Soar Technology
Thinking *inside* the box.

Comparing Modeling Idioms in ACT-R and Soar

Randolph M. Jones
rjones@soartech.com

Thanks to Christian Lebiere, Jacob Crossman, Robert Wray

Architectural Constraints on Cognition

- Make it easier to build “correct” cognitive models
- Make it harder to build “incorrect” cognitive models
- Behavior patterns that are difficult to code are assumed to be difficult for a good reason
- Commonly occurring behavior patterns lead to programming patterns or idioms
- Many idioms are relatively directly tied to architectural constraints
- When taken seriously, idioms can significantly alter a model’s predictions (timing, error rates and types, task representation, etc.)

Basic Constraints in ACT-R and Soar

- ACT-R
  - Cognitive constraint
    - One production instantiation fires at a time
  - Information constraints
    - One chunk at a time per architectural module/buffer available to be matched by productions
    - One goal buffer
    - One retrieval buffer
    - Declarative memory only accessible through buffers
- Soar
  - Cognitive constraint
    - One operator selection (per state) at a time
    - All operator selections go through the preference/selection mechanisms
  - Information constraints
    - Maintain internal logical consistency
    - Operator objects automatically deselect when preconditions unmatch
    - Direct working-memory changes restricted to “accept” and “reject”

Soar idiom: change-value

```
sp {operator*apply*change-value*reject-old-values
   (state <s> ^operator <o>
              ^change-value <cv>)
   (<cv> ^id <id>
         ^att <att>
         ^value <value>)
   (<id> ^<att> { <ov> <> <value> })
-->
   (<id> ^<att> <ov> -)
}
```

```
sp {operator*apply*change-value*assert-new-value
   (state <s> ^operator <o>
              ^change-value <cv>)
   (<cv> ^id <id>
         ^att <att>
         ^value <value>)
  -(<id> ^<att>)
-->
   (<id> ^<att> <value>)
}
```

```
sp {operator*apply*change-value*reject
   (state <s> ^operator <o>
              ^change-value <cv>)
   (<cv> ^id <id>
         ^att <att>
         ^value <value>)
   (<id> ^<att> <value>)
  -(<id> ^<att> { <ov> <> <value> })
-->
   (<s> ^change-value <cv> -)
}
```

ACT-R Idiom: Grouping conceptual actions

- Only one production fires at a time, so operate on multiple buffers at once

```
(p find-next-tower
   =goal>
      isa move-tower
      disk =disk
      peg =peg
      state nil
==>
   !output! "Retrieving disk
   smaller than ~S" =disk
   +retrieval>
      isa next-smallest-disk
      disk =disk
   =goal>
      state next)
```

Soar idiom: Modular parallel operator applications

- Multiple rules can fire in parallel, so prefer teasing apart each action with its own conditions, so they can compose if and only if necessary

```
sp {find-next-tower*apply*retrieve
   (state <s> ^operator <o>
            ^current-goal <g>
            ^next-smallest-disk <nsd>)
   (<o> ^name find-next-tower
        ^goal <g> ^disk <disk>)
   (<nsd> ^disk <disk>)
  -(<s> ^current-retrieval <nsd>)
   (<disk> ^name <dname>)
-->
   (write (crlf) |Retrieving disk 
smaller than | <dname>)
   (<s> ^change-value <cv>)
   (<cv> ^id <s> ^att current-retrieval
         ^value <nsd>)}
```

```
sp {find-next-tower*apply*change-state
   (state <s> ^operator <o>
        ^current-goal <g>
        ^next-smallest-disk <nsd>)
   (<o> ^name find-next-tower
        ^goal <g> ^disk <disk>)
   (<nsd> ^disk <disk>)
   (<s> ^current-retrieval <nsd>)
  -(<g> ^state next)
-->
   (write (crlf) |Moving to state 
"next"|)
   (<s> ^change-value <cv>)
   (<cv> ^id <g> ^att state
         ^value next)}
```

Soar Idiom: Serialization to avoid race conditions

- In examples like the previous Soar snippet, race conditions can arise
  - What if the “change-value” pattern takes longer to complete for one of the parallel threads than for the other?
  - The first one that completes may cause the operator to deselect
  - To avoid such a race condition, the “safe” approach is to force the actions to be implemented serially, each with its own operator

ACT-R Idiom: Query-Harvest rules for declarative-memory retrieval

- Because rules must match declarative memory through the retrieval buffer, every chunk the needs to be tested must first be fetched into the buffer

```
(p find-spare-peg
  =goal>
   isa clear-disk
   disk =disk current =on
   peg =peg state nil
  =retrieval>
   isa next-smallest-disk disk =disk next =next
==>
  !output! "Next smaller disk to ~S is ~S and
retrieving peg other than ~s and ~S" =disk =next
=on =peg
  =goal>
   disk =next state other
  +retrieval>
   isa spare-peg
   current =on destination =peg)
```

```
(p clear-tower
  =goal>
   isa clear-disk disk =disk
   current =on peg =peg
   state other parent =parent
  =retrieval>
   isa spare-peg current =on
   destination =peg other =other
==>
  !output! "Subgoaling move-tower with disk ~S
peg ~S parent ~S" =disk =peg =parent
  +goal>
   isa move-tower
   disk =disk
   peg =other
   parent =parent)
```

Soar Idiom: Simultaneous Query-Harvest

- Because Soar does not force matches to funnel through a retrieval buffer, a “retrieval” step is usually unnecessary. Simply match against the information that is already in declarative memory (sometimes implies large working-memory sets)

```
sp {clear-disk*propose*create-subgoal*move-tower
   (state <s> ^current-goal <g> ^disk <disk>
              ^next-smallest-disk <nsd>
              ^spare-peg <sp>)
   (<g> ^name clear-disk ^disk <disk>
        ^current <on> ^peg <peg> ^parent <parent>)
   (<nsd> ^disk <disk> ^next <next>)
   (<sp> ^current <on> ^destination <peg>
         ^other <other>)
   (<next> ^name <dname>)
   (<peg> ^name <pname>)
   (<other> ^name <oname>)
-->
   (write (crlf) |Create new subgoal move-tower
   disk | <dname> | to peg | <oname> | to replace
   clear-disk from peg | <pname>)
   (<s> ^operator <o>)
   (<o> ^name create-subgoal ^goal <ng>)
   (<ng> ^name move-tower ^disk <next> ^peg <other>
         ^parent <parent> ^clear-parent *yes*)}
```

Soar Idiom: Preference-based partial ordering

- Soar’s preference mechanism can use subsets of operator-relevance conditions to produce partial-ordering constraints. This can allow economical representation of fairly complex choices

```
sp {eat*propose
   (state <s> ^agent <a>)
   (<a> ^hungry yes)
-->
   (<s> ^operator <o> + =)
   (<o> ^name eat ^agent <a>)}

sp {drink*propose
   (state <s> ^agent <a>)
   (<a> ^thirsty yes)
-->
   (<s> ^operator <o> + =)
   (<o> ^name drink ^agent <a>)}
```

```
sp {prefer*eat*over*drink
(state <s> ^operator <o1> + <o2> +)
(<o1> ^name eat)
(<o2> ^name drink)
-->
(<s> ^operator <o1> > <o2>)}
```

ACT-R Idiom: Exhaustive enumeration of conjunctive conditions

- ACT-R does not have a Soar-like preference mechanism for creating contextual partial orderings. One approach is to enumerate all the conjunctive conditions represented by different possible orderings and test for all of them (through a serial sequence of retrievals).

```
(p check-hungry
   =goal>
      isa agent
      name =name
      state nil
==>
   +retrieval>
      isa property
      agent =name
      attribute hungry
      value yes
   =goal>
      state hungry)
```

```
(p check-thirsty
   =goal>
      isa agent
      name =name
      state hungry
   =retrieval>
      isa error
==>
   +retrieval>
      isa property
      agent =name
      attribute thirsty
      value yes
   =goal>
      state thirsty)
```

ACT-R Idiom: Ordering via partial matching

- An alternative method to ordering choices in ACT-R is to use the similarity-based partial-matching feature of the ACT-R retrieval module. This requires specifying a similarity (or dissimilarity) measure for the attributes and values that are relevant to the ordering constraints.

```
(setsimilarities (hungry thirsty -0.5))

(p choose-action
   =goal>
      isa agent
      name =name
      state nil
==>
   +retrieval>
      isa property
      agent =name
      attribute hungry
      value yes
   =goal>
      state unknown)
```

Soar Idiom: Exhaustive parallel processing of similar items

- Because Soar does not limit access to declarative memory and allows multiple rules to fire in parallel, some types of exhaustive processing are easy to do all at once

```
sp {handle-messages*apply
   (state <s> ^operator <o> ^message <m>)
   (<o> ^name handle-messages)
   (<m> ^text <t> ^message-handled false)
-->
   (write (crlf) | Message is: | <t>)
   (<m> ^message-handled false - true +)}
```

ACT-R Idiom: Exhaustive serial processing with Query-Harvest

- ACT-R limits access to declarative memory to one chunk at a time, each of which must be retrieved before it can be processed, and only one rule instantiation can fire at a time. For exhaustive processing of similar items, this leads to a series of query-harvest rules plus a rule to check when the process is done.

```
(p find-message-to-handle
  =goal>
   isa handle-message state nil
==>
  =goal>
   state harvest
  +retrieval>
   isa message handled false)

(p handle-message
  =goal>
   isa handle-message state harvest
  =retrieval>
   isa message text =text handled false
==>
  !output! "~S" =text
  =goal>
   state nil
  =retrieval>
   handled true)
```

```
(p finish-handle-message
  =goal>
   isa handle-message
   state harvest
  =retrieval>
   isa ERROR
   condition Failure
==>
  !output! "Done handling messages"
  =goal>
   state finished)
```

Conclusions

- Nuggets
  - Architectural constraints can have both subtle and significant implications for the details of a cognitive model
  - The modeling communities have developed a variety of idioms for common “behavior units”
  - Comparing these idioms gives some insights into the differences, strengths, and weaknesses of each architecture
- Lumps
  - Some of the architectural constraints are of questionable theoretical value, but still can have significant impact on the types of idioms that must be used, and hence the types of data models will produce
  - A “more complete” cognitive architecture would probably combine some of the constraints from ACT-R and Soar