In Proceedings of ICCM - 2007- Eighth International Conference on Cognitive Modeling. 49 - 54. Oxford, UK: Taylor & Francis/Psychology Press.

Comparing Modeling Idioms in ACT-R and Soar

Randolph M. Jones (rjones@soartech.com)
Soar Technology, 44 Burleigh Street
Waterville, ME 04901 USA

Christian Lebiere (cl@cmu.edu)
Psychology Department, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213 USA

Jacob A. Crossman (jcrossman@soartech.com)
Soar Technology, 3600 Green Court Suite 600
Ann Arbor, MI 48105 USA

Abstract

This paper examines some of the constraints on cognition assumed and imposed by the ACT-R and Soar cognitive architectures. In particular, we study how these constraints either encourage or require particular types of "modeling idioms" in the form of programming patterns that commonly appear in implemented models. Because of the nature of the mapping of the architectures to human cognition, each modeling idiom translates relatively directly into changes in model behavior data, such as decision timing, memory access, and error rates. Our analysis notes that both architectures have sometimes adopted extreme and opposed constraints, where the human architecture most likely relies on some mixed or more moderate set of constraints.

Implications of Architectural Constraints on Cognitive Modeling

Experienced cognitive modelers are well aware that "the devil is in the details," particularly when it comes to fine-grained models of deliberative behavior. Changes in the particular reasoning path chosen to model a task can manifest themselves as differences in task timing, type and rates of errors, and overall strategy differences. Cognitive architectures such as Soar (Laird, Rosenbloom, & Newell, 1987) and ACT-R (Anderson & Lebiere, 1998) implement constraining assumptions that encourage and sometimes require particular types of modeling idioms or patterns that in turn impact the data the model produces.

This paper compares some of the modeling idioms (perhaps alternatively described as programming patterns) that commonly appear in Soar and ACT-R models of decision making. We have come across many of these comparisons while developing HLSR, a language for building models that can compile both to ACT-R and to Soar (Jones et al., 2006). It is interesting that, although ACT-R and Soar are in some ways "close cousins", there are significant differences in how some types of low-level reasoning tasks must be modeled, although these differences are not necessarily obvious without getting into model details. In many cases, each architecture has adopted constraints that are diametrically opposed to each other, where an alternative architecture might encourage a more moderate or mixed approach.

Constraints in cognitive architectures often manifest themselves as computational bottlenecks that are inspired by assumed limitations on human processing. In ACT-R 6.0 (Anderson et al, 2004), there is a "cognitive bottleneck" that allows only one production rule instantiation to fire at a time (even if multiple rule instantiations currently match), and there are information bottlenecks that allow only one chunk per architectural module to be accessible for production matching through that module's buffer. In particular, that means that only one goal can be active at any time and that only one chunk can be retrieved from long-term declarative memory at a time. These limitations imply that complex logical decisions must be implemented via sequences of retrievals and actions, which in turn impacts the timing of retrieval/decision sequences.

In Soar 8.6, multiple rule instantiations can fire at once, and access to declarative memory is essentially unlimited. However, Soar imposes a cognitive bottleneck by allowing only one "operator object" to be selected at a time. Additionally, all operator selections must occur through Soar's preference/decision mechanism. Finally, in order to maintain logical self-consistency, operator objects can become automatically unselected if their logical preconditions become unmatched. This latter effect implies that individual Soar operators can usually not implement long sequences of actions. Such sequences must instead be implemented by series of operators, which can have an impact on the timing and granularity of decision sequences.

These combinations of imposed constraints and bottlenecks dictate some of the types of modeling idioms that programmers typically use when implementing decision-making processes in each architecture. The remainder of this paper provides examples contrasting four of these types of idioms.

Sequences of Decisions and Actions

ACT-R production rules are allowed to execute multiple actions at a time, but in a limited fashion. They can make changes to the contents of each architectural buffer (for this paper's purposes, we will concern ourselves only with the goal and retrieval buffers). Consider the following example rule from a Towers-of-Hanoi model, which makes a change to the chunk in the goal buffer while simultaneously initiating a new retrieval to the retrieval buffer.

```
(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)
```

Individual Soar rules can also implement multiple actions simultaneously. However, Soar rules are allowed to test complex logical patterns with more flexibility than in ACT-R, and multiple Soar rule instantiations can fire at the same time. As a result, a common Soar modeling idiom is to tease apart individual types of actions into separate rules. This allows the development of more adaptive code that does not introduce "artificial" conjunctions of conditions just because the modeler wants multiple things to happen at once. For the above example, the Soar idiom would typically divide into two separate rules, as shown below. These rules access a "current-retrieval" object that mimics ACT-R's retrieval buffer (there is no architectural requirement for such a buffer in Soar models). The first rule initiates the retrieval, while the second makes the change to the goal state. Notice that the first rule can fire even if some other set of conditions want to change the goal state. The second rule can fire whenever the appropriate information is in the retrieval buffer, regardless of which process might have initiated that retrieval.

```
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)}
```

In any rule-based system, combinations of conditional actions must either be implemented by a combinatorial number of rules covering the space of possible condition combinations or by a set of rules that reason through the combination of conditions. One important difference is that Soar models can sometimes execute such rule combinations in parallel where ACT-R must execute them in sequence. Either choice has an impact on the timing of decision making, as well as the types of errors and adaptivity that the model might produce.

The standard Soar idiom for implementing multiple actions can also encounter problems that impact timing, errors, and adaptivity. The typical approach in Soar would have a single operator object that has associated with it multiple rules that implement the conditional logic for various combinations of actions. However, different sequences of action may require rule-firing sequences of different lengths, some of which can cause the operator object to be deselected automatically (this is in fact the case in the above example, where there are additional rules that match against the "change-value" pattern). This can introduce "race conditions" where one stream of decision making does not get a chance to complete because another stream has deselected the operator. Consider the following, slightly more complicated, ACT-R rule, which implements three separate actions simultaneously.

```
(p clear-disk
   =goal>
      isa move-disk
      disk =disk
      peg =peg
      state peg
   =retrieval>
      isa disk-on-peg
      disk =disk
      peg =on
    - peg =peg
==>
   !output! "Subgoaling clear-disk with disk ~S on
peg ~S to peg ~S parent ~S" =disk =on =peg =goal
   +goal>
      isa clear-disk
      disk =disk
      current =on
      peg =peg
      parent =goal
   +retrieval>
      isa next-smallest-disk
      disk =disk
   =goal>
      state =retrieval)
```

Attempting to implement this with a single Soar operator would almost certainly lead to race conditions that would cause the model to break. The standard Soar idiom to respond to such a situation is to break these simultaneous conditional actions into individual operators, so they cannot race with each other. But because Soar only allows one operator at a time, this imposes sequential processing, where the initial desire was to implement a set of parallel actions. Again, a combination of constraints within the architecture directly leads to meaningful changes in the data that models will produce.

Sequential vs. Parallel Memory Retrieval

In ACT-R, the combined bottlenecks for individual rule firing and memory access through a retrieval buffer produce a common idiom for accessing and processing elements from long-term declarative memory. Before any memory object can be accessed, it must first be fetched into the retrieval buffer. Thus, the idiom is to include one rule (or more) to initiate the retrieval, and one rule (or more) to "harvest" the retrieved item, processing it in the desired way. Below are two example rules, again from a Towers-of-Hanoi model. These rules process a "clear-disk" goal by creating a subgoal to move the "next smaller tower" off of the current disk. In order to accomplish this, the ACT-R model must first find a peg to move the subgoal tower to. This is accomplished by searching for a spare peg and fetching it into the retrieval buffer, where it then becomes available to provide information for the new subgoal.

```
(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)
```

As in our first example above, a Soar model could be built similarly by mimicking the retrieval buffer within Soar's working memory. However, the more typical idiom in Soar would take advantage of Soar's unfettered access to all elements in declarative memory. In such a Soar model, a single rule can perform a complex conditional query and use the information to create the desired subgoal, without requiring the extra step of going through a retrieval buffer.

```
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*)}
```

In general, the lack of a retrieval buffer in Soar allows Soar models to be written in a more compact way with more opportunities for the reuse of individual operators and rules. The primary potential downside is that many Soar models do not take the memory-retrieval bottleneck seriously, as ACT-R models must. It is possible to find Soar models that have literally hundreds of accessible items in their declarative memory at one time, although this is generally truer for "applied" Soar systems than it is for serious cognitive models built in Soar. There are a number of Soar-based cognitive models that self-impose more declarative-memory constraints than the architecture itself requires (e.g., Wray & Chong, 2005; Young & Lewis, 1999). It is also worth noting that Soar models with large declarative memories are usually compensating for the fact that they do not use Soar's built-in learning mechanism. Models that use learning usually use the learned rules for declarative access, rather than relying on huge declarative memories. The situation is similar in ACT-R, except that ACT-R's constraints are more forceful in the sense that it is more difficult to "cheat" in the ways that you sometimes can when using Soar.

There are some senses in which loosely limited declarative memory access may be plausible, but other senses in which it certainly is not. On the other hand, the restriction in ACT-R to have a single retrieval buffer that can hold only a single chunk is probably overly restrictive in some cases. In the example above, it would seem reasonable that a model of even a slightly experienced Towers-of-Hanoi practitioner should just "know" what the third peg is. However, under the current architectural constraints, that is only possible by encoding in the production rules all the combinatorial possibilities of origin and destination pegs (admittedly a limited number with only three pegs, but still too large to be considered elegant or even plausible). It would seem plausible to have a small number of frequently and/or recently used chunks directly accessible from some sort of working memory, but that is currently only possible by having the modeler pack a given buffer with the content of those chunks, a practice that often leads to brittle and/or implausible models. Both assumptions lead to interesting models that are qualitatively different, but perhaps plausible and implausible in their own ways.

The main reason for the differing idioms in this case is that Soar implements its "retrieval process" through rules and rule conditions that can encode arbitrarily complex conjunctions of declarative memory elements. Retrieval in ACT-R is instead a sequential process that takes a set of cues as input and returns a single set of elements to fill the retrieval buffer. Both of these approaches to memory access manifest themselves in modeling idioms that predict different types of behavior. In this case, it is interesting to note that each architecture adopts a rather extreme approach to memory access, where a more accurate model of the human architecture would probably be somewhere in between the two. It seems unlikely that human memory is

limited to holding accessible a single chunk at a time (e.g. Miller, 1956), but equally unlikely that human memory is capable of unfettered retrieval of arbitrarily complex conjunctions.

Partial Matching vs. Preferences for Conflict Resolution

One of the more unique aspects of the Soar architecture involves its mechanisms for supporting symbolic rule-based preferences for conflict resolution. In Soar, all conflict resolution centers around deciding which operator object to select next, and this is generally accomplished by preference rules that propose binary comparisons between the various candidates (O1 is better than O2, O2 is just as good as O3, etc.). The rule-based preference mechanism is necessary because there is no architectural conflict resolution mechanism (other than the architectural component that makes a selection based on the symbolic preferences).

In ACT-R, conflict resolution centers around two types of choices: which rule instantiation should fire next and which chunk should be retrieved from declarative memory into the retrieval buffer. ACT-R includes architectural mechanisms to support both of these modes of conflict resolution. Both mechanisms are similar, being grounded in subsymbolic concepts (utility and activation, respectively) and including similar restrictions such as learning constraints. Thus, the idiom in ACT-R modeling is to create numerically oriented "preferences" that are assumed to reflect some sort of learning from prior experience. The Soar idiom is to encode the preferences as (sometimes complex) sets of logical ordering constraints (which are also assumed to be learned). The result is that we see some significant differences between ACT-R and Soar in conflict-resolution modeling, depending on the type of model. For purely symbolic models, ACT-R must include rule conditions that encode the combinations of constraints that could be represented as individual preference rules in a Soar model. However, ACT-R also provides a subsymbolic partial-matching idiom that is not directly available to Soar modelers. Similarly, the most recent versions of Soar have introduced the ability to specify numeric and probabilistic preferences, so there are some new opportunities to explore non-symbolic preference idioms in Soar, as well.

Following is a simple example of the relatively compact representation of preferences that can be encoded into a Soar model. In this example, the model is to select either an "eat" operator or a "drink" operator, but it prefers to eat before drinking.

```
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>)}
```

Note that, if Soar did not include its preference-based conflict-resolution mechanism, a modeler would be forced to encode the semantics of the various preferences into the operator proposal rules themselves. For example, in the above code, we would have to change the drink proposal rule to the following:

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

A potential problem with this approach to conflict resolution is that it will lead to a combinatorial explosion of conditions for complex preferences between multiple potential choices. In a purely symbolic ACT-R model, the approach would be similar, but with an added constraint. Because only one item can be in the retrieval buffer at a time, an ACT-R model must test the different logical conditions sequentially and either make each test depend on the results of the previous one(s) or accumulate the results in the goal (or some other) buffer for some final decision. In contrast, Soar proposals can each check their combinations of conditions with less restricted access to declarative memory. Thus the symbolic ACT-R approach might look as follows:

```
(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)
```

In the above example, the first rule's retrieval will succeed if and only if there is a "hungry" property with a value of "yes" in declarative memory. If that retrieval fails, the check-thirsty rule will look for a "thirsty" property with a value of "yes". However, ACT-R modelers are not restricted to doing symbolic conflict resolution. For choices like this, ACT-R also supports similarity-based partial matching for retrieval. It is possible to define a "similarity relationship" between different attribute values, which will in turn influence how the retrieval process executes. Using ACT-R's partial-matching mechanism, we can rëimplement the above example as follows:

```
(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)
```

In this case, the attribute values "hungry" and "thirsty" are set to be relatively dissimilar to each other. But the fact that they are defined with any similarity measure at all indicates that they are candidates to be substituted for each other in any partial-matching retrieval. Thus, the choose-action rule initiates a search for "hungry yes", and it will retrieve a perfectly matching chunk if one exists in declarative memory. But if there is no perfectly matching chunk, the retrieval process will instead look for the closest partial match. In this case, a chunk representing "thirsty yes" would be the next best match. Based on whichever chunk happens to get retrieved, the program can then choose to "eat" or "drink", as appropriate. However, if the set of options is so complex or heterogeneous that checking the options cannot be reduced to a single retrieval, then an outer loop must be explicitly maintained to access the various options sequentially, where in Soar they could be combined into a single complex conditional rule. The problem in ACT-R is that if each option involves checking some additional condition (such as perceptual or memory information), then the utility preferences are not helpful because they would attempt to check the same condition over and over again. Either an explicit round robin check of the various conditions has to be set up symbolically in the production conditions or learning of the utilities can be used to iterate through the options by having the failure of each option temporarily depress the utility of the production selecting that option (Lebiere et al., in press).

Exhaustive Processing and Search

The final pattern we investigate involves performing exhaustive iterative actions on a set of similar object or chunk types. For example, imagine that declarative memory contains a number of message objects, each with a text attribute. We would like to build a model that iterates through all of the messages and prints out the text value of each one. In a Soar program this can be done relatively simply because an individual operator application rule can match against multiple objects at a time, and each matching instantiation will execute simultaneously. For example, the following Soar rule simultaneously finds all "unhandled" message objects in declarative memory, prints their messages, and marks the message objects as "handled".

```
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 +)}
```

In contrast, ACT-R is restricted to matching one object at a time through the retrieval buffer. In older versions of ACT-R, this would be accomplished by iterating over a sequence of retrievals and harvests, tagging each chunk as it is processed. This approach also requires an additional rule that detects when the retrieval process has failed to find any further matching candidates for processing.

```
(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)
```

However, the most recent version of ACT-R does not allow non-monotonic changes (such as tagging) to chunks in the retrieval buffer, so new idioms are developing that rely on the subsymbolic processing of the retrieval mechanism.

These new idioms encounter additional confounding factors. A major problem is that the dynamics of the activation calculus, and in particular the learning of the base level to reflect frequency and recency of access, conspire against that iterative process. Recently accessed chunks become more active while chunks that have not been accessed decay and become less active, leading to the opposite dynamics of the iteration desired, namely a winner-take-all tendency to retrieve the same candidate(s) again and again. One typical idiom to get around this problem is to alter subsymbolic processing parameters such as noise, in order to "break out" of bad retrieval sequences. However, this is often only partially successful in moving the iteration along.

Another example of iteration comes again from the Towers of Hanoi. In this problem, it is useful to compute which disk is currently at the top of a particular peg. In a Soar model, the encoded logic is along the lines of "find a disk on the peg for which all other disks on the peg have a lower position". Although this gets a bit messy, the logic can be encoded in the conditions of a single Soar rule. In contrast, an ACT-R model must implement this logic using a sequential loop or by clever configuration of the partial-matching mechanism. Although the sequential iteration can be implemented in a relatively straightforward fashion, it again runs into the stumbling block that ACT-R prefers to retrieve the same disk repeatedly, instead of iterating through all of the disks on the peg.

Note that it is also possible to implement sequential iteration using operators in Soar. Soar does not include the restriction against altering declarative memory items, so the typical Soar idiom in such situations is to tag each object as it is processed in sequence. However, depending on the situation, the alternative idiom in Soar is to use a single rule to process everything at once. It is certainly a valid question, however, whether Soar *ought* to make it so easy to do this type of computation. It could be argued persuasively that humans in general cannot perform this type of exhaustive, instantaneous, massively parallel processing, and so it is a mistake for Soar to allow and even encourage this type of solution. On the other hand, there are certainly some types of massively parallel processing occurring in the human architecture. So once again, we are faced with two architectures that embody extreme constraints, where the truth is probably a combination or compromise.

It should also be noted that there are particular problems of this type that also *require* a sequential solution approach in Soar. For example, although a Soar program can easily use one rule to operate on a whole set of objects simultaneously, it currently has no way to *count* the number of objects in that set. For the task of counting the number of elements in a set, both ACT-R and Soar demand sequentially implemented solutions.

Conclusion

We have examined four classes of modeling idioms that arise relatively directly from the combination of assumed constraints on cognitive processing imposed by the ACT-R and Soar cognitive architectures. We hope that these examples provide a more detailed feeling to the modeling community about what some of the differences and similarities are between the architectures, particularly when it gets to the nitty-gritty of building detailed models. From a cognitive modeling perspective, this is not just an exercise in examining computationally equivalent modeling approaches. Each of the idioms implies measurable differences in the type of data the models will produce. We have also observed that the constraints and bottlenecks assumed by each architecture tend be rather extreme and often opposed to each other. We join others in recommending future work that includes finding more intermediate constraints on the cognitive architecture, which should translate to some variation in the common modeling idioms, and in turn to cognitive models that produce better matches to human data.

References

Anderson, J., & Lebiere, C. (1998). *The Atomic Components of Thought*. Mahwah, NJ: Lawrence Erlbaum.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y . (2004). An integrated theory of the mind. *Psychological Review 111,* (4). 1036-1060.

Jones, R. M., Crossman, J. A., Lebiere, C., & Best, B. J. (2006). An abstract language for cognitive modeling. *Proceedings of the Seventh International Conference on Cognitive Modeling*. Trieste, Italy.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence* 33(1): 1-64.

Lebiere, C., Archer, R., Best, B., & Schunk, D. (in press). Modeling pilot performance with an integrated task network and cognitive architecture approach. In Foyle, D. & Hooey, B. (Eds.) *Human Performance Modeling in Aviation.* Mahwah, NJ: Lawrence Erlbaum.

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review, 63*, 81-97.

Wray, R., & Chong, R., (2005). Comparing cognitive models and human behavior representations: Computational tools for expressing human behavior. *Proceedings of the Infotech@Aerospace 2005 Conference*, Arlington, VA. American Institute of Aeronautics and Astronautics.

Young, R. M., & Lewis, R. L. (1999). The Soar cognitive architecture and human working memory (1999). In A. Miyake & P. Shah (Eds.), *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control,* 224-256. Cambridge University Press.

Acknowledgments

This work was supported in part by contract N00014-05-C-0245 from the Office of Naval Research. Many thanks to Bob Wray for his helpful comments on an earlier draft.