An Abstract Language for Cognitive Modeling

Randolph M. Jones (rjones@soartech.com)
Jacob A. Crossman (jcrossman@soartech.com)
Soar Technology, Inc. 3600 Green Ct. Suite 600
Ann Arbor, MI 48105

Christian Lebiere (clebiere@maad.com)
Bradley J. Best (bbest@maad.com)
Micro Analysis and Design, 4949 Pearl E. Circle, Suite 300
Boulder, CO 80301

Abstract

Cognitive architectures provide a definition of an abstract machine to support programming of cognitive models and intelligent systems. The point of the abstract machine is to provide the most useful set of processes and representations for developing such models, and the machine usually comes hand in hand with a programming language. However, most cognitive architectural languages are specified at a very low level, which hinders model development in a number of ways. We have developed an abstract machine and language that generalizes across architectures, allowing modelers to move up a level in their model specification. This serves a variety of scientific and engineering goals.

Introduction

This paper describes our development of high-level abstractions for modeling cognitive processes and intelligent behavior, together with a formal computer language based on these abstractions. The work is in the spirit of past research into cognitive architectures, which provide functional components and data representations for the purpose of modeling human behavior. Each cognitive architecture defines an abstract machine together with a language for programming that machine. However, until recently, there has been little effort to identify the commonalities across existing cognitive architectures, which would also make it more clear which architectural differences are important from a theoretical point of view.

Our efforts to produce an abstract language for cognitive modeling have shown that there *are* important fundamental and theoretical differences between the most prominent cognitive architectures (Crossman et. al., 2004; Jones & Wray, in press). However, much of the work involved in building specific cognitive models is the same no matter which architecture one is using. This is particularly true for defining high-level knowledge representations, building a structured task analysis, and implementing this with a conventional sense-retrieve-act decision cycle. One goal of this research is to make it feasible for common modeling activities to be accomplished within a common framework and formal language. This should make it easier to build and maintain models, facilitate exploration of model variations within a particular cognitive architecture, and enable comparing models across architectures.

To this end, we are developing an abstract formal cognitive modeling programming language that generalizes the common structures and processes found in existing cognitive architectures. Our approach combines a high-level overview and analysis of a number of architectures for cognition and intelligent agents with a fine-grained analysis of two of the most prominent cognitive architectures. Our current work involves developing a language and compilers to specify high-level cognitive models and translate them into executable ACT-R (Anderson, 1998) and Soar (Newell, 1990; Wray & Jones, 2005) models. This has required us to be extremely careful about managing the theoretical differences and assumptions behind ACT-R and Soar, and generalizing those into a useful abstract framework that can be represented in a formal, high-level language.

Together with the development of the language and compilers, we are working with initial modeling examples to help refine and evaluate the language. This report presents an overview of some of the interesting language features we have identified so far, together with illustrative examples and initial evaluations of the language design.

Abstract Machines and Languages

A key concept in computer science is to define appropriate levels of abstraction for different types of tasks. This approach has led to the notion of an *abstract machine*; an interpreter that provides a fixed set of functional components, together with a "machine" language that operates on those components. The cognitive science field has also produced work in this spirit, leading to the development of *cognitive architectures*. A cognitive architecture can be considered as a virtual machine that provides functional components for the essential elements of the human mind

Cognitive architectures have added to our understanding of mental processes by providing formal abstractions of those processes. Current architectures, however, have necessarily provided low-level abstractions, meaning that their associated programming languages are also low-level, akin to assembly languages for intelligent systems. Such languages require undue effort on the part of model builders and make it difficult to develop high-level solutions that are not mired in details. In some respects, one of the strengths of cognitive modeling is that it forces the modeler to be precise in developing a theory. However, in many instances

it would be more useful to work at higher levels of abstraction when developing individual cognitive models.

Research Goals

The goals for our research can be divided into two broad categories. On one hand, developing higher level abstractions will provide scientific advantages for advancing our understanding of human thinking. On the other hand, the effort should also improve the efficiency and correctness of engineering applied human behavior models.

Scientific Goals

Our scientific goals focus on making it easier to develop, understand, reuse, and compare cognitive models and components of those models:

- Creating a clean distinction between the parts of a model that depend on the unique aspects of the architecture and those that do not.
- Eliminating or reducing the number of possible different ways to create a particular model, thereby reducing potential confounding factors when comparing models.
- Fostering reuse across cognitive models, especially in terms of high-level task knowledge.
- Allowing straightforward comparisons of the same model within two different cognitive architectures.
- Encouraging exploration and fine-tuning of architectures while holding a model's high-level abstractions constant.

Engineering Goals

Our engineering goals aim primarily at controlling the costs of development, maintenance, and deployment of cognitive models and applied knowledge-intensive agent systems:

- Fostering reuse, thereby reducing development expense.
- Decreasing number of lines of code and programming constructs necessary to implement a complete model.
- Improving compile-time and run-time error checking during model implementation.
- Decreasing software maintenance costs by allowing code updates at higher levels of abstraction and incorporating software engineering constructs.
- Allowing model specification only to the level of detail necessary for a particular application or research effort.
- Informing the creation of abstract-level design tools and integrated development environments.

Overview of Cognitive Architectures

To accomplish these goals, we are focusing on three elements: a language for specifying agent behavior at a high level with reusable components (HLSR), a mapping between this language and the underlying architectures (a compiler), and a methodology for developing agents.

Important constraints on HLSR's design arise from the fact that sophisticated cognitive models and agents must incorporate significant amounts of knowledge. There are many good engineering platforms for building software systems, including "lightweight" agent systems. However, these platforms are clearly not suitable for implementing agents that incorporate large amounts of knowledge, that must maintain sophisticated internal representations of situational awareness, and that must manage the maintenance and pursuit of complex sets of interacting goals. In contrast, cognitive architectures have traditionally focused on exactly such capabilities.

In addition, cognitive architectures perform in a *least commitment* (Weld 1994) manner, making context-sensitive decisions about behavior and resource allocations, and flexibly adapting those decisions in the face of a changing environment or assumptions. Least commitment mechanisms, in which control decisions are made at every decision opportunity, contrasts with traditional control logic, in which control decisions are fixed when the program is designed and compiled. Least commitment is a fundamental requirement for autonomous, flexible, adaptable behavior. Cognitive architectures also generally provide explicit mechanisms for relating parallel processing (for example, at the level of memory retrieval, pattern matching, or analysis of courses of action) to serial processing (where behavior systems must ultimately generate a serial set of commitments to action).

As platforms for knowledge-intensive models, cognitive architectures also support the encoding of knowledge into executable models. Many architectures focus on symbolic representations of this knowledge while others also support subsymbolic processing (e.g., the retrieval process in ACT-R's associative network). However, for HLSR, we are targeting a symbolic level of abstraction, leaving "subsymbolic" processes to inform the implementation level. Symbolic encoding of knowledge has a natural relationship to symbolic programming languages, which is ideal for systems that lead the "double life" of serving as human behavior models and application programs.

Knowledge in cognitive architectures is encoded *associatively*, as opposed to *procedurally* or *functionally*, as is standard practice in software engineering. Each architecture includes a mechanism for associative retrieval of potential courses of action, and a conflict resolution mechanism for choosing between the candidates. We argue (and research into cognitive architectures seems to confirm) that associatively represented knowledge is a fundamental key to capturing mixed-initiative commitment to action, which is expected of artifacts with human-like intelligence.

A final reason to focus on cognitive architectures is that they generally provide at least some account of all aspects of intelligent behavior, and provide explicit structures and processes for modeling them. In particular, this breadth includes learning and long-term adaptation to new environments, which will be a key part of future development of sophisticated human behavior models. Much additional research is needed before learning is used in robustly engineered, knowledge-intensive agents. However, learning is critical and successful efforts to design abstract frameworks for intelligent agents must address the challenges of learning early in design.

HLSR Language Constructs

The selection of constructs and execution semantics for HLSR has been driven by several factors. Initially, we identified a set of core elements that appear to be relatively similar across cognitive architectures, including:

- A declarative memory structure and a retrieval method
- Goals
- A procedural memory, particularly containing information to achieve goals
- Mechanisms for timely reaction to external events.
- A *decision process* that iteratively selects goals to achieve and actions to execute based on input and the contents of procedural and declarative memory

The key guiding principle behind identifying these components has been to find useful levels of abstraction, specifically the abstraction of low-level programming and architectural details. Appropriate abstractions will allow HLSR to compile models into multiple architectures and reduce the amount of code necessary. HLSR's design has also included an emphasis on the target architectures. The language not only needs to produce code that will execute on the target architectures, but should also take advantage of the unique capabilities of each target architecture whenever possible. Finally, we apply the principle of "least surprise" in our design, selecting constructs and semantics that are familiar and intuitive to cognitive modelers.

Following these constraints, we have developed a set of high-level primitive language features together with a code generation paradigm that exploits the strengths of the individual target architectures. In this section we provide three detailed examples of core language features in HLSR: the *relation*, the *transform*, and the *activation table*. Relations serve an abstraction of declarative memory structures, including goals. Transforms serve as an abstraction for procedural knowledge indexed by particular goals (and possibly other relations). Activation tables serve as an abstraction for pattern-based reaction that must cover a range of possible response situations.

A *relation*, shown in Figure 1, is an n-ary relationship between atomic symbols or other relations in declarative memory. HLSR relations are defined by listing a name and the attributes that the relation references, similar to Prolog syntax. A relation's declaration can optionally contain a *met condition*; a predicate logic statement that indicates when particular instantiations of the relation may exist in declarative memory. A relation can be used in three ways. First, it can be asserted as a *fact*; i.e., an assumed belief that the relation holds for the given arguments. Second, it can be asserted as a *goal*; i.e., a desire that the relation holds for given arguments. Third, it can be used as a declarative-memory *query*; i.e., a request to retrieve one or more known instantiations of the relation.

An important design decision for queries concerns how many instantiations a single query should attempt to retrieve. Currently, HLSR queries always retrieve a single instantiation, even if multiple instantiations exist in declarative memory. HLSR requires retrieval of the "best" single value that meets the conditions of the query. We refer to this as the *retrieve best* semantic. Each architecture may apply its own process (e.g. the subsymbolic activation process in ACT-R), together with retrieval and similarity semantics engineered into the model, to determine which instantiation is "best" under a given set of conditions.

Because relations can be both asserted (assumptions) or inferred from the *met* conditions (entailments), retrieval strategies must include logic about whether to retrieve pre-existing facts or to execute a more complex logical computation. The HLSR compiler defines this process, the *retrieve v. compute* decision, for each target architecture.

```
# Relations without a met condition
relation Peg (name isa string)
relation Disk(name isa string, size isa integer)
# Relations with a met condition
relation SmallerThan(a isa Disk, b isa Disk)
  ( met ( a.size < b.size ) )

relation TopDiskOnPeg(peg isa Peg,
                      top-disk isa Disk) (
met (DiskOnPeg(top-disk, peg)
     forall DiskOnPeg(other-disk, peg)
       if(other-disk != top-disk)
       then(SmallerThan(top-disk, other-disk))))
```

(A) Defining Relations

```
# "disk" is the top disk on "peg"
new<TopDiskOnPeg>(disk, peg)
# A desire to make "disk" top on "peg"
new-goal<TopDiskOnPeg>(disk, peg)
# Is "disk" the top disk on "peg?"
TopDiskOnPeg(disk, peg)
```

(B) Using Relations

Figure 1: Example of HLSR relations

The *transform*, shown in Figure 2, is a conditionally executed procedure. Transforms consist of a name, attributes, trigger conditions, and a body. The attributes behave like local variables for a transform and define the transform's interface to the rest of the model. Trigger conditions are a set of queries combined by logical conditions. These serve as a query, the instantiation of which indicates that the transform should be executed. The body is a list of queries and actions. Queries can be specified in any order (or in parallel) but actions execute serially, in the order specified. If a model requires parallel execution of particular actions, this can be accomplished by including each action in a separate transform, because multiple transforms may execute in parallel (if the target architecture supports this type of parallelism).

```
transform MoveDiskToPeg(d isa Disk,p isa Peg) (
  # Consider if a goal to put disk d on peg p
  consider-if ( goal<DiskOnPeg>(d, p) )
  body ( DiskClearToMoveToPeg(d, p)
         DiskIsOnPeg(d, other-peg)
         consider-instead(
           DiskIsOnPeg(d, other-peg),
           new<DiskIsOnPeg>(d, p)))
)

# If DiskCleartoMoveToPeg or DiskIsOnPeg
#  fails, an impasse is generated.  A query
#  can retrieve a goal to resolve this impasse,
#  where "trans" binds to the transform instance
impasse<MoveDiskToPeg>(trans)
```

Figure 2: An example of a transform in HLSR

All queries and actions must execute successfully for a transform to complete execution successfully. If a transform query fails to retrieve anything, the transform suspends and automatically creates a subgoal. This is similar to impasse-driven universal subgoaling in Soar or the automatic subgoaling of means-ends analysis.

The *activation table*, shown in Figure 3, combines concepts from truth tables and production rules as a mechanism for specifying conditional actions. Our aim is to provide an easy way to specify a number of conditional actions that span many possible situations. An activation table's condition block defines a set of logical queries. The subsequent action block lists labeled sets of actions where the label determines when it is executed. Each character of the label can be either T (true), F (false), or * (don't care). These characters are associated with the numbered labels in the condition from left to right (i.e. condition 1 is associated with the first character on the left, condition 2 with the second, etc). If the pattern defined by the action label matches the pattern formed by evaluating the logical conditions in the condition block, the action statements for that label execute (e.g., if conditions 1, 2, and 3 all evaluate to logical *true*, the actions defined for TTT would execute). The resulting action block resembles a truth table, making it easier for a modeler to detect gaps and inconsistencies in the action specification.

```
activation-table DecomposeMoveTower() (
 conditions (
   1:TowerOnPeg(tower, destination)
   2:DiskOnPeg(tower.base, destination)
   3:NextSmallestDisk(tower.base, next)
 ) actions (
  TTT:(new-goal<TowerOnPeg>
         new<Tower>(next),destination))
  TF*:(new-goal<TowerOnPeg>
         (tower.base, destination)) )
)
```

Figure 3: Example of an Activation Table

As with other queries in HLSR, the queries in an activation table use the *retrieve-best* semantic. However, depending on the goals of the model, transform actions may be intended to apply to *all* retrievable instantiations of the queries. For such cases, we must develop *exhaustive-search* code-generation fragments for each target architecture. Depending on the details of the architecture, it may search for all the instantiations in serial or in parallel.

Summary

The language constructs described above demonstrate how HLSR abstracts core elements of the underlying architectures, making them easier to program by hiding low-level details. HLSR does not dictate a particular goal-management system or set of query-retrieval strategies, leaving these implementations up to the code-generation templates created for each target architecture. However, the HLSR language does obviate the need for "information meta-tagging" (such as goal-achieved flags) that often bogs down the construction of individual cognitive models.

Compiling HLSR

The goal of compilation is to generate code compatible with each target architecture (for now, Soar and ACT-R) that executes within the constraints the HLSR language defines. This goal is sufficient for *logically correct* execution. But to be useful in practice, compilation should also:

- Generate code roughly equivalent to what a human trained in modeling for that architecture would generate.
- Generate code that takes advantage of the unique capabilities of the underlying architecture, such as automated reason maintenance in Soar and sub-symbolic activation in ACT-R.

In the longer term, our intention is to focus on *optimizing* compilers that can generate code that both executes efficiently and maximally exploits the architecture in ways that would be time consuming or difficult to do by hand.

Our approach to compilation is to define architecture-specific *microtheories* of compilation for HLSR constructs and constraints. A microtheory is a description of the structures, templates, and execution strategies that will be used at the architectural level to execute each HLSR construct. Without HLSR, developers have to define these structures and strategies manually on a case by case basis for each model and model sub-component, relying on expertise to effectively apply them. Using HLSR, the compiler does this for the modeler.

For each HLSR construct and constraint, there are often several ways to execute it on the target architecture. For example, goals in Soar can be implemented using either Soar's automated subgoaling system or by representing goals as "beliefs" using Soar's reason-maintenance system. Advanced HLSR compilers may support more than one microtheory for key constructs, allowing the HLSR developer to select the most appropriate construct for their model at compilation time. The modular design of microtheories encourages such variability in model design.

Initially, we are implementing only one microtheory per construct. To provide an illustrative example, we discuss below the microtheories associated with queries of the TopDiskOnPeg relation (defined above in Figure 1).

ACT-R Micro-theory for Relation Queries

HLSR's model of declarative memory is a single pool of relation instances (facts) that can be retrieved via queries. This is similar to ACT-R's declarative memory model, which includes data chunks and a retrieval mechanism. The challenge lies in mapping clusters of queries that share variables, such as the TopDiskOnPeg query, to ACT-R retrievals that function correctly and produce useful behavior and data. ACT-R does not provide direct support for the predicate logic used by queries. Typically, queries must instead be mapped to a series of retrievals and tests, with intermediate results and variables stored in ACT-R's goal buffer. The exact strategy used by an ACT-R modeler for the retrieval often depends on the structure and quantity of the chunks being queried and the behavior the modeler is interested in, with alternatives including:

- Cognitive looping, which executes each retrieval serially, checking the logical constraints after each retrieval. For

retrievals that fail the logical constraints, the system must backtrack to retrieve another instantiation.
- Reordering of conditions to reduce the number of retrievals for which multiple chunks could be retrieved. This requires knowledge of the cardinality and keys of each type of chunk.
- Using ACT-R's spreading activation and/or partial matching to retrieve a chunk that is "close" to the right value, and then just use this chunk.
- Restructuring declarative memory to store complex links between structures explicitly when it is known that they will be tested.

The current HLSR compiler implements a version of the first strategy, though it should be possible to support the first three by making minor changes to the HLSR language. The fourth can be done explicitly by the HLSR developer by redefining relations. The ACT-R microtheory for matching queries implements the following algorithm:

- Relations with no met condition map to chunk types in ACT-R, treating them like chunks (with some additional slots added to aid processing of meta-information).

```
;;; Goal to retrieve TopDiskOnPeg
(chunk Retrieve_TopDiskOnPeg_In_Context
    disk nil peg nil step intial
    processed nil supergoal nil)

;;; ACT-R productions doing the retrieve,
;;;  harvest and handle error pattern
(p retrieve-disk-on-peg-1R
   =goal>
      isa  Retrieve_TopDiskOnPeg_In_X_Context
      peg  =peg
      step initial
==>
   +retrieval>
      isa DiskOnPeg
      peg =peg
   =goal>
      step 1R)    ;;; retrieval
(p harvest-disk-on-peg-1S
   =goal>
      isa  Retrieve_TopDiskOnPeg_In_X_Context
      peg  =peg
      step 1R
   =retrieval>
      isa DiskOnPeg
      peg  =peg   ;;; make sure it succeeded
      disk =disk  ;;; harvest
==>
   =retrieval>
      processed  =goal  ;;; may rtv another
   =goal>
      disk =disk  ;;; store result
      step 1S)    ;;; Success for step
(p harvest-disk-on-peg-1F
   =goal>
      isa  Retrieve_TopDiskOnPeg_In_X_Context
      step 1R
   =retrieval>
      isa error
==>
   =goal>
      step 1F)    ;;; Failed at step 1
;;;; More productions
```

Figure 4: ACT-R retrieve, harvest, check pattern

- Nested HLSR queries (which have other queries in their met condition) are incorporated recursively, flattening the query into a set of fact retrievals and attribute tests.
- An ACT-R retrieval goal guides the multi-step retrieval process for the flattened query.
- Fact retrievals and their related attribute tests convert to ACT-R chunk retrievals.
- Logical conjunctions and shared variable storage convert to "harvest productions", which simultaneously test the result of a query and store the result in a goal slot.
- HLSR generates additional productions to handle retrieval failures. Failures can trigger other processing such as backtracking to find additional retrievals, or can indicate the end of a process.

The core of the ACT-R retrieval micro-theory is the "retrieve-harvest-check" pattern. An example of this pattern for the DiskOnPeg retrieval in the TopDiskOnPeg HLSR production is shown in Figure 4. This is a common pattern of productions found in ACT-R models, though it is often optimized by hand. Its compiler implementation uses more productions than an expert might use, but automates the pattern's implementation as a reusable package.

Soar Micro-theory for Relations

Compiling relations to Soar inverts the challenges of compiling to ACT-R. Soar has no long term declarative memory: all permanent knowledge is stored as productions. It has a short term declarative memory (working memory), structured as a graph accessed from a root node called the *state*. Retrievals in Soar involve link following rather than a general search over the entire pool of memory. Furthermore, these retrievals retrieve all values that match a pattern, not just the one best value as HLSR requires.

On the other hand, Soar productions support pattern matching over a subgraph of memory providing a fairly straightforward mapping of complex HLSR queries to Soar productions. The structure of relations in HLSR map naturally to Soar WMEs that share a common root symbol.

The essential aspects of the Soar microtheory for retrieval are the fact storage structure, the retrieval mechanism, and especially the strategy for retrieving one best value for each query. The standard approach to structuring declarative memory in Soar is to partition the state into sub-graphs based on data type and the context in which it is used. Processed sensory data and problem solving results are asserted deliberately using operators, while additional facts are inferred using truth-maintained elaboration productions.

To *retrieve the best*, the Soar microtheory must produce partitions of memory that allow for general retrievals while not generating excessive partial matches that would greatly reduce production matching efficiency. Our approach to facilitating general queries is to store all facts in a single pool partitioned by object type. That is, to access a fact the WME graph would be navigated from the root state through the object pool and the fact typename to the fact instance itself as seen in Figure 5. This limits partial matches to the number of simultaneously asserted facts of a given type.

The retrieval process is a three part process basd on the typical Soar modeling strategy described above.

```
# Create pool to store objects, but index
#  by type to reduce partial match costs
^top-state
 ^objects      ;# fact pool
  ^typename
   ^object     ;# a fact instance
    ^paramname
    ...        ;# params and internal tags

# Compute smallerthan when we need to
#  find the top disk on a peg
sp {topdiskonpeg*retrieve*smallerthan
  (state <s> ^retrieveal-request <rqs>)
  (<rqs>     ^request <rq>)
  (<rq>      ^type TopDiskOnPeg)
-->
  (<rqs>         ^request <new-request>)
  (<new-request> ^type SmallerThan)}

# One retrieval production (peg is known)
sp {retrieve*topdiskonpeg*peg*no-topdisk
  (state <s> ^retrieval-request <rqs>
             ^objects <objs>)
  (<rqs>     ^request <rq>)
  (<rq>      ^type TopDiskOnPeg
             ^params <param>)
  (<param>   ^peg <peg>)
  (<objs>    ^object <st-1>)
  (<st-1>    ^type DiskOnPeg
             ^disk <top-disk> ^peg <peg>)
 -{ (<objs>  ^object <st-2>)
    (<st-2>  ^type DiskOnPeg
             ^disk { <other> <> <top-disk> }
             ^peg <peg>)
    (<objs>  ^object <st-3>)
   -{ (<st-3>  ^type SmallerThan
               ^a <top-disk> ^b <other>) } }
-->
  (<objs>    ^object <new-object>)
  (<new-object> ^type TopDiskOnPeg
                ^peg <peg>
                ^top-disk <top-disk>)}
```

Figure 5: Soar structure and retrieval production

- Directly asserted facts (those stored in declarative memory via an action) are kept in the type-partitioned memory pools until they are explicitly retracted.
- Facts that can be inferred using HLSR relation *met* conditions are compiled into elaboration productions that assert the given relation if the met condition matches. These productions are constrained to fire only when needed (when an action using the assertions is executed). Figure 5, shows examples of these elaborations. Multiple values may be asserted for the same relation.
- Retrieval occurs when the retrieved values are needed to execute an action. Given an action and all of the facts asserted in the object pool, the compiler outputs productions to propose an operator for each combination of facts necessary to bind the variables used in the action. At this point, Soar's operator selection process will select only one operator for execution, the other operators are retracted, and Soar is left with one set of retrieved values to use (meeting HLSR's "retrieve one best" constraint).

Soar's retrieval process selects the one "best" value for a query randomly. It is possible to bias the selection of operators using preferences, including probabilistic. To do this effectively would require either the definition of a more sophisticated retrieval strategy (e.g., ACT-R's subsymbolic activation mechanism) or the ability to add preference knowledge through HLSR. We are currently exploring each of these possibilities.

Conclusions

HLSR provides an abstract architecture and language for developing cognitive models and intelligent systems. Its design has combined a top-down approach (analyzing similarities across a variety of cognitive and agent architectures) and a bottom-up approach (working in detail to provide a language that can compile to both ACT-R and Soar models). The language's design has already allowed us to characterize a number of subtle differences between ACT-R and Soar, such as their approaches to retrieval and exhaustive matching. But these results are in the context of a set of uniform abstract processes and representations that the architectures both share. Future work will produce more detailed comparisons of the architectures, as well as demonstrations of usability and reusability to accomplish the scientific and engineering goals of using an abstract machine and language as a basis for cognitive modeling.

Acknowledgements

This work is being funded by the Office of Naval Research under contract N00014-05-C-0245.

References

Anderson, J. and C. Lebiere, (1998) *The Atomic Components of Thought*. Lawrence Erlbaum.

Crossman, J., R.E. Wray, R.M. Jones, and C. Lebiere. (2004) A High Level Symbolic Representation for Behavior Modeling. In *Behavioral Representation in Modeling and Simulation Conference*. Arlington, VA.

Jones, R. M., & Wray, R. E. (in press). Comparative analysis of frameworks for knowledge-intensive agents. *AI Magazine*.

Morgan, G. P., Cohen, M. A., Haynes, S. R., & Ritter, F. E. (2005). Increasing efficiency of the development of user models. In *Proceedings of the IEEE System Information and Engineering Design Symposium*.

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

St. Amant, R., Freed, A. R., & Ritter, F. E. (2005). "Specifying ACT-R models of user interaction with a GOMS language." *Cognitive Systems Research*, 6(1), 71-88.

Salvucci, D. D., & Lee, F. J. (2003). Simple cognitive modeling in a complex cognitive architecture. In Human Factors in Computing Systems: CHI 2003 Conference Proceedings (pp. 265-272). New York: ACM Press.

Weld, D. (1994). "An Introduction to Least Commitment Planning." In *AI Magazine*. 15(4), 27-61

Wray, R. E., & Jones, R. M. (2005). Considering Soar as an agent architecture. In R. Sun (Ed.), Cognition and multi-agent interaction: From cognitive modeling to social simulation, 53–78. Cambridge, UK: Cambridge University Press.