default*top-goal*elaborate*state*name*top-goal*top-state
  Firing ht*propose-space*ht
   Soar> run 1
   Firing ht*propose-op*drink
   Firing ht*propose-op*eat
   Soar> run 1
   Firing ht*compare*eat*better*drink
  Soar> run 1
        1:    O: O2 (name: eat)
   Soar>
```

- This is a run of the hungry-thirsty model, where we are watching rule firings only. Note that 'r 1' tells Soar to run for one elaboration cycle, which as we can see may complete a decision cycle if quiescence is reached. The decision cycle is denoted by the number alongside it, so we can see that the first decision cycle ('0') was to select a state (with no name associated with it) for the context slot. After this, there are elaboration cycles to vote for which operator should be selected for the empty operator context slot. The second decision cycle ('1') then selects the 'eat' operator to fill the operator context slot, which is about to be applied to the state so that it becomes modified. Such processing will continue until the goal is satisfied, or until every possible manipulation has been made without success, at which point Soar will halt having failed to satisfy the goal.

Rule Syntax

- We now go down yet another level and examine the syntax of the rules.
- Consider a rule for the following statement:
- "In the blocks world, if one block is on top of another block of a different colour, then propose repainting the lower block to be the same colour as the upper block".
- When translated into a Soar model, it looks like this:

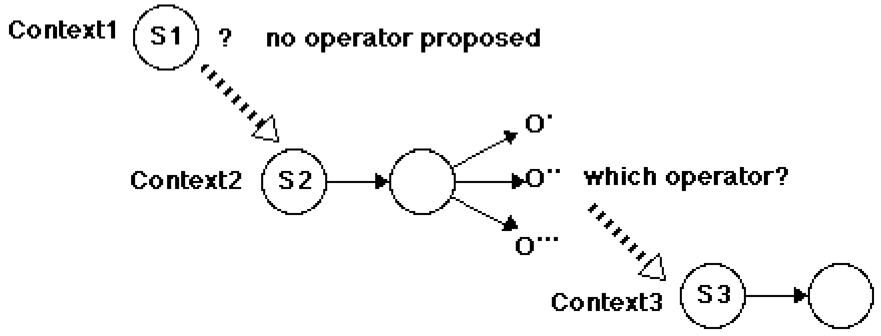
```
   sp {blocks*paint*propose
      (state <s> ^problem-space <p> ^desired <d>)
    (<p> ^name blocks-world)       (<s> ^block <block1> <block2>)
      (<block1> ^isa block ^colour <col1> ^above <block2>)       (<block2> ^isa block ^colour
        <>
```

- <col1>)   -->
- (<s> ^operator <o>)
- (<o> ^name repaint ^object <block2> ^colour <col1>)}
- Many of the basic rule features are shown in this rule:
- sp ("soar production"), followed by the production name.
- Attribute-value pairs, such as ^name blocks-world.
- Conditions of the rule, which appear before the -->, and the actions appearing after. Note that each condition and each action is enclosed in a set of parentheses. For the rule to fire, each condition must be satisfied.
- Variables, such as <s> and <block1>, which denote the state we are in and a block respectively.
- Logic features, such as <> which denotes 'not equal'; that is, the colour of <block2> must not be the same as <block1> for this rule to fire. There are other logic features in Soar, such as:
- ^colour << red blue >> denotes that the colour attribute should have a value of red or blue.
- -^colour red denotes that the condition is that there is not a colour attribute with a value of red.
- Further features exist which may be noted in the code for the exercises.
- Sometimes the syntax appears to get in the way by being verbose, but with practice rules become easier to read.
- Shorthand syntax of rules
- There are shorthands for several common constructs. We can just give path names to access local structures. For example, using some of the shorthand available, the above rule begins:

```
    (state <s> ^problem-space.name blocks-world
             ^block (<block1> ^isa block ...)
                    (<block2> ^isa block ...))
```

Impasses and sub-goals

- When Soar encounters an impasse in context level-1, it sets up a subcontext (or "subgoal") at level-2, which has associated with it a new state, with its own problem space and operators. Note that the operators at level-2 could well depend upon the context at level-1.
- The goal of the 2nd level context is to find knowledge sufficient to resolve the higher impasse, allowing processing to resume there. For example, we may not have been able to choose between between two operators, so the level-2 sub-goal may simply try one operator to see if it solves the problem, and if not, tries the other operator.
- The processing at level-2 might itself encounter an impasse, set up a subgoal 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 goal), and each context can have its own state, problem space and operators.
- For example:



Kinds of impasse

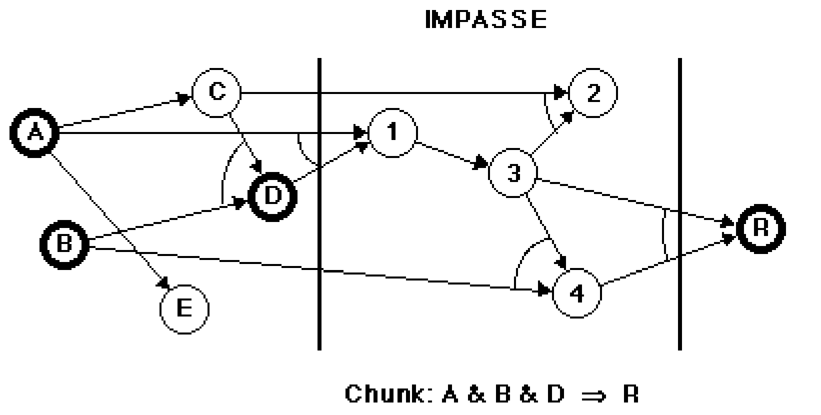
- Soar automatically creates subgoals in order to resolve impasses. This is the only way that subgoals get created. (Note, therefore, that production rules never vote directly for states. The only context slot rules vote for is for operators.)
- What types of impasses are there? Roughly, for the operator slot, there can be:
- No candidates for the slot ==> a no change impasse.
- Too many, undifferentiable candidates ==> a tie impasse.
- Inconsistency among the preferences (e.g. A > B and B > A) ==> a conflict impasse.
- The most common kinds (you're unlikely to meet any others) are:
- 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 of impasse, for its straightforward resolution, requires a particular kind of knowledge:
- An SNC/OZ needs operator proposal knowledge, which is sought by producing sub-goals to find or construct an operator.
- An OT needs control knowledge, which may be found by selecting each tied operator until one which resolves the sub-goal is applied.
- Interestingly, there are three possible reasons for an ONC:
- Knowledge of how to implement the operator may be lacking, in which case that's what's needed.
- The preconditions of the operator may not be satisfied, whereby a state must be found whereby the operator can apply (a case which requires operator subgoaling).
- The operator may be incompletely specified, and need to be augmented.
- It should be noted that there are other ways to deal with impasses, such as rejecting one of the context items that gives rise to it.

Chunking

- Soar includes a simple, uniform learning mechanism, called chunking. 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. This means that next time a sufficiently similar situation occurs, the impasse is avoided.
- The above diagram shows the WMEs that are created during an impasse, the resolution being the addition of the WME 'R', which solves the sub-goal. An arc between arrows denotes 'AND', so for WME 2 to be created, WME's C and 3 must exist. Notice that R is dependent upon the existence of WME's 3 and 4, so if we work our way back, we find that these elements require A, B and D only. This is why the chunk created will have these in its condition, and R in its action. You may think that C should also be included, since it is required in order for D to be added. However, D already exists at the time the impasse occurs, so it is not necessary to include C.
- 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).



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

Chunks and types of impasse

- Just as each kind of impasse, for its straightforward resolution, requires a particular kind of knowledge, so also it gives rise to a characteristic kind of chunk:
- A SNC/OZ resolved by operator proposal knowledge gives rise to an operator proposal chunk.
- An OT resolved by control knowledge gives rise to a control chunk.
- For the three varieties of Operator No Change:
- An ONC resolved by knowledge about how to implement the operator gives rise to operator application chunks.
- An ONC requiring operator subgoaling gives rise to ... don't ask about this case, at least until after you've finished the tutorial!
- An ONC resolved by fleshing out the details of the operator gives rise to operator modification chunks.
- Implications
- 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 in Soar, we expect to be able to model learning (cf. position in Cognitive Science until just recently).
- Even when no chunk is actually built, an internal chunk called a justification is formed. This can happen because learning is turned off, or bottom up (a learning state that only learns from the bottom problem space), or because the chunk is a duplicate of one that already exists, or whatever.

Rule templates

- Writing models in Soar typically does not proceed from scratch. Typically new models are built by copying old models and modifying them. There are also templates for the common actions in a problem space:
- State initialisation.
- State augmentation and problem space proposal.
- For each operator:
- Proposal. This is so that preferences for operators can be put forward.
- Implementation. Once an operator is selected, this will fire so that the operator can modify the state in some way. For example, we may eat if the eat operator was selected in the decision cycle.
- Termination. This tells Soar to reconsider what should be in the operator slot, and therefore means that another decision cycle to fill the context slot is required.
- Return result.
- Goal recognition.

The persistence of knowledge

- Information in WM is supported by a kind of Truth Maintenance System (TMS). When the conditions of a rule are satisfied, it fires, and produces various 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.
- Rules that modify the state can be divided into two categories:
- Elaboration rules, which make explicit information that is already implicit in the state. For example, whenever a block has any ^colour, we might 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.
- Operator application rules, which change the state from one configuration to another. For example, when we apply the repaint operator, we change the block from its old colour to the new.
- 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).
- Information put into WM by operator application rules is sticky . We want the information to remain even after the conditions that fired the application rules have ceased to hold. Preferences for such information are said to have o-support (o for operator).

Mapping between Soar and cognition

- One of the strengths of Soar is that the correspondence between the model and psychology is pinned down, not free floating. Newell explains this in detail in his Unified Theories of Cognition (1990) book. One of the most straightforward ways of achieving this correspondence is in terms of timescales, for example:
- Elaboration cycle ~~10ms, where ~~ means "within a factor of about 3".
- Decision cycle ~~100ms.
- Per-operator time 50-200ms.
- Constraints on a unified theory of cognition
- Newell explains this topic in detail in his Unified Theories of Cognition (1990) book. A shortened list is provided here.
- 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.