A High Level Symbolic Representation for Behavior Modeling

Jacob Crossman  Robert E Wray  Randolph M Jones
Soar Technology, Inc.
3600 Green Court, Suite 600
Ann Arbor, MI 48105
734-327-8000
{jcrossman, wray, rjones}@soartech.com

Christian Lebiere
Micro Analysis & Design
4949 Pearl East Circle, Suite 300
Boulder, CO 80301
303-442-6947
clebiere@maad.com

Keywords: Human behavior representation, cognitive architectures, software engineering, programming languages

ABSTRACT: An important goal of the human behavior modeling community is to develop methods and tools for reducing the expense involved in developing knowledge intensive human behavior models and agent systems. This paper reports an approach addressing this goal. The key feature that sets human behavior models apart from other software systems is the reliance on intensive amounts of knowledge encoded into appropriate structures and processes. We focus on cognitive architectures, because these are the systems that have most emphasized the development of structures and processes that support human-like levels of knowledge and reasoning. Our analyses of existing cognitive architectures demonstrate that they contain many similar components and processes when viewed at an appropriate level of abstraction. We use this level of abstraction to guide our development of a High Level Symbolic Representation for human behavior modeling. HLSR will ultimately provide a high-level knowledge-based language for specifying behavior models and compilers for translating HLSR models into executable code for different cognitive architectures.

1. Problem Statement
Building high-fidelity human behavior models is a difficult process. Often, the knowledge that is used to create such models derives from task lists and expert opinion. Normally, a model designer decides on a scenario and then selects and implements potential tasks and strategies that a human would perform in that situation. These tasks delineate the types of procedural and domain knowledge that a model must embody to generate behavior. However, transforming that knowledge to behavior in a computer-executable form is beyond the reach of typical model developers familiar with the subject matter. At this point, model development usually requires significant efforts by a specialist in knowledge engineering techniques.

There are many problems with this traditional approach: 1) human behavior models are difficult to create; 2) once created they are often inflexible (i.e., difficult to apply to similar problems or slight mission changes); 3) the knowledge built into any particular representation is difficult to transfer to another representation; and finally, 4) they are difficult for users to customize. This last problem is particularly egregious because the rapid pace of military development combined with the heightened need for information security makes proving new warfare concepts arduously slow. None of these problems are inherent in the development of human behavior models. Rather, they arise because of the complexity of the models, and model developers must be extremely careful if such problems in design and implementation are to be avoided. Our motivating assumption is to pursue approaches that encourage or assist model developers in avoiding such problems in the first place.

Our response to these problems is to develop tools and techniques that automate, as much as possible, the translation between high-level model design and implemented behavior systems, thus reducing the required involvement of a knowledge engineer, and improving the flexibility of the defined models. In particular, this report presents our efforts to develop a High Level Symbolic Representation (HLSR). The HLSR provides a formal level of description closer to the level of concepts that a subject-matter expert (SME) can specify. However, because the semantics of HLSR are defined in a specified language, we can then compile HLSR task specifications into executable code within one of several popular computer architectures for specifying human behavior. We are presently focusing on compilation to Soar (Laird, Newell et al. 1987; Newell 1990) and ACT-R (Anderson and Lebiere 1998).

Soar and ACT-R are two of the most mature cognitive architectures available, and they provide explicit support for a variety of computational mechanisms and representations that are directly relevant to human behavior modeling. One serious drawback to these architectures is that they provide very low-level programming languages (with primitive operators that are very close to the architecture’s details), that make developing a behavior model a task akin to programming in assembly language. From one
perspective, our goal with HLSR is to advance the state of the art in human behavior modeling in a manner similar to how high-level languages advanced software engineering. At one point in time, the idea of creating a formal design-level language that could be automatically compiled into an efficiently running computer program was a significant challenge, but the history of high-level language design has demonstrated that there was really no other choice for the future of software development. We are in a similar situation with current cognitive architectures. Although it remains a challenge to define abstract processes and representations that cut across the requirements for human behavior modeling, the payoffs for development in the long run will be more than worth the effort.

2. HLSR Vision

HLSR goals are to simplify development and reuse of capable agents. By capable, we mean agents that exhibit relatively high degrees of behavioral autonomy. They should adapt to their environments, they should behave robustly and flexibly, and they should be taskable. HLSR should simplify the development process by providing language constructs that make it easy to tell agents what to do and how to do it, to understand what an agent knows, and to identify knowledge components that can be used in the development of agent extensions. HLSR will include at least the following three elements: a language for specifying agent behavior at a high level and in reusable components (HLSR), a mapping between this language and the underlying architectures (a compiler), and a methodology for developing agents.

2.1 Focus on cognitive architectures

Given the realities of the state of the technology, it is particularly important for any paper about “agents” to define what it means by that term. There is no firm consensus as to what makes a computer program an “agent”, but many definitions include the notions that the agent is embedded in an environment, that it senses the environment, and that it autonomously interacts with the environment over time in pursuit of its goals, and to acquire more sensed information. Other notions of agency add the requirements that the software contain some type of explicit representation of beliefs, goals, and intended actions in the environment. Our own notion of agency is consistent with this collective definition, but tends to favor systems that handle fairly complex tasks that are normally assumed to require significant amount of knowledge to accomplish. We refer to such agents as knowledge-intensive agents, primarily in order to distinguish them from the popular notion of light agents (which are seen particularly in the multi-agent systems community). In light agents, one of the defining (and desirable) characteristics is that each individual agent is fairly simple, having very specific goals and constrained interactive abilities. The point of such agents is to solve complex problems through the emergent interactions of large groups of such simple agents (akin to an ant colony).

We are instead interested in individual computer programs that must address complex reasoning tasks on their own. The problems most appropriate for knowledge-intensive agents are characterized by a complex variety of interacting goals, and a comparatively large number of modes of interaction with the environment. In response to this level of complexity, knowledge-intensive agents generally do require explicit representations of beliefs, goals, and intentions, but these must further be structured into sophisticated relational (and often hierarchical), structured representations of situational awareness. In addition, such agents generally must be capable of mixed levels of commitment and interaction, generating appropriate combinations of purposeful and reactive behaviors.

These characteristics of knowledge-intensive agents in turn impact the types of primitive computational elements that are best suited for the development of such agents, and these elements are so far best represented by the relatively mature existing cognitive architectures. The desired primitive computational elements for knowledge-intensive agents include, but are not limited to, efficient, parallel, associative memories; structured relational representations of situational awareness; automated belief maintenance; and least-commitment planning and execution.

There are many other reasons that cognitive architectures are the best starting point for an HLSR. Human behavior models that perform in complex and dynamic environments require autonomy and flexiblity in execution. Existing cognitive architectures directly provide and support non-linear control constructs that aim to address such capabilities. Such control constructs include productions (Newell 1973) and other efficient, relational pattern matchers; rapid context switching for interruptibility and pursuit of simultaneous goals; and varying amounts of parallelism in pattern matching, reasoning, and behavior.

Importantly, cognitive architectures generally support the notion of least commitment (Weld 1994), which allows agents to make context-sensitive decisions about behavior and resource allocations, but also to be flexible about adapting those decisions in the face of a changing environment or assumptions. **Error! Reference source not found.**, contrasts least
commitment mechanisms, in which control decisions are made at every decision opportunity, with traditional control logic, in which control decisions are hard-coded when the program is designed and compiled. Least commitment is a fundamental requirement for autonomous, flexible, adaptable behavior.

Figure 1: Least commitment in comparison to traditional control logic.

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). In essence, cognitive architectures define processes to support knowledge-driven commitments to courses of action, mixed with appropriate reactive context switching when necessary.

As architectures for knowledge-intensive models, cognitive architectures also support the encoding of knowledge into executable models. Many architectures focus on symbolic representations of this knowledge and we have adopted this focus. Some architectures 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 as well as application programs.

Knowledge in cognitive architectures gets encoded associatively, as opposed to sequentially or functionally, as is standard practice in software engineering. Each architecture includes some mechanism for associative retrieval of potential courses of action, and then 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 the types of mixed-initiative commitment to action that are expected of artifacts with human-like intelligence.

A final reason to focus on cognitive architectures is that they generally attempt to provide at least some account of all aspects of intelligent behavior, and provide explicit structures and processes for modeling them. This breadth particularly includes learning and long-term adaptation to new environments. Learning 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 important, and successful efforts to design abstract frameworks for intelligent agents must address the challenges of learning early in design.

2.2 Building high-level abstractions
We are investigating existing cognitive architectures in order to identify properties, representations, and mechanisms that they share and to create abstractions of these shared computational features. The abstraction process will help rationalize years of research by agent, cognitive architecture, and human behavior representation research groups. By focusing on general categories of functions and representations within the scope of knowledge intensive, situated agents, we can explicitly identify and study much of what has been learned about computational intelligence.

Abstraction will also allow us to specify behavior models that are independent of the details of any particular cognitive architecture. Currently, model implementation must be performed by a knowledge engineer who is intimately familiar with the finest details of the implementation architecture, similar to a software engineer using assembly language. Additionally, once a model has been implemented, it will necessarily include many design features that closely tie it to the implementation architecture, which makes transfer of the model to other applications, architectures, or environments much more difficult.

Thus far, we have identified and enumerated a number of functional patterns and development patterns across existing cognitive architectures. Our initial analysis focuses on patterns within Soar and ACT-R, but has also been informed by patterns in non-cognitive architectures such as belief-desire-intention (BDI) (Bratman 1987; Wooldridge 2000) and planning systems (Erol, Hendler et al. 1994; Weld, Anderson et al. 1998). From this analysis we expect these patterns will also extend to these other systems as well. We describe what we mean by functional and development patterns below.
2.3.1 Functional Patterns
Cognitive architectures implement mechanisms for combining reactive and goal-driven behavior. While the details of these mechanisms vary among architectures, at least three constructs and mechanisms can be abstracted as the core elements of intelligent behavior. These constructs and mechanisms are goals, transforms, and reactive activation.

We define a goal as an explicit declaration of a desired state. We distinguish the process used to achieve a goal (the transform, below) from the goal itself. Most symbol-level intelligent systems define some construct that can be mapped to our definition of a goal. In ACT-R the architecture defines an explicit goal construct. There are a few alternative methods for representing task goals in Soar. Generally, goals are defined by the developer, and part of the design process involves identifying agent goals. A goal’s purpose is twofold: to direct behavior meaningfully and to provide metrics for determining whether behavior has been successful. Internal representations for goals vary from simple symbols with implicit semantics (e.g. AchievePoint) to detailed logical definitions that can be used by planning processes to determine desirable sequences of actions.

A transform is a collection of closely related actions intended to achieve or make progress toward a goal. Transforms are not tied to goals, emphasizing least commitment. Transforms take various forms in intelligent systems. In ACT-R no native construct represents the breadth of transform capabilities. However, individual productions designed to apply to a common goal are a close approximation. In Soar, collections of productions called “operators” map to the concept of a transform. Both ACT-R and Soar impose upper bounds on the execution times of individual transforms, ensuring an agent the opportunity to react quickly to changes in context.

Reactive activation is the activation of a knowledge construct (e.g., a belief, goal, or transform) based entirely on a sensed pattern. Soar and ACT-R both use production rules to perform reactive activation, and ACT-R supplements this with other activation mechanisms. Cognitive architectures constrain reactive activation by their respective decision processes. The result is that reactive processes must always be evaluated within the context of the current goal-driven behavior before making any significant changes to agent behavior.

2.3.2 Development Patterns
Development patterns include required low-level tasks directly relevant to the higher-level behavior, high-level tasks that developers would like to be made easier, and design patterns used to solve common problems in intelligent system development. In general HLSR should make the low-level tasks no longer necessary, the high-level tasks easier, and the micro-patterns either elements of the language or library components.

An example of a low-level task is symbolic tagging. Symbolic tagging is the process of creating and managing information about the processing of one or more objects. A simple example of symbolic process tagging is marking a message as “sent” after processing it with a “SendMessage” transform. Symbolic tags are common and often scattered widely throughout the declarative knowledge of an agent, and they serve important functional purposes such a organizing processing information related to knowledge structures and serving as markers for process control. While functionally necessary for managing processes within the least commitment framework of cognitive architectures, tags tend to clutter agent knowledge with small details and repetitive processing. HLSR can aid the developer by providing explicit constructs for organizing tags and primitive processes for automating the management of many classes of tags.

A high-level development pattern is the knowledge integration pattern. Efficient development of knowledge rich agents requires reuse of knowledge from various sources, including existing agents, domain ontologies, common sense knowledge repositories, and special purpose processing libraries. Currently this process includes several error prone and time consuming steps. First, desired functionality has to be identified. Because much of the most useful knowledge is embedded in larger knowledge bases and agent systems, this task can be very difficult. Second, the dependencies and interface for the behavior must be determined. The more dynamic and flexible the behavior, the more difficult this is to determine because of the high degree of knowledge coupling required in knowledge rich systems. Third the underlying details of the behavior must be understood. If the behavior was implemented on another architecture, these details end up having to be rewritten, thus only design reuse is achieved. If the behavior was implemented for the same architecture, the behavior must be examined to ensure it uses the architecture consistently with the rest of the agent (often not the case).

HLSR can help this process in several ways. First, the language can provide explicit packaging constructs that allow behavior encapsulation, thus the functionality is clearly defined and details of execution are hidden. Second, by specifying behavior units at a level above
the architecture, the compiler can ensure consistent use of architecture mechanisms. Third, the language can provide interfaces to common knowledge formats (e.g. OWL), importing knowledge from third party sources becomes possible.

An example of a micro-pattern is the retrieve vs. compute solution. In general, this micro-pattern expresses the choice between retrieving a previously stored solution to a specific problem and recomputing the answer using general procedural means. Both Soar and ACT-R use mechanisms to automatically store a problem solution after it has been computed. Often, these mechanisms are difficult to leverage and require considerable skill to master their subtle impacts. Furthermore, implementations that take advantage of precomputed solutions tend to result in knowledge that is more complex and entangled in the details of optimization rather than in the behavior description that is intended. The secret to an HLSR solution to this pattern is to have the compiler interact with the architecture’s retrieve-compute mechanisms, rather than the developer. The advantages are threefold. First, even novice developers can take advantage of the more advanced features of the architecture. Second, the HLSR knowledge is expressed independently of the optimization knowledge, thus leading to a more understandable and maintainable knowledge base. Third, the knowledge can be reused independent of any particular retrieve-compute mechanisms (e.g. compile to Soar to leverage it’s mechanisms, compile to ACT-R to leverage it’s mechanisms).

2.3 Commitment and reconsideration
We have developed a general framework for analyzing the processes of cognitive architectures. Our framework generalizes the notions of commitment (Wooldridge 2000) and reconsideration (Schut and Wooldridge 2001) in intelligent agents. Each behavioral function within a model can be viewed as the commitment or reconsideration of activating a particular type of knowledge structure. This guiding principle allows us to create abstract processes for each type of knowledge structure, and then further to categorize various implementations of processing by formally defining the commitment and reconsideration procedures provided by each cognitive architecture. Analysis of commitment and reconsideration procedures is important if we wish to remain in the least-commitment paradigm that identifies knowledge-intensive agents. We cannot (or should not) resort to functional or sequential formalisms because they violate the notion of least commitment, so we must uniformly instead formalize the commitment and reconsideration strategies for each representational structure that makes up an agent’s knowledge base. In addition, these dual levels of analysis provide exactly the information we need to both design HLSR (based around the abstractions) and various knowledge compilers that translate HLSR models into executable architecture-based models (based around the various implementations of commitment and reconsideration).

![Figure 2: Unifying view of memory operations via consider, commit, reconsider, unconsider, and store.](image)

We have defined five basic processes that appear to address the ways knowledge structures are handled during decision making. These basic processes correspond to transformation between three knowledge structure states, as shown in Figure 2. Our labels for these three knowledge states are latent, retrieved, and activated. The following five processes govern the various transitions that can change a single knowledge structure’s state.

**Consideration** is a retrieval process that instantiates a specific knowledge structure from an outside stimulus or long-term memory. Here “knowledge structure” is defined as any mental structure that an agent can reason about such as representations of real world objects, goals, and transforms. After a knowledge structure is considered, it is available for reasoning, but is not yet part of the agent’s beliefs (i.e., it is not activated). The process of consideration moves a particular knowledge structure from the **Latent** state to the **Retrieved** state. Consideration is a transformation process but implies an underlying creation process for constructing memory representations. Architectures implement the details of the creation process in different ways, but share the transformational concept of retrieving a knowledge structure for further reasoning. Examples of consideration include an agent considering a goal to reload a weapon during a fight, or considering whether to believe that a sensed visual entity is a soldier.

**Commitment** is the process of committing to a particular knowledge structure. After a knowledge structure is committed, it is said to be part of the agent’s beliefs (i.e., it is activated). Once “activated,”
it becomes a part of the current decision-making context and can be used as a basis for making further decisions. This process moves a particular knowledge structure from the Retrieved state to the Activated state. Commitment does not change the contents of memory, but rather the state of what is in memory, as driven by a deliberate decision-making process. For example, an agent may commit to a goal to load a weapon (from among other goals, such as moving forward) after deliberately determining that it is the goal most likely to help achieve the more general goal of disabling the enemy. On the other hand, the agent could not reactively commit to the goal without reasoning over it first. For example, an agent generally would not commit to loading a gun based purely on an external stimulus.

**Reconsideration** is the process of removing a knowledge structure from the current decision-making context. A deactivated knowledge structure can still be reasoned about by the agent, but it is no longer considered part of the agent’s beliefs. Reconsideration moves a knowledge structure from the Activated state to the Retrieved state (here we do not make a distinction between items that have been retrieved, but not yet activated, and those that have been deactivated via reconsideration). Examples of reconsideration include an agent deactivating the goal to load its weapon after it determines that the weapon is jammed, and reconsidering whether to believe an entity is a soldier after discovering that the entity does not carry a weapon. Reconsideration is important for coherence in behavior. When an agent is interrupted, it may deactivate some previously active goal. However, once the interruption is complete, the agent may want to commit again to the previous goal. Because that goal remains retrieved (it has not been unconsidered), the context associated with the previous goal remains available and does not require another instantiation.

**Unconsideration** is the process of removing a knowledge structure entirely from the decision-making process, so that it is no longer under consideration in the current decision-making context. Unconsideration moves a particular knowledge structure from the Retrieved state to the Latent state. Unconsideration, similar to consideration, is a transformation process that implies an underlying memory management process for removing memory representations from the current context. Examples of unconsideration include an agent unconsidering a goal to load a weapon after determining that the weapon is jammed, or unconsidering whether to believe that an entity is a soldier after committing to the belief that the same entity is a local civilian.

We have defined six core requirements for HLSR to guide the analysis and design process. Our initial efforts focus primarily on making HLSR general enough to generalize concepts shared by Soar and ACT-R. Later iterations of HLSR’s design will include the core concepts from additional architectures, and thus the requirements will be generalized appropriately.

1. **HLSR must be independent of the target implementation architectures.** HLSR must not depend on the existence of particular individual architectural capabilities and structures that are not generally shared by other architectures.

2. **HLSR must be a high-level language,** similar to high-level programming languages. That is, an HLSR developer must not be required to code any low-level knowledge to produce an executable model. Our current working definition of “low-level” includes constructs that could clearly be implemented in alternate ways (by different underlying architectures), as well as constructs that do not immediately provide a knowledge-level understanding of the model’s behavior. Forcing HLSR to be a high-level languages is intended to make producing agents more efficient (more time spent on the desired behavior and less on details) while at the same time making the agent models in HLSR more maintainable.

3. **HLSR must make it both possible and convenient to package knowledge into knowledge components.** A component-oriented approach should allow developers to construct libraries of reusable behavior components, and definite implementation-independent interfaces for those components.

4. **HLSR must guide and make it easy for the developer to take advantage of the specific features of cognitive architectures.** It is not sufficient merely to provide a high-level computational programming language for knowledge-intensive agents. Rather, it is necessary to include those components and processes that have been identified in the research on cognitive architectures as being critical for intelligent and autonomous behavior.

5. **HLSR must support incremental addition of knowledge efficiently and robustly.** Model developers must be able to improve the knowledge-base of particular behavior models over time, without having to refactor significant portions of the initial knowledge base. Incremental addition of knowledge should also facilitate automated agent learning.

6. **HLSR must be complete and transparent.** HLSR must be expressive enough that HLSR models can be compiled and executed on each
supported architecture without the need for additional architecture-specific code

These core requirements can be organized into three categories related to the developer, reuse, and architecture, as shown in Figure 3.

![Figure 3: HLSR requirements compete.](image)

Organizing the requirements this way shows HLSR is pulled in six distinct directions, as all six requirements compete with each other. For example, leveraging cognitive architectures (requirement 4) necessarily involves drawing the developer’s attention away from high-level behavior and toward architecture details. Furthermore, it leads to highly complex and unmaintainable interdependencies between knowledge that are necessary for context driven behavior, thus competing with reuse requirements. Even within the architecture slice, exploiting architectural details competes with architecture independence in that maximal exploitation requires intimate knowledge and control of the architecture’s features.

This competition among requirements means that HLSR cannot possibly fulfill any of the requirements completely. Rather, our approach in HLSR is to balance the core requirements, thus creating a language that will be useful in general for intelligent system implementation.

This goal to balance the requirements separates HLSR from many of the other attempts to provide tools for developing intelligent systems. For example, approaches that are based on traditional OO techniques strongly concentrate on decomposition and reuse concerns, but have left the details of implementing high-level constructs and intelligent behavior processes to the developer. Other systems (e.g., Yost 1993) have focused on making the specification of intelligent behavior higher-level and easier, but sacrifice the developer control necessary to generalize solutions to other problems and domains. Cognitive architectures and their associated languages have provided very powerful core capabilities for intelligent behavior, at the cost of increasing the complexity and interdependencies among behavior components.

4. Evaluation

The success of HLSR ultimately depends on the usefulness of the language for developing behavior models. Because, we will not be able to assess overall measures of this type of usefulness until the language is complete and (hopefully) being used, we have, in the interim, identified a useful set of desired language features that we can use as evaluation metrics. Examples include **generality** of the language (Is a particular feature general overall multiple architectures?), **encapsulation** (Does a feature encapsulate an important concept in intelligent system design or implementation?), and **completeness** (Does a feature require any additional low-level code or parameters to be useful in general?). We have collected ten general requirements. Individual feature evaluation will assess each potential language feature with respect to each of the language requirements, measuring whether (and how much) the feature has a positive or negative impact with respect to each requirement. Many language features may have both positive and negative impacts with respect to individual requirements, so the evaluation will include an analysis of the tradeoffs inherent in each feature.

6. Conclusion

Our efforts so far have concentrated on defining an appropriate level of abstraction and initial symbolic components for HLSR, a language for representing human behavior models. We have adopted an empirical approach, by analyzing two cognitive architectures and using them to guide our analysis of the important similarities and differences between architectures. We have particularly focused our analysis on those aspects of the architectures that seem most germane to supporting reasoning with large amounts of knowledge, because that is where HLSR will set itself apart from “standard” programming languages. Although in the preliminary stages of this project, we have already identified important dimensions for analysis, and abstract components that appear to cut across the specific architectures. These common components will provide the basis for the initial version of HLSR. By analyzing two specific architectures, we have also been able to consider the different ways in which the architectures implement some of the abstract representations and components. This will in turn guide our efforts to create an HLSR “compiler” for translating HLSR knowledge into executable models within each target architecture.

The ultimate goal of this work is to provide useful tools that reduce greatly the expense involved in
developing, testing, and maintaining human behavior models. As part of this work, we are also creating a useful framework for comparing and evaluating agent architectures, and defining the similarities cognitive architectures have to each other at high levels of abstraction. Eventually, we will add additional cognitive and agent architectures to our analysis, giving us a rich framework for comparing similarities, differences, strengths, and weaknesses of a variety of approaches to human behavior modeling.

7. Acknowledgements
This work was funded the Defense Modeling and Simulation Office (DMSO) under contract F33615-03-C-6343.

8. References

Author Biographies

**JACOB CROSSMAN** is a research engineer at Soar Technology where he designs and develops agent-based systems and formalisms for specifying agent behavior. He received an M.S. in computer science from the University of Michigan Dearborn.

**ROBERT E. WRAY, PhD,** is a Senior Scientist at Soar Technology, Inc. His research and development experience includes work in agent-based systems, machine learning, knowledge representation and ontology. Previously, he was assistant professor of computer science at Middle Tennessee State University. He received a Ph.D. in computer science and engineering from the University of Michigan.

**RANDOLPH M. JONES, PhD,** is Chief Scientist of Soar Technology, Inc., and Assistant Professor of Computer Science at Colby College. He has worked with a variety of agent architectures and models, including having participated in the TacAir-Soar project since its inception. He has nearly 20 years of experience involved in research on agent architectures, intelligent agents, machine and human learning, graphical user interfaces, cognitive modeling, and a variety of related areas.

**CHRISTIAN LEBIERE** is a Scientist at Micro Analysis and Design. During his graduate career, he worked on the development of connectionist models, including the Cascade-Correlation neural network learning algorithm that has been used in hundreds of scientific, technical and commercial applications. Since 1990, he has worked on the development of the ACT-R hybrid cognitive architecture and is coauthor with John R. Anderson of the 1998 book The Atomic Components of Thought. His main research interest is cognitive architectures and their applications to psychology, artificial intelligence, human-computer interaction, decision-making, game theory, and computer-generated forces.