The Pennsylvania State University

The Graduate School

College of Information Sciences and Technology

A THEORY-BASED ENVIRONMENT FOR CREATING REUSABLE COGNITIVE MODELS

A Thesis in

Information Sciences and Technology

by

Mark A. Cohen

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2008

The dissertation of Mark A. Cohen was reviewed and approved* by the following:

Frank E. Ritter
Associate Professor of Information Sciences and Technology
Associate Professor of Computer Science and Engineering
Associate Professor of Psychology
Dissertation Advisor
Chair of Committee

Steven R. Haynes
Assistant Professor of Information Sciences and Technology

Mary Beth Rosson
Professor of Information Sciences and Technology

Richard A. Carlson
Professor of Psychology

John Yen
Professor of Information Sciences and Technology
Associate Dean for Research and Graduate Programs in
Information Sciences and Technology

*Signatures are on file in the Graduate School

ABSTRACT

Intelligent agents and cognitive models are useful for a number of purposes. Unfortunately, limited theory-based tool and language support for the creation of intelligent agents has made it difficult for modelers to create, debug, and reuse agent software. This dissertation explores how to make it easier to create intelligent agents, and especially cognitive models, by taking advantage of established software engineering principles. The benefits of applying software engineering principles to intelligent agent development is demonstrated with the creation of a high-level language and development environment that embodies these principles, and with an evaluation of this language and environment, in use, by students and cognitive modelers.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Fortunately for me, my advisor Dr. Frank Ritter takes advising very seriously. It is not uncommon for him to spend large portions of lab meetings handing out reading material on topics such as the importance of networking at conferences, how to avoid common writing mistakes, and tips for surviving graduate school. Over the past six years, his honest concern for his students and his dedication to mentorship have taught me a tremendous amount about science, scholarship, success, and life.

Thanks Frank.

I was also fortunate enough to have a second mentor in IST, Dr. Steven Haynes, who dedicated a lot of time helping me develop my research interests, and providing me with additional guidance and encouragement when I needed it.

Thanks Steve.

I would also like to thank my parents for instilling in me the importance of education, and inspiring me to pursue academic excellence.

Thanks Mom and Dad.

My deepest gratitude goes to my wife Lisa, whose patience and love kept this research fun.

Thanks Lisa.

Chapter 1

Introduction

This dissertation explains how to make it easier to create intelligent agents, and especially cognitive models, by taking advantage of established software engineering principles. The benefits of applying software engineering principles to cognitive modeling is demonstrated with the creation of a high-level language and development environment that embodies these principles, and with an evaluation of this language and environment, in use, by students and cognitive modelers.

Intelligent Agents and Cognitive Models

An intelligent agent is defined as a piece of software that perceives its environment via sensors and acts on that environment by way of effectors (Russell & Norvig, 2003). As shown in Figure **1-1**, the mapping from an agent's sensor readings to its actions produces intelligent behavior. Due to the complexity of intelligent behavior, the implementation of this mapping is both interesting and challenging. The easier it is to define this mapping, the easier it will be to create agents.

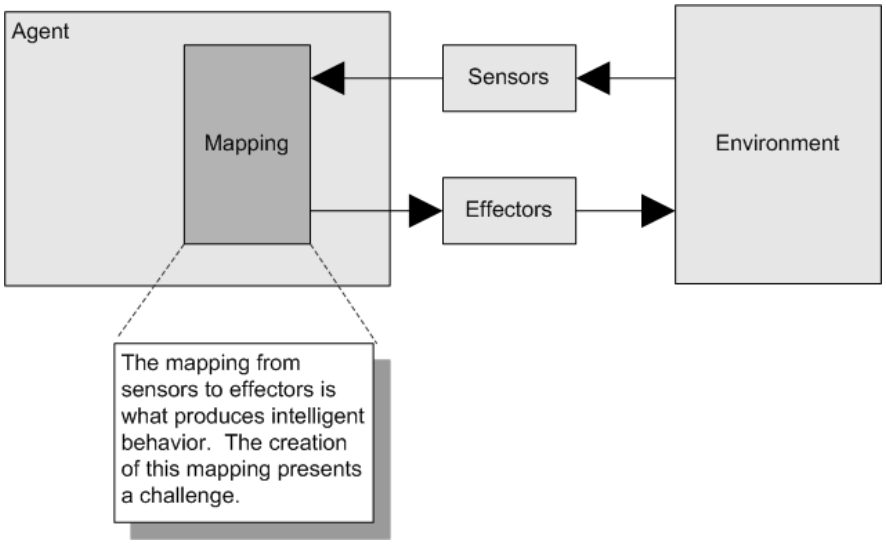


Figure **1-1**: A simple definition of an intelligent agent.

What constitutes an intelligent mapping varies depending on the type and purpose of the agent. As shown in Figure **1-2,** a cognitive model is a special type of intelligent agent with a distinctive definition for intelligence: Cognitive models are agents designed to simulate human behavior. Cognitive models succeed when they *precisely* exhibit human behavior: both the good and bad (Das & Stuerzlinger, 2007; Lindsay & Connelly, 2002; Ritter, Baxter, Jones, & Young, 2000).

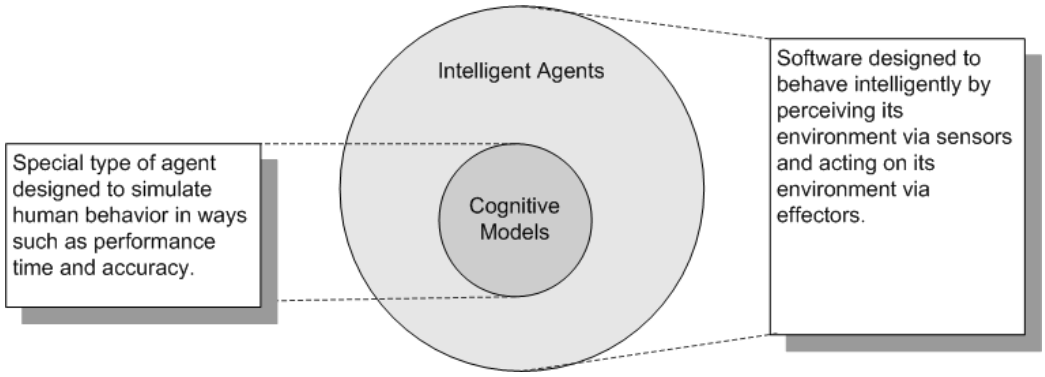


Figure **1-2**: The difference between intelligent agents and cognitive models.

An actual example can help illustrate the difference between an intelligent agent and a cognitive model. Consider the task of dialing a cell phone. An intelligent agent designed to perform this task might utilize a sophisticated algorithm to help it dial numbers in less than one second and with 99% accuracy. A cognitive model, on the other hand, such as the one written by Das and Stuerzlinger (2007), for example, can use psychologically plausible algorithms that lead to cell phone dialing in the same time and accuracy exhibited by a typical novice user. Like a human, the cognitive model will make errors, and this is desirable so that modelers can understand and predict errors, and create better systems.

Both the agent and the cognitive model are useful, but for different reasons. The impressive accuracy of an intelligent agent can help humans dial phones quickly and accurately when driving a car, while the cognitive model can predict commons errors, and their reasons, which can lead to better cell phone design for novice users. Prediction and psychological insight are two important outcomes of a cognitive model that separate it from other types of intelligent agents.

The different requirements of cognitive models lead to unique applications. Cognitive models can be used for training and simulation in domains where actual human participation could be dangerous (Jones et al., 1999). Computer games equipped with opponents that follow predictable scripts can be made more interesting using cognitive models of human adversaries (Laird, 2001). In addition, computer interfaces can be tested more efficiently using models of human users (Ivory & Hearst, 2001; St. Amant & Ritter, 2004).

The many uses for cognitive models lead to a varied set of potential model developers and users. For example, military strategists, pilots, game programmers, human factors experts, and psychologists, all stand to benefit from the use of cognitive models. Unfortunately, cognitive modeling is hard (Pew & Mavor, 1998; Ritter et al., 2003; Salvucci & Lee, 2003; Yost, 1993). I propose that the lack of good software engineering practices in the field of cognitive modeling has made the creation and use of cognitive models more difficult than it need be.

Obstacles Facing Efficient Cognitive Modeling

Figure **1-1** defines agent behavior as the mapping from sensor readings to actions. Given the complexity of human behavior, creating this mapping under the unique constraints of a cognitive model is challenging (Gluck & Pew, 2001; Jones, Crossman, Lebiere, & Best, 2006; Jones & Wray, 2003). Figure **1-3** suggests that two important reasons for this challenge are: (1) there are multiple competing cognitive architectures, (2) cognitive modeling requires a variety of skills. The literature review in Chapter 2 supports the existence of these two obstacles.

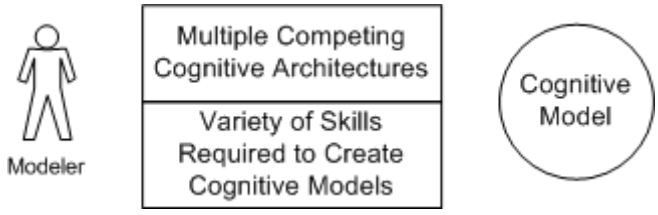


Figure **1-3**: Two important obstacles inhibiting the creation of cognitive models.

One important obstacle facing cognitive modelers is the use of many different cognitive architectures, and their differing theories about how human behavior arises. The use of different architectures makes it difficult to compare, reuse, and integrate models (Gluck & Pew, 2001; Jones, Crossman, Lebiere, & Best, 2006; Jones & Wray, 2003). For example, modelers cannot reuse behavior written for the ACT-R cognitive architecture within the Soar architecture.

A second important obstacle facing modelers is the varied skill set required of those who build cognitive models. Creating cognitive models often requires significant training in many areas (Salvucci & Lee, 2003). In most modeling environments (e.g., ACT-R, Soar, EPIC, JACK, and Jess), creating cognitive models requires computer programming skills; knowledge of psychology; and expertise in the domain being modeled. Unfortunately, few people possess all of these skills. Chapter 2 provides a detailed and supported discussion of this obstacle.

Viewing cognitive modeling as a complicated software engineering problem seems to be a natural approach. Creating complex software is not a new problem, and the software engineering community has developed strong theories and principles about how to solve complex problems with software solutions. This dissertation illustrates how applying these lessons appropriately will help alleviate the obstacles shown in Figure **1-3**.

A Theory for Simplifying Cognitive Modeling

The aim of the research presented here is to alleviate the obstacles shown in Figure **1-3**. This research accomplishes this by applying three broad software

engineering principles to the cognitive modeling task. The focal theory[1] offered here alleviates these obstacles by implementing a high-level language, a maintenance-oriented development environment, and by providing strong support for reuse within and across models. Figure **1-4** depicts this theory and Table **1-1** summarizes this focal theory as a set of hypotheses.

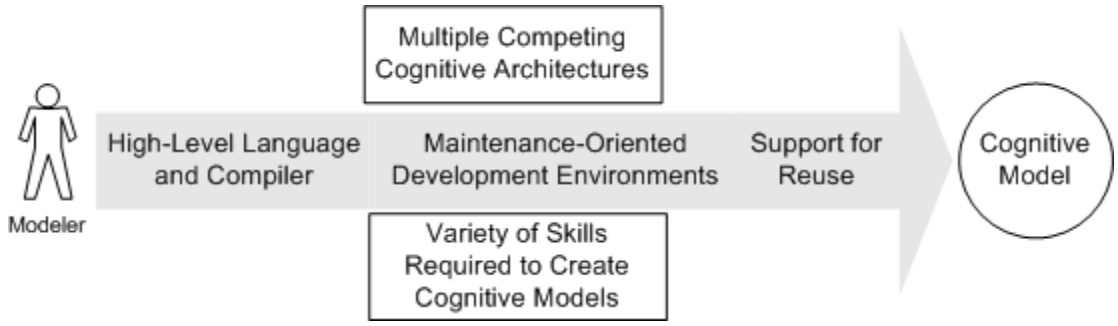


Figure **1-4**: An illustration of the focal theory.

Table **1-1**: The hypotheses forming the foundation of the focal theory.

Hypotheses
1. Cognitive modeling is difficult because there are multiple cognitive architectures in use. This makes it difficult to compare, reuse, and integrate models.
2. Cognitive modeling is difficult because the modeling task requires modelers to acquire a variety of complex skills.
3. Applying three broad software engineering principles to the cognitive modeling task will alleviate the two obstacles listed above. Specifically, the use of high-level languages, maintenance-oriented development environments, and integral support for reuse, will simplify the cognitive modeling task.

[1] Focal theory is used here as defined in Phillips and Pugh (2005): a precise description of the nature of a problem and the focus of the research, along with any related hypotheses.

The next section further introduces this focal theory. Chapter 2 and Chapter 3 provide a more detailed background of the existing research that supports this focal theory.

High-Level Languages and Compilers

Software engineers have used high-level languages and compilers with great success (Auerbach et al., 1991; Beck & Perkins, 1983; Brooks, 1987; Daly, 1977; Maxwell, Wassenhove, & Dutta, 1996). A high-level language and compiler allows the programmer to express a model using abstract terminology. Consider the two methods shown in Table **1-2** used to describe the process of fetching a beer from the refrigerator.

Table **1-2**: Two different ways to describe the process of fetching a beer.

High-Level Description	Low Level Description
1. Walk to fridge	1. Take one step forward
2. Open door	2. Take another step forward
3. Get beer	3. Take another step forward
4. etc., …	4. Raise arm
5.	5. Grasp door handle
	6. Pull
	7. etc., …

The high-level description in the first column is more concise and easier to specify. However, to carry out these instructions a system must translate them into the lower-level counterparts shown in the second column. A piece of software called a compiler often performs this translation

Table **1-3** provides a more realistic example that illustrates the use of a high-level programming language called Java to describe the process of taking the factorial of an integer. A compiler translates the Java description into a low-level intermediate code. Table **1-3** shows the low-level representation that the compiler generates, as well as its high-level counter part. It is clear in this table just how much easier the high-level representation is to create, read, and understand than the low-level representation that is ultimately executed by the Java virtual machine.

Table **1-3**: Two different ways to describe the factorial calculation.

High-Level Description	**Low-Level Description**

public static int factorial(int d)
{
  if (d == 1)
  return 1;
  else
  return d * factorial(d-1);
}
	0: iload_0
	1: iconst_1
	2: if_icmpne 7
	5: iconst_1
	6: ireturn
	7: iload_0
	8: iload_0
	9: iconst_1
	10: isub
	11: invokestatic #25;
	14: imul
	15: ireturn

The focal theory presented here proposes that the same principle will simplify the cognitive modeling process by allowing a domain expert to describe the model's behavior more concisely using a high-level modeling language. If modelers use a high-level language appropriately, compilers can make it easier to compare, integrate, and reuse models across the low-level representations used by architectures.

Of course, it is important to consider the tradeoffs when using high-level languages. For example, a compiler can translate a high-level representation to a low-

level counterpart in many different ways. The compiler chooses one way, but it may not always be what the programmer desired. As a result, many popular high-level representations (e.g., C, C++, and Java) make it possible to override the translation provided by the compiler, and mix low-level representations with high-level representation when needed. The same option should exist for high-level modeling languages. The ability to program at different levels of abstraction is important.

Maintenance-Oriented Development Environments

For complex systems, the process of software maintenance is the most expensive phase of a system's development life cycle (Boehm, 1987; Brooks, 1995). Developers spend a lot of time performing maintenance (Ko, Myers, Coblenz, & Aung, 2006; Tassey, 2002). During maintenance, developers typically perform three types of tasks: fixing flaws, implementing incremental updates, and adapting to changes in the system's environment (Brooks, 1995). Table **1-4** lists examples of these types of tasks.

Table **1-4**: A description of typical software maintenance tasks.

Maintenance Types	Example Tasks
Fixing flaws in the system	1. Fix the system so that it does not crash when saving a file on the network 2. Change the address book so that names are sorted properly 3. When editing the employee list, disable the delete button if no employee is selected
Performing incremental updates	1. Change the existing employee payroll report so that it includes employee numbers 2. Add the ability to print reports in landscape orientation 3. Provide the current date as a default when entering a new meeting note in the system
Adapting to changes in the system's environment	1. Change the report generator to support the newly installed printer 2. Change the interface to take advantage of the new operating system widgets 3. Change the interface to take advantage of new wide-screen monitors

Fortunately, the cost of maintenance has not gone unnoticed by the software engineering community, and modern software development tools have incorporated rich support for the tasks listed in Table **1-4** (Coblenz, Ko, & Myers, 2006; Cubranic, Murphy, Singer, & Booth, 2005; DeLine, Czerwinski, & Robertson, 2005; Ko & Myers, 2004; Lawrance, Bellamy, Burnett, & Rector, 2008; Lewis, 2003; Reiss, 2006). Development tools with this type of support are called maintenance-oriented development environments, and at least one has been successful at reducing maintenance programming tasks by as much as 35% (Ko, Aung, & Myers, 2005). Chapter 3 discusses these tools in more detail.

Modelers are also likely to spend considerable time maintaining their models and should benefit equally from similar tools. The theory presented here suggests that creating cognitive modeling environments that are maintenance-orientated can reduce the obstacles shown in Figure **1-3**.

Support for Reuse

Support for reuse is the final component of the theory illustrated in Figure **1-4**. The software engineering community has continually reaffirmed the value of software reuse (Boehm, 1987; Brooks, 1995; Krueger, 1992). As illustrated in the following example, reuse of software is common in today's applications. Consider the typical web-based application, shown in Figure **1-5**, built almost entirely from existing components.

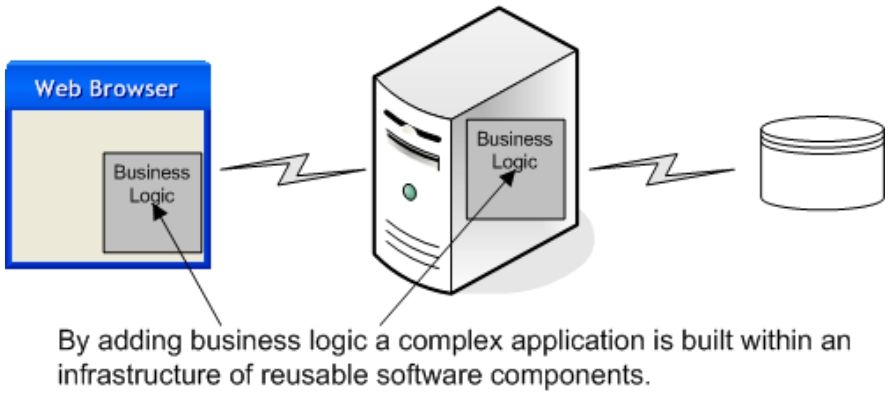


Figure **1-5**: An example of software reuse in a modern web-based application.

On the client side of the application, the graphical user interface is provided by an existing web browser that offers reusable functionality such as scrolling, HTML rendering, printing, and hyperlink navigation. Reusable software components also support the networking infrastructure used by web-based applications. This software

reuse provides developers with reliable and ubiquitous networking for a very low cost. Finally, on the server side, application servers and database management systems provide security, transaction management, and data management.

The combination of all of these reusable components provides an infrastructure that allows developers to spend more time on the business logic unique to the application, and less time developing solutions that already exist.

The hypotheses shown in Table **1-1** state that cognitive modeling can benefit from the similar gains provided by reuse in software engineering. For example, if a renowned vision researcher creates a cognitive model of character recognition, and a famous psychologist specializing in motor skills creates a model of manual dexterity, then a graduate student in human computer interaction should be able to reuse these models to create an agent that uses a keyboard.

This thesis proposes that this type of reuse would help simplify cognitive modeling. However, as will be shown in the following two chapters, existing modeling environments and languages do not support reuse. Although some counter-examples exist (John, Remington, & Steier, 1991; Lewis, Newell, & Polk, 1989), reuse is an infrequent occurrence in the modeling community.

Preview of Contributions and the Structure of this Thesis

This dissertation demonstrates the benefits of applying software engineering principles to cognitive modeling development with the creation of a high-level language and development environment, and with evaluations of this language and environment by

students and cognitive modelers. Several contributions arise from this work in areas such as modeling languages, maintenance-oriented modeling environments, model reuse, and education. This section only previews these contributions. Chapter 8 provides a more detailed discussion.

This work's contributions towards better modeling languages include the only high-level modeling language with both explicit support for a popular theory of cognition and the ability to translate models into multiple architectures. This contribution includes an empirical validation of the high-level language with positive results.

This work's contributions towards maintenance-oriented development environments include the only cognitive modeling environment that has support for simultaneously creating models at three different levels of abstraction: graphically, textually at a high-level, and textually at a low-level. In addition, this environment brings recent research in software engineering to the modeling community to support better code navigation. Results from empirical studies show this environment is both useful and usable.

With respect to model reuse, this dissertation contributes extensions to a popular theory of cognition that provide an additional level of granularity. This added granularity allows for better reuse within and across models. In addition, the high-level language presented here is the only library-centric modeling language. Modelers must create libraries, and can search these libraries for relevant components using methods not yet applied to modeling environments.

This dissertation also makes educational contributions. For example, this work presents new graphical environments that are based on examples given in a popular

textbook and a popular modeling tutorial. This allows students to work in graphical environments that mirror the examples given in learning materials. In addition, Faculty has exposed 89 undergraduates and 9 graduates to modeling using this work, and more will follow in the fall 2008 semester. Students at Tufts University have also used this work to gain a better understanding of high-level behavior representation languages. Finally, the three levels of abstraction that this work supports appear to be useful for teaching low-level rule-based programming. Observations of participants have showed that editing the high-level graphical representation directly, and then viewing the generated low-level productions, is a useful way to learn the low-level representation.

This dissertation is composed of four sections: the focal theory, the background theory, the data theory, and contributions/future work[2]. This chapter introduces the focal theory (shown in Figure **1-4**), which describes the nature of the problem along with any related hypotheses.

Chapter 2 and Chapter 3 give the background theory, which supports the focal theory by synthesizing the current methods used by the cognitive modeling community, with useful software engineering principles.

Chapter 4 thru Chapter 7 provides the data theory, which describes the precise methods used to realize the focal theory and the results of the evaluations of these methods.

Chapter 8 concludes this dissertation with a description of the contributions that this research has made as well as ideas for future work.

---

[2] The use of this structure for a Ph.D. is described and justified by Phillips and Pugh (2005).

Chapter 2

The Current State of Cognitive Modeling

To understand the problems facing cognitive modeling, this chapter presents the techniques currently used by modelers. The following reviews behavior representations methods, low-level behavior representations, popular cognitive architectures, modeling environments, and reuse in cognitive modeling.

Methods of Behavior Representations

To produce software that exhibits intelligent behavior, a representation that describes the behavior is required. Modelers typically develop models using connectionist representations, symbolic representations, or some combination of the two.

Connectionist Representations

Parallel Distributed Processing (PDP) (Rumelhart & McClelland, 1987), also known as neural networks, uses computer programs to represent behavior by mimicking the parallel processing in the brain. Modelers often refer to this method of behavior representation as a connectionist approach because it uses networks of connected neurons.

The brain is made up of some 10 billion neurons and 60 trillion connections between them (Shepherd & Koch, 1990), and these connections form a network that

allows the brain to function. Neural networks work in the brain as follows: Signals propagate from one neuron to the next using electro-chemical reaction. Each neuron connects to other neurons via an axon and the connection takes place at a synapse. The synapse releases a chemical substance that changes the electrical potential of the cell body. When this potential reaches some threshold, a pulse sends electricity down an axon and towards other neurons. This signal changes the electrical potential of other neurons, that may or may not reach their potential, leading to a continuation of the electrical signal (Negnevitsky, 2004).

Neural networks in the human brain learn by changing the threshold at which they fire, and by forming new connections or even migrating to other parts of the network. The flexibility of the network is what makes it possible for the network to change and for humans to learn. Because the changes are linked to feedback about what is "right", the network will change to produce new and more correct outputs (Negnevitsky, 2004).

Artificial neural networks consist of a set of interconnected simulated neurons. Each neuron has several inputs and only one output. All networks have a set of input neurons (the input layer) that can be activated and propagate a signal through a weighted link (the simulated axon). These signals eventually reach a set of internal neurons (the middle or hidden layer). When activated, the internal neurons produce a set of output signals that move across connections that lead to the final output of the network (called the output layer) (Bigus & Bigus, 2001; Negnevitsky, 2004; Ripley, 1993). Figure **2-1** shows a neural network with a single hidden layer.

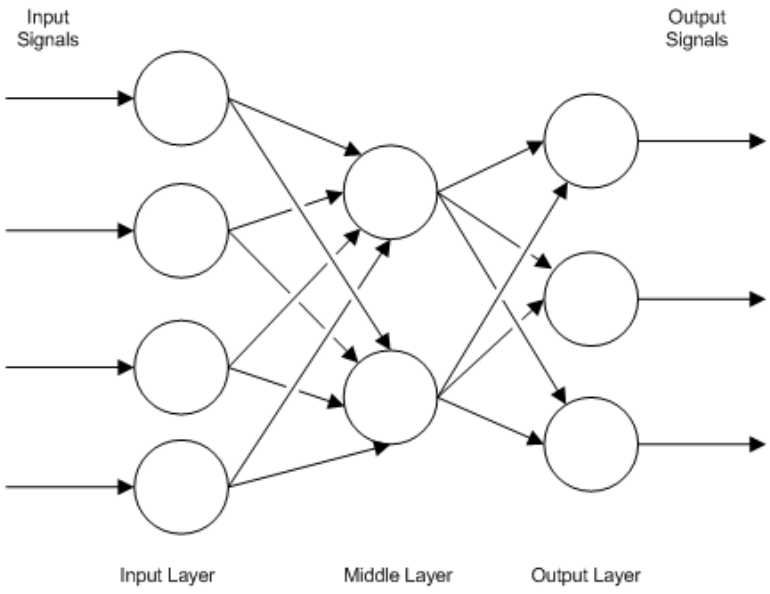


Figure **2-1**: The architecture of a typical artificial neural network (Negnevitsky, 2004).

Using known input/output pairs, researchers can train a network with a single hidden layer to model a continuous function. It takes two hidden layers to model a discontinuous function (Minsky & Papert, 1987; Negnevitsky, 2004).

The network calculates the total input activation energy for a neuron using the weighted sum of the inputs. The neuron does not propagate the signal unless the input signal exceeds some threshold. If the threshold is met, the neuron emits "+1"; otherwise it emits "-1". This type of output (based on some threshold) is called an activation function. Specifically, the activation function described above (+1 or -1 based on a threshold) is called the sign function. There are many other types of activation functions including step, sign, linear and sigmoid. Different activation functions are useful for different types of problems (Bigus & Bigus, 2001; Negnevitsky, 2004; Ripley, 1993).

Researchers use back propagation to train neural networks. Back propagation compares the produced output to a desired output, and the error between these outputs

propagates through the network (from right to left) so that the weights on the links between the neurons change to reduce the error. This back propagation is continued for several iterations until the desired criteria are met (Bigus & Bigus, 2001).

The formula used to adjust the weights during back propagation results in each weight being adjusted using an error gradient; which is calculated using the error signal (the difference between actual and desired output) and the derivative of the activation function (Bigus & Bigus, 2001).

A major advantage of using neural networks to represent behavior is that they are easy to use. Neural networks do not require the explicit encoding of domain knowledge. Instead, the network learns this knowledge with a matrix of coefficients and a set of training data. Modelers present training data to the network, the network produces a result, and the network updates its coefficients based on the correct result according to the training data. In addition, many neural network systems (see, for example, Rumelhart & McClelland, 1987) allow networks to be created and trained by simply entering information using a graphical user interface; no programming is required.

However, neural networks also have disadvantages. The main criticism of neural networks is what Minsky called the problem of opacity (1990). The problem of opacity is that a network represents knowledge by way of numerical weights, and this knowledge has very little apparent meaning to an observer, when related to the domain in which the model operates. Essentially, neural networks serve as a black box that accepts input and produces output, but the relationship between the input and the output is not intuitive. This makes it difficult to understand, explain, and trust behavior generated by connectionist approaches.

Symbolic representations, which the next section covers, are much less opaque. Figure **2-2** illustrates his difference by comparing how symbolic and connectionist representations might represent the concept of an apple.

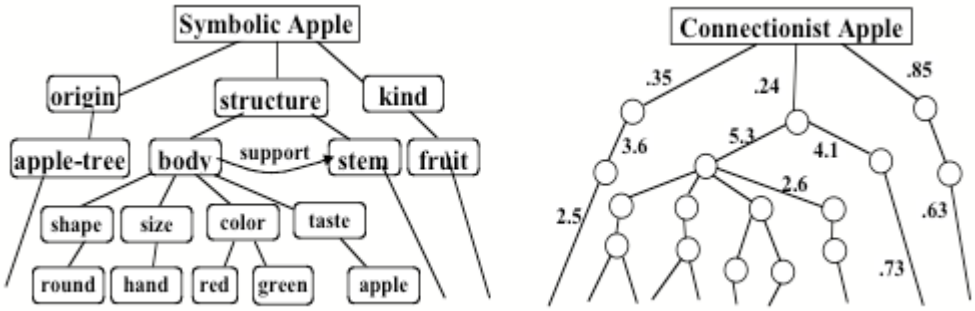


Figure **2-2**: An illustration of the opacity problem with neural networks (Minsky, 1990).

Notice in Figure **2-2** how transparent the symbolic representation is. This transparency can be essential for cognitive models, especially when the purpose of the model is to provide insight into human cognition. It is also important for intelligent agents when users require an explanation of the agent's behavior for the justification of the agent's actions. Unfortunately, the explicit encoding of domain knowledge that provides for this transparency leads to new challenges. The rest of this chapter introduces symbolic representations and the challenges they create, and the rest of this dissertation deals with alleviating these challenges.

Symbolic Representation

Haugeland (1987) provides an excellent description of the use of symbolic manipulation in computational psychology. Haugeland explains symbolic manipulation

by introducing the concept of a formal system. A formal system here consists of a set of symbols; a starting point for the arrangement of the symbols; and a set of rules about how the symbols can be manipulated. Games such as chess and checkers are formal systems.

Table **2-1** shows an example of a formal system in a particular state. This system assigns meanings to symbols such as *H* and *S* (e.g., Hungry and Sally). Relationships can also be modeled using symbols. For example, Sally is hungry might be represented using *H(S)*.

Table **2-1**: An example of a formal system in a particular state.

Symbol	**Meaning**
A	Apple
T	Table
S	Sally
H(S)	Sally is Hungry
E(S, A)	Sally eats the Apple
O(A, T)	The apple is on top of the table

Programs used for symbolic modeling can take the form of production systems (Russell & Norvig, 2003). A production system is a formal system that consists of two categories of symbol groupings: facts and rules. Facts represent declarative memory: memory that aids in the recollection of simple facts similar to those shown in Table **2-1**. Rules, also called productions, represent procedural memory, which is memory that aids in the recollection of procedures. The formal system introduced in Table **2-1** could include the following production: *IF H(S) and O(A, T) THEN E(S, A) and REMOVE H(S)*

This production declares that if Sally is hungry, and there is an apple on the table, then Sally should eat the apple, and as a result, she should no longer be hungry. Rules

like the one shown above interpret and manipulate symbols, and when meaning is assigned to these symbols (e.g., *A* means apple), mental processes such as memory, reasoning, decision making, and problem solving can be modeled (Newell & Simon, 1972).

Artificially Intelligent systems (AI systems) can be production systems that attempt to model expert reasoning without concern for cognitive plausibility. These models do not attempt to generate predictions about human behavior, but instead model perfectly rational thought. One common type of AI system is the expert system, and there are several examples of the successful use of expert systems.

DENDRAL is an expert system that reasons over mass spectrometer data to analyze chemicals. Its creation was funded by NASA to analyze chemicals found in the soil on Mars (Buchanan, Sutherland, & Feigenbaum, 1969). DENDRAL embodied a set of productions based on the experience gained by analytical chemists, and could quickly and accurately analyze chemicals using these rules of thumb. Other successful uses of expert systems include MYCIN (Shortliffe, 1976), an expert system used for medical diagnosis, and PROSPECTOR (Duda, Gaschnig, & Hart, 1979), an expert system used for mineral exploration.

AI systems do not typically use theories of cognition and are less interesting as true models of human behavior.

Low-Level Behavior Representation Languages

Two categories can classify behavior representation languages: low-level behavior representation languages and high-level behavior representation languages. Many computer programs use a low-level programming language called assembly language, which is not that far removed from the language of ones and zeros understood by the computer. High-level languages, on the other hand, contain instructions that map more explicitly to a problem domain, and therefore create a level of abstraction from the actual implementation of the system. Chapter 3 covers in detail the advantages high-level languages have over low-level languages.

There are a wide variety of behavior representation languages in use today (some connectionist, some symbolic, and some a hybrid of both), some are ideal for the creation of intelligent agents, while others are useful for cognitive models. The following is a review of four popular representations (i.e., Jess, Soar, ACT-R, and EPIC) that provides a clear picture of the state-of-the-art in low-level behavior representation and illustrates the problems noted in the introduction.

The languages reviewed here are high-level languages with respect to the level of abstraction they provide above the machine code. However, from the perspective of the cognitive modeler, these languages are considered low-level behavior representation languages because they are programmed using rules rather than higher-level descriptions of behavior (Jones, Crossman, Lebiere, & Best, 2006). The fact that the support for behavior provided by these languages is implicit in a set of rules is one reason why they are so difficult to use.

The Jess Expert System Shell

Jess is an example of a popular expert system that can be used to create a variety of different intelligent agents (Friedman-Hill, 2003). Jess is an expert system shell written in Java. It is fast, lightweight, and easy to integrate with existing Java applications. Programmers do not typically base expert systems written in Jess on psychological theories and therefore are not usually presented as psychologically plausible.

The behavior representation language used by Jess derives from an older rule-based language called CLIPS (Giarratano & Riley, 1998). This functional programming language consists entirely of function calls specified as parenthesized lists. Jess includes strong support for rules and, as stated earlier, this dissertation considers them a low-level behavior representation language.

Unlike traditional programming languages that solve problems in a straightforward and predictable way, rule-based languages, like the one used by Jess, are well-suited for problems that consist of complicated control-flows and a tangled web of possible decisions. Control flow in rule-based languages, like Jess, emerges from the rules governing a particular problem. The creation and encoding of these domain specific rules requires both computer programming skills and domain expertise. Unfortunately, domain experts typically do not possess strong computer programming skills and programmers typically do not possess the required domain expertise.

Figure **2-3** shows the Jess architecture, which is typical of many rule-based systems. The inference engine operates in cycles in which the pattern matcher finds rules

in the rule base that apply to the particular situation, and therefore belong to the agenda. A rule moves to the agenda when its antecedent (the if-part of the rule) has support from facts in working memory. If multiple rules in the agenda, a conflict resolution strategy selects a single rule for execution. Finally, the selected rule it is executed leading to changes in working memory, and the cycle is repeated (Friedman-Hill, 2003).

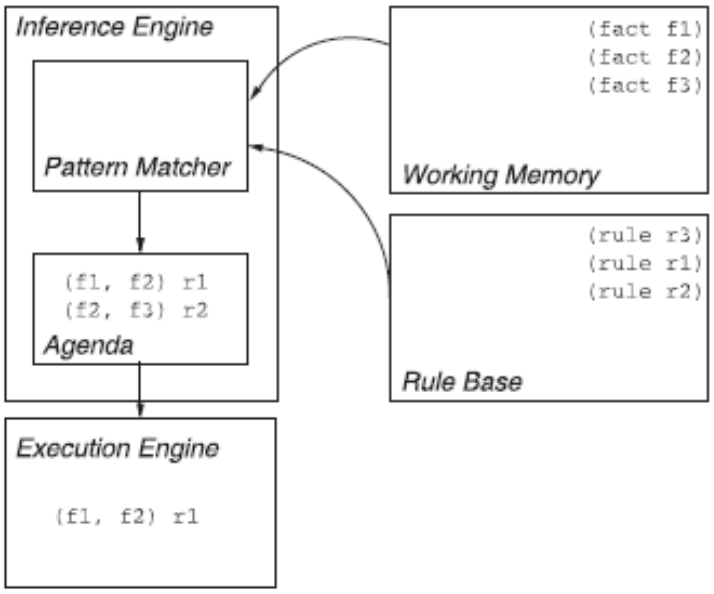


Figure 2.1
The architecture of a typical rule-based system. The pattern-matcher applies the rules in the rule-base to the facts in working memory to construct the agenda. The execution engine fires the rules from the agenda, which changes the contents of working memory and restarts the cycle.

Figure **2-3**: The architecture of the Jess rule-based system (Friedman-Hill, 2003).

Unfortunately, Jess can be difficult to use, especially for people without considerable programming experience. The lack of organization of the rules is a major reason for the difficulty encountered by rule-based programmers. Clancey (1981) argues that the complexity inherent in traditional production systems like Jess is a direct result of the fact that problem solving strategy is hidden implicitly within the structure of the rules (e.g., the order in which they fire).

Clancey (1981) studied the MYCIN expert system from the perspective of a teacher and discovered that "there are points of flexibility in the (rule-based) representation that can be easily exploited to embed structural and strategic knowledge in task rules" (p. 64). As a result, people other than the original programmers find it hard to understand and maintain the rules.

To simplify the creation and maintenance of production systems, there is a need to formalize the structure and strategy used by programmers so that it is explicit in the code. Clancey (1981) asserts that "Making explicit this structural, strategic and support knowledge enhances the ability to understand and modify the system" (p. 1). However, the tradeoff is that the implicit structure of these rules leads to flexibility and emergent behavior.

A major problem when programming Jess (and other rule-based systems) is the effort spent on translating the problem solving strategies used by the domain expert into a set of interrelated Jess rules. An abstract representation, in the form of a high-level language, that made it possible to explicitly represent a given problem solving strategy, could make Jess, and other rule-based systems, easier to program.

Cognitive Architectures

Software systems created to implement unified theories of cognition are referred to as cognitive architectures (Newell, 1990), and they provide the infrastructure to create models that are based on the supporting theory. The theory embedded in these architectures makes it possible to create models that mirror and predict human behavior.

Unlike AI systems (e.g., Jess), cognitive architectures are supposed to provide a theoretical base that is used to express the structure and problem solving strategy of the agent, rather than making this strategy implicit within a set of rules. As will be demonstrated shortly, popular cognitive architectures do not actually achieve this goal.

Despite this overarching theory, current cognitive architectures (e.g., Soar, ACT-R, EPIC) are programmed at the production level and suffer from the same exploitation problem (introduced by Clancey) that plagues rule-based programming languages like Jess. In addition, there are a number of cognitive architectures currently in use, and it is very difficult to compare, reuse, or integrate models created using different architectures.

There are several different cognitive architectures available, including Soar, ACT-R, and EPIC, and the next few sections briefly summarize them. The review provided here is highly representative of the state of cognitive architectures. More detailed reviews of cognitive architectures are also available (Morrison, 2003; Newell, 1990; Ritter et al., 2003).

Soar

Soar is an instantiation of Allen Newell's Unified Theory of Cognition (UTC) (Newell, 1990), and as such, provides the modeler with the mechanisms and structures necessary to use Newell's theory of cognition to model behavior. Soar implements this theory implicitly using rules rather than explicitly in the constructs proposed by the theory.

Soar supports the Problem Space Computational Model (PSCM) (Lehman, Laird, & Rosenbloom, 1996; Newell, Yost, Laird, Rosenbloom, & Altmann, 1991). In Soar, behavior is defined as movement through a problem space (see Figure **2-4**), which is a high-level organizational tool purported to be used by the brain to partition knowledge in goal-relevant ways.

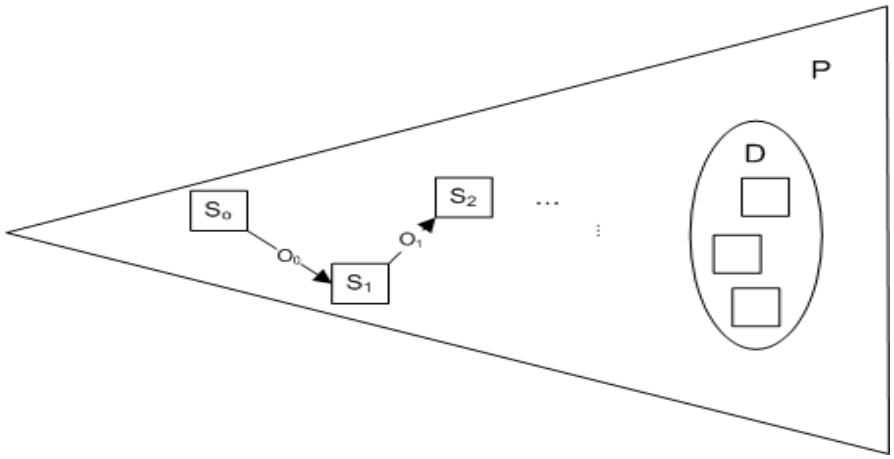


Figure **2-4**: Behavior as movement through a problem space (Newell, Yost, Laird, Rosenbloom, & Altmann, 1991).

As shown in Figure **2-4**, a problem space is a set of states (i.e., S_0, S_1) and a set of operators (i.e., O_0, O_1). A task is formulated when a problem space (P) is adopted, a desired goal (D) is set, and the state of the problem space (S_0) is initialized. The task is attempted as operators are selected and applied to the current state, transforming the problem space into a new state. Finally, the task terminates when the current state matches the goal (Newell, Yost, Laird, Rosenbloom, & Altmann, 1991).

Soar supports two different types of memory: long-term memory (LTM) and working memory (WM). Applying general knowledge in LTM leads to changes to WM

that can result in the application of operators that move the goal-context towards the goal. This process takes place in regular intervals defined as the decision cycle. The decision cycle simulates rational behavior, which consists of applying general knowledge to all know facts in a situation to generate possible responses. Soar evaluates these possible responses and chooses the best response.

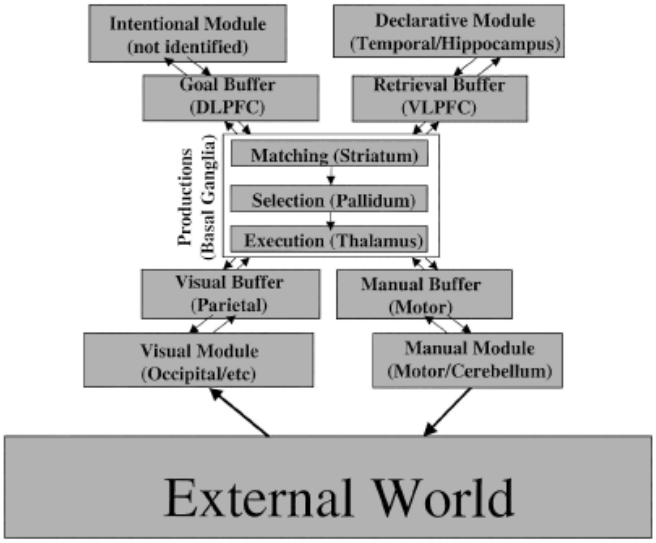
Unfortunately, there is a large gap between the theory defined by the PSCM and the language actually used to program Soar. Despite the high-level approach proposed for the architecture, modelers program Soar at the symbolic level as a production system. The language used by Soar is an extremely expressive low-level behavior representation language that allows for the creation of powerful cognitive models. However, the elements of the PSCM are not obvious when examining Soar productions. If given the code for a Soar model, a novice would not be able to point out the basic PSCM components that form the structure of the model. Greg Yost (1993) described this problem succinctly: "Soar productions do not correspond in any obvious way to PSCM concepts. The productions have a uniform structure with no syntactic differentiation with respect to problem space concepts" (p. 29).

Naturally, the problem discussed above makes it more difficult to create models in Soar. For example, a psychologist that is already well versed in PSCM concepts must learn how to represent these concepts using Soar productions. Because these productions do not map very clearly to the PSCM, this translation can be difficult, and might require skills that psychologists and other domain-experts may not have, or might not care to have.

ACT-R

Developed by John Anderson at Carnegie Mellon University, ACT-R is another popular cognitive architecture that is based on a theory of cognition initially called: Adaptive Control of Thought-Rational (ACT-R) (1993). ACT-R models rationality using a cost-benefit model of decision-making. ACT-R models choose between strategies by maximizing the probability of success and minimizing the costs in computation (Morrison, 2003).

ACT-R supports declarative and procedural knowledge. ACT-R represents declarative knowledge using chunks and procedural knowledge using rules. ACT-R allows for a modular representation of behavior, where each module is responsible for a specific function. These modules communicate using a central production system and by placing information in data in buffers. The ACT-R theory consists of several modules that communicate via buffers and a central production system (Anderson et al., 2004). Figure **2-5** shows these modules, buffers, and production system. Interestingly, many of the architectural pieces map directly to regions of the brain.



(DLPFC stands for dorsolateral prefrontal cortex; VLPFC stands for ventrolateral prefrontal cortex)

Figure **2-5**: The organization of information in ACT–R 5.0 (Anderson et al., 2004).

ACT-R is a hybrid system in which the symbolic reasoning of a formal system is teamed with sub-symbolic connectionist learning to produce behavior. As described earlier, sub-symbolic methods rely on assigning weights to various components (e.g., facts or rules) and those components become active only when their weights reach certain activation levels. By adjusting the weights based on feedback, the behavior of the system changes, resulting in learning.

The programming language used by ACT-R is implemented in Lisp (McCarthy, 1960), which is a functional programming language popular in mathematics and artificial intelligence. Similar to Jess, Lisp code consists of functions written as parenthesized lists. Lisp does not contain a rule-based component. Instead, ACT-R augments Lisp with constructs that add chunks and rules. ACT-R interprets code resulting in rule-based processing that forms the ACT-R production system.

Like Soar, modelers program ACT-R at the rule level. As a result, some of the higher-level cognitive constructs in ACT-R are not as explicit as they could be with a high-level language. This makes it difficult to ascertain the structures and problem solving strategies used by an ACT-R model when looking at the ACT-R Lisp code. This can be especially difficult for modelers without extensive programming skills. However, understanding and utilizing the psychological theory supported by ACT-R often requires knowledge of psychology, and implementing a model for a specific domain often requires a subject matter expert. The need for programming skills, knowledge of psychology, and domain knowledge complicates the model creation process.

EPIC

The Executive Process Interactive Control Architecture (EPIC) is another popular cognitive architecture (Kieras & Meyer, 1997). The design of EPIC couples information processing, and perceptual and motor activity, with a cognitive theory of procedural skill. EPIC provides perceptual processors, such as a visual processor and an auditory processor, and motor processors for the hands, eyes, and vocal organs. In addition, EPIC simplifies the interaction between the model and the computer interface by simulating screen elements and keys.

Similar to ACT-R and Soar, EPIC is a production system. Modelers are required to provide a description of the simulated task environment, task-specific sensory parameters, and a set of productions (rules). EPIC represents productions using the Parsimonious Production System (PPS) interpreter. There is no high-level language

support for the theory supported by EPIC: modelers are required to program at the rule-level. Consequently, EPIC is also difficult to program.

Summary

Table **2-2** summarizes the previous review of low-level behavior representation languages. Importantly, not one of the languages shares the same theory of cognition. This makes it difficult for modelers to compare, reuse, and integrate models across architectures. In addition, all languages require models to be represented using rules that form a low-level representation of the actual theory the architecture supports. This results in languages that are difficult to use, especially for modelers with little or no programming experience.

Table **2-2**: A summary of low-level behavior representations.

Architecture	Theory	Language	Type
Jess	None	Rule-based	Symbolic
Soar	PSCM	Rule-based	Symbolic
ACT-R	ACT-R	Rule-based with activations	Hybrid (Symbolic and Connectionist)
EPIC	EPIC	Rule-based	Symbolic

High-Level Behavior Representation Languages

The low-level behavior representation languages just reviewed do not provide explicit support for the structures and problem solving strategies used by the modeler to

produce agent behavior. In all these cases, the structures and strategies are implicit within a set of rules.

The absence of higher-level languages that incorporate cognitive theory as an explicit object in the language (instead of using rules) has not gone entirely unnoticed (Ritter et al., 2006). In response, researchers have begun developing higher-level languages that simplify the encoding of behavior by creating representations that map more directly to a theory of how behavior arises in humans. In other words, the theory is explicit in the language.

The following is a review of current high-level languages used to develop cognitive models. While this is not a complete review of all high-level modeling languages, it is representative of the current state of high-level languages in use.

RAPs

A good example of a high-level agent programming language that reduces the role of rules is the language used by Reactive Action Packages (RAPs). RAPs is a plan and task representation language that is designed to specify tasks and plans in a way that is flexible enough to deal with the uncertainty of an agent's interaction within a complex and unpredictable world (Firby, 1989).

The RAPs language makes it possible to create a hierarchical set of building blocks that combine in different ways to generate a plan for achieving a task. As the environment changes, different methods of achieving a task make the plan dynamic.

Each RAP divides into three parts: a task goal, a success clause that determines if the goal is satisfied, and one or more methods that accomplish the goal. Each method further divides into two associated sections: the steps involved in the method and the context in which those steps apply.

One key lesson from RAPs is that the high-level language provided by RAPs decreases the degree of programming skills needed to create agents. Importantly, RAPs accomplishes this without the use of explicit rules. In addition, RAPs can be combined easily to form more complicated, dynamic behaviors (Firby, 1989).

RAPs also uses language constructs to accomplish common needs, which can be difficult to implement using rules. For example, consider a looping construct, which is not only difficult to implement in rules, but is also difficult to recognize within a rule-based program: RAPs makes looping constructs explicit using a simple REPEAT-WHILE language construct. According to Firby (1989), "The primary reason for having an explicit repeat clause is thus to notify a planner that the RAP is explicitly designed to loop" (p. 129). Firby's statement may seem like common sense, but explicit looping structures like REPEAT-WHILE are not common in rule-based cognitive modeling languages.

JACK

Created by the Agent Oriented Software Group, JACK implements the Belief-Desire-Intention (BDI) framework (Norling, 2004). In the BDI framework, an agent defines a set of beliefs, desires, intentions, and plans. Driven by goals, and what it

believes about the world, an agent formulates intentions to execute certain plans. Intentions lead to the execution of plans that eventually lead to completing a goal (Norling, 2004).

The programming language used by JACK is a modified version of Java, which is a modern object-oriented programming language. Because JACK is not rule-based, it does not suffer from the problems discussed by Clancey (1981). In other words, JACK directly supports a framework (BDI) that makes the problem solving strategy explicit, as opposed to hiding this strategy implicitly within a collection of rules.

In JACK, the BDI level of abstraction was added directly to the Java programming language (Howden, Ronnquist, Hodgson, & Lucas, 2001) providing direct access to the high-level framework underlying JACK. BDI concepts such as belief sets and plans are explicit in the JACK language, making the code easier to comprehend and reducing the problem of hiding structure and strategy within rules.

Unfortunately, because BDI is derived from a folk-psychological view of reasoning, or an ordinary person's idea of cognition that may not be based on sound psychological theory (Goldman, 1993), JACK is not well-suited for creating psychologically plausible models. While JACK provides a high-level language that is suitable for a wider range of users (not just experienced rule-based programmers), it is not an ideal language for creating cognitive models. A more comprehensive architecture that accounts for the low-level details of human cognition may be better suited.

Efforts are underway to augment JACK to be more psychologically plausible (Norling & Ritter, 2001). COJACK is an agent-based cognitive environment that extends JACK with psychologically plausible human variability (Norling & Ritter, 2004; Ritter &

Norling, 2006). COJACK constrains JACK agents using a set of parameters that vary agent behavior across all agents, or within an individual agent. COJACK is a good example of how a cognitive overlay provides new levels of abstraction, such as support for individual differences. The ability to overlay different levels of abstraction on top of a high-level behavior language is an important lesson from the COJACT work.

GOMS-Based Languages

G2A (St. Amant, Freed, & Ritter, 2005) is a high-level representation language that allows for the creation of ACT-R models using the Goals, Operators, Methods, and Selection Rules (GOMS) description. GOMS is a high-level behavior representation language that can be used to model skilled users performing error-free tasks. GOMS excels at individual, user-paced, passive systems (John, 2003). One of the strengths of GOMS is its simplicity. The GOMS language is abstract and maps directly to the task. However, this simplicity is also a weakness: GOMS is not well suited for modeling novice users learning or performing complex, interactive tasks. This is because GOMS does not model the errors that novice users often make, the change in performance that occurs because of learning, or the exceptions that can occur during interactive tasks (John, 2003).

The use of GOMS by G2A allows modelers to create behavior using an explicit representation of a theory. G2A translates this representation into the low-level rules required by ACT-R. One naturalistic experiment has shown that G2A has significantly

reduced the amount of effort required to produce ACT-R models (St. Amant, Freed, & Ritter, 2005).

Currently, G2A only supports the creation of ACT-R models. In addition, the high-level language supported by G2A (GOMS) is limited to modeling expert behavior in simple tasks. However, the significant speed up in ACT-R model development afforded by G2A, along with its use of a high-level language and compiler, provides important lessons for the work presented in this dissertation.

ACT-Simple is another example of a high-level language designed to simplify cognitive modeling (Salvucci & Lee, 2003). ACT-Simple is similar to G2A in that it provides a GOMS-based higher-level language that can be complied into low-level ACT-R rules. ACT-Simple has been shown to be useful for quickly building models that predict expert performance (Salvucci & Lee, 2003).

ACT-Simple and G2A are good examples of how combining the simplicity of an abstract behavior representation language, with the complexity of a lower-level cognitive architecture, can simplify the modeling task. Figure **2-6** illustrates how the languages and compilers used by ACT-Simple and G2A provide a layer of abstraction above ACT-R that simplifies the programming task.

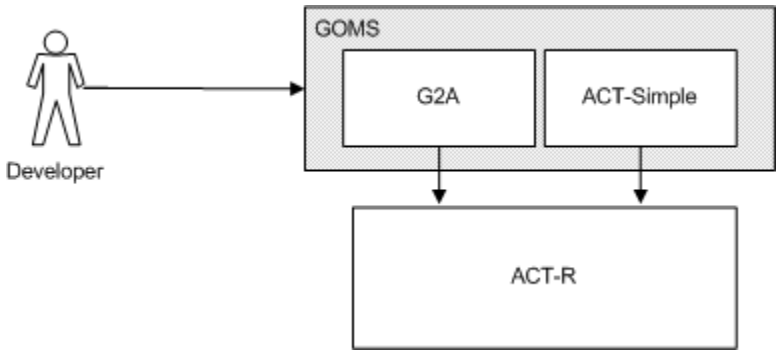


Figure **2-6**: G2A and ACT-Simple provide a GOMS-level abstraction on top of ACT-R.

Unfortunately, both ACT-Simple and G2A suffer from the same problem: the higher-level languages they use derive from GOMS, which is limited in the types of tasks it can model. A richer cognitive theory would be useful when modeling certain tasks.

However, the concept of implementing a high-level language on top of an existing architecture is an important step towards simplifying cognitive modeling. Salvucci and Lee provide three significant benefits to this approach: theoretical consistency, inheritance of architecture, model refinement, and model integration (2003).

Theoretical consistency specifies that, regardless of the level used to create the model, the underlying theory of cognition (for ACT-Simple this is ACT-R) should be consistent.

Inheritance of architecture specifies that the simplicity of the higher-level representation does not necessarily limit its predictive power. By compiling the simpler model into low-level productions for a sophisticated architecture like ACT-R, modelers get predictions that go beyond what is in the higher-level representation.

Model refinement allows the modeler to resort to working in the low-level language when the higher-level language is not sophisticated enough to handle issues that

arise during model creation. This allows the modeler to create a model using the high-level representation and then refine it when they need lower-level control. This is an essential component of any high-level representation, and is another example of the need to allow programming at multiple levels of abstraction.

Model integration relates to model reuse, which the end of this chapter covers in more detail. All three of these benefits are important for reducing the degree of programming skills required by cognitive modelers, and can help with the integration and reuse of models written across theories and architectures.

TAQL

One major difficulty with programming Soar is the low-level production language it uses. As mentioned earlier, Soar's language does not explicitly express the PSCM theory. This problem was addressed by Greg Yost (1993) using the Task Acquisition Language (TAQL).

TAQL is a high-level language designed to map directly to the PSCM and to compile into Soar productions. Yost chose the PSCM for two reasons: It represents the underlying theory used by Soar, and it provides a simple and flexible problem solving method.

The evaluations done by Yost on TAQL's effectiveness were encouraging. The use of TAQL significantly reduced the amount of time developers spent creating Soar productions, and this improvement persisted as the problem size and complexity increased (Yost, 1993).

TAQL maintenance ceased when Soar moved to C and Tcl/Tk. As a result, TAQL only supports Soar 5 (a much older version of Soar). In addition, the syntax of TAQL is complex (Ritter, 1992) and does not include a visual development environment to help developers with this complexity. However, TAQL showed that a high-level language, based on the PSCM and used for Soar development, could be effective.

HTAmap

HTAmap (Heinath, Dzaack, Wiesner, & Urbas, 2007) is a high-level representation based on the Sub Goal Template (SGT) task analysis method (Ormerod & Shepherd, 2004). Using HTAmap, SGT task descriptions are transformed into an intermediate XML-based representation called Cognitive Activity Patterns (CAP). These patterns can be compiled into lower-level ACT-R productions. Like many of the high-level languages introduced here, HTAmap is designed to make modeling available to a wider range of users.

COGENT

COGENT is a graphical environment for creating cognitive models. The primary goal of COGENT is to provide a tool that simplifies the cognitive modeling process (Cooper & Fox, 1998).

With the use of a box-and-arrow notation, COGENT makes it possible for developers to model cognitive functioning quickly, and with little or no programming.

Using COGENT, a modeler can build a model based on a set of cognitive processes. COGENT processes can use either symbolic or connectionist representations. These processes are connected using communication links (see Figure **2-7**). Once a model is sketched, it is configured and then executed within the COGENT environment to analyze its behavior (Cooper & Fox, 1998; Cooper & Yule, 2007). The COGENT environment provides several different visualizations to support behavior analysis.

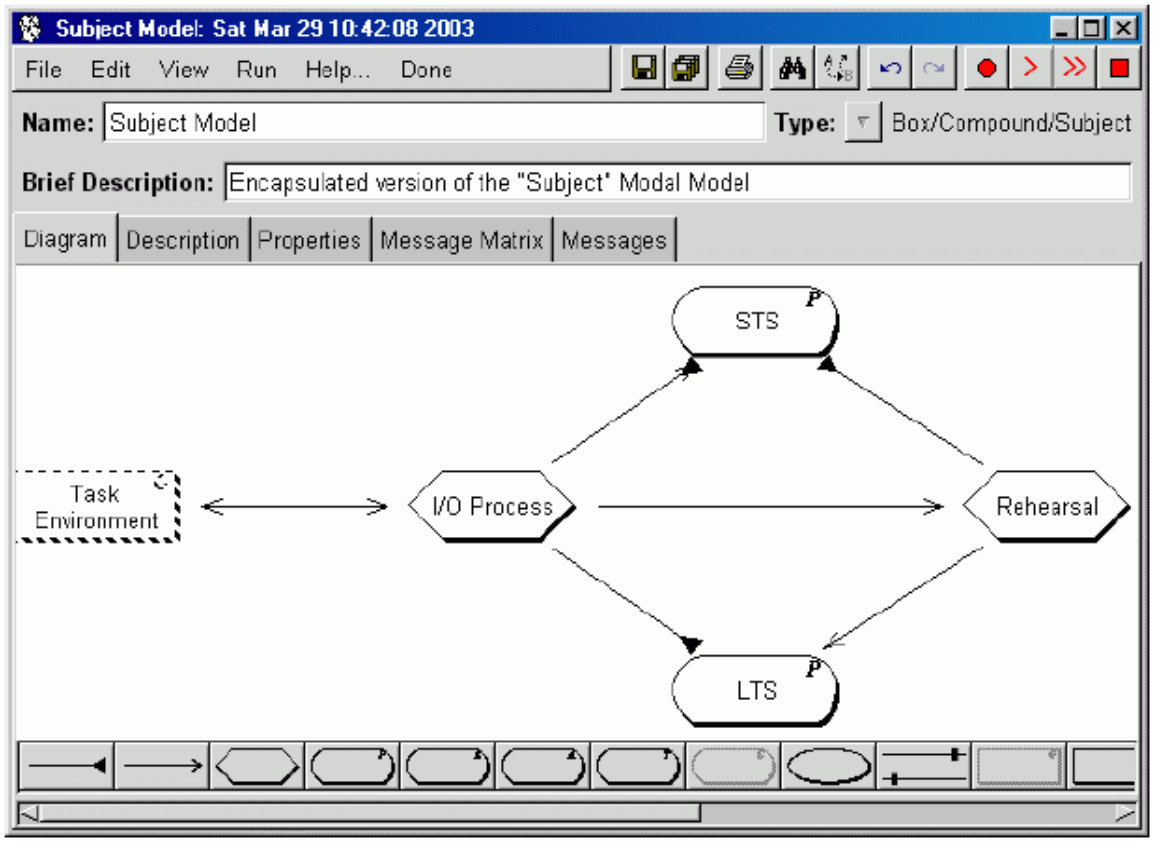


Figure **2-7**: A box-and-arrow diagram of the Modal Model of memory created using COGENT (Cooper & Yule, 2007).

COGENT supports several different model components such as rule-based processes, memory buffers, connectionist networks, I/O sources and sinks, sockets, and inter-module communication links (Cooper & Yule, 2007).

COGENT is a modeling environment and graphical language, not a cognitive architecture. As a result, COGENT does not enforce a set of architectural constraints based on psychological theory. Instead, COGENT allows modelers to build their own theory of cognition within COGENT and create models based on these constraints. For example, in theory COGENT could be use to implement existing architectures such as ACT-R and Soar (Cooper & Yule, 2007).

The COGENT project provides many lessons. Making cognitive modeling easier and more accessible is a goal shared by both COGENT and this dissertation. COGENT approaches this problem using an environment in which users sketch models graphically, thus allowing programming at a higher level of abstraction. In addition, modelers build COGENT models from a library of existing components. This flexibility makes it possible for models that utilize different theories of cognition to interact within the same environment, and can alleviate the problem of reuse and integration across the many different theories currently in use. Lastly, COGENT provides visualizations to help with the analysis of running models. The benefits achieved by COGENT using graphical development environments, component reuse, theory integration, and visualizations are valuable.

Unfortunately, COGENT does not take the idea of a graphical development environment far enough. As will be discussed in Chapter 3, recent research in software engineering proposes the use of maintenance-oriented development environments that

offer more than just model sketching (Ko, Aung, & Myers, 2005; Ko & Myers, 2004; Lewis, 2003; Reiss, 2006; Robillard, Coelho, & Murphy, 2004). In addition, COGENT models can only be graphical. There is no option to create models using a lower-level representation, which as discussed in Chapter 1 and reinforced by Salvucci's concept of model refinement, is important. Again, we see the lack of support for programming at multiple levels of abstraction.

Lastly, COGENT is a meta-architecture. It does not provide a cognitive architecture for its modelers, nor does it support existing architectures such as Soar and ACT-R. Modelers must implement their own architectural constraints, and cannot easily integrate their models with more established, theory-based architectures.

HLSR

The High Level Symbolic Representation (HLSR) project aims at creating a formal language that encompasses a wide variety of modeling tasks using a variety of cognitive architectures. Importantly, HLSR strives to make it easier to create models by providing high-level language support for common modeling problems. HLSR consists of three core elements: relations, transforms, and activation tables (Jones, Crossman, Lebiere, & Best, 2006).

A compiler exists that translates HLSR to an underlying architecture. Currently, HLSR creates Soar and ACT-R productions. HLSR supports two architectures using something called microtheories, which describe how an HLSR architectural construct will compile into a specific architecture.

The ability of HLSR to use microtheories to remove the architectural dependencies from the code that represents the cognitive model is an important accomplishment. This allows modelers to implement a model once, yet executed in different architectures. In addition, the high-level language used by HLSR is rich enough to model complex behavior.

However, the architectural neutrality of HLSR results in the lack of explicit support for a popular unified theory of cognition (e.g., PSCM and ACT-R). Instead, microtheories hide this theory. Because modelers often use one of these theories of cognition to understand how to perform a task, they must take an extra step to translate their understanding of the task into a description using HLSR. This gap between the modeler's conceptualization of behavior, and its realization in the high-level language, is exactly what a high-level language is supposed to prevent.

Summary

Table **2-3** summarizes this review of high-level behavior representations. Notice that only three of these high-level representations explicitly support a theory of cognition, and two of these three are based on GOMS, which is limited in scope. HLSR supports programmable theories using "microtheories", which makes it extremely flexible, but removes explicit theory support in the language.

Table **2-3** also shows that only one representation can compile productions for multiple architectures, which is required if modelers want to be able to compare, reuse, and integrate behavior across architectures. It is clear from the review that the cognitive

modeling community lacks is a high-level representation language that explicitly supports a well-known theory of cognition, allows for the reuse of behavior, compiles into productions for multiple well-tested architectures, supports model refinement, and allows programming at several different levels of abstraction.

Table **2-3**: A summary of high-level behavior representations.

Representation	Explicit Theory Supported	Architectures Supported
RAPs	Plan based	RAPs
JACK	BDI	JACK
G2A	GOMS	ACT-R
ACT-Simple	GOMS	ACT-R
TAQL	PSCM	Soar
HTAmap	STG	ACT-R
COGENT	None	COGENT
HLSR	Programmable	Soar and ACT-R

Cognitive Modeling Environments

While high-level languages have helped simplify the development of intelligent agents and cognitive models, their use alone is not enough. Model development can be further simplified using development environments that simplify the programming task. These environments help developers build behavior by transforming the task of rule creation into an interactive process using a graphical user interface (GUI). This reduces the modeler's need for advanced programming skills. In addition, some of these environments help developers maintain their agents using visual debuggers and code

navigation techniques. Software developers call these environments Integrated Development Environments (IDE).

Many of the behavior representation languages discussed above are combined with development environments to simplify the development process. While helpful, these environments do not implement recent findings by researchers in the software engineering community (Chapter 3 discusses these findings in detail). The following is a review of existing cognitive modeling and intelligent agent environments. Chapter 3 discusses the recent software engineering developments that these environments are lacking.

Jess Environments

There are several development environments available for use with the Jess expert system shell. For example, JessPad (http://www.ida.liu.se/~her/JessTab/) provides integration between Jess and a popular ontology editor called Protégé. This allows Jess developers to create knowledge using the graphical interface provided by Protégé.

Another Jess development environment is the JessDE (http://herzberg.ca.sandia.gov/jess/docs/70/eclipse.html). JessDE is a plug-in for the popular Java development environment Eclipse (www.eclipse.org). JessDE provides low-level programming features such as syntax coloring, code assistants that automatically find and correct syntax errors, automatic code formatting, graphical code navigation, and an integrated debugger.

An ACT-R Environment: CogTool

CogTool (John, Prevas, Salvucci, & Koedinger, 2004) is a graphical environment that allows user interface designers to develop a GUI and at the same time, a cognitive model that predicts skilled performance of a user utilizing the GUI. Using CogTool, complex interfaces can be mocked-up using storyboards that demonstrate the interface and how users can interact with it. From this storyboard, CogTool generates ACT-Simple code that a compiler translates into ACT-R productions (Figure **2-8**). The resulting ACT-R represents a cognitive model that utilizes the interface. This allows for the rapid creation and evaluation of user interfaces without the need for programming or expensive user studies.

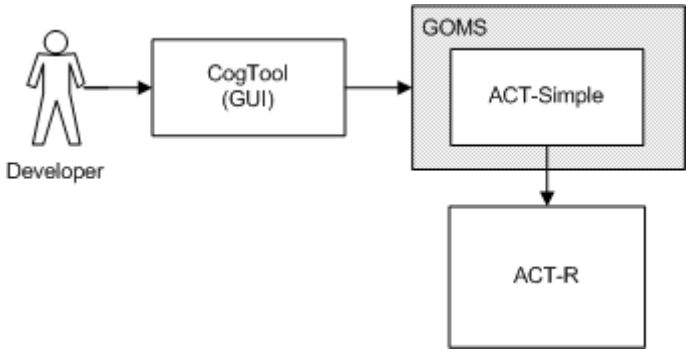


Figure **2-8**: CogTool provides a graphical environment that produces ACT-Simple code automatically.

Because CogTool does not require any programming, this system promises to bring modeling to a community (UI designers) that has been limited in its ability to test interfaces using simulated users. Unfortunately, because CogTool generates ACT-Simple, it is subject to the limitations of GOMS discussed earlier. In addition, CogTool focuses primarily on creating and testing user interfaces and does not apply to a wider

range of model types. Finally, CogTool only supports programming at the visual level, and therefore does not support model refinement.

Soar Environments

Soar developers also benefit from development environments. For example, there is ViSoar (Hirst, 1999), which is a Tcl/Tk dialogue-driven interface for generating Soar code automatically, debugging Soar code, and reverse engineering existing Soar productions. ViSoar provides a GUI interface for creating Soar productions.

Visual Soar (Laird, 1999) is a Java based development environment that simplifies the creation of Soar productions. Visual Soar makes it easier to maintain a collection of Soar source code files and a hierarchy of Soar operators. In addition, Visual Soar supports a Data Map editor that allows some code generation and helps add much needed type checking when working with elements in working memory. Lastly, Visual Soar makes it easier to write Soar productions by providing syntax highlighting and other formatting techniques.

A recent addition to the set of available Soar development environments is Soar IDE (Knudsen, Quist, Ray, & Wray, 2007). Soar IDE is an Eclipse plug-in similar to JessDE, that it provides syntax coloring, code assistants that automatically find and correct syntax errors, automatic code formatting, graphical code navigation, and an integrated debugger.

Unfortunately, all Soar IDEs are designed for experienced Soar programmers. Not one is built to support users with a variety of skills and experience.

Summary

Table **2-4** summarizes the review of cognitive modeling development environments. Importantly, this table shows that not one of the environments provides good support for programming at multiple levels of abstraction. In addition, the environments discussed above do not implement recent findings by researchers in the software engineering community. Chapter 3 presents the recent software engineering work that can greatly improve the effectiveness of many of these environments.

Table **2-4**: A summary of cognitive modeling environments.

Environment	Architecture	Support for programming at multiple levels of abstraction?
JessPad	Jess	No. Only supports graphical programming.
JessDE	Jess	No. Only supports text-based programming of the low-level productions.
CogTool	ACT-R (via ACT-Simple)	No. Only supports graphical programming.
ViSoar	Soar	No. Only supports dialogue-driven programming.
Visual Soar	Soar	No. Only supports text-based programming of the low-level productions.
Soar IDE	Soar	No. Only supports text-based programming of the low-level productions.

Reuse in Cognitive Modeling

Unfortunately, compared to traditional software development there has been very little reuse of models of human behavior (Jones, Crossman, Lebiere, & Best, 2006). A major reason for this is that low-level representation languages are rule-based, yet many of these languages do not provide support for the reuse of rules.

For example, modern programming languages such as Java and C# make it possible to package a set of classes into libraries that can be easily included and reused by other systems. In addition, these libraries contain standard interfaces that allow development environments to discover the contents of the libraries and how to share them.

Engineers have also applied the concept of self-describing reusable libraries to the World Wide Web in the form of web services. These services are discoverable and self-describing. Unfortunately, none of the low-level behavior representation languages described here (e.g., Jess, Soar, ACT-R, EPIC) provide support for this type of reuse.

The nature of rule-based representation languages also makes it difficult to reuse the components of a rule. For example, the two rules shown in Table **2-5** share a similar condition: an enemy tank that is nearby and aggressive. However, what defines a tank as nearby and aggressive can consist of a complex set of sensor readings and logical tests. In addition, these definitions may change over time as operational procedures evolve or sensors become more sophisticated. Unfortunately, rule-based languages require that the modeler repeat the conditions in every rule in which they are used. As a result, when the definition of nearby or aggressive changes, all the rules that rely on these definitions must

also change. The same thing is true for actions contained in the consequent of a rule. This makes it difficult to reuse conditions and actions within and across models.

Table **2-5**: Problems with reuse in rule-based languages.

Rules That Share Conditions
If an *enemy tank* ***is nearby and aggressive*** and this tank is in trouble then retreat.
If an *enemy tank* ***is nearby and aggressive*** and this tank is healthy then attack the enemy tank.

Reuse in rule-based languages is also difficult because of the dependencies that exist between the conditions in the rule's antecedent and the actions in the rule's consequent. For example, in the second rule shown in Table **2-5** the attack action relies on the condition identifying the enemy tank. This dependency between conditions and actions makes it difficult to reuse the condition or the action within a new context (e.g., the attack action must be used in a rule whose antecedent identifies an enemy tank).

As languages become more high-level, reuse becomes more common (Brooks, 1987). For example, because JACK uses object-oriented concepts supported by a robust object-oriented language (Java), it is easier to create and share libraries of JACK behavior. In addition, HLSR provides support for the creation of named relations and transformations that modelers can reuse within and across models.

Summary

This review covers a set of modeling architectures, behavior representations, and environments that is representative of the current state of cognitive modeling. In

addition, it illustrates the current problems with cognitive modeling and several recent developments that help simplify modeling.

One major problem presented here is that low-level symbolic languages are the norm in the most popular and powerful architectures (see Table **2-2**). This increases the required skill set for modeling and reduces the population of potential modelers (Pew & Mavor, 1998; Ritter et al., 2003; Salvucci & Lee, 2003; Yost, 1993). In addition, the architectures all support different underlying theories, which makes it very difficult for models written for different architectures to be compared, shared, and reused (Gluck & Pew, 2001a; Jones, Crossman, Lebiere, & Best, 2006; Jones & Wray, 2003).

The good news is that high-level languages are emerging that help support reuse and more closely map the theory or framework to the representation (Ritter et al., 2006). Some of these languages (e.g., RAPs, and JACK) also support higher-level abstractions such as plans, beliefs, desires, and intentions.

However, only one language reviewed here (HLSR) creates models that run in more than one architecture (see Table **2-3**). The cognitive modeling community lacks is a high-level representation language that explicitly supports a well-known theory, allows for the reuse of behavior, compiles into productions for multiple well-tested architectures, supports model refinement, and allows programming at multiple levels of abstraction.

In addition to high-level languages, development environments are also important for supporting a larger audience of modelers. Once again, there is some good news. Development environments for modeling are emerging (e.g., JessDE, Visual Soar, and CogTool). Unfortunately, there is a strong need for environments that strike a better balance between the support for experienced modelers (e.g., Visual Soar, Soar IDE) and

environments that support end-user programmers (e.g., COGENT and CogTool) by supporting different levels of programming to ease the transition as users gain experience.

There are many lessons here, and their impact on the direction of this dissertation will become even more evident in Chapter 3, which reviews important software engineering principals. Research in software engineering is rich in methods for creating high-level languages that simplify programming, support reuse, and work with multiple platforms. In addition, there is significant software engineering research that can help with creating environments that support both the beginner and experienced programmer. The next chapter reviews this research and completes the theoretical foundation for this dissertation.

Chapter 3

Important Lessons from Software Engineering

The focal theory offered in this dissertation suggests that utilizing well-established software engineering principles can simplify cognitive modeling and intelligent agent development. This chapter reviews important literature about software engineering principles such as high-level languages and maintenance-oriented development environments. By combining the lessons from Chapter 2, this chapter builds a foundation for the focal theory presented in this dissertation.

High-Level Languages

Software developers regularly use high-level languages because they simplify the creation of complex systems. The influence high-level languages have had on programming is made clear by Brooks (1987, p. 14):

> Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility.

Empirical evidence of the advantages of high-level languages also exists in the literature. For example, as far back as 1977 a 20% reduction in development time was reported by GTE Automatic Electric Laboratories when development was done using a high-level language as opposed to assembly language (Daly, 1977). In addition, a survey

of professional programmers showed that developers rated high-level languages as an effective method for software development (Beck & Perkins, 1983). More recent empirical evidence also exists in the literature (see, for example, Maxwell, Wassenhove, & Dutta, 1996).

Unfortunately, as mentioned in Chapter 2, many of the cognitive modeling environments in use today rely on low-level rule-based programming languages, and this is a major reason for the difficulty encountered by modelers. The problem with the low-level production systems is simple: The concepts that are embodied in the low level-language have little to do with the concepts that are used by the programmer to solve the problem. Modelers need higher-level representations.

The Conceptual Gap

A study by Petre and Blackwell (1997) showed that programmers create mental images when they design programs, and that these images rarely match the programming language being used. According to Petre and Blackwell (1997), "it appears that the experts are not designing 'close to the code'; they are thinking abstractly and strategically, in some cases with a substantial translation to the implementation" (p. 110).

The conceptual gap between the ideas used to solve the problem, and the ideas supported by the language, forces the programmer to keep track of two distinct models: the one embodied in the program and the one in their heads. The research done by Petre and Blackwell (1997) reinforces the idea that the model used by programmers is primarily visual. This visual representation is rich enough to allow the programmer to

engage in mental simulation: the act of "building and exploring structures 'in their heads' before making commitments to external representations" (Petre & Blackwell, 1997, p. 111).

The use of mental simulation in programming is also evident in work done by Salomon (1992). Salomon provided good advice on the importance of well-designed high-level languages in his discussion of the interplay between machine and human independence. Salomon (1992) argues that "when one designs a programming language, one should design it not only for execution by machines, but also for execution by humans" (p. 49).

Successful Use of High-Level Languages

There are many examples of how researchers have successfully applied high-level languages to a specific domain. For example, in the early 1990's, traditional programming environments were geared towards the creation of single applications. As the need to create distributed applications (applications that work together across processes or computers) increased, programmers had to step away from the current high-level language of the time (in this case the C programming language) and program using low-level assembly language. The need for assembly language made distributed application development very difficult because of the low-level nature of assembly language.

As distributed systems became more prevalent, researchers began to experiment with new languages that directly supported the distributed programming paradigm.

Auerbach and colleagues (1991) introduced a solution in which "The complexity (of distributed programming) is hidden inside the implementation of a small number of higher-level language constructs" (p. 173). The languages created by Auerbach et al. (Hermes and Concert-C) were successful because they simplified the task of distributed programming by directly supporting the distributed programming paradigm.

Modern object-oriented languages provide a more recent example of the successful application of high-level representations. Object-oriented analysis has the advantage of simplifying the mapping of the real world domain to the code designed to model it (Coad & Yourdon, 1991). Object-oriented techniques accomplish this by making classes of objects, and objects themselves, explicit structures of the representation. Researchers theorize that this direct mapping between the objects in the real world, and the objects in the actual representation, simplifies the creation and comprehension of software (Coad & Yourdon, 1991).

Interestingly, the application of object-oriented programming languages has not been universally successful. A study done by Agarwal, et al. suggests that the success of the adoption of a high-level language depends on the experiences of the programmers and the nature of the domain being modeled (2000). In other words, the language not only must map to the domain, but also to the programmers' preconceived way of viewing and modeling reality. The successful adaption of object-oriented representations requires a good fit between the type of problem and also the modelers themselves (Agarwal, De, Sinha, & Tanniru, 2000). This lesson is very applicable to the development of high-level languages for cognitive modelers. The review given in Chapter 2 provides lessons about

the theories cognitive modelers use to represent behavior. A successful high-level language must ground itself on one of these popular cognitive theories.

Maintenance-Oriented Environments

Programmers spend considerable time performing software maintenance. According to Brooks (1995), the total cost of software maintenance is often at least 40% of the total cost of developing it the software. A recent study done by the National Institute of Standards and Technology (Tassey, 2002) showed that U.S. programmers spend over 70% of their time testing and debugging. One reason for this large cost is that fixing a bug, which on average takes 17.4 hours to do (Tassey, 2002), results in a considerable chance of introducing a new bug (Brooks, 1995).

According to the National Institute of Standards and Technology, programmers blame testing and debugging tools for this problem (Tassey, 2002). As a result, researchers often describe the process of software maintenance as a one-step forward and one-step back affair (Brooks, 1995, pp. 122-123).

Fortunately, the use of high-level languages can help with maintenance (Brooks, 1995). For example, a literature review by Hordijk and Wieringa (2005) categorized the factors that influence the maintainability of a software system. Included in these factors were code-level properties such as code complexity and duplication. High-level languages help here because they reduce code complexity and duplication.

In addition to high-level languages, the survey done by Hordijk's and Wieringa's (2005) also identified development environments as a factor that influences

maintainability. This implies that creating environments that explicitly support software maintenance (referred to as maintenance-oriented development environments) can help reduce the cost of software development.

Cause/Effect Chasm, Program Slices, and Editing Above the Code

Weiser (1982) attempted to understand how programmers encode and process information during software maintenance. Weiser performed an experiment to test his hypothesis that programmers use something called program slicing during software debugging. According to Weiser (1982, p. 446), program slicing is the process of striping a program of code that has no influence on the particular problem being debugged, thus being left with relevant program slices. Weiser's study suggested that providing support for the slicing process, as part of the development environment, would be beneficial to programmers engaged in debugging.

Additional work on supporting maintenance-oriented tasks was done by Boshernitsan (2003). Boshernitsan proposed the use of a scripting language that modifies source code at a level above the syntax, allowing the programmer to alter code at a much higher-level. Boshernitsan proposed graphical user interfaces to simplify the use of such a scripting language. As demonstrated next, editing a program at a level above the syntax has been shown to be a useful concept, not just for maintenance but also for teaching programming to novices.

Alice (Conway et al., 2000), an innovative approach to teaching programming concepts, is a paragon of the concept of editing above the level of syntax. Students create

Alice programs by placing objects in a 3D world, and then visually programming them to interact. The Alice development environment hides the syntax from the programmer, and eliminates the possibility of syntax errors. At the same time, the concepts students use to animate their 3D world closely match the concepts of a modern object-oriented programming. This makes it possible for students to learn object-oriented programming without the frustration that fragile, text-based environments often cause entry-level programmers (Dann, Cooper, & Pausch, 2008).

One major goal of Alice is to introduce programming to a much wider audience. For example, a study by Kelleher, Pausch, and Kiesler (2007) was successful using Alice to introduce programming to middle-school girls, which may help bring a traditionally under-represented group to computer science. However, transitioning students from the Alice environment to one of the more traditional high-level languages (e.g., C++, Java) has been shown to be problematic (Powers, Ecott, & Hirshfield, 2007).

Powers, Ecott and Hirshfield (2007), have encountered trouble transitioning students from the Alice environment to a more traditional programming language such as Java. They found that students struggled with the shift to advanced object-oriented concepts, and had trouble seeing the relationship between Java code and Alice code. In addition, by deemphasizing syntax, Alice may have contributed to students having trouble adjusting to the syntax intensive Java programming environment.

There are two very important lessons that the Alice project contributes. First, it is possible to bring computer programming to a wider audience by creating a visual environment that allows programming at a level above syntax. Second and most important, successful environments must provide a bridge between the abstract visual

programming they provide and the concepts that the programmers may ultimately need to grasp.

This dissertation proposes that an environment can simplify this transition by supporting programming at multiple levels, instead of just at the visual one, thus allowing the environment to better accommodate programmers as they gain experience.

Program Navigation

Much recent work pertaining to the simplification of the software maintenance task has been done (Coblenz, Ko, & Myers, 2006; Ko, Aung, & Myers, 2005; Ko & Myers, 2003, 2004; Ko, Myers, Coblenz, & Aung, 2006; Reiss, 2006; Robillard, Coelho, & Murphy, 2004). On particularly rich area is program navigation.

For example, a study by Ko, Myers, Coblenz, and Aung (2006) found that developers spend 35% of their time (much of which could have been avoided given better tools) navigating source code to find fragments relevant to a specific task. This illustrates the need to provide better support for the navigation task.

Importantly, the development environment played a large role in what the developer perceived as relevant information. Interestingly, the use of poor or incomplete terms when searching for relevant code fragments caused much of the navigational overhead incurred by the developer.

Working Sets and JASPER

Ko, Aung, and Myers (2005) also looked at how Java programmers approached the task of software maintenance. Their study suggested that programmers perform maintenance by forming a working set of task relevant code fragments. This working set was typically built by the programmer using the find and replace dialog or visually searching the program's source code. This type of methodical, structured investigation of code, in which developers kept a record (or working set) of their findings, was also reported in an earlier study conducted by Robillard, Coelho, and Murphy (2004), and is similar to the concept of program slices (Weiser, 1982).

After the developer formed the working set, they navigated the working set to uncover direct and indirect dependencies. An example of a direct dependency would be the link between an element's use and its declaration. An example of an indirect dependency would be the link between an element's use and the place where code manipulated the element's value.

An outcome of their study is a set of design guidelines that they suggest will enhance support for finding, navigating, and editing working sets of task relevant code fragments. Table **3-1** lists these findings.

Table **3-1**: Design requirements that help support the use of working sets during software maintenance (Ko, Aung, & Myers, 2005).

Table 5. Design requirements for maintenance-oriented tools, elicited from the empirical trends in the study.

#	Empirical Result	Design Requirement for Maintenance-Oriented Tools
R1	Programmers formed working sets of task-relevant methods and statements.	Provide a working set interface that supports the quick addition and removal of task-relevant code fragments.
R2	Because programmers had to store their working sets in the interactive state of file tabs and package explorer, when they changed tasks, they lost their working set.	Automatically save and recover of working sets of task-relevant code fragments, ensuring that the tools used to *navigate* working sets are distinct from the tools used to *represent* working sets.
R3	When programmers found task-relevant code, they tended to glance at its dependencies. Also, more than 60% of navigations of *indirect* relationships were for the purpose of comparison. All of these incurred significant visual search costs.	When programmers add code to a working set interface, automatically add its direct and indirect dependencies. Then, directly or indirectly related code could be placed side-by-side, avoiding the interactive overhead of opening and closing file tabs.
R4	When copying code, programmers often left indistinct or off-screen references unchanged. Because they believed the copied code was correct, it was the last place they checked for errors.	Copied code should maintain a dependency with its "original" so that unchanged references can be marked as "suspect" until verified. These markers should be apparent even when off-screen.
R5	When copying a pattern of code that was distributed within or between files, programmers often duplicated only part of the pattern, leading to *dead-end data*.	When programmers copy code, the IDE should check if the programmer is neglecting any dependencies in the copied code and offer to help collect them.
R6	Programmers searched for task-relevant *names*, but only half of such searches led to task-relevant code. Programmers also used surface features of *Paint*'s output to deduce the cause of failures, but only a ¼ of such features correlated with the cause.	Let programmers ask about program output of interest and have the IDE gather all of the code that was directly responsible for the output in question. This way, the correct working set can be built *automatically* by the IDE.

The theories resulting from the empirical studies conducted by Coblenz, Ko, Aung, and Myers (2006) are embodied in a tool called JASPER. Based on the guidelines given in Table **3-1**, JASPER makes it possible for developers to gather artifacts that are relevant to the maintenance task into a working set. Each working set can consist of code fragments, uniform resource locators (URL), and free-form notes. Coblenz et al. are currently evaluating of JASPER's effectiveness in reducing the time required to perform software maintenance tasks.

Despite the demonstrated importance of the maintenance task, not one of the cognitive modeling environments reviewed in Chapter 2 support any of the design requirements listed in Table **3-1**.

Group Memory and Information Scent

Researchers have attempted to address the problem of navigational overhead incurred by the developer. Specifically, researchers are looking at reducing the use of poor or incomplete terms when searching for relevant code fragments.

For example, DeLine, Czerwinski, and Robertson (2005) created Team Tracks, a system that helps new developers comprehend programs by recording and presenting code navigation patterns of fellow developers. Visualizations in Team Tracks are based on two ideas. First, the more often developers view a code fragment, the more important that fragment is for new developers. Second, how often a developer visits two different code fragments in sequence, determines the strength of the relationship between these two fragments. The results of two user studies showed that Team Tracks helped developers navigate to areas of code that were relevant to their current goals.

Hipikat is another example of leveraging the group history of developers performing maintenance (Cubranic, Murphy, Singer, & Booth, 2005). Hipikat is a recommender system that uses the history of a project's development as a basis for its recommendations. In addition to the source code, Hipikat also includes other forms of history such as requirements specifications, email and discussion postings, test plans, and bug reports. Most interesting is the fact that Hipikat uses information about the current task as a basis for the kinds of recommendations it makes. Like Team Tracks, researchers have demonstrated Hipikat's usefulness in two user studies.

Another way of reducing the navigation overhead is to extend information foraging theory (Pirolli & Card, 1999). Programmer Flow by Information Scent (PFIS) is

a model that predicts how programmers navigate while performing software maintenance (Lawrance, Bellamy, Burnett, & Rector, 2008). Based on information foraging theory, the theory suggests that programmers use "scent" to determine where to navigate in the source code to solve a particular problem.

Building on the Web User Flow by Information Scent (WUFIS) (Chi, Pirolli, Chen, & Pitkow, 2001), PFIS uses the source code's topology and its scent to predict where a programmer will navigate. According to Lawrance et al., the concept of scent relates linguistically to the words used to express a developer's task. PFIS relies on the topology of the source code in the same way that WUFIS relies on the topology of the Web. Links in the source code are a means by which the programmer can navigate from one place in the code to another via a single click. Links in PFIS are dependent on the actions allowed by the programming environment. For example, a programmer using the Eclipse development environment will have more links available to them than if they were using a simple text editor such as VIM.

From user studies, PFIS did a good job of predicting aggregated human navigation decisions and had better performance than the individual programmer had. Lawrance et al., also suggest that different tasks may rely more or less on information scent. For example, it may be that fixing a bug requires the developer to follow scent more than adding a new feature.

Another way to streamline the software maintenance process is to make it easier for programmers to discover the intent of the programmer when creating the code. This intent can provide "scent" that helps developers search for relevant code fragments. Without access to this intent, it can be difficult for developers to perform software

maintenance task (Ko & Myers, 2004; Lewis, 2003). Using the intent recorded in the names of the software components, or in comments created by developers, it would be possible to further reduce the overhead of source code navigation (Ko, Myers, Coblenz, & Aung, 2006).

Unfortunately, developers currently spend a lot of time figuring out the rationale that other developers implicitly embedded within the source code. In a study of software developers conducted by LaToza, Venolia, and DeLine (2006), respondents conveyed that understanding the rationale behind code was a serious problem faced during maintenance. In addition, they found that developers spend a lot of effort understanding why the code is implemented the way it is, how the code works, and what the code is trying to accomplish.

Ko and Myers confirmed this in another study of how programmers debug. Ko and Myers found that 68% of the questions asked by programmers were about "why didn't" something happen or "why did" something happen (2003). Answers to these questions rely, in part, on understanding the rationale behind the design of the program, and the easier it is to get at this design rationale, the easier it will be for the programmer to find and fix the bug.

Making design rationale more explicit using documentation would help alleviate this problem, but surprisingly, developers do not take the time to consult existing documentation to uncover the rationale behind the code they are working on. The reasons given are that the documentation is hard to locate or often out of date. According to LaToza, Venolia, and DeLine (2006, p. 499), "even if developers thought there was a

possibility of a design document containing the information they cared about, it was not worth looking for."

Ko and Myers developed an Interrogative Debugging environment (a debugger that allows developers to debug by asking questions) called Whyline (2004). Whyline allows programmers to ask "why did" and "why didn't" questions during debugging, allowing the programmer to gain a better understanding of the intent and purpose of the code being debugged.

Another project that has contributed to better maintenance-oriented environments, and allows the programmer to interrogate the program to infer rationale about its design, is the Omniscient Debugger for Java. Created by Bil Lewis, the Omniscient Debugger changes the typical paradigm of a debugger from finding out what is going on at a specific point in time, to keeping track of the complete history of a running program. Using the Omniscient Debugger, programmers can navigate through time (forward and backward) keeping an eye on values of interest (Lewis, 2003).

Creating facilities to help developers understand the rationale of the design of a system, and to maintain executing systems, has not been limited to traditional software development. Researchers have also done work on explanation facilities used for explaining the behavior of intelligent agents. Haynes, Cohen, and Ritter (2008) provide a review of these systems, along with a novel approach to supporting explanation in agent development environments. According to Haynes et al. (2008), the guidelines presented "support creating more usable and more affordable intelligent agents by encapsulating prior knowledge about how to generate explanations in concise representations that can

be instantiated or adapted by agent developers" (p. 1). These explanations can be used to help developers find relevant code fragments quickly.

Not one of the cognitive modeling environments reviewed in Chapter 2 take advantage of any of the code navigation techniques reviewed here.

Cognitive Dimensions

A discussion of high-level languages and maintenance-oriented environments is not complete without a review of notational systems and cognitive dimensions. A notational system consists of a high-level language, a development environment, and a medium of interaction (Blackwell & Green, 2003).

According to Blackwell and Green (2003), the problem with notational systems is that "every notation highlights some kinds of information, at the cost of obscuring other kinds" (p. 104). The degree in which this tradeoff exists was evident in the review of current modeling languages presented in Chapter 2. Blackwell and Green introduce cognitive dimensions as a way of helping the language designer deal with this tradeoff.

Designers can also use cognitive dimensions as a questionnaire-based evaluation tool. Kadoda, Stone, and Diaper (1999) were among the first to use cognitive dimensions as an evaluation tool. Using only the dimensions deemed relevant to their system, Kadoda et al. presented users with a questionnaire that paraphrased the dimensions in terms of the system under consideration. Blackwell and Green (2000) took this evaluation technique a step further by creating a questionnaire that presented all of the dimensions, leaving it to the user to decide which ones were relevant. By using cognitive

dimensions to structure feedback questionnaires, designers can get detailed feedback using a common vocabulary for evaluating the quality of a notational system. Designers can use this feedback inform design changes and improve the system.

Examples of cognitive dimensions include Closeness of Mapping, which is a dimension that measures how closely the notational system maps to the result it represents. For the reasons already discussed, this dimension measures a critical attribute of a well-designed high-level language. Another relevant cognitive dimension is Viscosity, which measures how easily a high-level language and environment allows change. A language or environment that makes it easy for developers to perform maintenance tasks would have low Viscosity.

Table **3-2** gives a summary of some of the more useful dimensions for evaluating high-level behavior representations and cognitive modeling environments.

Table **3-2**: Useful cognitive dimensions for evaluating a high-level behavior representation language and modeling environment.

Cognitive Dimension	Description
Visibility	How easy is it to view the elements in a model, including their internal details?
Viscosity	How easy is it to make changes to an existing model? The less the viscosity, the easier it is to change the model.
Diffuseness	How many symbols or how much space does the notation require to produce a certain result or express a meaning?
Hard-mental operations	How much hard mental processing lies at the notational level, rather than at the semantic level? Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what is happening?
Error-proneness	How easy is it to make errors using the behavior representation language?
Closeness of mapping	How closely does the behavior representation language match the way that the modeler describes the behavior?
Role-expressiveness	How easy is it to discover why a modeler has chosen a particular design? Explicit support for design rationale, as discussed earlier, improves a systems role-expressiveness.
Progressive evaluation	How easy is it to evaluate and obtain feedback on an incomplete solution?
Premature commitment	How often is the developer forced to make a commitment in the model before there is enough information to make the commitment?

Software Reuse

One way to simplify software maintenance is to support code reuse (Boehm, 1987). Reuse of code, even within a single program, can reduce development and maintenance costs. As far back as 1969, the importance of software reuse was presented as an invited paper by McIlroy (1968). In his talk, McIlroy (1968) described a scenario of the future of software reuse: "...the purchaser of a component from a family will choose one tailored to his exact needs. He will consult a catalogue offering routines in

varying degrees of precision, robustness, time-space performance, and generality" (p. 140).

Progress towards this vision has since been made in software engineering and the value of software reuse has continually been reaffirmed (Brooks, 1995; Krueger, 1992). Empirical evidence of the advantages of software reuse is also evident in the literature. For example, reusable libraries created by Raytheon have resulted in the development of new applications using as much as 60% preexisting code (Boehm, 1987). This resulted in cost savings of 10% in the design phase, 50% in the code and test phase, and 60% in the maintenance phase (Boehm, 1987). In addition, Toshiba's library of reusable components for industrial process control has also resulted in significant productivity gains (Boehm, 1988b).

Four Dimensions of Reuse

A useful framework for considering reuse was developed by Krueger (1992). Krueger breaks software reuse into four dimensions: (a) abstraction, (b) selection, (c) specialization, and (d) integration.

According to Krueger (1992), "Abstraction is the essential feature in any reuse technique" (p. 133). Abstraction allows programmers to consider a programming task at a more general level, separate from the concrete realities of the modeling language, and is a reason why high-level languages provide such great support for reuse. Although often taken for granted, abstraction in high-level languages is one of the most successful vehicles of software reuse (Brooks, 1987; Krueger, 1992).

Selection, as defined by Krueger, helps programmers locate and select reusable components. Reusable components are not useful when they are difficult to locate, select, and compare to other software components. According to Kruger, good maintenance-oriented environments should make it easy for clients to search for reusable components based on a number of criteria.

Krueger (1992) defines specialization as allowing programmers to tailor reusable components to their specific needs. Specialization is essential for reuse because the reusability of a component is dependant on its generality. Without specialization, developers would be unable to configure a component for a specific use: transforming the component from a general artifact to a more specialized object.

The fourth and final of Krueger's dimensions is integration, which designers can accomplish with an environment that allows developers to combine reusable components into a working program. The usefulness of reusable components depends on how easy it is to integrate these components. Again, a maintenance-oriented environment can play a major role in simplifying the integration piece of reusable software. All four of Krueger's dimensions are necessary for effective software reuse and high-level behavior representation languages and cognitive modeling environments must support them.

Reuse with Design Patterns

Design patterns provide another effective way to promote reuse and, in some cases even reduce defects (Vokac, 2004). Design patterns are reusable templates that provide solutions to recurring problems. A design pattern consists of four elements: The

pattern name, which makes it possible for developers to identify and communicate about a pattern; a problem, which helps developers recognize when a particular pattern is useful; a pattern solution, which provides an abstract description of the pattern and how it can be used to solve the problem; and the consequences, which discuss the trade-offs related to the pattern's use (Gamma, Helm, Johnson, & Vlissides, 1995).

An example is useful to illustrate how design patterns can promote reuse. The Strategy design pattern, described in the seminal book on design patterns (Gamma, Helm, Johnson, & Vlissides, 1995), is a rather simple design pattern meant to solve the following dilemma: A programmer wishes to create a system that implements many different strategies for solving a particular problem, and would like to decouple these strategies from the program that solves them. In other words, the programmer needs to be able to create new strategies without breaking the code that uses them to solve the problem.

Figure **3-1** illustrates (using the Universal Modeling Language, a standard graphical language for illustrating object-oriented designs) the Strategy pattern, an object-oriented pattern that provides a solution to this general problem.

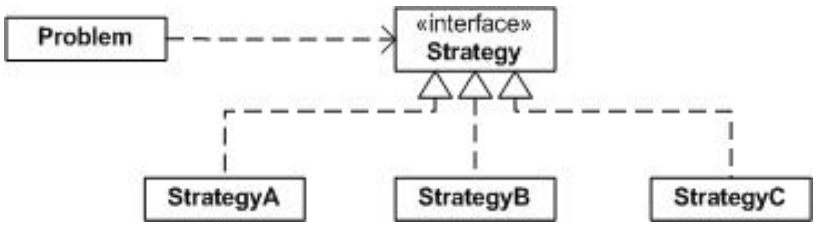


Figure **3-1**: The strategy design pattern.

In Figure **3-1**, the entity that solves the problem (Problem) uses an abstract definition of a strategy to find a solution. This abstract entity can take many forms: StrategyA, StrategyB, and StrategyC. However, the problem itself is not aware of the details surrounding these different strategies. Designers can add, remove, or alter concrete strategies without affecting the entity that solves the problem. This decouples the problem from the strategies used to find a solution making it easy to reuse strategies.

Developers can adopt the general configuration of object-oriented entities shown in Figure **3-1** to solve any number of related problems. Gamma et al. (1995) cataloged several object-oriented design patterns like this in a single publication that has since served as a cookbook of reusable solutions for object-oriented developers.

Unfortunately, a similar collection of reusable solutions for cognitive modelers is absent and its creation would help promote the reuse of behavior in the cognitive model community. Of course, before modelers can build this collection, there must be support for reuse within the behavior representation languages.

Summary

This chapter presents a representative subset of the large body of literature supporting the use of high-level languages, maintenance-oriented environments, and reuse in software development.

High-level languages have played an important role in improving the productivity, reliability, and simplicity of software. However, the review of popular

agent and cognitive architectures in Chapter 2 shows that they are all programmed using low-level rule-based languages.

A conceptual gap between the ideas used to model human behavior, and the ideas supported by the language, can force the programmer to keep track of two distinct models (Petre & Blackwell, 1997). Therefore, a successful high-level behavior representation language should closely match a widely adopted theory of cognition. In addition, the success of the adoption of a high-level language depends on both the representation and the experiences of the programmers (Agarwal, De, Sinha, & Tanniru, 2000). As a result, the paradigm supported by the high-level behavior representation language should be familiar to cognitive scientists. Unfortunately, in the review in Chapter 2 only one of the high-level behavior representations reviewed (TAQL) explicitly supported a theory of cognition (other than those that support GOMS, which is limited in scope).

Recent research has aimed at improving development environments by adding features that simplify all aspects of development. The environments reviewed in Chapter 2 (e.g., JessPad, CogTool, Visual Soar) do provide cognitive modelers with environments that help with the creation of model. However, the fact that developers might be spending 35% of their time navigating source code (Ko, Myers, Coblenz, & Aung, 2006), underscores the need for maintenance-oriented environments that support code navigation. Researchers are looking at techniques such as working sets, group memory, and information scent to improve the modeler's ability to navigate code quickly to find task relevant fragments. Yet not one of the cognitive modeling environments in Chapter 2 takes advantage of these techniques.

The software engineering literature also clearly documents the benefits of reuse. Traditional software development has been able to reuse software by using high-level languages, following Krueger's dimensions of reuse, and taking advantage of libraries of design patterns. Fortunately, many of the high-level behavior representation languages reviewed in Chapter 2 help make reuse within a cognitive model possible. However, only one of these languages (HLSR) supports multiple architectures. The lack of cross-architecture languages, makes reuse across architectures difficult. In addition, not one of the languages or environments reviewed in Chapter 2 provides special support for design patterns.

This detailed look at the state of cognitive modeling (Chapter 2) and software engineering (this chapter) has illuminated two important problems, and uncovered well-defined and tested solutions to these problems. Table **3-3** summarizes these problems and the solutions, and explicitly outlines the direction taken by this dissertation.

Table **3-3**: A succinct description of the problems facing cognitive modeling and the software engineering solutions that will make a difference.

The Problems	The Solutions
1. The cognitive modeling community lacks is a high-level representation language that explicitly supports a well-known theory, allows for the reuse of behavior, compiles into productions for multiple well-tested architectures, supports model refinement, and allows programming at several different levels of abstraction.	1. Close the conceptual gap by designing a high-level representation language that provides a good fit between the type of problem and the modelers themselves. A successful high-level language must ground itself on a popular cognitive theory, support multiple architectures, allow for model refinement, support reuse, and support a variety of additional high-level behavioral abstractions.
2. The cognitive modeling community lacks a maintenance-oriented development environment that supports both novice and experienced programmers using recent software engineering research (e.g., graphical environments and code navigation techniques).	2. Create a maintenance-oriented environment that makes it possible for modelers to edit and browse a program graphically and at a level above the actual text-based code.
	3. Create a maintenance-oriented environment that facilitates code navigation using some of the design guidelines listed in Table **3-1**, and by taking advantage of working sets, group memory, or information foraging theory.
	4. Ensure proper support for reuse by paying attention to Krueger's dimensions of reuse and by supporting reusable templates that provide solutions to recurring problems.
	5. Conduct formative and summative evaluations of the resulting notational system using validated methods based on cognitive dimensions.

The next chapter describes, in detail, how I embedded portions of an existing tool with the software engineering theories reviewed here to alleviate the obstacles that are facing cognitive modelers.

Chapter 4

Herbal: A Theory-Based System for Simplifying Cognitive Modeling

The detailed look at the state of cognitive modeling and software engineering presented in Chapter 2 and Chapter 3 has illuminated two important problems, and uncovered well-defined and tested solutions to these problems (see Table **4-1**). This chapter describes how I have implemented these solutions in an existing tool called Herbal (version 3.0.0). Importantly, the formative evaluation described in Chapter 5 has also guided this implementation.

Table **4-1**: Summary of the solutions resulting from the literature review.

1. Close the conceptual gap by designing a high-level representation language that provides a good fit between the type of problem and the modelers themselves. A successful high-level language must ground itself on a popular cognitive theory, support multiple architectures, allow for model refinement, support reuse, and support a variety of additional high-level behavioral abstractions.
2. Create a maintenance-oriented environment that makes it possible for modelers to edit and browse a program graphically and at a level above the actual text-based code.
3. Create a maintenance-oriented environment that facilitates some of the code navigation using the design guidelines from (Ko, Aung, & Myers, 2005), and by taking advantage of working sets, group memory, or information foraging theory.
4. Ensure proper support for reuse by paying attention to Krueger's dimensions of reuse and by supporting reusable templates that provide solutions to recurring problems.
5. Conduct formative and summative evaluations of the resulting notational system using validated methods based on cognitive dimensions.

Herbal: A High-Level Behavior Representation Language

To simplify agent programming and cognitive modeling, I implemented a high-level behavior representation language, and associated parser and compiler. The Problem Space Computational Model (PSCM) forms the basis for the Herbal high-level language, and the Extensible Markup Language (XML) (W3C, 2004a) specifies the syntax of this language. The Herbal system compiles this language into productions that execute within two popular agent architectures: Soar (sitemaker.umich.edu/soar) and Jess (herzberg.ca.sandia.gov/jess/).

The Problem Space Computational Model

The high-level language supported by the Herbal Toolset is based on the PSCM (Lehman, Laird, & Rosenbloom, 1996; Newell, 1990; Newell, Yost, Laird, Rosenbloom, & Altmann, 1991). As explained in detail in Chapter 2, the PSCM is a unified theory of cognition that defines behavior as movement through a problem space. The choice to use the PSCM will help close the conceptual gap between the language used by modelers to describe behavior and the language used by architectures to represent that behavior. The PSCM is a robust, and well-tested cognitive theory that closely maps the modeling domain to the people who typically create cognitive models.

The PSCM also serves as an organizational structure for intelligent agents in general. Explicit support for PSCM constructs allows all modelers to partition behavior into a hierarchy of problem spaces, operators, states, and desired goals. The ability to

create componentized programs hierarchically is something that should resonate with all software engineers that are used to creating modularized programs.

XML and XSchema

The PSCM-based high-level language supported by the Herbal Toolset takes the form of an XML application. Herbal uses XML to provide explicit support for the PSCM, and to translate the PSCM into a low-level rule-based representation for execution within an agent environment (see Figure **4-1**).

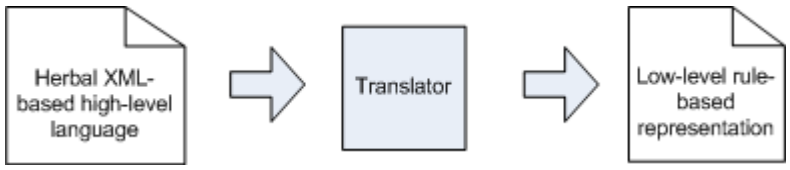


Figure **4-1**: A High-level XML representation translated into low-level rule-based representations.

Choosing an XML-based language provides many benefits. For example, XML allows for the creation of structured documents that can directly represent the hierarchical structure of the PSCM. In addition, compared to low-level rule based languages, the portable text format used by XML is easily readable by both people and computers. In fact, there are a large number of robust XML editors that parse XML and provide a graphical environment for quickly and safely editing the XML (e.g., XMLSpy, oXygen, XMetaL, XMLBuddy, and XML Notepad). Research promises to provide even better support for XML editing (Chidlovskii, 2003; Quint & Vatton, 2004). As a result, a wide

range of existing graphical programming tools can support editing the Herbal high-level language.

Programmers can also transform XML into other formats using the Extensible Stylesheet Language (XSL) (Royappa, 1999; W3C, 2004b). For example, Herbal agent code can be easily transformed into HTML documentation, making it easy for developers to generate documentation directly from source code. In addition, Herbal agent code can be easily transformed into Scalable Vector Graphics (SVG) (W3C, 2003) which can be an effective way for creating visualizations of complex data (Jackson, 2002; Vullo & Bogaard, 2004). Finally, the popularity of XML helps reduce the learning curve that might otherwise form a barrier to the adoption of Herbal.

XSchema (W3C, 2004c) defines the Herbal high-level language specification. The use of XSchema for this language was advantageous for many reasons. XSchema provides a clear documentation of the structure and content of XML documents (Campbell, Eisenberg, & Melton, 2003). In addition, XML parsers can use the Herbal XSchema to validate the content of an Herbal program. This eliminates the common problem of the specification becoming "out of sync" with the implementation because the XSchema serves as both the language specification and the documentation of the language specification. Lastly, most commercial and open source XML editors utilize XSchema to provide features such as syntax highlighting and auto completion to help programmers quickly create valid XML documents. Because the Herbal language uses XSchema, these features are immediately available to the Herbal programmer.

Six different types of XML documents make up an Herbal program, each defining a set of reusable components including namely types, conditions, actions, operators,

problem spaces, and agents. These documents represent libraries in the Herbal language and these libraries give the Herbal high-level language explicit support for Krueger's first dimension of reuse: abstraction (Krueger, 1992).

XSchema defines the allowed structure and content of each of these library types. In many cases, the components in these libraries mirror the elements of the PSCM, and should be familiar to most cognitive scientists. However, there are components in the Herbal high-level language (such as types, conditions, and actions) that extend the PSCM by providing additional levels of abstraction (further support of Krueger's first dimension). The choice to add these components specifically addresses the reuse problem introduced in Chapter 2, which describes how the nature of rule-based representation languages makes it difficult to reuse the conditions and actions in a rule.

As an example of this new granularity, the left-hand side of Table **4-2** shows an operator element, which has a unique name, and child elements of *ifType* and *thenType*. The *ifType* element contains references to conditions and the *thenType* element contains references to actions. These references point to conditions and actions that reside in a separate XML document (library), and whose syntax specification is in a separate XSchema. Unlike operators, conditions and actions are not explicitly part of the PSCM, but were included in the Herbal high-level language to provide granularity that supports reuse at the condition/action level.

Table **4-2**: XSchema describing an operator and an XML instance of an operator.

XSchema Specification for an Operator

```
<xs:complexType name="operatorType">
  <xs:sequence>
   <xs:element name="if"
               type="ifType"
               minOccurs="1"
               maxOccurs="1"/>
   <xs:element name="then"
               type="thenType"
               minOccurs="1"
               maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name"
type="xs:ID" use="required"/>
</xs:complexType>

<xs:complexType name="ifType">
  <xs:sequence>
   <xs:element name="conditionref"
               type="conditionRefType"
               minOccurs="0"
               maxOccurs="unbounded"/>
  </xs:sequence>
  </xs:complexType>

<xs:complexType name="thenType">
  <xs:sequence>
   <xs:element name="actionref"
               type="actionRefType"
               minOccurs="0"
               maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Instance of an Operator

```
<operator name='driveRight'>
  <if>
   <conditionref
       conditioN='okRight'/>
  </if>
  <then>
   <actionref actioN='moveRight'/>
  </then>
</operator>
```

The right-hand side of Table **4-2** lists a typical section of Herbal source code. The XML shown here declares an instance of an operator called *driveRight,* and obeys the Schema given in the left-hand side of Table **4-2**.

The *driveRight* operator will be proposed when the condition *okRight* is true, and when the operator is applied an action called *moveRight* will move the agent to the right. The details of the *okRight* condition and the *moveRight* action are encapsulated in the libraries that contain their instantiations.

The XSchema and associated XML code shown in Table **4-2** can be edited graphically using any of the commercial or open source XML editors. For example, Figure **4-2** shows an XML document, containing instantiations of several operators edited in XML Notepad. XML Notepad is using the XSchema to determine the required syntax for the declaration of operators, and can even find problems and help the programmer fix them. In Figure **4-2**, XML Notepad is indicating that an operator is missing the required name attribute, and can help the programmer add this attribute.

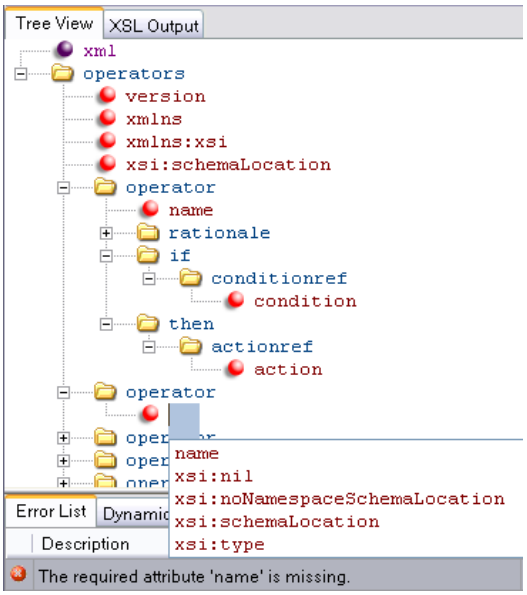


Figure **4-2**: Herbal programming using XML Notepad.

The Herbal Parser and Compiler

Herbal can transform code written in the Herbal high-level language into executable productions for either the Soar or Jess agent architectures. The ability to compile to multiple architectures is essential for reuse across architectures. Herbal

supports the Soar architecture because it is widely used for cognitive modeling, and Herbal supports Jess because of its popularity as an intelligent agent architecture. Support for two very different types of architectures was intentional because it emphasizes the ability for high-level languages to support architectural-neutral reuse. In addition, it allows for architectural comparison.

The first phase in this transformation (shown in Figure **4-3**) consists of parsing the XML code and creating a Document Object Model (DOM) of the PSCM. I used Java to create the Herbal parser, and the DOM consists of a hierarchical collection of Java objects.

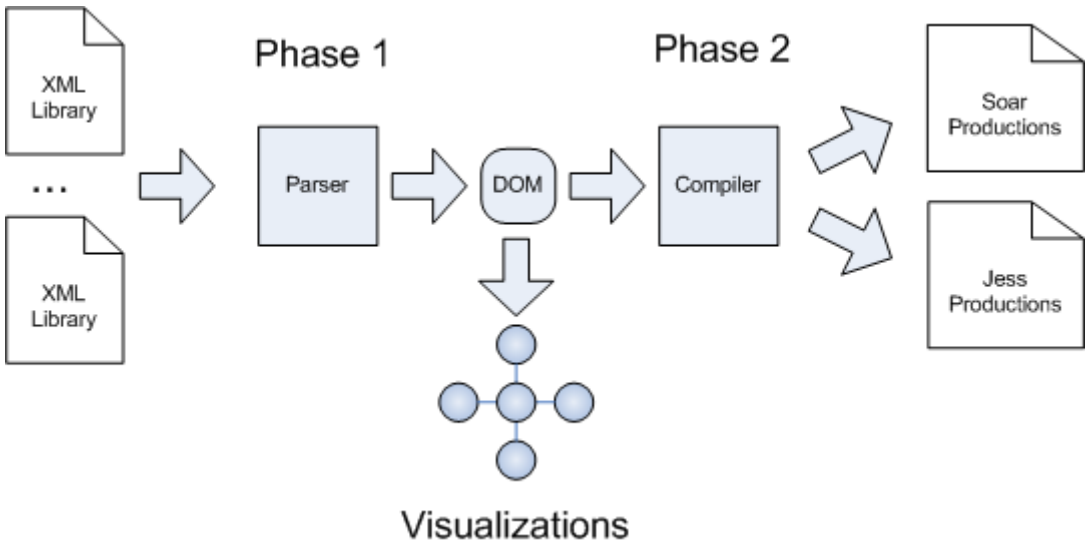


Figure **4-3**: Parsing and compiling Herbal XML source code.

The parser validates the XML based on the associated XSchema. In addition, custom logic extends the parser for additional validation. This validation makes it possible to check for semantic errors that the XSchema cannot not specify.

Once in memory, Herbal accesses the DOM for many different purposes including the creation of useful visualizations, and the creation of executable productions. The Herbal compiler is responsible for the transformation of the in-memory DOM into executable code.

The main challenge in creating a compiler is deciding how to transform the PSCM DOM into a semantically equivalent set of productions for a specific architecture. The degree of difficulty of this transformation relates to the underlying language support for the PSCM. For example, the Soar architecture is an exemplar of the PSCM, while Jess provides no explicit support for the PSCM.

Table **4-3** provides a few examples that illustrate how the Soar and Jess compilers transform the PSCM into appropriate productions. Consider the Herbal XML code shown in Table **4-3**. This code defines a condition, called *dirty,* that tests if a vacuum cleaner agent (Cohen, 2005) is on a dirty square. Table **4-3** also shows the resulting Soar and Jess code produced by the Herbal compiler. This translation is straightforward because both Soar and Jess have clear support for the concept of a condition.

Table **4-3**: A translation from an Herbal condition to Soar and Jess source code.

Architecture	Source Code

Herbal XML Language	`<condition name='dirty'>`
	`  <match type='vacuum.types.spot'>`
	`  <restrict field='status'>`
	`  <eq>dirty</eq>`
	`  </restrict>`
	`  </match>`
	`</condition>`

Compiled Soar Code	`(<vacuum-types-spot2> ^status <status2> |dirty| )`
Compiled Jess Code	`(topspace::vacuum.types.spot (status ?status1&:(eq* ?status1 "dirty")))`

A second example, given in Table **4-4**, illustrates how the Soar and Jess compilers transform Herbal XML code for an action called *clean*. This translation is less straightforward because Soar and Jess have different support for the interaction between an agent and its environment.

Soar defines explicit structures to support an agent's communication with its environment. These structures take the form of an input and output link. As a result, the Herbal compiler adds the *clean* working memory element directly to the output link (labeled <*i2*> in Table **4-4**). Jess, on the other hand, has no special language constructs that deal with agent/environment interaction so the c*lean* command is treated like any other fact in working memory.

Table **4-4**: A translation from an Herbal action to Soar and Jess source code.

Architecture	Source Code

Herbal XML Language	`<action name=clean'>`
	`  <add type='vacuum.types.action'>`
	`  <set field='move'><value>clean</value></set>`
	`  </add>`
	`</action>`

Compiled Soar Code	`(<i1> ^output-link <i2>)`
	`-->`
	`(<i2> ^|vacuum.types.action| <vacuum-types-action20>)`
	`(<vacuum-types-action20> ^move |clean| )`

Compiled Jess Code	`(assert (topspace::vacuum.types.action (move "clean") ))`

The third example, shown in Table **4-5**, demonstrates how the Herbal compiler transforms an Herbal operator. Recall that the operator is an important component of the PSCM. Unlike Soar, the Jess language has no concept of operators. As a result, the Herbal compiler must simulate the operator concept in Jess using a basic production.

Table **4-5**: A translation from an Herbal operator to Soar and Jess source code.

Architecture	Source Code

Herbal XML Language	`<operator name='clean'>`
	`  <if>`
	`  <conditionref conditioN=dirty'/>`
	`  </if>`
	`  <then>`
	`  <actionref actioN='suck'/>`
	`  </then>`
	`</operator>`

Compiled Soar Code	`sp {propose*clean`
	`  (state <local> ^top <top> ^parent <parent> ^name cleanps)`
	`  (<top> ^io <i1>)`
	`  (<i1> ^input-link <i2>)`
	`  (<top> ^|origvac.types.status| <vacuum-types-spot2>)`
	`  (<vacuum-types-spot2> ^status <status2> |dirty| )`
	`-->`
	`  (<local> ^operator <o> + =)`
	`  (<o> ^name clean)`
	`  (<o> ^count <count>) }`
	`sp {apply*clean`
	`  (state <local> ^top <top> ^name cleanps ^operator <o>)`
	`  (<o> ^name clean)`
	`  (<top> ^io <i1>)`
	`  (<i1> ^output-link <i2>)`
	`-->`
	`  (<i2> ^|vacuum.types.action| <vacuum-types-action20>)`
	`  (<vacuum-types-action20> ^move |suck| )}`

Compiled Jess Code	`(defrule clean`
	`  (topspace::vacuum.types.spot`
	`  (status ?status18&:(eq* ?status18 "dirty")))`
	`=>`
	`  (assert (topspace::vacuum.types.action (move "suck") )) )`

Table **4-5** shows the differences between how the Herbal compiler produces operators in Soar and Jess. For Jess, the compiler translates the Herbal operator directly into a simple production. However, in Soar an operator consists of a proposal rule and an application rule (Lehman, Laird, & Rosenbloom, 1996). The proposal rule fires when the operator is appropriate for the current situation. The application rule contains knowledge about how the operator changes working memory. The distinction between operator proposal and operator application allows for interruptability, which is an important part

of the psychological plausibility of Soar agents and is necessary to support learning in Soar.

The examples given in Table **4-3**, Table **4-4**, and Table **4-5** illustrate how the Herbal high-level language has augmented the PSCM. In some cases (i.e., the addition of conditions and actions as explicit objects), these modifications have added greater granularity, which allows for better reuse. While in other cases (e.g., simulated operators in Jess), sacrifices were made in the richness of the problem solving abilities and psychological plausibility of the PSCM. These sacrifices are apparent when supporting architectures that do not provide direct support the PSCM. These trade-offs are common throughout the design of the Herbal Toolset.

There will always be times when the modeler will be unhappy with the sacrifices made by the compiler for the sake of architectural neutrality. Model refinement (Salvucci & Lee, 2003) allows the modeler to create a model using a high-level representation and then refine it when they need lower-level, architecture specific, control. This is an essential component of any high-level representation and the Herbal tool supports this using prescripts and postscripts. Herbal's prescript and postscript files allow the modeler to inject architecture specific code that will be automatically included into the generated low-level representation.

Herbal: A Tool for Supporting Maintenance

The Herbal Toolset includes an Integrated Development Environment (IDE) that provides a graphical environment for creating and maintaining agents. This environment

supports the creation and maintenance of agents by extending the popular Eclipse extensible platform (Shavor et al., 2003). The Herbal IDE provides support for developers to modify source code at a level above the syntax, and support for code navigation using working sets and information scent.

The Herbal IDE

I have implemented the Herbal IDE as an Eclipse plug-in. Eclipse is a universal platform providing an open and extensible IDE. Basing the Herbal IDE on Eclipse provides many advantages. First, Eclipse provides a robust framework for the creation of powerful development tools. This framework consists of many of the modern IDE features expected by developers, including project management, multiple views, and real-time compilation. In addition, the popularity of the Eclipse IDE has grown considerably, and as a result, the learning curve for using the Herbal IDE is small for users who are already familiar with the Eclipse environment. Finally, Eclipse is free and executes on a variety of different platforms, making the Herbal IDE available to a wide range of potential users.

The Herbal IDE supports the creation of Herbal agents either graphically or by programming directly in the Herbal high-level language. Agent programmers can freely switch between these two modes at any time.

Modelers perform graphical editing in Herbal with the Herbal GUI Editor (shown in Figure **4-4**). The Herbal graphical editor provides support for modifying source code at a level above the syntax (Boshernitsan, 2003).

Like the Herbal high-level language, the GUI Editor is library centric. Using the editor, programmers can use wizards to create or modify existing library components (i.e., types, conditions, actions, operators, problem spaces, and agents). Developers can also create agents without having to write code in the Herbal high-level language. Herbal creates the Herbal XML code automatically as the developer interacts with the GUI Editor.

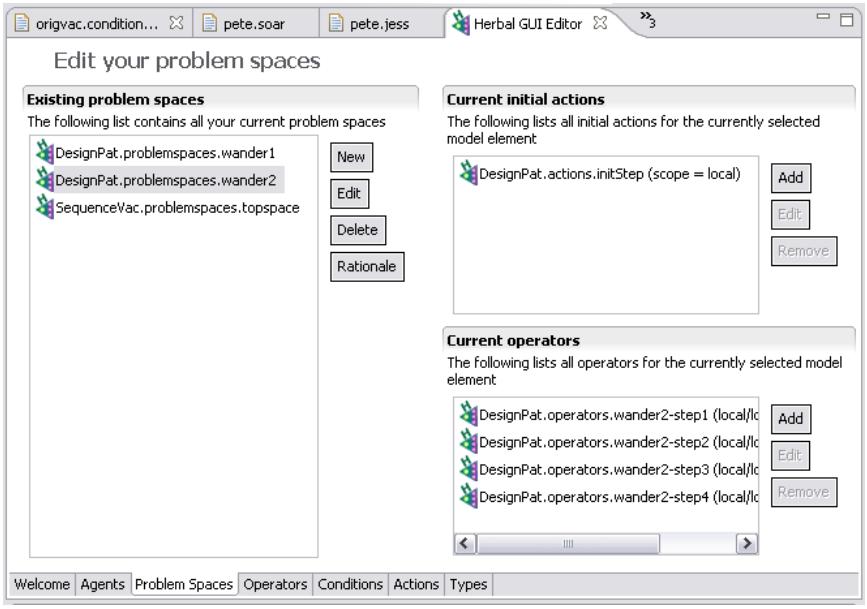


Figure **4-4**: The Herbal GUI Editor.

In addition to saving time and reducing programming errors, modelers can use the GUI Editor as a means for learning the Herbal XML language. Developers can create a PSCM component using the GUI Editor and then inspect the XML code created. By switching between the editor and the generated code, programmers can quickly learn the syntax of the Herbal high-level language.

While the editor simplifies the creation of PSCM components, some developers may prefer to work directly with the Herbal high-level language. This is another example

of support for model refinement (Salvucci & Lee, 2003), but at a level of abstraction higher than what Salvucci and Lee proposed. At any time during development, programmers can edit the Herbal XML code directly, and the GUI Editor immediately displays the changes (see Figure **4-5**).

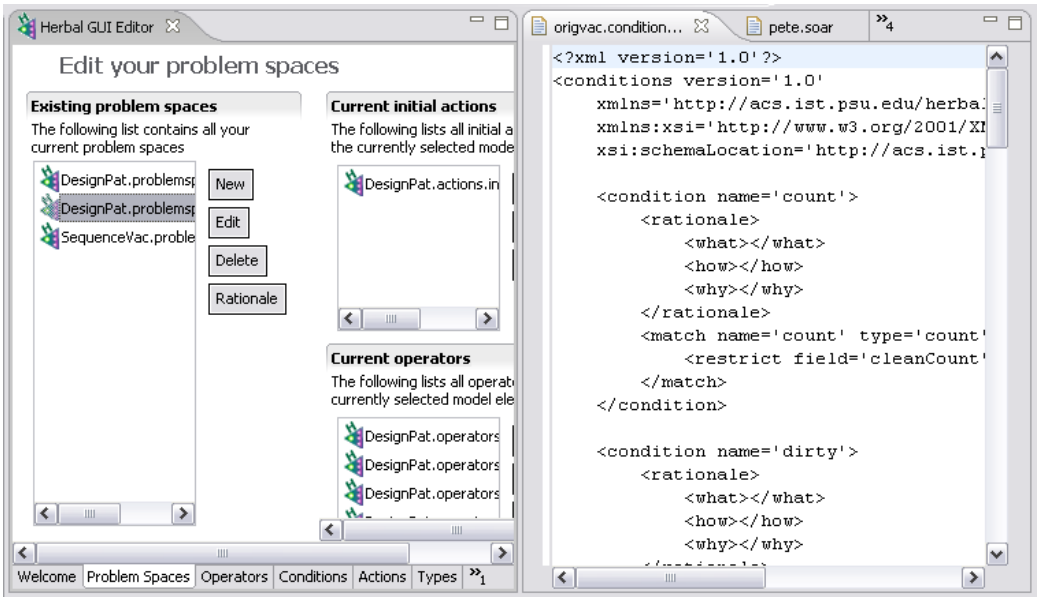


Figure **4-5**: Developing agents using both the GUI Editor and by editing the Herbal XML by hand.

Typical of most Eclipse plug-ins, Eclipse automatically invokes the compiler as the programmer is working. In other words, with each change made by the agent developer, the Herbal IDE compiles the Herbal XML code into both Soar and Jess productions. This feature serves as an excellent mechanism for learning the underlying Soar or Jess programming languages. Herbal programmers can create PSCM constructs using either the Herbal GUI Editor or the Herbal high-level language, then inspect the generated Soar and Jess code to learn how Herbal implements these constructs in the underlying architectures.

Figure **4-6** shows the Herbal IDE displaying multiple views of an Herbal library. The top left view shows the Herbal GUI editor. To the right of the GUI Editor is a snapshot of some Herbal high-level XML code. The bottom two views in Figure **4-6** show the generated Jess and Soar code. Finally, along the very bottom of Figure **4-6** is a list of current warnings and errors. In this case, a typo made by the developer has generated a warning. Double-clicking on this warning will open an editor to the appropriate location so the warning can be resolved.

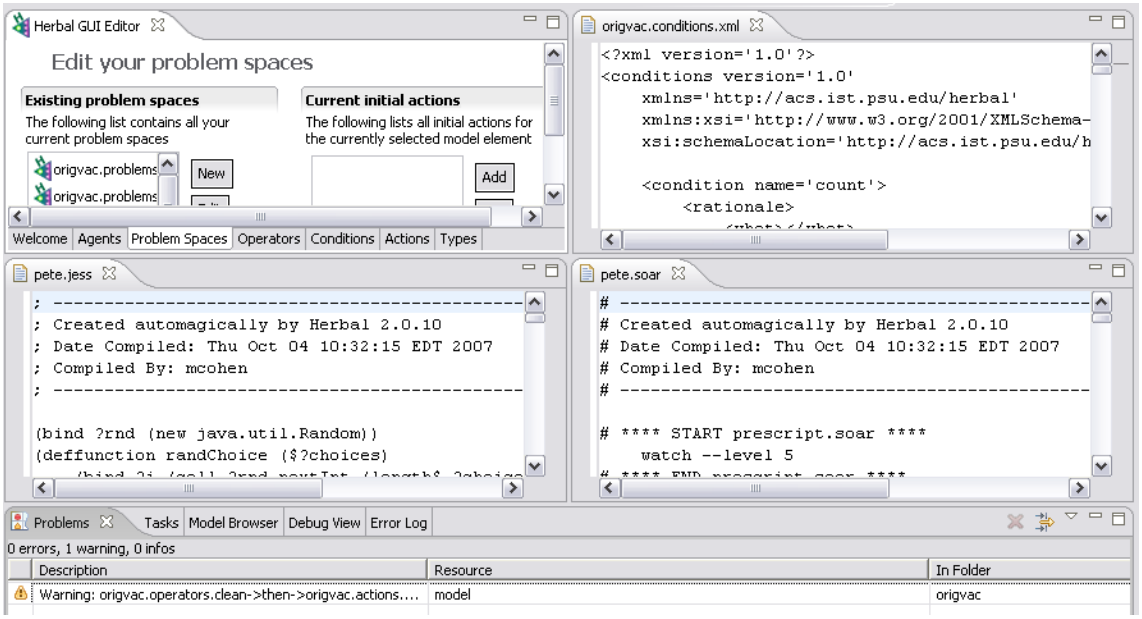


Figure **4-6**: The Herbal IDE showing multiple views of an Herbal library.

The Model Browser View shown in Figure **4-7** makes it easy to browse the static PSCM structure of an Herbal agent and, therefore, may simplify the maintenance of these structures.

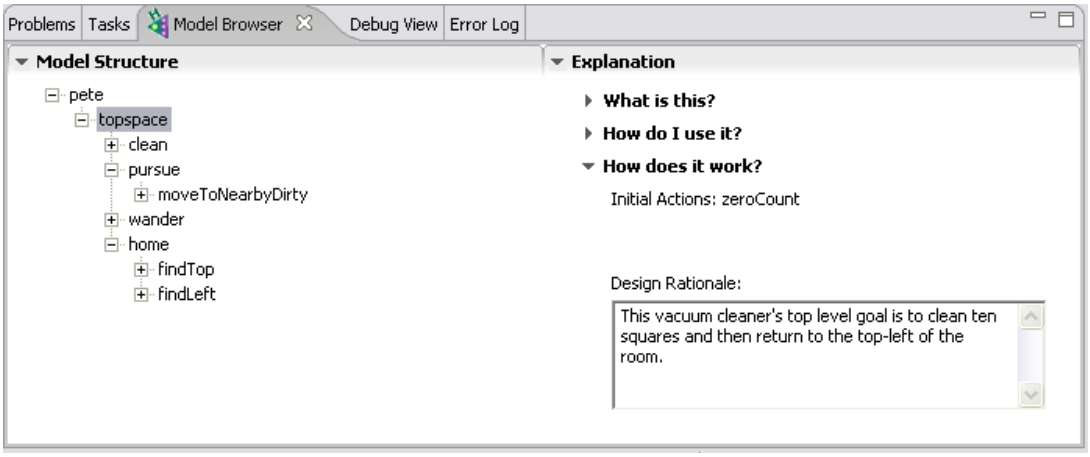


Figure **4-7**: Viewing the static PSCM structure using the Model Browser View.

Working Sets and Intent as Information Scent

Chapter 3 clearly documents the need to support code navigation. Working sets are one method for providing this support. Studies done by Ko, Aung, and Myers (2005) suggest that better support for working sets can help simplify the code navigation task. Ko, Aung, and Myers (2005) present a set of design requirements for maintenance-oriented tools, three of which I have implemented in Herbal. The following is a list of the design requirements that Herbal supports for better code navigation.

1. Provide a working set interface that supports the quick addition and removal of task-relevant code fragments.
2. Automatically save and recover working sets of task-relevant code fragments, ensuring that the tools used to navigate working sets are distinct from the tools used to represent working sets.
3. When programmers add code to a working set interface, automatically add its direct and indirect dependencies. Then, directly or indirectly related code can be placed side-by-side avoiding the interactive overhead of opening and closing file tabs.

As shown in Figure **4-8**, the Herbal IDE makes it possible for developers to build a working set of task relevant code fragments. The key design question when building working set interfaces is how to help developers find relevant code fragments. As outlined in Chapter 2, several different approaches have been taken including leveraging group memory (Cubranic, Murphy, Singer, & Booth, 2005; DeLine, Czerwinski, & Robertson, 2005), information foraging theory (Lawrance, Bellamy, Burnett, & Rector, 2008), and design rationale (Ko, Myers, Coblenz, & Aung, 2006). I have chosen to utilize the design rationale feature that already exists in Herbal to help modelers find and follow scent when building working sets of relevant code fragments.

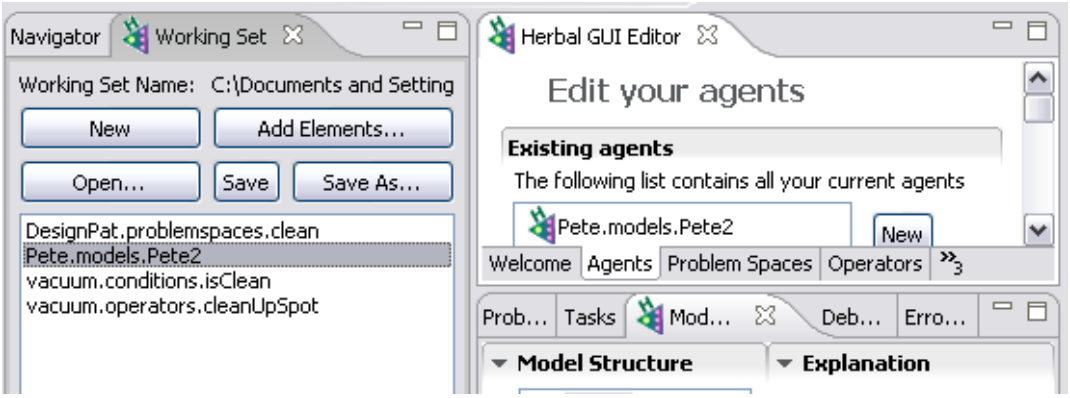


Figure **4-8**: Support for working sets in the Herbal IDE.

In Herbal, developers build working sets manually or by executing a search through the libraries using keywords related to the current maintenance task. Relying on a search of component names using keywords can be fragile because it requires the modeler to use descriptive names that exactly match the keyword (Ko, Myers, Coblenz, & Aung, 2006). Fortunately, Herbal's existing support for design rationale, based on the work done by Haynes, Cohen, and Ritter (2008), make it possible to also search the

component's design rational for the specified keyword. If the modeler previously entered design rationale, Herbal will use it to increase the relevance of the search results.

Herbal also takes into account the topology of the model when searching for relevant components based on keywords. The items returned in the result set either contain or reference the keywords themselves or are dependent on items that contain or reference the keywords.

The modeler can save the collection of code fragments as a named working set and share them between developers or recall them for future use. Finally, double-clicking on items in the working set will open the code fragment in the Herbal GUI editor for inspection.

Interestingly, the working set interface implemented in Herbal supports Kruger's second dimension: selection. In addition to facilitating code navigation, Herbal's working set interface also helps programmers locate and select reusable components. This is especially effective because of the inclusion of intent by the selection algorithm used by the working set search.

Herbal: A Tool for Supporting Reuse

The design of Herbal includes support for several different forms of reuse including the reuse of low-level PSCM components, the creation of libraries, and the instantiation of behavior design patterns.

Libraries

The Herbal high-level language is library centric, in that Herbal projects must consist of XML documents that define several libraries of reusable components. There are six different types of Herbal libraries: type libraries, condition libraries, action libraries, operator libraries, problem space libraries, and agent libraries.

Figure **4-9** shows the dependencies between these libraries. The foundation of all the Herbal libraries is the type library. This library contains the set of data types available to the agent programmer. This is a major improvement over the lack of data type checking typical in most rule-based languages. From these types, the programmer can define conditions and actions that can add, edit, remove, or test for the existence of instances of the defined types. Modelers build operators from these conditions and actions, and problems spaces from a set of operators and conditions. Finally, the developer defines agent behavior using a hierarchy of problem spaces. This layered approach allows developers to choose and reuse behavior at just the level of abstraction.

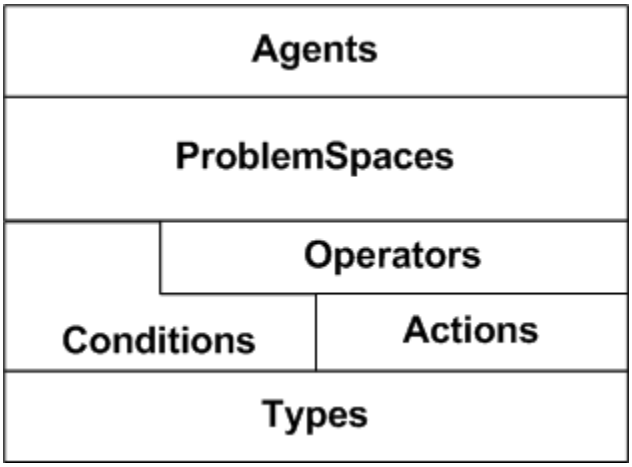


Figure **4-9**: The dependencies between the six different types of libraries in Herbal.

Herbal libraries are uniquely qualified using a namespace. This allows developers to create any number of libraries and share them across models. The Herbal IDE supports library sharing graphically using wizards for the importing and exporting of libraries across projects. As shown in Figure **4-10**, this feature automatically detects library dependencies, thus ensuring that the required library components are included in the export.

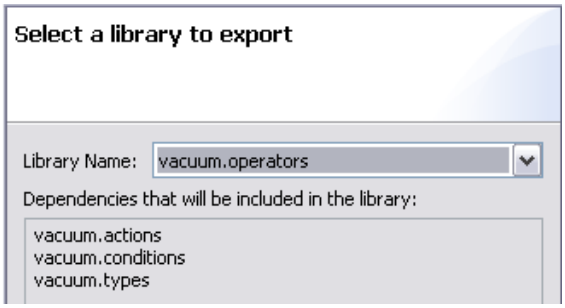


Figure **4-10**: Exporting a library and its dependencies.

The following example illustrates how Herbal supports library reuse. In this example, a modeler creates libraries of basic reusable components for the vacuum cleaner agent environment (Cohen, 2005) and prefixes them with the namespace 'vacuum'. A different modeler then uses these components to build additional higher-level libraries and new, more aggressive vacuum cleaner agents. The modeler prefixes the new libraries with the namespace 'aggressive' (see Figure **4-11**).

This type of layered reuse is common in traditional software development. This is a great example of support for the Krueger's fourth dimension – integration – because of the way it facilitates the combination of reusable components into a working model.

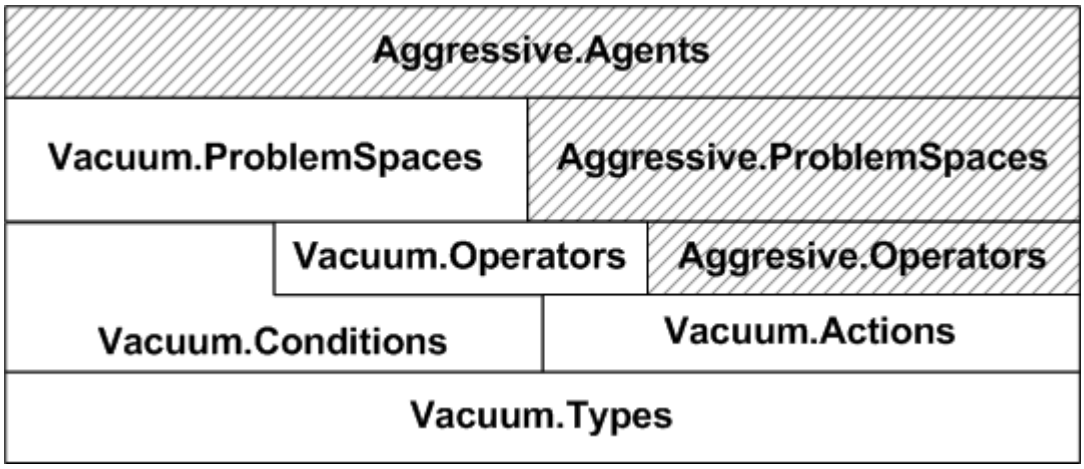


Figure **4-11**: Building custom agents by reusing libraries.

Behavior Design Patterns

In addition to the PSCM level, Chapter 2 and Chapter 3 illustrated the importance of supporting additional high-level behavioral abstractions, like the procedural patterns implemented in RAPs (Firby, 1989), the BDI framework supported by JACK (Norling, 2004), and the activation tables supported in HLSR (Jones, Crossman, Lebiere, & Best, 2006).

Structured programming paradigms like looping constructs can be useful in agent programming, but can be a challenge to program in a typical rule-based language. In addition, modelers often copy looping constructs throughout an agent program. High-level support for these constructs can allow modelers to reuse complex behavior, as opposed to duplicating it.

For example, agents created for graphical agent environments such as the Vacuum Cleaner Environment (Cohen, 2005) and the dTank environment (Ritter, Kase,

Bhandarkar, Lewis, & Cohen, 2007) often implement looping constructs. For the vacuum cleaner agents, behaviors like "*while the vacuum is on a clean square search for dirt using this pattern of movement*" are common, and for the dTank agents, behaviors like "*while no enemy tank is spotted search for an enemy using this search strategy*" are common.

To address this problem and to promote the reuse of high-level meta-behaviors such as looping, the Herbal development environment utilizes a Behavior Design Pattern Wizard (see Figure **4-12**). This wizard makes it possible for the agent developer to generate instantiations of useful meta-behaviors using existing PSCM components. The Wizard automatically creates these PSCM components to produce behavior within a problem space.

The Design Pattern Wizard is also an excellent example of support for Krueger's third dimension of reuse (specialization) (1992). Using the wizard, modelers can transform a general design pattern into a more specialized object tailored to their needs.

I have also designed Design Pattern Wizard to be extensible. This gives ambitious users the ability to plug-in support for the instantiation of additional patterns of high-level behavior (e.g., BDI constructs and HLSR activation tables).

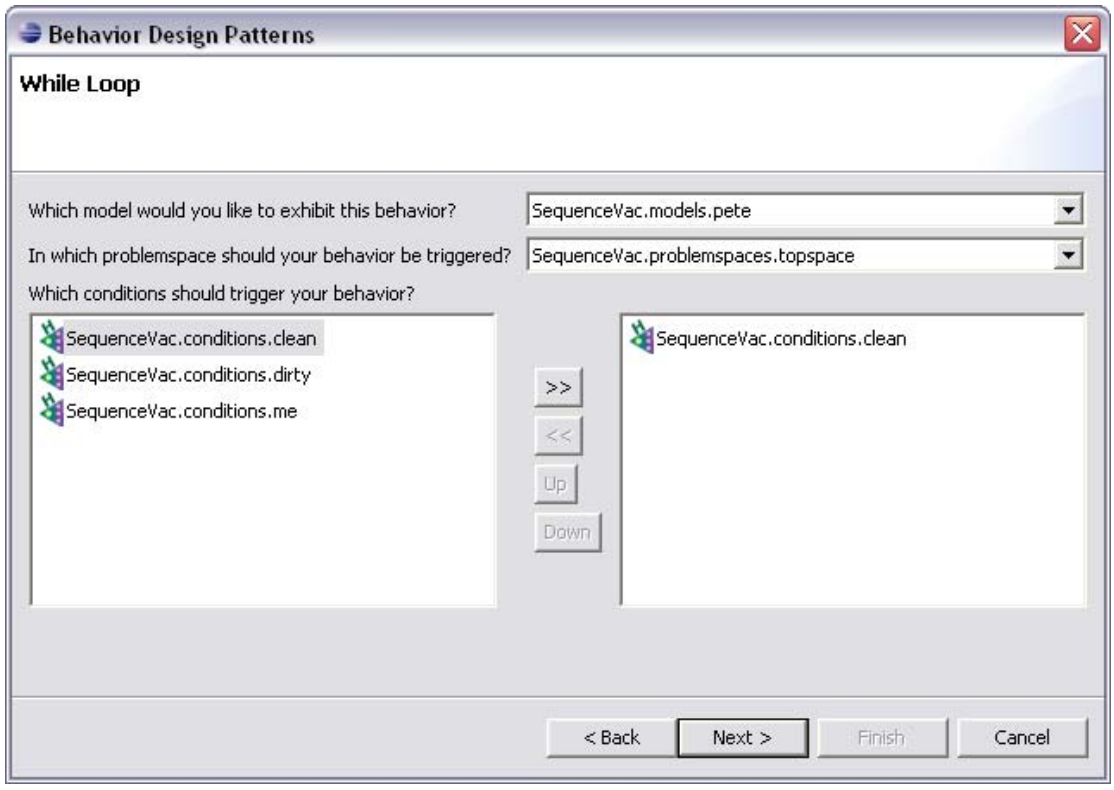


Figure **4-12**: The Behavior Design Pattern Wizard.

Herbal: A Tool for Supporting Programming at Various Levels of Abstraction

The design of Herbal includes support for programming at several different levels of abstraction. Figure **4-13** summarizes these levels and Appendix A gives examples of a model represented at each of these levels.

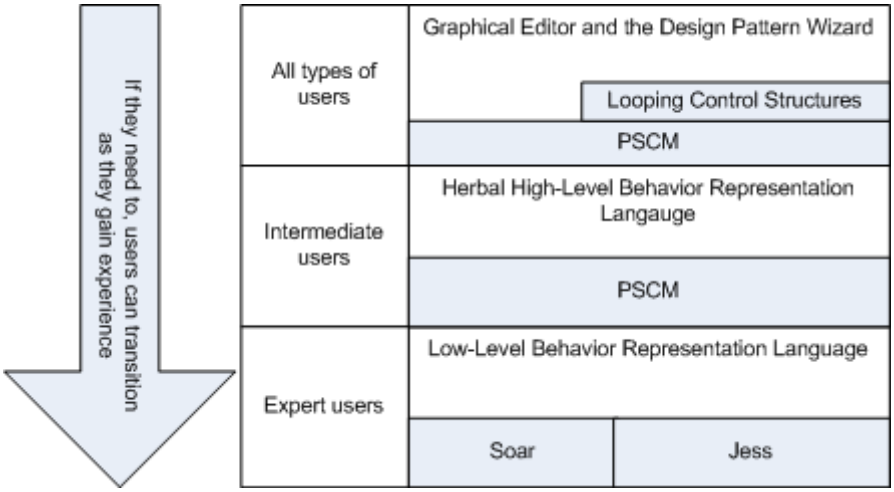


Figure **4-13**: Supporting multiple levels of abstraction in the Herbal Development Environment.

The graphical level represents the highest level of abstraction supported by Herbal. The Herbal GUI Editor (Figure **4-4),** Model Browser View (Figure **4-7**)**,** and Behavior Design Pattern Wizard (Figure **4-12)** form the core of this level. Using these graphical tools, modelers can interact with the model visually and at a level above the syntax (Boshernitsan, 2003; Dann, Cooper, & Pausch, 2008) using two different abstract behavior representations (i.e., the PSCM and Looping Control Structures). The modular design of the Design Pattern Wizard allows for the addition of other abstract representations in the future, such as the BDI framework (Norling, 2004) and the constructs supported by HLSR (Jones, Crossman, Lebiere, & Best, 2006).

The middle layer represents a level of abstraction that allows programming with XML using the PSCM (Friedrich, Cohen, & Ritter, 2007). The use of XML in this layer allows modelers to interact with the model using a variety of specialized XML editors, or simply an ordinary text editor. The use of the PSCM here helps reduce the conceptual

gap between a theoretical representation of the behavior, and the underlying rule-based code.

The bottom layer allows for programming using a low-level representation language (Jess or Soar). This level allows for model refinement (Salvucci & Lee, 2003) by fine tuning the resulting low-level code produced automatically by the Herbal compiler.

Modelers are free to interact with the model using all three of these layers. The graphical nature of the first layer is well suited for novice modelers, but is useable by modelers with any level of experience. As modelers gain experience and encounter situations that need more control, they can interact with the model at progressively lower levels. This structure provides support for users as they gain experience and transition from novice to expert (Powers, Ecott, & Hirshfield, 2007).

Summary

The design and implementation described in this chapter is based on theories developed by the software engineering community about how to solve complex problems with software solutions (Chapter 3). Theories about high-level languages, reuse, and maintenance-oriented environments are central to Herbal's design. In addition, Herbal leverages theories from cognitive science, such as the PSCM to make it easier to develop useful agents and cognitive models (Chapter 2).

The focal theory introduced in this dissertation proposes that the combination of these theories can simplify agent and cognitive model development. The evolution and

ultimate success of this focal theory must be guided and evaluated using both formative and summative evaluations. The design presented in this chapter incorporates the lessons learned during the formative evaluation of Herbal (Chapter 5). In addition, the success of this theory has been measured using two different summative evaluations (Chapter 6 and Chapter 7). The next three chapters explain these evaluations in detail

Chapter 5

Evaluating Design: A Formative Evaluation of Herbal

Producing useful and usable software requires continuous and iterative evaluation (Boehm, 1988a; Rosson & Carroll, 2002). It is helpful to categorize evaluation as either formative evaluation or summative evaluation (Scriven, 1967). Formative evaluation is useful during the design of a system. Designers use feedback from formative evaluations to inform future design. It is common to perform several formative evaluations during system development (Boehm, 1988a; Rosson & Carroll, 2002).

Summative evaluations evaluate the quality of a completed design, and typically take place when system development is complete. The results of a summative evaluation provide a measurement of how well the system meets specific design objectives (Rosson & Carroll, 2002). Figure **5-1** illustrates the difference between formative and summative evaluation.

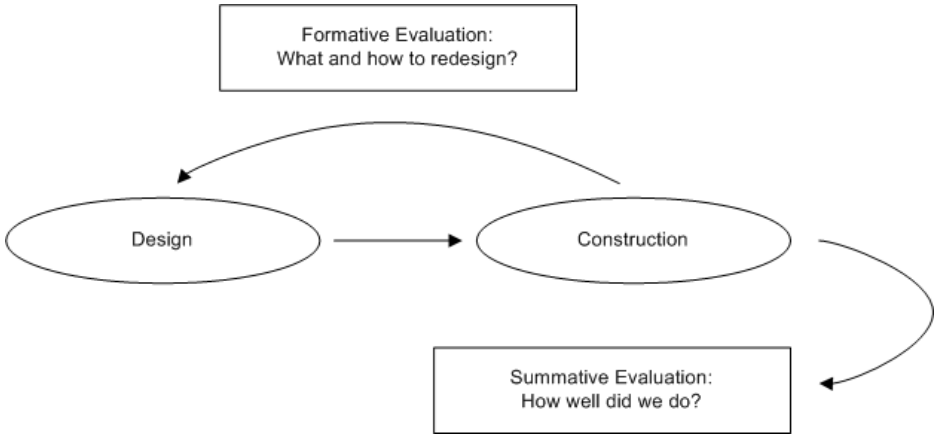


Figure **5-1**: Formative and summative evaluation (Rosson & Carroll, 2002).

Following these guidelines, the design and implementation of Herbal underwent both formative and summative evaluation during its development. The formative evaluation of Herbal is described in detail in this chapter.

In the fall of 2006, an empirical formative evaluation of a prototype of Herbal, and some of the theories it embeds, was conducted to inform the design of Herbal. The Vacuum Cleaner Environment (discussed next) (Cohen, 2005) was used as a basis for the tasks used to evaluate Herbal. This environment was chosen because it is simple enough to introduce to undergraduates, yet complicated enough to allow for the creation of interesting agents. In addition, this environment is colorful and entertaining, thus holding the interest of the study participants. Understanding the formal evaluation of Herbal requires a basic understanding of the Vacuum Cleaner Environment. A description of this environment follows.

Overview of the Task

The Vacuum Cleaner Environment is based on a very simple world that was introduced in a widely used Artificial Intelligence text book, *Artificial intelligence: A modern approach* by Stuart Russell and Peter Norvig (2003). In the Vacuum Cleaner World, a vacuum cleaner resides in an environment that contains two squares: A and B. Each square can be either clean or dirty. The vacuum cleaner's percepts allow it to detect what square it is in and the state of the square (i.e., clean or dirty). In addition, the vacuum cleaner can perform four actions: move left, move right, suck, or do nothing. This environment is useful because its entire state space, consisting of only eight states, can be easily illustrated and explored. In addition, if a performance measure is used, the concept of agent rationality (Russell & Norvig, 2003) can be introduced.

There are several implementations of the Vacuum Cleaner World available. For example, the Pyro robotics toolkit (Blank, Kumar, Meeden, & Yanco, 2006) includes an implementation in Python. Another interesting extension of the Vacuum Cleaner World, created by Musicant and Exley (2004), allows students to program a physical robot to navigate a simplified version of the Vacuum Cleaner World. Additional implementations, in a variety of languages, are included on the official website for Artificial Intelligence: A Modern Approach (aima.cs.berkeley.edu).

While these implementations are useful for introducing basic agent programming concepts, they are either too simplistic for more advanced rule-based programming, or require the overhead of expensive hardware. To effectively evaluate Herbal, a custom graphical agent environment was created in Java (Cohen, 2005). This environment adds

complexity to the Vacuum Cleaner World described earlier. In addition, this environment supports rule-based programs written in two widely used agent architectures: Jess and Soar. A screenshot of the Vacuum Cleaner Environment is shown in Figure **5-2**.

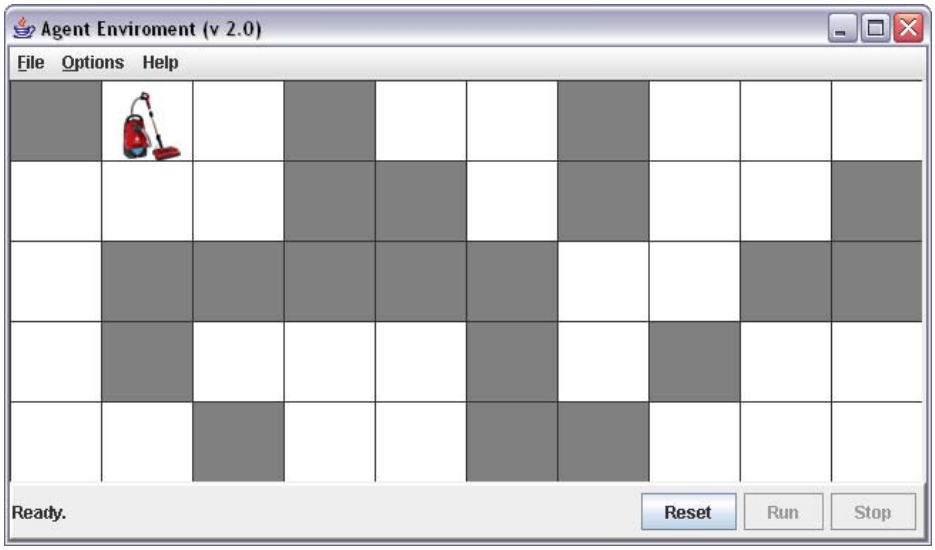


Figure **5-2**: The Vacuum Cleaner Environment.

Method

This section describes the method used for a formal study conducted in parallel with an undergraduate artificial intelligence class. The ultimate goal of this study was to improve the design of Herbal and the Vacuum Cleaner Environment. Specifically, this study was designed to measure four different factors:

- The students' impressions of rule-based programming in general, and Jess specifically
- The students' impressions of graphical development environments in general, and Herbal and the Vacuum Cleaner Environment specifically

- The students' impressions of higher-level methods for organizing rules in general, and the use of the PSCM specifically
- The students' impressions of the Herbal high-level language.

This study took advantage of cognitive dimensions research (Blackwell & Green, 2003; Blackwell & Green, 2000) to evaluate the Herbal Integrated Development Environment (Cohen, Ritter, & Haynes, 2005). These dimensions provide a framework and a common vocabulary that can be used to judge the design of a notational system like Herbal (Blackwell & Green, 2003).

Table **5-1** shows the eight cognitive dimensions selected as usability evaluation criteria. These dimensions were chosen because they measure the degree in which the principles that mediated the design of Herbal were achieved (i.e., embracing high-level languages, enabling reuse, and supporting maintenance-oriented development).

Table **5-1**: The Cognitive dimensions used to evaluate the design of Herbal.

Cognitive Dimension	Description
Closeness of mapping	How closely does the behavior representation language match the way that the modeler describes the behavior?
Error-proneness	How easy is it to make errors using the behavior representation language?
Hidden dependencies	How easy does the behavior representation language make it to create hidden dependencies between model entities?
Premature commitment	How often is the developer forced to make a commitment in the model before there is enough information to make the commitment?
Provisionality	How easy is it to make provisional commitments that can be corrected at a later time? Provisionality allows modelers to easily examine design options and construct what-if scenarios.
Role-expressiveness	How easy is it to discover why a modeler has chosen a particular design? Explicit support for design rationale, as discussed earlier, improves a systems role-expressiveness.
Viscosity	How easy is it to make changes to an existing model? The less the viscosity, the easier it is to change the model.
Visibility	How easy is it to view the elements in a model, including their internal details?

Participants

The seven participants recruited for this study were undergraduate students majoring in Computer Science (CS) or Computer Information Science (CIS) at Lock Haven University and were enrolled in an upper-level Artificial Intelligence course at Lock Haven. Participants were not paid for taking part in this study. Seven students in

the class agreed to participate: one CIS student and six CS students. Each participant was assigned a Participant ID and this ID is the only way that participants can be associated with the data collected during the study. The Lock Haven University Institutional Review Board (IRB) approved the study prior to its implementation.

Apparatus

Participants used Dell Desktop computers running Linux to complete the required tasks. These desktops are all located in the Lock Haven Penguin Lab and are equipped with a keyboard, a mouse, a 100MB external hard-drive, and a 17-inch flat screen monitor.

The required software for this experiment was installed on each machine. The software was Eclipse (3.2.1), Java (1.5), Herbal (2.0.2 Pre-release D), Jess (6.1), the Vim text editor, and the Vacuum Cleaner Environment (2.0).

Design

As part of the course requirements, all students were asked to complete four assignments. The assignments turned in by the students who agreed to participant in this study were used for the formative evaluation. The first assignment asked the participants to create a Jess program that simulated customers entering a bank and waiting in a queue for service. This assignment measured the participants' initial impressions of rule-based programming in Jess, and of graphical development environments in general.

The second assignment required the participants to create two vacuum cleaner models. The purpose of this assignment was to measure the participants' impressions of rule-based programming in Jess, graphical development environments, and the Vacuum Cleaner Environment.

The third assignment asked the students to use Jess modules to create a vacuum cleaner agent that operated in the PSCM. The purpose of this assignment was to measure the participants' impressions of problem spaces and the PSCM from the perspective of organizing and modularizing code.

The fourth assignment was to repeat assignment number three, but to use an early prototype of the Herbal high-level language and development environment (Version 2.0.2 Pre-Release D) to create the agent. The purpose of this assignment was to measure the participants' impressions of Herbal.

Data collection consisted of participant observation and quantitative and qualitative survey questions. Participant observations and open-ended survey questions were coded based on the cognitive dimensions in Table **5-1**. Portions of the assignments were completed during class time so that participant observation could be conducted. Upon completion of each assignment, surveys were administered to the participants.

Table **5-2** provides a summary of the four tasks performed by the participants.

Table **5-2**: Summary of the experimental design for the formative evaluation.

Exp	Task	Data Collected	Purpose
1	Create a Jess program that models customers entering a bank and waiting in a queue for service	The Jess source code Completed survey Participant observations	To measure student impressions of rule-based programming and graphical development environments
2	Create a vacuum cleaner agent that cleaned a room Create a second vacuum cleaner agent that cleaned a room and also kept track of how many squares it cleaned so that it would halt when the room was clean	The Jess source code Completed survey Participant observations	To see if the participants' impressions of rule-based programming and graphical development environments changed after using the Vacuum Cleaner Environment To measure the students' impressions of the Vacuum Cleaner Environment
3	Use Jess modules to create a vacuum cleaner agent that operated in problem spaces	The Jess source code Completed survey Participant observations	To measure the participants' impressions of problem spaces and the Problem Space Computational Model
4	Use the Herbal Graphical Development Environment to create a vacuum cleaner agent that operated in problem spaces	The Jess source code Completed survey Participant observations	To measure the participants' impressions of Herbal

Procedure

On the first day of class, participants were recruited from the group of students enrolled in the course. The study began with each participant reading and signing the consent form as well as completing a User Background Survey, which collected basic

information about his or her background and expectations prior to participating in the study.

During the semester, participants were assigned each of the four assignments in order. Assignments were completed both during class time, and outside of class. When participants were given class time to work on the assignments, observations about the participant's performance, as well as the interactions between the experimenter and the participant, were /noted by the experimenter. When participants finished each assignment, they were asked to complete a different user reaction survey for each assignment. The surveys were designed to measure the four objectives given in the Methods section. The Results section contains details about the content of these surveys.

The first assignment asked the participants to create a Jess program that simulated customers entering a bank and waiting in a queue for service. The simulation operates by generating random numbers that determine how much time will elapse before the next customer enters the bank, and how much time it will take for the teller to service the current customer. For example, customers can arrive at the bank in intervals between 1 and 10 minutes, and tellers can take between 1 and 7 minutes to service a customer. The simulation was run for 1000 simulated minutes, and during this time customers were added to a queue when they enter the bank and, as the teller becomes available, customers were removed from the queue so they can be serviced by the teller. The wait time for each customer was be calculated as the amount of time the customer spends on line, and did not include the time the customer spends with the teller.

Participants worked alone on this assignment and used the Vim text editor to create their programs. Although hard to control, participants were asked not to use

graphical development environments and debuggers. When the assignment was finished, participants were each asked to complete User Reaction Survey #1.

The second assignment required the participants to create two vacuum cleaner agents. The first agent was a simple agent that cleaned a dirty room. This agent was run with no state, no penalty for movement, no radar sensor, and in an environment two squares wide and one square tall. Participants were asked to record the best possible score for a run of 10 steps and the average score of their agent. The second agent operated in the same environment; however, this agent was allowed to maintain state and was assigned a penalty for each movement. Students were asked to minimize the penalty by remembering where the vacuum had been so it stopped moving when all squares were visited. Participants worked alone on this assignment and used the Vim text editor to create their programs. Again, graphical development environments and debuggers were forbidden. When the assignment was finished, participants were each asked to complete User Reaction Survey #2.

Problem spaces are simulated in Jess using Jess modules (Friedman-Hill, 2003). The third assignment asked the students to use Jess modules to create a vacuum cleaner agent that operated in problem spaces. The problem space hierarchy and the relationships between them are shown in Figure **5-3**.

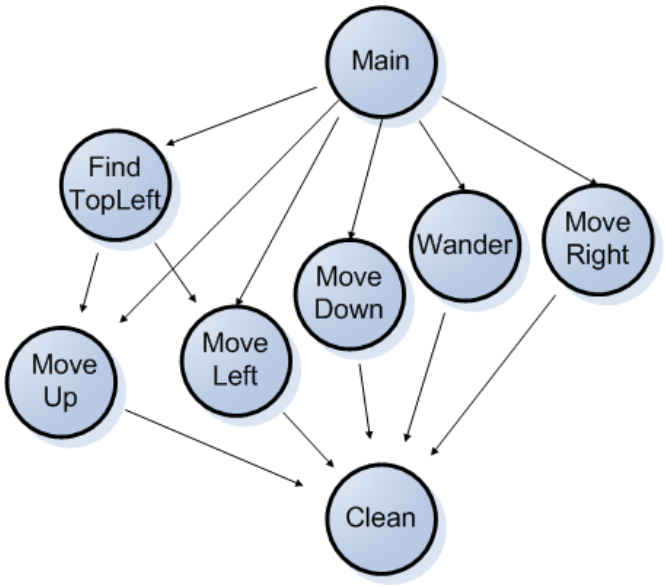


Figure **5-3**: Problem space hierarchy for assignments 4 and 5.

When the agent in the third assignment started, it entered the FindTopLeft problem space which caused it to go immediately to the top left square on the board; cleaning dirty squares along the way. The FindTopLeft problem space used the MoveUp and MoveLeft problem spaces to accomplish its goal and the MoveUp and MoveLeft problem spaces used the Clean problem space to make sure squares were cleaned a long the way.

After the agent arrived at the top left square, it walked the perimeter of the board, cleaning any dirty squares it encountered during its travels. While the agent walked the perimeter, it was asked to assert the following three facts: a fact that represents the height of the board, a fact that represents the width of the board, a fact that represents the total number of squares on the board. The MoveUp, MoveLeft, MoveDown, and MoveRight problem spaces accomplished this behavior.

After the agent walked the entire perimeter, it entered a problem space called Wander that caused the agent to explore the board using the following algorithm. If the agent was on a dirty square, it cleaned it. If there was a dirty square adjacent to the agent, it should move to that square. If there were no dirty squares near the agent, it should randomly move to a new square, if the agent had visited every square on the board since it began to wander, it should stop moving.

As in the first two assignments, participants worked alone on assignment three and used the Vim text editor to create their programs. Graphical development environments and debuggers were forbidden. When the assignment was finished, participants were each asked to complete User Reaction Survey #3.

The fourth assignment was to repeat assignment number three, but to use the Herbal development environment (Version 2.0.2 Pre-Release D) to create the agent, instead of Vim. Participants worked alone on assignment four. When the assignment was finished, participants were each asked to complete User Reaction Survey #4.

Results

Throughout this study, data were collected using surveys and participant observation. Many of the questions in the surveys were designed to measure the cognitive dimensions listed in Table **5-1**. Although all of the participants completed each of the four required assignments, not all participants choose to complete each survey (despite constant reminders). Table **5-3**, Table **5-4**, Table **5-5**, and Table **5-6** show quantitative results for each of the four surveys. The number of participants that

completed each survey is indicated in the caption of each table. In addition, if a question or result mapped to a cognitive dimension, it is indicated in the Table.

Table **5-3**: Quantitative results from User Reaction Survey #1 (N=6).

Impressions of rule-based programming and graphical development environments

I understand the main constructs in Jess but I find it difficult to implement them because the Jess syntax is difficult.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**2**	**1**	**3**	0
Programming agents would be easier if the behavior of my running agent was displayed visually in a graphical environment.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**4**	**1**	0	0
Using print statements to print the progress of my agent in a console window is all want in order to help me create and debug my agents.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	0	**3**	**3**	0
I would enjoy programming in Jess more if there was a better development environment.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**2**	**3**	0	0

Table **5-4**: Quantitative results from User Reaction Survey #2 (N=7).

Impressions of rule-based programming, graphical development environments, and the Vacuum Cleaner Environment

I understand the main constructs in Jess but I find it difficult to implement them because Jess syntax is difficult.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**1**	**1**	**5**	0
Programming agents would be easier if the behavior of my running agent was displayed visually in a graphical environment.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
2	**4**	**1**	0	0
Using print statements to print the progress of my agent in a console window is all want in order to help me create and debug my agents.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	0	**2**	**5**	0
The vacuum cleaner graphical agent environment made programming agents more fun.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
4	**3**	0	0	0
The vacuum cleaner graphical agent environment made it easier to learn how to create rule-based agents.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**4**	**2**		
The vacuum cleaner graphical agent environment had just the right amount of complexity to make it possible to create interesting agents without getting distracted by the details of the environment.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**6**	0	0	0

Table **5-5**: Quantitative results from User Reaction Survey #3 (N=6).

Impressions of problem spaces and the Problem Space Computational Model				
The ability to group a set of operators and behavior into a problem space makes it easier to create complicated agents.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**4**	**2**	0	0
A graphical environment that simplified the use of problem spaces, operators, and impasses is needed to make them useful in Jess.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**1**	**3**	**1**	0
Breaking my agent code into problem spaces made it possible to breakup complicated agent behavior into smaller, less complicated parts.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
3	**2**	**1**		
It would be easier to use problem spaces if there was a graphical debugger that showed my agent as it moved from problem space to problem space.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**3**	**2**	0	0

Table **5-6**: Quantitative results from User Reaction Survey #4 (N=4).

Impressions of the Herbal Prototype

If given the choice, I would rather use Herbal that pure Jess in order to complete the agent programming assignments given in this course.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	0	0	**2**	**1**
Herbal would be easier to use if there were better visualizations of the agent structure.			***Measures Visibility***	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
3	**1**	0	0	0
It takes less time to create an agent using Herbal that to write code in Jess.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**1**	**1**	**1**	0
It takes less time to learn how to use Herbal than to learn how to write Jess Code.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**2**	**1**	**1**	0
The Herbal GUI editor makes it easier than Jess programming to recognize components of my agent (problem spaces, operators, etc.).			***Measures Visibility***	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**1**	**2**	**1**	0
Herbal makes it easier than Jess to reuse conditions and actions in my agent.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**1**	**2**	0	0
Herbal's XML language is easy to read/understand.			***Measures Closeness of Mapping***	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**2**	**2**	0	0
I would rather write code in Herbal using thee XML high-level language than with the GUI editor.			***Measures Closeness of Mapping and Viscosity***	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**2**	0	**2**	0
Herbal makes it easier than Jess to change my agent.			***Measures Viscosity***	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	0	0	**3**	0
Herbal placed very little restrictions on the order in which I created my agent.			***Measures Provisionality and Premature Commitment***	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**1**	**1**	**2**	0

Table **5-7** shows the qualitative results from Survey #4, and Table **5-8** shows the observations made while the participants were working on the assignments. The

responses to the open-ended questions, and the observations made while programming, were coded based on the related cognitive dimensions. This coding is displayed in Table **5-7** and Table **5-8**.

Table **5-7**: Qualitative results from User Reaction Survey #4.

Impressions of the Herbal Prototype

What part of Herbal did you find most useful?		
Response	**# Responding**	**Cognitive Dimension**
Syntax becomes a non-issue	2	Closeness of mapping
Wiring aliases	1	N/A
What part of herbal did you find most confusing?		
Response	**# Responding**	**Cognitive Dimension**
Understanding the order in which to create components	2	Provisionality and Premature Commitment
Wiring aliases	2	N/A
Getting a high-level picture of the agent structure	1	Visibility
If you were in charge of programming Herbal, what improvements would you make?		
Response	**# Responding**	**Cognitive Dimension**
Visual representation of the model structure	3	Visibility
Wizard or flow-chart that helps you create components	2	Provisionality and Premature Commitment

Table **5-8**: Observation of participants completing assignment 4.

Observation	Cognitive Dimension
Participants had problems understanding what an alias is in Herbal. They struggled with this term. Discussions with participants revealed that it helped them to think of them as input and output variables.	Closeness of Mapping
Participants had problems understanding when you would want to use a problem space as opposed to just an operator. Thinking of the problem space as a behavior seemed to be very helpful.	Closeness of Mapping
Participants had a hard time understanding the term impasse. It helped to explain the impasse as a set of conditions that cause entry into a problem space.	Closeness of Mapping
Participants had problems debugging common problems. For example, they struggled figuring out why an agent was not entering a specific problem space or why an operator was not firing.	Role-expressiveness Hidden Dependencies
Participants were frustrated by the requirement to fully specify a component when it was created.	Provisionality, Premature Commitment Viscosity Hidden Dependencies
Participants were frustrated when the system forced them to delete all references to a component before they could delete the component.	Provisionality, Premature Commitment Viscosity
Participants were frustrated by the lack of warnings. The system produced errors for situations that occur during development but were easily corrected later in the development process. There errors were highly dependent on the order in which the model was created. The participants would prefer these to be reported as warnings.	Provisionality, Premature Commitment Viscosity
In some cases, participants were allowed to make certain mistakes that caused the visual editor to stop functioning and could only be fixed using the XML code.	Error-proneness
Participants continued to express the need for a high-level visualization of the model and its structure.	Visibility
Participants continually commented that they would have rather learned Herbal and then Jess instead of the other way around. They all felt that Herbal is useful in learning how to program in pure Jess.	N/A

Discussion

Responses to the first two surveys (Table **5-3** and Table **5-4**) indicate that after the first assignment, participants were divided about their comfort level with Jess syntax. Two out of six found the syntax challenging, one was neutral, and three did not find the syntax difficult at all. The level of comfort with Jess syntax was not surprising: especially because this evaluation was conducted in an upper-level, CS/CIS course using students with considerable programming experience.

The participants comfort level with Jess syntax increased after completing the second assignment, with five out of seven disagreeing with the statement that Jess syntax is difficult. Reasons for becoming more comfortable with Jess syntax could be related to gaining more experience with the language. One might expect less comfort with Jess given a more diverse set of participants.

In addition, participants agreed that being able to view a running agent visually in a graphical environment would help make agent programming easier. They also expressed the need for more than just console output for debugging their agents. Responses to these same questions remained strong after they were introduced to the Vacuum Cleaner Environment in the second assignment.

Survey #2 (Table **5-4**) shows participants were positive about the effectiveness of the Vacuum Cleaner Environment. Participants found that the environment made the programming assignments easier and more enjoyable. In addition, participants felt that the Vacuum Cleaner Environment was created with just the right amount of complexity.

Results from Survey #2 supported the belief that graphical agent environments can make agent programming more enjoyable for students.

Responses from Survey #3 (Table **5-5**) validated the use of the PSCM as the foundation for the Herbal high-level language. Participants agreed that the PSCM made agent programming easier because it componentized their agents. In addition, responses showed that participants favored the idea of a development environment and debugger that supported the PSCM. Results from Survey #3 illustrate that a higher-level language that allows programmers to organize rules into higher-level structures was appreciated, and that the PSCM is a good choice for this purpose. This was a very encouraging result because it suggests that the PSCM may be a familiar construct in computer science students, not just psychology.

Results from Survey #4 (Table **5-7**), which are directly related to the design of Herbal, are mixed. Most participants felt that they would rather program using pure Jess than the Herbal Development Environment. The also felt strongly that Herbal needed better visualizations of the agent's structure. In addition, participants were not convinced that Herbal made it easier to make changes to agent code. They also felt that Herbal forced them to work in a particular order when developing agents. This means that Herbal poorly supports the Visibility, Viscosity, Provisionality, and Premature Commitment dimensions. In addition, mixed responses from participants about the time it takes to learn and use Herbal also indicated a need for design changes.

However, some responses in Survey #4 were positive. For example, participants found it easier to reuse model components using Herbal than when programming using

pure Jess. In addition, participants found the XML high-level language used by Herbal to be easy to read and understand.

Interestingly, in Survey #4 half of the participants preferred programming by editing the Herbal XML high-level language, while the other half preferred the GUI editor. Herbal was designed to support both methods of programming because it was believed that preferences, and requirements, for both styles of programming exist (Powers, Ecott, & Hirshfield, 2007; Salvucci & Lee, 2003). These results support this design choice.

Responses to the open-ended questions (Table **5-7**) and the participant observations (Table **5-8**) were used to help discover the reasons behind some of the negative responses in Survey #4. These reasons were used to help improve the design of Herbal. For example, the frustration with the order that Herbal enforced while creating agents is evident in both the open-ended questions and the participant observations. Participants did not like having to provide a complete specification for a component at the time it was created. They also did not like having to remove references to a component before the component could be deleted. These problems made it difficult to create and change an agent. This feedback suggests the need for design changes for better support of the Viscosity, Provisionality, and Premature Commitment cognitive dimensions.

Another problem indicated in both the open-ended questions and participant observations was poor support for the Visibility cognitive dimension. Specifically, participants requested better visualizations of the model structure. The need for this type of visualization was also evident during participant observation.

Participants also had trouble getting comfortable with some of the terminology used by Herbal. For example, participants struggled with the difference between a problem space and an operator. Discussions with participants suggested that it helped to refer to problem spaces as behaviors or goals. Participants also indicated that they preferred to think of aliases as input/output variables and impasses as a set of conditions for entry into a new behavior.

Participant observations and survey responses indicated that participants had trouble finding and fixing errors in their agents. One possible factor was that the console method of debugging caused execution to be traced using rules instead of the PSCM terminology used when the agent was created. Participants were trying to see what problem space their agent was in, and which operator was recently applied, but the trace they were using contained a list of rules. This mismatch between the behavior representation language and the way that trace describes the model's behavior resulted in poor support for the Closeness of Mapping dimension. A good debugger or tracing tool that maps directly to the PSCM rather than the rules would help here.

Finally, bugs within the GUI editor, that allowed fatal mistakes to be made (e.g., the GUI editor stopped functioning) that could only be fixed in the XML code, frustrated participants, and this frustration was evident during participant observation and in open-ended responses. Experimenter observations indicated that this frustration was a major cause for some of the negative responses in Survey #4.

Conclusions

There were several important lessons learned during the formative study described here, and many of these lessons resulted in changes in the Herbal design. For example, participants felt strongly that Herbal needed a better visualization of the agent structure. This feedback resulted in the development of the Model Browser View in Herbal. This window shows a hierarchical view of a model's structure, giving the programmer a high-level picture of the model and its components. This view was specifically designed to improve Herbal's support for Visibility.

In addition, participants were annoyed by the fact that Herbal forced them to work in a particular order when developing agents. To correct this problem, "soft" warnings were implemented in Herbal. During normal development, an agent is often only partially completed. With the addition of soft warnings, an incomplete agent produces a message that is passively displayed in the Eclipse output window. When a warning is displayed, the developer is allowed to continue without interruption. This makes it possible for developers to work in any order by building or editing models that are not yet complete. The warnings remind the programmer which components are not complete, but they do not prevent the developer from continuing. This should lead to better support for Viscosity and Premature Commitment.

The participants' difficulty debugging models indicated poor support by Herbal for Role Expressiveness and Closeness of Mapping. Participants had problems finding and fixing problems in existing agents, which indicates poor understanding of how the model works and what the model was doing. To correct this problem, working sets that

leveraged existing design rationale, were added to Herbal. These working sets should make it easier for modelers to find task relevant model components during maintenance. In addition, a graphical debugger was built that traces model execution using PSCM components rather than rules. These additions directly targeted better support for Role Expressiveness and Closeness to Mapping.

To correct the participants' problems with terminology some of the model components were renamed. For example, aliases were renamed to input and output variables, and the concept of an impasse is now presented as a set of conditions for entry into a problem space. In addition, the concept of a behavior was introduced to help users understand problem spaces, and a Design Pattern Wizard was created to make it easy for users to create new model behaviors that are ultimately represented as problem spaces. All of these changes were implemented to improve Herbal's support for Closeness of Mapping.

Finally, several bugs in the GUI Editor were discovered during this study. These bugs frustrated participants and made it difficult for them to complete the tasks. All the bugs identified during the formative evaluation were fixed.

Results from this study also indicated strengths in the current Herbal design. These results helped confirm many of the design decisions that were made early on in development process. For example, the choice to use the PSCM as the basis of the Herbal high-level language was confirmed by participants, as they indicated that the PSCM made agent programming easier. In addition, the emphasis on reuse during Herbal's design was successful as participants found it easier to reuse model components using Herbal.

The decision to use XML for Herbal's high-level language was also supported by this study. Participants clearly found the XML high-level language used by Herbal to be easy to read and understand. Finally, the design decision to allow users to edit Herbal code using both the GUI Editor and by directly editing the XML code was appreciated by participants. Half of the participants preferred programming by editing the Herbal XML high-level language, while the other half preferred the GUI editor.

Table **5-9** summarizes the lessons learned during this study, and the changes that were implemented to address these lessons.

Table **5-9**: Summary of the design changes resulting from the formative study.

Formative Result	Design Change
Herbal needed a better visualization of the agent structure	Added a Model Browser View
Herbal forced them to work in a particular order	Implementation of "soft" warnings
Difficulty debugging models	Implementation of working set feature that leverages existing design rationale, and a graphical debugger based on the PSCM
Problems with some PSCM terminology	Aliases renamed to input/output variables, impasses presented as conditions for entry, and behavior design pattern associates problem spaces with agent behaviors
Participants encountered frustrating bugs in the GUI editor	Bugs fixed

The formative evaluation presented here resulted in the confirmation of many of the design decisions made during the implementation of the Herbal prototype. In addition, survey results and participant observations lead to several changes in Herbal.

Of course, the success of these changes needed to be evaluated. Two different summative evaluations were conducted to measure the overall effectiveness and usability of the Herbal system. These evaluations are described in detail in the next two chapters.

Chapter 6

Evaluating Functionality: Herbal as a Cognitive Modeling Tool

Two summative evaluations were performed to evaluate both the functionality and the usability of Herbal. The functionality of Herbal as a cognitive modeling tool was evaluated by creating a cognitive model (using only Herbal) that was capable of learning in a competitive environment.

The usability of Herbal was evaluated with the design and implementation of a summative usability study. The criteria for success in this evaluation were acceptable ratings for a collection of cognitive dimensions. This chapter describes the evaluation of the functionality of Herbal as a cognitive modeling tool. The next chapter covers the usability evaluation.

Cognitive models of people interacting in competitive environments can be useful, especially in games and simulations (Jones et al., 1999; Laird, 2001a, 2001b; Ritter et al., 2003). To be successful in such environments, it is necessary to learn the strategy used by the opponent. In addition, as the opponent adjusts its tactics it is equally important to unlearn opponent strategies that are no longer used.

Learning by reflection (or introspection) is one technique that can be used to learn and unlearn an opponent's changing strategies while at the same time preserving the variability in which people learn (e.g., Bass, Baxter, & Ritter, 1995; Cox & Ram, 1999; Ritter & Wallach, 1998).

Learning by reflection is a form of metacognition that allows the model to learn by reflecting on its performance, and adjusting accordingly. When reflection reveals previous actions that were beneficial, the model should be more likely to repeat those same actions in similar situations. However, when reflection reveals poor performance, the actions that lead to that performance are less likely to be repeated. Thus, learning by reflection is a form of reinforcement learning (Russell & Norvig, 2003).

Reflective learning requires that both the cognitive model and its environment are fully observable (Russell & Norvig, 2003). In other words, the model must be able to observe the effects of its actions on the environment and other models.

Because reflective learning strategies are based on probability, the behaviors they generate are not deterministic. This allows reflective models to exhibit variability in learning and thus performance. When playing a game or participating in a simulation, variable behavior is a crucial part of the realism that these systems must portray.

For all the reasons already discussed, using Soar to create non-deterministic models that learn is a challenge. Therefore, this problem serves as a good test of the functionality and usefulness of Herbal.

Overview of the Task

Lehman, Laird, and Rosenbloom (1996) in their A Gentle Introduction to Soar use baseball repeatedly as an example. This inspired me to implement a simple version of a baseball game to study adversarial problem solving and support people learning Soar. In

a broader context, this environment provides an accessible platform for the future study of cognitive models interacting with other agents in a social simulation (Sun, 2006).

Figure **6-1** shows the basic interface and one of the feedback screens. In this game, participants play the role of the pitcher competing against a series of agent-operated batters. The goal of this game, as in baseball, is to get batters out.

The baseball game described here was written in Java and interacts with the Soar cognitive architecture using the Soar Markup Language (SML Quick Start Guide, 2005). The software and instructions on how to use it are available online (acs.ist.psu.edu/herbal).

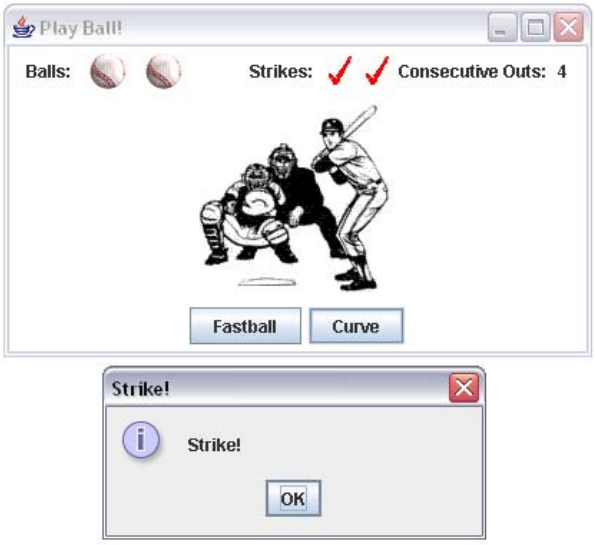


Figure **6-1**: The baseball game task.

There are two ways to get a batter out in this game: The batter can get three strikes (a strike results when a batter either swings and misses or does not swing at a good pitch), or the batter can hit the ball directly at a fielder who catches the ball.

There are also two ways for a batter to get on base in this game: The batter can get four balls (a ball results when the batter does not swing at a bad pitch), or the batter can hit the ball in a way that prevents the fielders from catching it.

The pitcher has a choice of throwing either a fastball or a curveball to the batter. Once it threw a pitch, the batter had a choice of either swinging at the pitch or letting it go by. Both the pitcher and batter are always aware of how many balls and how many strikes the batter has. The rules shown in Table **6-1** describe how to determine the outcome of each pitch.

Table **6-1**: Determining the outcome of a pitch.

Pitcher	**Batter Response**	**Outcome**
Fastball	Batter swings	Contact is made that may result in either an out (50% of the time) or a hit (50% of the time).
Fastball	Batter does not swing	The pitch will result in a strike.
Curveball	Batter swings	The pitch will result in a strike.
Curveball	Batter does not swing	The pitch will result in a ball.

Based on the rules described in Table **6-1**, the most certain way to get a batter out is to throw a curveball when the pitcher thinks the batter will be swinging and to throw a fastball when the pitcher thinks the batter is not going to swing. Naturally, if the participant can learn what strategy the batter is using then they have a better chance of getting them out.

Method

The purpose of this study was to measure how quickly participants learned batter strategies while performing the baseball task described above and to compare this performance to a cognitive model designed to do the same task with similar performance and with equal variability.

Participants

Participants were recruited from the set of undergraduate students majoring in Computer Science (CS) and Computer Information Science (CIS) at Lock Haven University. Ten participants were chosen: nine of the 10 participants were male. Each participant was assigned a Participant ID based on the order in which they requested to participate in the study. This ID is the only way that participants can be associated with data collected during the study. The Lock Haven University Institutional Review Board (IRB) approved the study prior to its implementation.

Apparatus

A Lenovo T60p laptop computer was used by the participants to perform the required task. This laptop was in a docking station equipped with a keyboard, a mouse, speakers, a 250MB external hard-drive, and a 17-inch flat screen monitor.

Design

In this study, participants and several models performed the baseball task described above. As the participants and the model performed the task, each pitch thrown, the batter's reaction, and the outcome of the pitch was recorded in a log file.

Using the data recorded in the log file, the model's and the participant's ability to learn a particular strategy was measured quantitatively using a measure of pitching efficiency (*PE*). The following formula was used to calculate pitching efficiency:

$$PE = Ns / Ts$$

Where *Ns* is the number of batters using strategy *s* that were faced by the participant, and *Ts* is the consecutive out threshold for strategy *s*. A decrease in *PE* indicates an increase in the efficiency of the pitcher. A value of 1.0 for *PE* indicates the most efficient pitching strategy because it means the pitcher retired every batter they faced that was using a particular strategy. For example, if a participant faced 14 Aggressive batters before they could retire seven in a row, the participant's pitching efficiency would be 14 / 7, or 2.

Procedure

The study began with each participant reading and signing the consent form. Next, each participant was given instructions explaining the rules of the baseball task as described above. After reading the instructions, the participants were ready to perform the baseball task.

During the task, each participant faced five different batter strategies, each represented by a different model created using Herbal. Each participant faced the same set of strategies and in the same sequence. As the task progresses, strategy changes took place based on the number of consecutive outs that the participant recorded against a given strategy. When a predetermined out threshold was reached, a strategy shift by the batter would take place. The exact sequence of batter strategies and their corresponding out thresholds was defined in a configuration file that was used by the baseball environment, but the participant did not know what type of strategies to expect, or when strategy changes would take place. The batter strategies, along with their consecutive out thresholds, are shown in Table **6-2**.

Table **6-2**: Batter strategies in the baseball environment.

Name	**Strategy**	**Out Threshold**
Hacker	Always swings	4
Aggressive	Swings at the first pitch and when there are fewer strikes than balls, unless there are three balls and two strikes	7
Random	Randomly chooses when to swing	5
Chicken	Never swings	4
Alternate	Swings if the last pitch was a fastball and does not swing if it is the first pitch or the last pitch was a curve	7

Participants were given as much time as needed to complete the task and were allowed to consult the instruction sheet during play. All the participants seemed to have no problem understanding the game and no questions were asked while performing the task.

Models

Six cognitive models were written to conduct the study described here. All six models were written using the Herbal high-level language and development environment and were completed in just one day. Because of the use of the Herbal high-level/ language and graphical editor, the creation of the models described here required only an understanding of the PSCM and some visual modeling techniques. This serves as an example of how Herbal can provide modelers without a strong programming background access to the complicated machinery used by architectures that may traditionally be out of their reach.

Batter Models

Five cognitive models were written to represent the strategies used by the batter (Hacker, Aggressive, Random, Chicken, and Alternate). These models are not capable of learning and served only as opponents that exhibit the behavior described in Table **6-2**

Pitcher Model

A sixth model was written in Herbal to play the role of the pitcher. The goal of the pitcher model was to exhibit behavior similar to that demonstrated by the participants. Unlike the batter strategy models, the pitcher model was able to learn using reflection. More specifically, this model operated within two problem spaces: one to deliberate what pitch to throw next, and one to reflect on recent performance and modify future deliberation.

The pitcher model started with an equal probability of throwing a curveball or a fastball. Within the explicit reflection problem space, the pitcher model considers the following features of the environment: the previous number of balls and strikes on the batter, the previous pitch thrown, and the outcome of that pitch. If the outcome is positive (i.e., a strike was called or the batter struck out) the pitcher adjusts a probability so that it is more likely to throw the same pitch the next time it encounters this situation. If, on the other hand, the outcome was negative (i.e., a ball was called, or contact was made by the batter, including contact resulting in an out), and the pitcher had previously experienced a positive outcome in this situation (a strike or a strikeout), the probability of throwing the same pitch in that situation was decreased.

Model Parameters

The pitcher model takes two parameters: the learning rate and the unlearning rate. The learning rate specifies how quickly the model will commit to throwing a particular pitch in a particular situation; in other words, how quickly the probability increases given

a positive outcome. The unlearning rate specifies how quickly the model will reduce this learned commitment. The best values for these learning rates almost certainly depend on the nature of the particular task.

Considering the relatively simple rules in the baseball task described above, it was expected that participants would be able to learn strategies quickly. In addition, it was hypothesized that participants would at first be reluctant to unlearn until they were sure that a strategy shift has taken place. Given persistent negative feedback on a previously learned response, participants should eventually accelerate their unlearning rate.

The fact that four of the five batter strategies are deterministic further justifies these parameter values. When a particular pitch works for a batter in a specific situation, it will continue to work until a strategy shift takes place. After a particular pitch stops working for a batter, it can be assumed that a strategy shift has occurred.

As a result, in an effort to match human behavior the pitcher model described here was equipped with a fast learning rate and an initially stubborn, but later accelerating, unlearning rate. Figure **6-2** depicts the learning and unlearning rates used by the model.

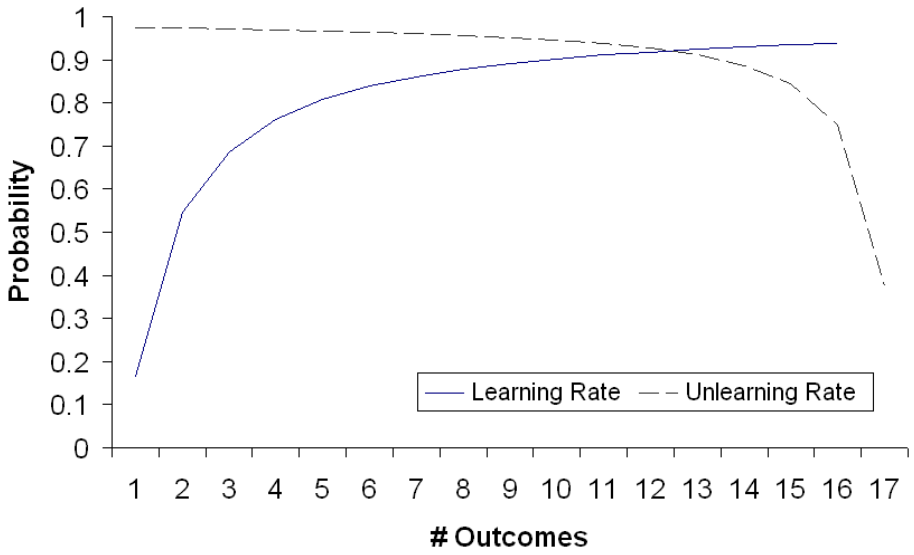


Figure **6-2**: Learning and unlearning rates used by the model.

Results

Because a primary goal of this work was to produce a model that not only matches the average pitching efficiency, but also matches the variability in pitching efficiency, the cognitive models created here are not deterministic. This made it possible to consider each run of the model as being equivalent to a participant run. To reduce any sampling error with this theory, the model was run 100 times.

Table **6-3** shows the results of the participant study and of the model executions. The mean pitching efficiency and the standard deviation of the pitching efficiency are listed for all participants and all model runs. Recall that the smaller the pitching efficiency the more efficient the pitcher, and the most efficient strategy has a *PE* equal to 1.0. In addition, Table **6-3** shows the out threshold for each strategy in square brackets.

Figure **6-3** visualizes the data listed in Table **6-3**. Each bar in Figure **6-3** represents the mean pitching efficiency. White bars represent the participant data and shaded bars represent the model data. The error bars in Figure **6-3** signify one standard deviation from the mean pitching efficiency.

Table **6-3**: Pitching efficiency for the participants and the learning model.

	Participants (N = 10)		**Model (N = 100)**	
Strategy	**Mean**	**StdDev**	**Mean**	**StdDev**
Hacker [4]	1.53	0.80	1.69	0.70
Aggressive [7]	1.81	1.62	1.13	0.20
Random [5]	5.00	6.24	5.36	4.67
Chicken [4]	1.03	0.08	1.25	0.33
Alternate [7]	1.54	0.72	3.53	2.01

Discussion

Analysis of Figure **6-3** reveals that the model's behavior matched both the participant's mean performance, and variability in performance, for three of the five presented strategies. However, for two of the strategies the model did not satisfactorily reflect the participant's performance.

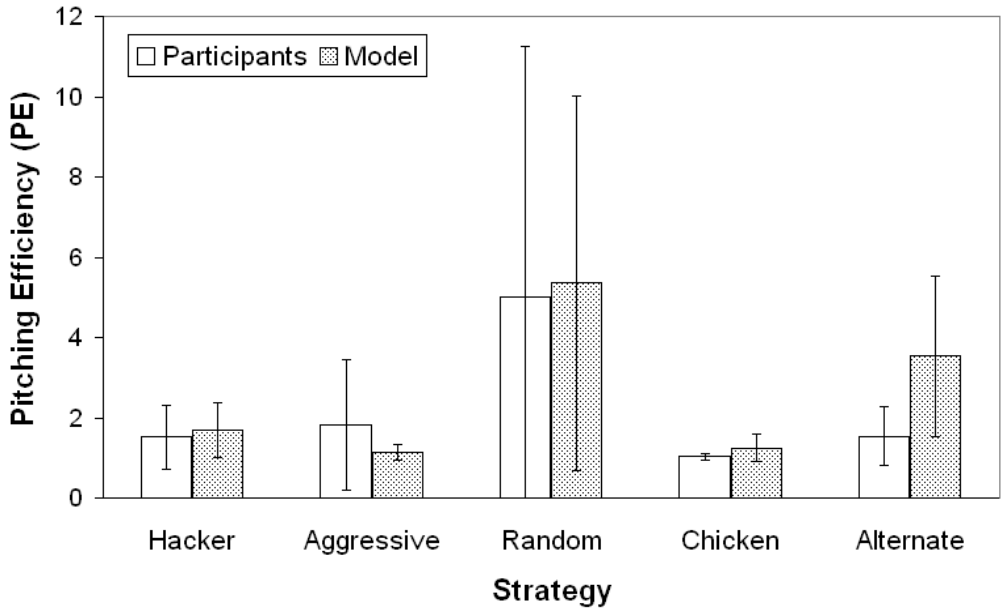


Figure **6-3**: Comparison of the model and participants for each batting strategy.

Hacker and Chicken Strategies

The model's performance matched very well for both the Hacker and Chicken strategies. Given the simplicity of the learning strategy used, this is an interesting result. Both the participants and the model were able to retire the requisite number of consecutive batters quickly and without much variability. Interestingly, the Hacker strategy proved to be more difficult for both the participants and the model. This may be because the very aggressive strategy used by the Hacker makes it more likely for the batter to get a hit when the pitcher made a mistake. On the other hand, the reserved approach used by the Chicken strategy only punishes mistakes with a single ball as

opposed to a hit. In this baseball task, an aggressive batter strategy is more dangerous to the pitcher than a timid one.

Random Strategy

As expected, the variation of the pitching efficiency against the Random strategy was quite large for both the participants and the model. Both the participant and the model could not consistently figure out the random strategy, because, well, it was random. The difference between the pitching efficiency for the model, and that of the participants, might be related to the number of participants run. Due to the random nature of this strategy, additional participants might cause these means to match more closely.

Aggressive and Alternate Strategies

Unexpectedly, the model did not do as good of a job matching the Aggressive and Alternate strategies. The order in which these strategies are presented may play an important role here. One possible explanation for these problems is that the unlearning rate used by the model is not fast enough. While good enough to match the transitions between some strategies, the unlearning rate may need to be faster in other cases. To understand this theory, the transitions from the Hacker strategy to the Aggressive strategy, and from the Chicken strategy to the Alternate strategy, need to be examined more closely.

Transition from Hacker to Aggressive

Because the Hacker strategy always swings, the pitcher must learn to throw a series of curveballs to get a batter out consistently. In addition, the inability to quickly unlearn the Hacker strategy is not immediately detrimental when an Aggressive batter follows the Hacker strategy. For example, if the pitcher continues to throw a series of curveballs to an Aggressive batter, the batter will not get on base until after the sixth curveball is thrown. This gives the pitcher several pitches, and therefore a lot of time to unlearn the strategy.

On the other hand, if the participant or model quickly unlearns the Hacker strategy, it will lead to throwing an early fastball, which will result in a 50% chance of the batter getting a base hit. In other words, in this particular case quickly unlearning the previous strategy is not beneficial. This might explain why the model performed better against the Aggressive strategy; the model simply does not unlearn as quickly as the participants did, and this proved to be more efficient in this particular ordering of strategies.

Transition from Chicken to Alternate

The opposite can be said about the transition from the Chicken strategy to the Alternate strategy. A series of consecutive fastballs will get a batter out using the Chicken strategy because this strategy never swings. However, if this knowledge goes unlearned, the same series of fastballs thrown to an Alternate batter will result in frequent hits because the Alternate batter swings immediately after a fastball is thrown. In this

particular case, failure to quickly unlearn the Chicken strategy results in poor performance and might explain why the model did not perform as well as the participants in this case. Once again, it appears as if the model did not unlearn the learned strategy quickly enough in this particular ordering of strategies.

Unfortunately, the reflective learning strategy is fundamentally limited in how quickly it can unlearn. This limit may be a major reason for the model's inability to unlearn the Chicken strategy quickly enough. The learning algorithm used here cannot unlearn unless it has already encountered positive feedback. This causes a problem if the model's initial encounter with a strategy involves a series of negative outcomes, which is precisely the case when transitioning from Chicken to Alternate. Augmenting the algorithm to use Soar's numeric-indifferent preference might eliminate this limitation and possibly improve the model's fit.

Additional Explanations

Factors other than unlearning rate may have also had an effect on the model's inability to match the participant's behavior. For example, if the pitcher follows the simple pattern of throwing a fastball, followed by curve, followed by fastball, they will always get the Alternate batter out. While speculative, it is possible that participants were quick to recognize this alternating pattern while the model did not treat alternating patterns any differently from other patterns.

Conclusions

A major outcome of this study is a demonstration that cognitive models that compete in adversarial environments using introspective learning can be written quickly and easily using Herbal. All the cognitive models used in this study were created in a single day using only the Herbal toolset. This serves as an example of how Soar models that learn can be written without directly writing Soar productions, hopefully making Soar available to a wider audience of modelers. In addition, the pitcher model was compared to participants' performance and was shown to match both the participants' mean performance and variability in performance against many of the presented batting strategies.

In addition, for strategies that the model did not satisfactorily master, insight into the limitations of the algorithm used, and how people possibly perform this task, was gained. These results motivate future work that will lead to improvements in the learning algorithm, and in the Herbal high-level language. For example, one way to improve the learning algorithm is to take advantage of Soar's numeric-indifferent preference.

However, Herbal's usability by a range of modelers with different backgrounds was still in question. The next section describes a summative usability study, conducted with first-time Herbal users, designed to measure the general usability of Herbal.

Chapter 7

Evaluating Usability: A Summative Usability Evaluation of Herbal

As mentioned in Chapter 5, producing useful and usable software requires continuous and iterative evaluation. Chapter 5 described the implementation of a formative evaluation to ensure continual improvement, in both the functionality and usability, of the Herbal system. Chapter 6 described the implementation of a summative evaluation of the usefulness of Herbal. This chapter describes the results of a summative evaluation that uses a new method for analyzing cognitive dimension data, and provides a measurement of the final usability of the Herbal system.

Overview of the Task

This main task in this study was to create a working intelligent agent that operates a vacuum cleaner in the Vacuum Cleaner Environment (Cohen, 2005). Chapter 5 describes the Vacuum Cleaner Environment in detail.

The study design divides the main task into three subtasks: creating a reusable library of agent components; creating a vacuum cleaner agent using this library; and finding and fixing a bug in the resulting vacuum cleaner agent. These three subtasks exercise all of the features of the Herbal Development Environment and closely mirror

the different phases of agent/model development: creating reusable components, using these components to build agents, and debugging the resulting agents.

The first subtask exercised Herbal's library creation facilities. In this subtask, participants created a new library and populated it with low-level components needed to build vacuum cleaners. The task instructions encouraged the inclusion of design rationale in the library throughout this task.

The second subtask exercised Herbal's model creation facilities. This task consisted of building a vacuum cleaner agent out of higher-level model components. Participants created these higher-level components by reusing the low-level components contained during the first subtask. The design rationale located in the library was available to participants to help them during the task. In addition, participants were encouraged to include further design rationale about the newly created components.

The third subtask exercised Herbal's model maintenance facilities. This task consisted of finding and fixing an error in an existing vacuum cleaner model (the experimenter injected the same error before the start of the third subtask). The broken model did not clean dirty squares when the vacuum cleaner encountered them. During this task, the participants used Herbal's debugger and working set feature to fix the broken vacuum cleaner. In addition, the participant was encouraged to view design rationale when needed.

Method

This study evaluated the usability of the Herbal Integrated Development Environment (Cohen, Ritter, & Haynes, 2005) based on Cognitive Dimensions research (Blackwell & Green, 2003; Blackwell & Green, 2000).

The data generated by a user reaction survey, and by participant observations, measured support for the Cognitive Dimensions shown in Table **7-1**. The user reaction survey was based on a validated and generalized Cognitive Dimension evaluation done by Blackwell and Green (2000). Their evaluation showed that a generalized Cognitive Dimension questionnaire of system users is useful for evaluating usability.

Table **7-1** shows the nine Cognitive Dimensions that this study used for usability evaluation criteria. These nine dimensions best measure the degree in which Herbal achieves the principles that mediated its design (i.e., embracing high-level languages, enabling reuse, and supporting maintenance-oriented development).

Table **7-1**: Cognitive dimensions used as evaluation criteria for the summative usability study.

Cognitive Dimension	Description
Visibility	How easy is it to view the elements in a model, including their internal details?
Viscosity	How easy is it to make changes to an existing model? The less the viscosity, the easier it is to change the model.
Diffuseness	How many symbols or how much space does the notation require to produce a certain result or express a meaning?
Hard-mental operations	How much hard mental processing lies at the notational level, rather than at the semantic level? Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening?
Error-proneness	How easy is it to make errors using the behavior representation language?
Closeness of mapping	How closely does the behavior representation language match the way that the modeler describes the behavior?
Role-expressiveness	How easy is it to discover why a modeler has chosen a particular design? Explicit support for design rationale, as discussed earlier, improves a systems role-expressiveness.
Progressive evaluation	How easy is it to evaluate and obtain feedback on an incomplete solution?
Premature commitment	How often is the developer forced to make a commitment in the model before there is enough information to make the commitment?

Because of the difficulty involved in observing when a participant is experiencing Hard-Mental Operations, only survey questions measured this dimension. The criteria used to categorize dimensions were different depending on the type of data analyzed: survey or observation.

A five-level Likert scale structures the survey questions. The term *Dimension of Concern* was used to categorize a dimension in which more than 20% of the participants

responded negatively. A negative response is an answer in the bottom two levels of the Likert scale.

For participant observations, the difference between the number of participants experiencing positive events and the number of participants experiencing negative represents a score for each dimension. In this study, a Dimension of Concern has a negative score whose absolute value is larger than 20% of the total participants.

For example, if a participant experienced six positive events and nine negative events related to Viscosity, the score for Viscosity would be negative four. If there were 10 participants, Viscosity would be a Dimension of Concern because the score is negative and its absolute value is greater than 20% of the number participants (2 in this example).

For the Herbal system, a reasonable goal is to have fewer than 20% of the participants experience problems. However, in general researchers should select a threshold based on the needs of the users, and the importance of the dimensions considered in the context of the task. This is especially true because of the tradeoffs that exist between different dimensions. A very high threshold might lead to frequent inability to complete a task, as well as very frustrated users. A very low threshold may be necessary for critical tasks where mistakes can be catastrophic, but this can also lead to a less flexible user experience(Blackwell & Green, 2003). The method presented here is a general approach because other researchers can adjust the threshold for a wide variety of dimensions, tasks, and systems.

How well Herbal supports the nine dimensions listed in Table **7-1** determines the success of this evaluation. The final analysis of this study gives special attention to the areas of weakness found during the formative evaluation described in Chapter 5. Recall

that the formative evaluation concluded a need for better support for Visibility, Premature Commitment, Role Expressiveness, and Closeness of Mapping.

Participants

The participants recruited for this study had limited or no experience developing cognitive models or intelligent agents. Participants recruited were from the set of undergraduate students majoring in Computer Science (CS), Computer Information Science (CIS), Management Information Science (MIS), and Bachelor of Science in Psychology (PSYC) at Lock Haven University. These majors represent likely users of the Herbal system. Participants received $10 for taking part in this study, which required approximately an hour.

Twenty-four students participated: 12 PSYC students and 12 CS/CIS/MIS students. The number of college credits completed by the participants ranged from 42 to 132. The average number of hours per week spent using a computer was 4.00 for PSYC majors and 10.00 for CS/CIS/MIS majors. The average number of programming courses previously taken was 0.25 for PSYC majors and 4.75 for CS/CIS/MIS majors. Finally, nine of the 14 PSYC majors are female and one of the CS/CIS/MIS students is female.

A Participant ID identified each participant based on the order in which they requested to participate in the study. This ID is the only way that participants can be associated with data collected during the study. The Lock Haven University Institutional Review Board (IRB) approved the study prior to its implementation.

Apparatus

Participants used a Lenovo T60p laptop computer to perform the required subtasks. This laptop was docked in a docking station equipped with a keyboard, a mouse, speakers, a 250MB external hard-drive, a 17-inch flat screen monitor, and a microphone.

Camtasia Studio 2.0.2, created by TechSmith, recorded both the screen and audio while the participants perform the subtasks. Additional software that was required for this experiment was Eclipse (3.2.1), Java (1.5), Herbal (3.0), and the Vacuum Cleaner Environment (2.0).

Design

The study design placed participants into groups of three. Groups contained either all PSYC majors or all CS/CIS/MIS majors. This resulted in eight groups of three participants: four groups consisting of PSYC majors and four groups consisting of CS, CIS, and MIS majors.

As described earlier, the main task for this study was to create a working intelligent agent that operates a vacuum cleaner in the Vacuum Cleaner Environment. This main task consists of three subtasks: creating a reusable library of agent components; creating a vacuum cleaner agent using this library; and finding and fixing a bug in the resulting vacuum cleaner agent.

Each group completed the main task by finishing each subtask independently, and in turn, as shown in Figure **7-1**.

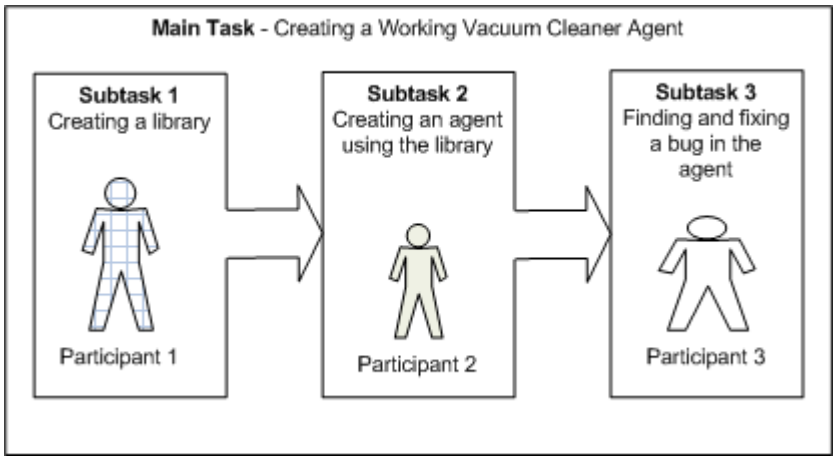


Figure **7-1**: Different participants work in turn to complete the main task.

Procedure

The study began with each participant reading and signing the consent form as well as completing the User Background Survey (see Appendix B), which collected information about his or her background and expectations prior to participating in the study.

Next, each participant watched a 15-minute video before performing his or her subtask. The video provided the participant with a high-level introduction to intelligent agents, the Vacuum Cleaner Environment, the Problem Space Computation Model, and creating and maintaining libraries and agents in the Herbal Integrated Development Environment.

After the video completed, the experimenter informed the participant what subtask he or she will be performing and gave each participant the General Task Instructions (see Appendix B). These instructions ask the participants to think-aloud

during the experiment (Ericsson & Simon, 1993; Newell & Simon, 1972), and to ask questions if they were confused at any time during the experiment. After the participant read the General Task Instructions, the experiment asked them to practice thinking aloud while performing a simple memory recollection exercise that was unrelated to this study.

Next, the experiment provided each participant with Specific Task Instructions (see Appendix B) that provided systematic instructions about how to perform the subtask, and the participants began their subtask.

The experimenter noted observations about the participant's performance during the execution of the task and recorded all observations in a Data Collection Form (see Appendix B). The experimenter focused on recording events related to the Cognitive Dimensions of interest. The same experimenter ran all participants.

Upon completion of the subtask, the experimenter asked the participants to complete a User Reaction Survey (see Appendix B). The User Reaction Survey contained 17 questions that mapped directly to the Cognitive Dimensions shown in Table **7-1**. A five-level Likert scale structured 11 of these questions. The remaining six questions were open-ended and sought information explaining responses to the scaled questions.

Models

As mentioned earlier, the vacuum cleaner agent that participants built in this study reused components contained in a library. The library that participants created and used to build the vacuum cleaner agent consisted of four data types, four vacuum cleaner

actions, and three environmental conditions. Table **7-2** summarizes these library components.

Table **7-2**: The vacuum cleaner library components created by the participants.

Component	Component Type	Description
action	data type	Used by the agent to perform actions like moving or cleaning a square
position	data type	Specifies the location of the vacuum cleaner agent
radar	data type	Contain information about the clean or dirty status of the squares around the vacuum cleaner
spot	data type	Specifies the clean or dirty status of the square currently occupied by the agent
up	action	Causes the vacuum to move up one square
down	action	Causes the vacuum to move down one square
left	action	Causes the vacuum to move left one square
right	action	Causes the vacuum to move right one square
isClean	condition	True if the current square is clean
isDirty	condition	True if the current square is clean
isAlive	condition	True if the vacuum cleaner is operational

The resulting vacuum cleaner agent consisted of higher-level model components, such as operators and problems spaces, that participants built on top of the low-level components contained in the library. Specifically, three high-level behaviors: Survive, Wander, and Clean were created. Table **7-3** summarizes these three high-level behaviors.

Table **7-3**: The high-level model behaviors created by the participants.

Behavior	Description
Survive	This is the model's top-level behavior and is composed of the Wander and Clean behaviors. The model survives by wandering the board and cleaning dirty squares as they are encountered
Wander	Causes the agent to wander in a random fashion as long as it is on a clean square
Clean	Causes the agent to clean the current square

The completed vacuum cleaner model survived by wandering randomly until it encountered dirt. When dirt was encountered the agent cleaned the dirty spot and then resumed wandering, looking for more dirt.

Results

All participants completed the tasks successfully. The mean time to complete a task was 24.38 minutes with a standard deviation of 9.40 minutes. Means (with standard deviations in parenthesis) for the library creation task, the model creation task, and the model maintenance task, took 34.38 (7.76), 23.13 (2.23), and 15.63 (4.63) minutes, respectively.

Table **7-4** lists completion times for all participants by major and by task. Means (with standard deviations in parenthesis) for PSYC majors and CS/CIS/MIS majors, across all tasks, were 27.08 (10.27) and 21.67 (7.95) minutes, respectively. The Soar model contained 13 productions and the Jess model contained 16 productions resulting in approximate mean times per production of 2 minutes / Soar production and 1.5 minutes / Jess production.

Table **7-4**: Task performance times in minutes.

ID	Major	Task	Time	ID	Major	Task	Time
8	PSYC	Library	45	1	CS	Library	30
10	PSYC	Model	25	2	CS	Model	20
11	PSYC	Maint.	15	3	CS	Maint.	20
12	PSYC	Library	40	4	CIS	Library	30
14	PSYC	Model	23	5	CIS	Model	25
15	PSYC	Maint.	20	6	CIS	Maint.	10
16	PSYC	Library	35	7	CIS	Library	35
17	PSYC	Model	20	9	CS	Model	25
18	PSYC	Maint.	15	13	CIS	Maint.	10
19	PSYC	Library	40	20	CIS	Library	20
21	PSYC	Model	25	22	MIS	Model	22
23	PSYC	Maint.	22	24	MIS	Maint.	13
			All Tasks				
Mean	27.08	**STDEV**	10.27	**Mean**	21.67	**STDEV**	7.95
			Library Task				
Mean	40.00	**STDEV**	4.08	**Mean**	28.75	**STDEV**	6.29
			Model Task				
Mean	23.25	**STDEV**	2.36	**Mean**	23.00	**STDEV**	2.45
			Maintenance Task				
Mean	18.00	**STDEV**	3.56	**Mean**	13.25	**STDEV**	4.72

Importantly, there was no statistical evidence of a difference between the mean task times, across all tasks, of the PSYC and CS/CIS/MIS participant groups: $t(22) = -1.44$, $p = .163$, (t-tests for mean task times in the subgroups are not included due to the small number of participants in each subgroup).

The user reaction survey described earlier, and the coded observations of participants performing the subtasks, generated a score for each Cognitive Dimensions shown in Table **7-1**. The following sections discuss the survey and observation results.

Survey Results

Upon completion of the subtask, the experimenter asked the participants to complete a User Reaction Survey based on the questionnaire validated by Blackwell and Green (2000). The User Reaction Survey contained 17 questions that mapped directly to the Cognitive Dimensions shown in Table **7-1**. A five-level Likert scale structured 11 of these questions. The remaining six questions were open-ended and sought information explaining responses to the scaled questions. Unfortunately, participants provided very little useful information in the open-ended questions, making it difficult to provide rationale for the participant's responses.

Table **7-5** lists the 11 scaled questions. When more than 20% of the participants responded negatively to any question pertaining to a specific dimension, that dimension became a Dimension of a Concern. A negative response is a response in the bottom two levels of the scale. Table **7-5** uses shading to represent the positive and negative response ranges for each question. Dark shading identifies positive responses and light shading identifies negative responses.

Table **7-5**: The survey questions used to measure support for various cognitive dimensions. Shading indicates the positive (dark shading) and negative (light shading) response ranges.

Visibility				
Q#1	How easy was it to see or find the various parts (e.g., problem spaces, operators, conditions) of your agent or library while it was being created, changed, or debugged?			
very easy	easy	neutral	difficult	very difficult
Q#2	If you needed to compare different parts (e.g., problem spaces, operators, conditions) of your agent or library, you could easily see these parts at the same time.			
strongly agree	agree	neutral	disagree	strongly disagree
Viscosity				
Q#3	How easy was it to make changes to your agent or library?			
very easy	easy	neutral	difficult	very difficult
Diffuseness				
Q#4	The elements (e.g., problem spaces, operators, and conditions) you used to build your agent or library allowed you to say what you wanted to say reasonably briefly.			
strongly agree	agree	neutral	disagree	strongly disagree
Hard-Mental Operations				
Q#5	In general, the task you performed did not seem especially complex or difficult to work out in your head.			
strongly agree	agree	neutral	disagree	strongly disagree
Error Proneness				
Q#6	During this task, you often found yourself making small mistakes that irritated you or made you feel stupid.			
strongly agree	agree	neutral	disagree	strongly disagree
Closeness of Mapping				
Q#7	The notation (e.g., problem spaces, operators, and conditions) you used to describe your agent or library was closely related to how you might describe the agent or library naturally.			
strongly agree	agree	neutral	disagree	strongly disagree
Role Expressiveness				
Q#8	During the task, you often did not know what many of the agent or library pieces meant (e.g., problem spaces, operators, conditions) but you put them in anyway.			
strongly agree	agree	neutral	disagree	strongly disagree
Progressive Evaluation				
Q#9	It was easy to stop in the middle of creating the agent or library, and check your work so far.			
strongly agree	agree	neutral	disagree	strongly disagree
Q#10	During this task, it was easy to find out how much progress you made, or check what stage in your work you were in.			
strongly agree	agree	neutral	disagree	strongly disagree
Premature Commitment				
Q#11	When working on this task, there were times when you felt like you could have changed the order you performed the steps without breaking the agent or library.			
strongly agree	agree	neutral	disagree	strongly disagree

The upcoming sections summarize participant responses based on each dimension. Reponses were analyzed the same way, using six different groupings: (1) all participants; (2) participants performing the library creation task; (2) participants performing the model creation task; (3) participants performing the model maintenance task; (4) participants majoring in PSYC, (5) for participants majoring in CS, CIS, or MIS.

A single table for each grouping shows the results. For each question, the tables show the number of responses in each of the five Likert levels. The tables use shading to distinguish the positive and negative responses ranges.

Ideally, all responses should be in the heavily shaded area (the positive range). When more than 20% of the responses fall inside the negative response range (the lightly shaded area), the dimension is marked as a Dimension of Concern, and bold font emphasizes the responses that exceed the 20% threshold.

The following six tables present the data for all six groupings.

Table **7-6**: Survey responses for all participants and all tasks (N = 24).

Visibility					
Q#1	3	13	6	2	0
Q#2	1	17	1	**5**	1
Viscosity					
Q#3	12	10	2	0	0
Diffuseness					
Q#4	6	15	3	0	0
Hard-Mental Operations					
Q#5	7	13	0	3	1
Error Proneness					
Q#6	0	4	6	8	4
Closeness of Mapping					
Q#7	8	12	2	2	0
Role Expressiveness					
Q#8	0	4	2	10	8
Progressive Evaluation					
Q#9	10	13	1	0	0
Q#10	7	14	3	0	0
Premature Commitment					
Q#11	3	8	6	**7**	0

Darkly shaded areas indicate the positive response range. Bold values indicate negative responses that exceed the threshold. Visibility and Premature Commitment are Dimensions of Concern because each had more than 20% of their responses within the negative response range.

Table **7-7**: Survey responses for participants performing the library creation task (N = 8).

Visibility					
Q#1	0	3	4	1	0
Q#2	0	4	1	**3**	0
Viscosity					
Q#3	5	1	2	0	0
Diffuseness					
Q#4	2	6	0	0	0
Hard-Mental Operations					
Q#5	1	6	0	1	0
Error Proneness					
Q#6	0	0	4	4	0
Closeness of Mapping					
Q#7	5	2	1	0	0
Role Expressiveness					
Q#8	0	1	1	4	2
Progressive Evaluation					
Q#9	5	3	0	0	0
Q#10	2	5	1	0	0
Premature Commitment					
Q#11	2	3	1	**2**	0

Darkly shaded areas indicate the positive response range. Bold values indicate negative responses that exceed the threshold. Visibility and Premature Commitment are Dimensions of Concern because each had more than 20% of their responses within the negative response range.

Table **7-8**: Survey responses for participants performing the model creation task (N = 8).

Visibility					
Q#1	2	5	1	0	0
Q#2	1	5	0	**2**	0
Viscosity					
Q#3	5	3	0	0	0
Diffuseness					
Q#4	2	6	0	0	0
Hard-Mental Operations					
Q#5	4	3	0	1	0
Error Proneness					
Q#6	0	**2**	1	1	4
Closeness of Mapping					
Q#7	2	5	0	1	0
Role Expressiveness					
Q#8	0	**2**	1	2	3
Progressive Evaluation					
Q#9	4	4	0	0	0
Q#4	3	2	0	0	4
Premature Commitment					
Q#11	1	1	4	**2**	0

Darkly shaded areas indicate the positive response range. Bold values indicate negative responses that exceed the threshold. Visibility, Error Proneness, Role Expressiveness, and Premature Commitment are Dimensions of Concern because each had more than 20% of their responses within the negative response range.

Table **7-9**: Survey responses for participants performing the model maintenance task (N = 8).

Visibility					
Q#1	1	5	1	1	0
Q#2	0	8	0	0	0
Viscosity					
Q#3	2	6	0	0	0
Diffuseness					
Q#4	2	3	3	0	0
Hard-Mental Operations					
Q#5	2	4	0	**1**	**1**
Error Proneness					
Q#6	0	**2**	2	2	2
Closeness of Mapping					
Q#7	1	5	1	1	0
Role Expressiveness					
Q#8	0	1	0	4	3
Progressive Evaluation					
Q#9	1	6	1	0	0
Q#10	1	6	1	0	0
Premature Commitment					
Q#11	0	4	1	**3**	0

Darkly shaded areas indicate the positive response range. Bold values indicate negative responses that exceed the threshold. Hard Mental Operations, Error Proneness, and Premature Commitment are Dimensions of Concern because each had more than 20% of their responses within the negative response range.

Table **7-10**: Survey responses for participants majoring in PSYC (N = 12).

Visibility					
Q#1	2	3	5	2	0
Q#2	1	10	1	0	0
Viscosity					
Q#3	5	5	2	0	0
Diffuseness					
Q#4	4	6	2	0	0
Hard-Mental Operations					
Q#5	3	7	0	2	0
Error Proneness					
Q#6	0	**3**	3	4	2
Closeness of Mapping					
Q#7	4	7	0	1	0
Role Expressiveness					
Q#8	0	**3**	1	6	2
Progressive Evaluation					
Q#9	6	5	1	0	0
Q#0	6	4	2	0	0
Premature Commitment					
Q#11	1	4	3	**4**	0

Darkly shaded areas indicate the positive response range. Bold values indicate negative responses that exceed the threshold. Error Proneness, Role Expressiveness, and Premature Commitment are Dimensions of Concern because each had more than 20% of their responses within the negative response range. In total, there were 117 positive responses and 15 negative responses.

Table **7-11**: Survey responses for participants majoring in CS, CIS, or MIS (N = 12).

Visibility					
Q#1	1	10	1	0	0
Q#2	0	7	0	**5**	
Viscosity					
Q#3	7	5	0	0	0
Diffuseness					
Q#4	2	9	1	0	0
Hard-Mental Operations					
Q#5	4	6	0	1	1
Error Proneness					
Q#6	0	1	4	3	4
Closeness of Mapping					
Q#7	4	5	2	1	0
Role Expressiveness					
Q#8	0	1	1	4	6
Progressive Evaluation					
Q#9	4	8	0	0	0
Q#10	1	10	1	0	0
Premature Commitment					
Q#11	2	4	3	**3**	0

Darkly shaded areas indicate the positive response range. Bold values indicate negative responses that exceed the threshold. Visibility and Premature Commitment are Dimensions of Concern because each had more than 20% of their responses within the negative response range. In total, there were 119 positive responses and 13 negative responses.

A chi-square test of independence was performed to check for a correlation between the number of negative and positive responses and the participants' major (PSYC or CI/CIS/MIS). Table **7-12** summarizes the results of this analysis. Importantly, there is no statistical evidence that there is a correlation between the number and type of

survey responses, and the participants' major: $X^2 = 0.160$, $DF = 1$, $p = .689$ (Howell, 1987).

Table **7-12**: A 2x2 chi-square contingency table used to test for independence between survey responses and participant major.

Major	# Positive Responses	# Negative Responses	Total
PSYC	117	15	132
	118.00	14.00	
	0.008	0.071	
CS/CIS/MIS	119	13	132
	118.00	14.00	
	0.008	0.071	
Total	236	28	264

Expected counts are printed below observed counts and chi-square contributions are printed below expected counts. $X^2 = 0.160$, $DF = 1$, $p = .689$

Summary of Survey Results

Table **7-13** lists Dimensions of Concern based on the survey results. Dimensions of Concern are shown using six different groupings: (1) all participants; (2) participants performing the library creation task; (2) participants performing the model creation task; (3) participants performing the model maintenance task; (4) participants majoring in PSYC, (5) for participants majoring in CS, CIS, or MIS. For example, survey responses from participants majoring in CS, CIS, or MIS indicated that Visibility and Premature commitment were Dimensions of Concern.

Table **7-13**: Dimensions of Concern as measured by survey results.

Dimension	All	Library Creation	Model Creation	Model Maintenance	PSYC	CS/CIS/MIS
Visibility	X	X	X			X
Viscosity						
Diffuseness						
Hard-mental operations				X		
Error-proneness			X	X	X	
Closeness of mapping						
Role-expressiveness			X		X	
Progressive evaluation						
Premature commitment	X	X	X	X	X	X

Observation Results

Upon completion of the study, the experimenter coded the data collection forms and the screen and audio recordings based on the cognitive dimensions shown in Table **7-1**. These observations compliment the survey results previously presented. Unlike the survey results, information from the context of the observations helps explain the positive or negative participant experiences related to a dimension.

The experimenter identified thirty-seven unique event types during observation, and these types were either a negative or a positive contribution to a particular cognitive

dimension. A positive contribution to a cognitive dimension means the software helped a participant in a fashion consistent with the definition of the dimension. A negative contribution to a dimension means the software was a hindrance to the participant in a fashion consistent with the definition of the dimension. To generate these event types, the experimenter relied on the participants' actions and utterances.

Table **7-14** lists the 37 event categories along with their associated dimensions. It is important to keep in mind that this table shows types of events not actual instances. These event types (or codes) are useful for providing the rationale behind the classification of an observation, and for suggesting improvements to future releases of the software.

For example, while editing model components several participants got lost when specifying properties in the model properties dialog box. This was because the dialog box does not contain the name of the component the participant was editing. This problem became apparent when the participant moved the dialog box out of the way in order to see what component they had selected before the dialog box appeared. Category 10 (the participant became confused about what specific component they were working on) in Table **7-14** coded this particular observation, and the suggested design change would be to add the component name to the dialog box.

As another example, category 3 (the participant became confused about what specific component they were working on) in Table **7-14** coded observations of participants having problems interacting with the search feature in the new project dialog box. Several participants entered the project name in the search field, rather than in the project name field. It is likely that the placement of the search field in a location

typically occupied by the project name in other dialog boxes caused this error. The recommended design change might be to rearrange the order of the fields in the dialog box.

Table **7-14**: Event codes used during participant observation.

ID	Event Description	Dimension	+/-
23	Participant appeared confused by notation used to represent agent	Closeness of mapping	-
27	Participant preferred a term not used by high-level language	Closeness of mapping	-
14	The design pattern wizard allowed participants to create several components using a brief terminology	Diffuseness	+
18	Participant was thankful for code automatically created by Herbal	Diffuseness	+
17	Participant found design rationale to be verbose and/or redundant	Diffuseness	-
24	Participant used copy and paste when entering design rationale	Diffuseness	-
2	The consistency in the Herbal interface helped reduce errors	Error prone	+
15	The design pattern wizard prevented errors creating components	Error prone	+
3	The search feature of the new project dialog lead to errors	Error prone	-
11	Participant had problems distinguishing types of design rationale	Error prone	-
12	Participant confused Eclipse export with Herbal library export	Error prone	-
21	Poor design in the working set dialog lead to errors	Error prone	-
25	Trouble locating the Herbal GUI Editor	Error prone	-
28	Instructions mislead participant to create action instead of type	Error prone	-
31	Participant selected wrong problem space in design pattern wizard	Error prone	-
37	Participant entered literal value in local variable edit box	Error prone	-
38	Participant confused by wire screen when there is nothing to wire	Error prone	-
29	Participant confused by conditions with no restrictions	Error prone	-
6	Lack of required order made it easier to fix mistakes	Premature commitment	+
8	Participant changed order of steps in task without problems	Premature commitment	+
19	Participant was confused by order required to run debugger	Premature commitment	-
7	It was easy for participants to check the status of the model, and for any errors, by looking at what has been done so far	Progressive evaluation	+
10	The participant became confused about what specific component they were working on, or what step they were doing	Progressive evaluation	-
13	Participant viewed rationale to learn more about model	Role expressiveness	+
16	Participant demonstrated strong understanding of the model	Role expressiveness	+
30	Commented that components are self-explanatory	Role expressiveness	+
33	Participant entered quality design rationale	Role expressiveness	+
26	Participant demonstrated poor understanding of the model	Role expressiveness	-
5	Participant had trouble understanding component/subcomponent	Role expressiveness	-
32	Participant viewed rationale but did not find it helpful	Role expressiveness	-
36	Participant misunderstood interface between model/environment	Role expressiveness	-
20	Behavior of agent was easy to see using the debugger	Visibility	+
1	A portion of the Herbal GUI editor was hidden	Visibility	-
4	Easy for participant to make a change to a component	Viscosity	+
34	Working sets helped participant find location of a problem	Viscosity	+
9	Participant had problems editing an action	Viscosity	-
35	Participant had problems editing an operator	Viscosity	-

Of the 37 event types shown in Table **7-14**, 12 are related to the Error Prone cognitive dimension, eight to Role Expressiveness, four to Viscosity and Diffuseness, three to Premature Commitment, and two to Closeness of Mapping, Progressive Evaluation, and Visibility. In addition, 23 of the 37 types represented negative events and 14 represented positive events.

For each dimension, the difference between the number of participants experiencing positive events and the number of participants experiencing negative represented a score for that dimension. A Dimension of Concern has a negative score whose absolute value is larger than 20% of the total participants.

Table **7-15** lists the total number of events observed for all subjects and all tasks. The table groups these events by their associated dimension and sorts them by total number of events. In addition, the table shows the total number of participants experiencing positive and negative events and the final score for each dimension.

For example, for all participants performing all tasks, there were 69 events observed that relate to the Error Prone cognitive dimension. Of these 69 events, 16 participants experienced a positive event related to Error Proneness and 21 experienced a negative event related to Error Proneness. This results in a score of minus five, which has an absolute value greater than 20% of the total number of participants. Therefore, Error Proneness is as a Dimension of Concern. Overall, there were 241 event instances.

Table **7-15**: Number of occurrences by cognitive dimension for all participants and all tasks (N = 24).

Dimension	Total Events	# Participants Experiencing Positive Events	# Participants Experiencing Negative Events	Final Score
Error prone	69	16	21	-5
Progressive evaluation	53	21	9	12
Viscosity	47	16	7	9
Premature commitment	27	10	3	7
Role expressiveness	21	10	6	4
Diffuseness	13	7	4	3
Visibility	8	5	2	3
Closeness of Mapping	3	0	3	-3
Total	**241**			

Participants experienced more negative events than positive events for two dimensions: Error Proneness and Closeness of Mapping. Only Error Proneness exceeded the 20% threshold. As a result, Error Proneness is a Dimension of Concern.

Table **7-16** and Table **7-17** provide a more detailed look at the dimensions listed in Table **7-15**. These data provide rationale for the classification of each dimension, as well as suggestions for improvements in future releases. The first table lists the positive events by event type and the second table shows the negative events by event type. For example, of the nine positive Diffuseness events, six were the result of interaction with the Design Pattern Wizard (event #14) and three the result the participants' reaction to the automatically generated code (event #18). In addition, for all participants, event number seven was the most observed positive event; 21 participants experienced this event.

Table **7-16**: Coded positive observations for all participants and tasks (N = 24).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Progressive evaluation	40	7 - It was easy for participants to check the status of the model, and for any errors, by looking at what has been done so far	21
Viscosity	36	4 - Easy for participant to make a change to a component	13
Error prone	12	2 - The consistency in the Herbal interface helped reduce errors	11
Premature commitment	12	6 - Lack of required order made it to fix mistakes	8
Premature commitment	12	8 - Participant changed order of steps in task without problems	9
Role expressiveness	7	16 - Participant demonstrated strong understanding of the model	7
Diffuseness	6	14 - The design pattern wizard allowed participant to create several components using a brief terminology	6
Error prone	6	15 - The design pattern wizard prevented errors creating components	6
Visibility	6	20 - Behavior of agent was easy to see using the debugger	5
Viscosity	4	34 - Working sets helped participant find location of a problem	4
Diffuseness	3	18 - Participant was thankful for code automatically created by Herbal	3
Role expressiveness	3	33 - Participant entered quality design rationale	3
Role expressiveness	2	13 - Participant viewed rationale to learn more about model	2
Role expressiveness	1	30 - Commented that components are self-explanatory	1
Total	**150**		

Table **7-17**: Coded negative observations for all participants and tasks (N = 24).

Dimension	Total Events	Event Description	# Participants Exper. Event
Progressive evaluation	13	10 - The participant became confused about what specific component they were working on, or what step they were doing	9
Error prone	13	11 - Participant had problems distinguishing between types of design rationale	12
Error prone	9	21 - Poor design in working set dialog lead to errors	6
Error prone	8	3 -The search feature of the new project dialog lead to errors	8
Error prone	4	28 - Instructions mislead participant to create action instead of type	4
Error prone	4	29 - Participant confused by conditions with no restrictions	4
Error prone	4	31 - Participant selected wrong problem space in design pattern wizard	4
Viscosity	4	35 - Participant had problems editing an operator	4
Role expressiveness	3	5 - Participant had trouble understanding component/subcomponent	3
Viscosity	3	9 - Participant had problems editing an action	3
Error prone	3	12 - Participant confused Eclipse export with Herbal library export	3
Premature commitment	3	19 - Participant was confused by order required to run debugger	3
Role expressiveness	3	26 - Participant demonstrated poor understanding of the model	3
Error prone	3	37 - Participant entered literal value in local variable edit box	3
Diffuseness	3	17 - Participant found design rationale to be verbose and/or redundant	3
Visibility	2	1 – A portion of the Herbal GUI editor was hidden	2
Error prone	2	25 - Trouble locating the Herbal GUI Editor	2
Closeness of mapping	2	27 - Participant preferred a term not used by high-level language	2
Closeness of mapping	1	23 - Participant appeared confused by notation used to represent agent	1
Diffuseness	1	34 - Participant used copy and paste when entering design rationale	1
Role expressiveness	1	32 - Participant viewed rationale but did not find it helpful	1
Role expressiveness	1	36 - Participant misunderstood interface between model/environment	1
Error prone	1	38 - Participant confused by wire screen when there is nothing to wire	1
Total	**91**		

Table **7-18** lists the total number of events observed for subjects performing the library creation task. The table groups these events by their associated dimension and sorts them by total number of events. In addition, the table shows the total number of participants experiencing positive and negative events and the final score for each dimension.

Table **7-18**: Number of occurrences by cognitive dimension for all participants performing the library creation task (N = 8).

Dimension	**Total Events**	**# Participants Experiencing Positive Events**	**# Participants Experiencing Negative Events**	**Final Score**
Error prone	33	8	8	0
Viscosity	33	8	4	4
Progressive evaluation	25	8	4	4
Premature commitment	18	6	0	6
Role expressiveness	8	2	4	-2
Diffuseness	2	0	2	-2
Visibility	1	0	1	-1
Closeness of Mapping	1	0	1	-1
Total	**121**			

Participants experienced more negative events than positive events for four dimensions: Role Expressiveness, Diffuseness, Visibility, and Closeness of Mapping. Role Expressiveness and Diffuseness exceeded the 20% threshold. As a result, Role Expressiveness and Diffuseness are Dimensions of Concern.

Table **7-19** and Table **7-20** provide a more detailed look at the dimensions listed in Table **7-18**.

Table **7-19**: Coded positive observations for participants performing the library creation task (N = 8).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Viscosity	29	4 - Easy for participant to make a change to a component	8
Progressive evaluation	19	7 - It was easy for participants to check the status of the model, and for any errors, by looking at what has been done so far	8
Error prone	9	2 - The consistency in the Herbal interface helped reduce errors	8
Premature commitment	9	6 - Lack of required order made it easy to fix mistakes	6
Premature commitment	9	8 - Participant changed order of steps in task without problems	6
Role expressiveness	1	33- Participant entered quality design rationale	1
Role expressiveness	1	30 - Commented that components are self-explanatory	1
Total	**77**		

Table **7-20**: Coded negative observations for participants performing the library creation task (N = 8).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Error prone	6	11 - Participant had problems distinguishing between types of design rationale	6
Progressive evaluation	6	10 - The participant became confused about what specific component they were working on, or what step they were doing	4
Error prone	5	3 - The search feature of the new project dialog lead to errors	5
Error prone	4	28 - Instructions mislead participant to create action instead of type	4
Error prone	3	29 - Participant confused by conditions with no restrictions	3
Role expressiveness	3	5 - Participant had trouble understanding component/subcomponent	3
Viscosity	3	9 - Participant had problems editing an action	3
Role expressiveness	3	26 - Participant demonstrated poor understanding of the model	3
Error prone	3	37 - Participant entered literal value in local variable edit box	3
Error prone	2	12 - Participant confused Eclipse export with Herbal library export	2
Viscosity	1	35 - Participant had problems editing an operator	1
Diffuseness	1	17 - Participant found design rationale to be verbose and/or redundant	1
Diffuseness	1	24 - Participant used copy and paste when entering design rationale	1
Visibility	1	1 - A portion of the Herbal GUI editor was hidden	1
Closeness of mapping	1	27 - Participant preferred a term not used by high-level language	1
Error prone	1	38 - Participant confused by wire screen when there is nothing to wire	1
Total	**44**		

Table **7-21** lists the total number of events observed for subjects performing the model creation task. The table groups these events by their associated dimension and sorts them by total number of events. In addition, the table shows the total number of

participants experiencing positive and negative events and the final score for each dimension.

Table **7-21**: Number of occurrences by cognitive dimension for participants performing the model creation task (N = 8).

Dimension	Total Events	# Participants Experiencing Positive Events	# Participants Experiencing Negative Events	Final Score
Error prone	25	8	7	1
Progressive evaluation	20	7	5	2
Diffuseness	11	7	2	5
Role expressiveness	10	6	1	5
Premature commitment	6	4	0	4
Viscosity	2	2	0	2
Closeness of Mapping	1	0	1	-1
Visibility	0	0	0	0
Total	**75**			

Participants experienced more negative events than positive events for one dimension: Closeness of Mapping. None of the dimensions exceeded the 20% threshold and therefore no dimensions are Dimensions of Concern.

Table **7-22** and Table **7-23** provide a more detailed look at the dimensions listed in Table **7-21**.

Table **7-22**: Coded positive observations for participants performing the model creation task (N = 8).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Progressive evaluation	13	7 - It was easy for participants to check the status of the model, and for any errors, by looking at what has been done so far	7
Diffuseness	6	14 - The design pattern wizard allowed participant to create several components using a brief terminology	6
Error prone	6	15 - The design pattern wizard prevented errors creating components	6
Role expressiveness	5	16 - Participant demonstrated strong understanding of the model	5
Error prone	3	2 - The consistency in the Herbal interface helped reduce errors	3
Premature commitment	3	8 - Participant changed order of steps in task without problems	3
Diffuseness	3	18 - Participant was thankful for code automatically created by Herbal	3
Premature commitment	3	6 - Lack of required order made it easy to fix mistakes	2
Viscosity	2	4 - Easy for participant to make a change to a component	2
Role expressiveness	2	33 - Participant entered quality design rationale	2
Role expressiveness	2	13 - Participant viewed rationale to learn more about model	2
Total	**48**		

Table **7-23**: Coded negative observations for participants performing the model creation task (N = 8).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Error prone	7	11- Participant had problems distinguishing between types of design rationale	6
Progressive evaluation	7	10 -The participant became confused about what specific component they were working on, or what step they were doing	5
Error prone	4	31 - Participant selected wrong problem space in design pattern wizard	4
Error prone	3	3 - The search feature of the new project dialog lead to errors	3
Diffuseness	2	17 - Participant found design rationale to be verbose and/or redundant	2
Error prone	1	12 - Participant confused Eclipse export with Herbal library export	1
Closeness of mapping	1	27 - Participant preferred a term not used by high-level language	1
Error prone	1	25 - Trouble locating the Herbal GUI Editor	1
Role expressiveness	1	32 - Participant viewed rationale but did not find it helpful	1
Total	**27**		

Table **7-24** lists the total number of events observed for subjects performing the model maintenance task. The table groups these events by their associated dimension and sorts them by total number of events. In addition, the table shows the total number of participants experiencing positive and negative events and the final score for each dimension.

Table **7-24**: Number of occurrences by cognitive dimension for all participants performing the model maintenance task (N = 8).

Dimension	Total Events	# Participants Experiencing Positive Events	# Participants Experiencing Negative Events	Final Score
Viscosity	12	6	3	-3
Error prone	11	0	6	-6
Progressive evaluation	8	6	0	6
Visibility	7	5	1	4
Role expressiveness	3	2	1	1
Premature commitment	3	0	3	-3
Closeness of Mapping	1	0	1	-1
Diffuseness	0	0	0	0
Total	**45**			

Participants experienced more negative events than positive events for four dimensions: Viscosity, Error Proneness, Premature Commitment, and Closeness of Mapping. Viscosity, Error Proneness, and Premature Commitment exceeded the 20% threshold. As a result, these three dimensions are Dimensions of Concern.

Table **7-25** and Table **7-26** provide a more detailed look at the dimensions listed in Table **7-24**.

Table **7-25**: Coded positive observations for participants performing the model maintenance task (N = 8).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Progressive evaluation	8	7 - It was easy for participants to check the status of the model, and for any errors, by looking at what has been done so far	6
Visibility	6	20 - Behavior of agent was easy to see using the debugger	5
Viscosity	5	4 - Easy for participant to make a change to a component	4
Viscosity	4	34 - Working sets helped participant find location of a problem	4
Role expressiveness	2	16 - Participant demonstrated strong understanding of the model	2
Total	**25**		

Table **7-26**: Coded negative observations for participants performing the model maintenance task (N = 8).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Error prone	9	21 - Participant had problems distinguishing between types of design rationale	6
Viscosity	3	35 - Participant had problems editing an operator	3
Premature commitment	3	19 - Participant was confused by order required to run debugger	3
Error prone	1	25 - Trouble locating the Herbal GUI Editor	1
Error prone	1	29 - Participant confused by conditions with no restrictions	1
Visibility	1	1 - A portion of the Herbal GUI editor was hidden	1
Role expressiveness	1	36 - Participant misunderstood interface between model/environment	1
Closeness of Mapping	1	23 - Participant appeared confused by notation used to represent agent	1
Total	**20**		

Table **7-27** lists the total number of events observed for subjects majoring in PSYC. The table groups these events by their associated dimension and sorts them by total number of events. In addition, the table shows the total number of participants experiencing positive and negative events and the final score for each dimension.

Table **7-27**: Number of occurrences by cognitive dimension across all tasks for participants majoring in PSYC (N = 12).

Dimension	Total Events	# Participants Experiencing Positive Events	# Participants Experiencing Negative Events	Final Score
Error prone	32	8	10	-2
Progressive evaluation	27	12	4	8
Viscosity	25	10	4	6
Premature commitment	14	6	0	6
Role expressiveness	12	3	5	-2
Diffuseness	6	4	1	3
Visibility	4	3	1	2
Closeness of Mapping	1	0	1	-1
Total	**121**			

Participants experienced more negative events than positive events for three dimensions: Error Proneness, Role Expressiveness, and Closeness of Mapping. However, none of the dimensions exceeded the 20% threshold. As a result, no dimensions are Dimensions of Concern.

Table **7-28** and Table **7-29** provide a more detailed look at the dimensions listed in Table **7-27**.

Table **7-28**: Number of positive occurrences by cognitive dimension across all tasks for participants majoring in PSYC (N = 12).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Progressive evaluation	22	7 - It was easy for participants to check the status of the model, and for any errors, by looking at what has been done so far	12
Viscosity	18	4 - Easy for participant to make a change to a component	7
Premature commitment	9	6 - Lack of required order made it to fix mistakes	5
Error prone	6	2 - The consistency in the Herbal interface helped reduce errors	6
Premature commitment	5	8 - Participant changed order of steps in task without problems	5
Error prone	3	15 - The design pattern wizard prevented errors creating components	3
Diffuseness	3	14 - The design pattern wizard allowed participant to create several components using a brief terminology	3
Viscosity	3	34 - Working sets helped participant find location of a problem	3
Role expressiveness	3	16 - Participant demonstrated strong understanding of the model	3
Visibility	3	20 - Behavior of agent was easy to see using the debugger	3
Role expressiveness	2	33 - Participant entered quality design rationale	2
Diffuseness	2	18 - Participant was thankful for code automatically created by Herbal	2
Total	**79**		

Table **7-29**: Number of negative occurrences by cognitive dimension across all tasks for participants majoring in PSYC (N = 12).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Error prone	6	11 - Participant had problems distinguishing between types of design rationale	6
Progressive evaluation	5	10 - The participant became confused about what specific component they were working on, or what step they were doing	4
Error prone	4	21 - Poor design in the working set dialog lead to errors	3
Error prone	3	3 - The search feature of the new project dialog lead to errors	3
Error prone	3	31 - Participant selected wrong problem space in design pattern wizard	3
Error prone	3	29 - Participant confused by conditions with no restrictions	3
Viscosity	3	35 - Participant had problems editing an operator	3
Role expressiveness	3	26 - Participant demonstrated poor understanding of the model	3
Role expressiveness	2	5 - Participant had trouble understanding component/subcomponent	2
Error prone	2	28 - Instructions mislead participant to create action instead of type	2
Error prone	2	37 - Participant entered literal value in local variable edit box	2
Closeness of Mapping	1	27 - Participant preferred a term not used by high-level language	1
Viscosity	1	9 - Participant had problems editing an action	1
Visibility	1	1 - A portion of the Herbal GUI editor was hidden	1
Diffuseness	1	17 - Participant found design rationale to be verbose and/or redundant	1
Role expressiveness	1	32 - Participant viewed rationale but did not find it helpful	1
Role expressiveness	1	36 - Participant misunderstood interface between model/environment	1
Total	**42**		

Table **7-30** lists the total number of events observed for subjects majoring in CS, CIS, or MIS. The table groups these events by their associated dimension and sorts them

by total number of events. In addition, the table shows the total number of participants experiencing positive and negative events and the final score for each dimension.

Table **7-30**: Number of occurrences by cognitive dimension across all tasks for participants majoring in CS, CIS, or MIS (N = 12).

Dimension	Total Events	# Participants Experiencing Positive Events	# Participants Experiencing Negative Events	Final Score
Error prone	37	8	11	-3
Progressive evaluation	26	9	5	4
Viscosity	22	6	3	3
Premature commitment	13	4	3	1
Role expressiveness	9	7	1	6
Diffuseness	7	3	3	0
Visibility	4	2	1	1
Closeness of Mapping	2	0	2	-2
Total	**120**			

Participants experienced more negative events than positive events for two dimensions: Error Proneness and Closeness of Mapping. Only Error Proneness exceeded the 20% threshold. As a result, Error Proneness is a Dimension of Concern.

Table **7-31** and Table **7-32** provide a more detailed look at the dimensions listed in Table **7-30**.

Table **7-31**: Number of positive occurrences by cognitive dimension across all tasks for participants majoring in CS, CIS, or MIS (N = 12).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Progressive evaluation	18	7 - It was easy for participants to check the status of the model, and for any errors, by looking at what has been done so far	9
Viscosity	18	4 - Easy for participant to make a change to a component	6
Premature commitment	7	8 - Participant changed order of steps in task without problems	4
Error prone	6	2 - The consistency in the Herbal interface helped reduce errors	5
Role expressiveness	4	16 - Participant demonstrated strong understanding of the model	4
Error prone	3	15 - The design pattern wizard prevented errors creating components	3
Diffuseness	3	14 - The design pattern wizard allowed participant to create several components using a brief terminology	3
Premature commitment	3	6 - Lack of required order made it to fix mistakes	3
Visibility	3	20 - Behavior of agent was easy to see using the debugger	2
Role expressiveness	2	13 -Participant viewed rationale to learn more about model	2
Diffuseness	1	18 - Participant was thankful for code automatically created by Herbal	1
Role expressiveness	1	30 - Commented that components are self-explanatory	1
Role expressiveness	1	33 - Participant entered quality design rationale	1
Viscosity	1	34 - Working sets helped participant find location of a problem	1
Total	**71**		

Table **7-32**: Number of negative occurrences by cognitive dimension across all tasks for participants majoring in CS, CIS, or MIS (N = 12).

Dimension	Total Events	Event Description	# Participants Experiencing Event
Progressive evaluation	8	10 - The participant became confused about what specific component they were working on, or what step they were doing	5
Error prone	7	11 - Participant had problems distinguishing between types of design rationale	6
Error prone	5	3 - The search feature of the new project dialog lead to errors	5
Error prone	5	21 - Poor design in the working set dialog lead to errors	3
Premature commitment	3	19 - Participant was confused by order required to run debugger	3
Error prone	3	12 - Participant confused Eclipse export with Herbal library export	3
Diffuseness	2	17 - Participant found design rationale to be verbose and/or redundant	2
Viscosity	2	9 - Participant had problems editing an action	2
Error prone	2	25 - Trouble locating the Herbal GUI Editor	2
Error prone	2	28 - Instructions mislead participant to create action instead of type	2
Error prone	1	31 - Participant selected wrong problem space in design pattern wizard	1
Error prone	1	37 - Participant entered literal value in local variable edit box	1
Error prone	1	38 - Participant confused by wire screen when there is nothing to wire	1
Error prone	1	29 - Participant confused by conditions with no restrictions	1
Diffuseness	1	24 - Participant used copy and paste when entering design rationale	1
Viscosity	1	35 - Participant had problems editing an operator	1
Visibility	1	1 - A portion of the Herbal GUI editor was hidden	1
Role expressiveness	1	5 - Participant had trouble understanding component/subcomponent	1
Closeness of Mapping	1	23 - Participant appeared confused by notation used to represent agent	1
Closeness of Mapping	1	27 - Participant preferred a term not used by high-level language	1
Total	**49**		

A chi-square test of independence was performed to check for a correlation between the number of negative and positive observed events and the participants' major (PSYC or CI/CIS/MIS). Table **7-12** summarizes the results of this analysis. Importantly, there is no statistical evidence that there is a correlation between the number and type of observed events during task completion, and the participants' major: $X^2 = 0.961$, $DF = 1$, $p = .327$ (Howell, 1987).

Table **7-33**: A 2x2 chi-square contingency table used to test for independence between observations and participant major.

Major	# Positive Observations	# Negative Observations	Total
PSYC	79	42	121
	75.31	45.69	
	0.181	0.298	
CS/CIS/MIS	71	49	120
	74.69	45.31	
	0.182	0.300	
Total	150	91	241

Expected counts are printed below observed counts and chi-square contributions are printed below expected counts. $X^2 = 0.961$, $DF = 1$, $p = .327$

Summary of Observation Results

Table **7-34** lists Dimensions of Concern based on participant observations. Dimensions of Concern are shown using six different groupings: (1) all participants; (2) participants performing the library creation task; (2) participants performing the model creation task; (3) participants performing the model maintenance task; (4) participants majoring in PSYC, (5) for participants majoring in CS, CIS, or MIS. For example,

observations of participants majoring in CS, CIS, or MIS indicated that Error Proneness was the only Dimensions of Concern.

Table **7-34**: Dimensions of Concern as measured by observations.

Dimension	All	Library Creation	Model Creation	Model Maintenance	PSYC	CS/CIS/MIS
Visibility						
Viscosity				X		
Diffuseness		X				
Hard-mental operations						
Error-proneness	X			X		X
Closeness of mapping						
Role-expressiveness		X				
Progressive evaluation						
Premature commitment				X		

Discussion

Table **7-35** lists Dimensions of Concern based on both survey results and participant observations. The table indicates concerns based on survey results with the letter S, and concerns based on observations with the letter O. The table indicates concerns recorded based on both survey results and observations with the letter B.

Table **7-35** shows Dimensions of Concern using six different groupings: (1) all participants; (2) participants performing the library creation task; (2) participants performing the model creation task; (3) participants performing the model maintenance task; (4) participants majoring in PSYC, (5) participants majoring in CS, CIS, or MIS. For example, both survey responses and observations of participants performing model maintenance indicated that Error Proneness and Premature Commitment were Dimensions of Concern.

Table **7-35**: Summary of Dimensions of Concerns based on survey results (S), participant observations (O), and both (B).

Dimension	**All**	**Library Creation**	**Model Creation**	**Model Maintenance**	**PSYC**	**CS/CIS/ MIS**
Visibility	S	S	S			S
Viscosity				O		
Diffuseness		O				
Hard-mental operations				S		
Error-proneness	O		S	B	S	O
Closeness of mapping						
Role-expressiveness		O	S		S	
Progressive evaluation						
Premature commitment	S	S	S	B	S	S

Table **7-35** presents 54 possible dimension/condition pairings (nine dimensions * six conditions). Of these 54 pairings, 64.8% (35) show agreement between the survey results and the observations. According to the surveys and observations, the Herbal

system appears to be strong with respect to Viscosity, Diffuseness, Progressive Evaluation, Closeness of Mapping, and Hard-Mental Operations.

Observations related to Viscosity show that participants found it easy to make changes to components. In addition, the working set feature made it easy for participants to change a collection of related components. Only during the model maintenance task was there an indication (by observation) of difficulty making changes to the model. During this task, several participants encountered a problem when editing the operator that was causing the vacuum cleaner to malfunction.

Observations of appreciation for the code automatically created by Herbal provided evidence of Positive support for Diffuseness. Several participants mentioned that they were very happy they did not have to generate the Soar code manually. The Design Pattern Wizard also proved to be a compact way of expressing complicated behavior. Problems with Diffuseness took place only during library creation when the participant attempted to enter design rationale. Several participants commented on the redundancy of the design rational task, relying on copy/paste to hasten design rationale entry. Perhaps with a more complicated model, entering design rationale would have been a more interesting task.

The system also met the needs of the participants with respect to Progressive Evaluation. Observations confirmed that participants could easily check the progress of the model at any point in time, regardless of task. In addition, the participants were able to browse the model for potential errors when needed.

Closeness of Mapping was another dimension that Herbal supported well. Survey results were positive with respect to the high-level language used to describe agents

written in Herbal. In addition, only three participants experienced negative events with respect to Closeness of Mapping. Two participants thought the term "mode" or "state" would be more useful than problem space, and one participant became confused by the notation used to represent the agent and its behavior.

Finally, participants indicated in the surveys that, as a whole, they did not find the tasks particularly complex. This is due in part to the fact that the tasks tested the usability of the system, and were not problem solving exercises (aside from the maintenance task). A poorly designed interface can make basic tasks complex, and in this respect, Herbal scored well. On the surveys, participants did indicate some complexity in the model maintenance task. This is not surprising because this is the one task where participants were asked to solve a problem (i.e., what was wrong with the vacuum cleaner) rather than exercise an interface.

Herbal did appear to lack good support two dimensions: Error Proneness, and Premature Commitment. Table **7-35** lists these two dimensions as Dimensions of Concern in nearly every column.

The classification of Error Proneness as a concern was due to both observations and survey results. Only the library creation tasks did not suffer from problems with Error Proneness. Recordings of the observed events give reasons for the problems participants had with Error Proneness. For example, 13 errors resulted from participants having problems distinguishing between the different types of design rational. Twenty-one errors resulted from problems with the design and layout of the Working Set, New Project, and Design Pattern Wizard dialogs. Four errors resulted from confusion caused

by the use of a condition with no restrictions. Finally, four errors took place because of a problem in the wording of one of the steps in the instructions.

Survey data in all six conditions classified Premature Commitment as a concern, yet participant observations only classified Premature Commitment as a concern during the model maintenance task. Due to a lack of useful responses to the open-ended portions of the survey, it is difficult to tell why participants felt restricted to order. The fact that the task instructions consisted of ordered steps most likely played a role.

Observations told a different story with respect to Premature Commitment. Eight participants were able to fix mistakes more easily because Herbal did not enforce order. In addition, nine participants changed the order of the steps in the task, on their own, and without problems.

A task where order did appear to present problems (during observation) was the model maintenance task. Connecting the debugger to the Vacuum Cleaner Environment requires a fixed order and three participants were observed having problems with this rigid order.

Because Survey data indicated a problem with Premature Commitment, and observations did not, further exploration of this concern is required.

Table **7-35** shows mixed Results for Role Expressiveness and Visibility. Three conditions classified Role Expressiveness as a concern, and four conditions classified Visibility as a concern. Role Expressiveness was a concern during library creation, where three participants demonstrated a poor understanding of the model in general, and three participants demonstrated trouble understanding the relationship between components and subcomponents. Surveys indicated a problem with Role Expressiveness

during model creation and for PSYC students as a whole. Visibility was a concern in survey data for all conditions except model maintenance and for PSYC students as a whole. However, observations did not indicate Visibility as a concern for any conditions. As a result, further exploration of this concern is required.

Finally, a two-sample t-test did not show a difference between the mean performance times of the two groups: $t(22) = -1.44$, $p = .163$ (two-tailed). In addition, chi-square tests of independence did not show a relationship between the participants' major and the survey results ($X^2 = 0.160$, $DF = 1$, $p = .689$), or the observed events ($X^2 = 0.961$, $DF = 1$, $p = .327$). These results are encouraging because they support the design goal that Herbal is usable by users with varying backgrounds and skill sets.

Conclusions

The Herbal system appears to be very strong in five of the nine dimensions: Viscosity, Diffuseness, Progressive Evaluation, Closeness of Mapping, and Hard-Mental Operations. This is a very positive result with respect to the overall usability of Herbal.

The data show mixed results on Role Expressiveness, and Herbal's support for Error Proneness was of concern. Finally, the observations contradicted the survey results for Visibility and Premature Commitment, opening the door for further evaluation of these two dimensions.

The strong ratings in five of the nine dimensions are very encouraging, especially the improvement in the problems with Closeness of Mapping reported during the formative study (Chapter 5). In addition, the lack of Visibility concerns during

participant observations also shows improvement over the findings during the formative study. The changes recommended by the formative study, such as changes to the terminology used by Herbal and its high-level language, and the addition of a model browser view, appear to have improved the usability of the system.

Another encouraging result came from the statistical analysis of the data based on major. Herbal's implementation of reuse, visual programming, working sets, graphical displays, and a well-designed high-level representation, appears to be helping people use the tool independently of their skill set.

One discouraging result was the apparent poor support for the Premature Commitment dimension, despite changes made because of the formative evaluations. Unfortunately, most of the data indicating poor support for Premature Commitment is from the surveys, and the lack of good open-ended responses makes it difficult to determine the reason for this result. Contradicting the survey results, observations showed that eight participants were able to fix mistakes more easily because Herbal did not enforce order. In addition, nine participants changed the order of the steps in the task on their own and without problems. Additional work is required, perhaps with a more open-ended task, to evaluate this dimension.

Finally, Error Proneness was a concern in both observations and survey results. Only the library creation task did not suffer from problems with Error Proneness. Fortunately, the observations revealed several ways to address this issue. For example, a better explanation of the difference between the three different design rationale types would have eliminated 13 errors. In addition, improvements to the design and layout of the Working Set, New Project, and Design Pattern Wizard could eliminate 21 errors.

Chapter 8

Contributions, Lessons, and Future Work

Cognitive models are useful for a number of purposes. Unfortunately, limited theory-based tool and language support for the creation of cognitive models has made it difficult for modelers to create, debug, and reuse cognitively plausible software (Pew & Mavor, 1998; Ritter et al., 2003; Salvucci & Lee, 2003; Yost, 1993). In addition, the use of multiple cognitive architectures has further complicated cognitive modeling by making it difficult to compare, reuse, and integrate models (Gluck & Pew, 2001a; Jones, Crossman, Lebiere, & Best, 2006; Jones & Wray, 2003).

This dissertation demonstrated the benefits of applying software engineering principles to cognitive modeling development, with the creation of a high-level language and development environment, and with evaluations of this language and environment, in use, by students and cognitive modelers. The upcoming sections detail the specific contributions and lessons generated by the work presented in this dissertation, along with opportunities that arise from this work.

Contributions towards Better Modeling Languages

The high-level language presented here is a significant contribution to better modeling languages because it allows modelers to program using a high-level representation that is compiled for multiple architectures (presently Soar and Jess). This

allows modelers to create a model using a well-known theory of cognition (PSCM) and translate this model into different architectures unifying different theories. For example, a modeler might choose Soar because of its learning mechanism and its emphasis on psychological plausibility. Alternatively, an agent developer might choose Jess because of its cross-platform strengths and ease at which it integrates with existing Java applications. With Herbal, modelers can share components across both models despite the fact that they execute in different architectures.

Currently, only one other high-level cognitive modeling language supports multiple architectures: HLSR, which was reviewed in Chapter 2 (Jones, Crossman, Lebiere, & Best, 2006). However, the high-Level language and environment presented in this dissertation has two distinct advantages over HLSR.

First, Herbal has explicit support for a well-established theory of cognition. The PSCM forms the basis of the Herbal high-level language, and the components of the PSCM are explicit in the code. This helps close the conceptual gap between a theory commonly used by cognitive modelers and the representation used to express behavior. The high-level language used by HLSR does not explicitly support a theory of cognition. Instead, HSLR hides this theory within programmable microtheories. Although the HLSR language does simplify cognitive modeling, there is still a conceptual gap between the theories commonly used to describe behavior and the HLSR representation.

Second, the Herbal high-level language uses XML to represent models. XML is a free and open standard specification that provides the foundation for nearly all modern markup languages and open-document formats. Models written in the Herbal high-level language can immediately benefit from a large set of open-source and commercial tools

(e.g., editors, graphics engines, mathematical notations, and databases). Developers can edit their models using existing XML editors, easily translate their code into a variety of formats (e.g., HTML, SVG, PDF, or even productions for additional cognitive architectures), and create their own Herbal tools using one of the many programming languages that support XML.

Another contribution of this work is confirmation of the usefulness of the PSCM as a high-level behavior representation language and hierarchical organization tool. This contribution arises from the evaluations completed in support of this dissertation. In general, results from the formative evaluation described in Chapter 5 illustrate that participants appreciated the PSCM for its ability to organize rules into higher-level structures, structures often obscured by rule-based languages. In addition, results from the summative evaluation presented in Chapter 7 show that Closeness of Mapping and Diffuseness were dimensions of strength for the system presented here. This is a further indication that the PSCM is a good choice for a behavior representation language because it closely matches the way that the modeler describes behavior naturally, and provides a brief way to produce results or express behavior.

Table **8-1** summarizes this dissertation's contributions towards better modeling languages.

Table **8-1**: Contributions towards better modeling languages.

Contributions
1. A high-level modeling language based on the PSCM and represented in XML
2. The ability to translate models written in this language into two popular, yet different, architectures
3. Empirical validation of the high-level language and the choice of the PSCM for this representation

Contributions towards Better Maintenance-Oriented Modeling Environments

Programmers spend considerable time performing software maintenance. As mentioned in Chapter 3, the total cost of software maintenance is often at least 40% of the total cost of developing it the software (Brooks, 1995) and U.S. programmers spend over 70% of their time testing and debugging (Tassey, 2002). As a result, strong support for usability and maintenance is an important part of simplifying cognitive modeling. This dissertation makes four contributions towards better maintenance-oriented modeling environments: support for multiple levels of editing source code; support for better code navigation; a strong emphasis on usability; and a novel method of analyzing the results of a cognitive dimension evaluation.

The Herbal development environment is currently the only cognitive modeling environment that has support for simultaneously creating models at many different levels of abstraction (Figure **4-13**). Support for programming at these levels makes Herbal useful for both end-user programmers, who can create models visually, and expert

programmers, who may prefer to build models using multiple levels. This also may provide better support for users as they transition from novice to expert user.

This dissertation contributes the only maintenance-oriented cognitive modeling environment that supports the creation, maintenance, and persistence, of working sets for the development of cognitive models. Studies have shown both the need, and the benefit of the use of working sets for software maintenance (Ko, Aung, & Myers, 2005; Ko, Myers, Coblenz, & Aung, 2006). The Herbal working set feature includes information about the intent of the model's components during a search. This gives the modeler access to additional information "scent" when building working sets. Modelers can also save these working sets and share them with other modelers or recall them for future use.

In addition, Herbal is the only cognitive modeling environment evaluated by three different studies, and is the only research effort that used cognitive dimensions as the basis for its evaluation. A semester long formative usability study has informed Herbal's design. In addition, this project has subjected Herbal to two different summative evaluations, one evaluating the usefulness of the environment and the other evaluating the usability of the environment. The summative usability evaluation showed that Herbal was strong in the Viscosity, Hard-mental Operations, Closeness of Mapping, and Progressive Evaluation cognitive dimensions. This same study has also identified methods for the improvement of some of these dimensions.

Finally, the method this project used to analyze data generated by the summative evaluation is novel. Specifically, when negative responses or negative observations about a dimension exceeded a threshold, this method classified that dimension as a Dimension of Concern. The concept of a Dimension of Concern, as described above, is

new and can be useful to other researchers that are implementing a similar evaluation based on cognitive dimensions. By adjusting the threshold based on the importance of the task, and the needs of the users, other researchers should be able to reuse this method of analysis, and perhaps the term "Dimension of Concern" will become common vocabulary for dimension-based evaluations.

Table **8-2** summarizes this dissertation's contributions towards better maintenance-oriented environments.

Table **8-2**: Contributions towards better maintenance-oriented environments.

Contributions
1. A cognitive modeling environment that has support for simultaneously creating models at three different levels of abstraction
2. A cognitive modeling environment with support for better code navigation using working sets that leverage model dependencies and the developer's intent
3. A cognitive modeling environment with a strong emphasis on usability
4. A novel and useful method of analyzing the results of a cognitive-dimension-based evaluation

Contributions towards Better Model Reuse

The Herbal high-level language and environment has also contributed towards better model reuse. Using Krueger's dimensions of reuse, Herbal's language and environment facilitate the reuse of behavior across models in several novel ways.

Herbal's high-level PSCM-based language, including the addition of conditions and actions, is an example of Krueger's first dimension of reuse: abstraction. The

extension of the PSCM to include conditions and actions as standard objects has added a level of granularity that allows for better reuse within and across PSCM models. Operators that utilize similar conditions and actions no longer need to duplicate the whole operator, previously the smallest unit in the PSCM. This type of reuse is difficult to achieve because of the dependencies between actions and conditions (e.g., modelers often design actions to work with specific conditions). Herbal's ability to "wire" conditions to actions is a new contribution that makes this possible.

Herbal is also the only high-level cognitive modeling language that is library centric. All of Herbal's model components must reside within a library, and modelers can identify each component using a unique namespace that simplifies the reuse of these components. The required use of libraries, and the ability to assemble components from these libraries to create models, is an example of abstraction and integration (Krueger's first and fourth dimensions).

Another contribution made by this dissertation is the support for the creation and reuse of even higher-level behavior patterns created on top of the PSCM. For example, structured programming patterns can be instantiated, thus creating looping structures within traditionally unstructured rule-based environments. Herbal builds these looping constructs out of standard PSCM components that are reusable, provides a graphical wizard so simplify their creation, and translates these constructs into productions that run in two widely used architectures. The ability for modelers to tailor reusable behavior design patterns to their specific needs is an example of specialization (Krueger's third dimension of reuse), and this is a new contribution to reuse in cognitive modeling.

Herbal's working set functionality also contributes to better reuse by supporting Krueger's second dimension: selection. Modelers can browse libraries looking for components related to their needs by using the working set search feature. Because this search feature includes design rationale and component dependencies, modelers can discover reusable components quickly and efficiently. No other cognitive modeling environment supports this type of component selection.

An evaluation using an early version of Herbal has empirically confirmed the benefits of reuse in Herbal. In a study done by Morgan, Haynes, Ritter, and Cohen (2005), a Soar model consisting of 29 productions was created using Herbal. In this study, the authors showed a reduction in the time it took to create productions as the library of reusable components (e.g., conditions and actions) expanded. This reduction in time was primarily due to the increased reuse provided by Herbal over standard Soar. In addition, the overall average time per production was less than that reported in a similar study of graduate students programming in Soar (Yost, 1993).

Table **8-3** summarizes this dissertation's contributions towards better model reuse.

Table **8-3**: Contributions towards better model reuse.

Contributions
1. The extension of the PSCM to include conditions and actions as standard objects has added another level of granularity that allows for better reuse within and across PSCM models
2. Library-centric modeling language
3. Support for the creation and reuse of even higher-level behavior patterns created on top of the PSCM
4. The ability to browse libraries looking for components related to their needs using the search feature of working sets

Contributions towards Education of Modelers

This dissertation has also made several contributions towards education. The baseball environment created to evaluate the usefulness of the Herbal modeling environment extends the baseball examples presented in the Soar tutorial (Laird & Congdon, 2005). Students using the tutorial to learn Soar now have a graphical Soar environment to work with that mimics examples given in the tutorial.

Another outcome of this work is the Vacuum Cleaner Environment (Cohen, 2005). The Vacuum Cleaner Environment also extends a popular learning example, in this case the vacuum cleaner world presented in a widely used artificial intelligence textbook (Russell & Norvig, 2003). Students learning AI using this textbook can now execute textbook examples in a dynamic graphical environment using two very different modeling languages.

Herbal is used extensively as a teaching tool in classes at Lock Haven University and Penn State University. To date, professors have exposed 89 undergraduates and 9 graduates to modeling using Herbal. In addition, in the fall of 2008 another graduate class at Penn State will use Herbal.

Students in a Cognitive and Brain Sciences course at Tufts University have used Herbal as a tool for learning Soar and gaining a better understanding of high-level behavior representation languages. Audrey Girouard and Noah W. Smith took a well-written Soar model and decompiled it into an equally functional Herbal high-level representation. This helped students understand the tradeoffs between high-level and low-level representations, and obtain a better understanding of how Soar productions represent the PSCM.

During observations of students using Herbal, I have also discovered some unexpected instructional benefits. Initially designed to reduce the need to program at a low-level, the Herbal high-level language and GUI Editor also appear to be valuable for teaching low-level rule-based programming. Working with the Herbal GUI editor and the Herbal high-level language editor side-by-side, students have used the tool to learn the Herbal language by making changes graphically and then viewing the generated Herbal representation. In addition, by editing the Herbal representation directly, and then viewing the generated low-level productions, students have been able to learn native Jess and Soar programming.

Table **8-4** summarizes this dissertation's contributions towards education.

Table **8-4**: Contributions towards education of modelers.

Contributions
1. Students using the Soar tutorial now have a graphical environment to work with that mimics the baseball examples given in the tutorial
2. Eighty-nine undergraduates and nine graduates have been exposed to modeling using Herbal, and more will follow in the fall 2008 semester
3. Herbal has been used at Tufts to learn Soar and to gain a better understanding of high-level behavior representation languages
4. The Herbal GUI Editor is also useful for teaching low-level rule-based programming, by editing the Herbal representation directly, and then viewing the generated low-level productions

External Users

Maik Friedrich (2008), a masters student in Germany, used Herbal for the modeling portion of his dissertation. In his dissertation, Friedrich (2008) re-implemented a model (Ritter & Bibby, 2008) that was created to solve a diagrammatic reasoning task (this older model was written for Soar 6 and was no longer supported).

The first phase of Friedrich's work involved creating a library of components using Herbal. This phase took six weeks to complete. The second phase involved creating four different models based on this library. The most complex model contained 80 productions. Each model used a different strategy to solve the diagrammatic reasoning task. This phase took two weeks to complete. Overall, the modeling effort took eight weeks to complete, which was significantly less that the six months required to build the original model (based on comments in the original source code). Part of this

improvement can be attributed to the ability to reuse the design work in the original model, but some of this improvement is very likely attributable to the use of Herbal.

In addition, four external sites have downloaded or are using Herbal to conduct research (NYU, the Netherlands Government, the Laboratory for Telecommunications Sciences at UMD, and Pace University).

Lessons and Future Work

In addition to the positive empirical results, the implementation of Herbal produced and reinforced some areas of future research. Five categories classify this future work: high-level languages, maintenance-oriented environments, reuse, usability and evaluation, and graphical agent environments. The next few subsections describe this future work in detail.

Future Work in High-level Modeling Languages

The most pressing area of work is support for more cognitive architectures. One possibility is ACT-R. ACT-R is a very popular cognitive architecture that supports a theory quite different from Soar and Jess. In addition, the hybrid nature of ACT-R allows modelers to more easily explore variability in behavior. By adding support for more cognitive architectures like ACT-R, modelers can further realize a main goal of this research: the reusability of behavior across architectures.

The development of the Herbal high-level modeling language continually reinforced the trade-off between the power of programming close to the architecture and the simplicity of programming at a higher-level. On the one hand, basing the Herbal high-level language on the PSCM provided some much needed structure and organization to a traditionally rule-based programming environment. However, the absence of underlying architectural support for the PSCM in Jess created a need to limit or simulate portions of the PSCM. Understanding this trade-off, and looking for ways of minimizing it, is an excellent task for future research.

Future Work in Maintenance-Oriented Modeling Environments

Based on recent research, the Herbal maintenance-oriented development environment has been equipped with a working set feature that simplifies the creation of a navigable and related set of components.

As discussed in Chapter 3, researchers have been working on additional techniques that can enable environments to generate these working sets automatically. For example, a tool might linguistically analyze a description of the modeler's current task, perhaps as described in a bug report, to build working sets automatically. In addition, the use of code navigation history by project team members, and recent adaptations of information processing theory (e.g., PFIS), can also provide automatic working sets generation (Cubranic, Murphy, Singer, & Booth, 2005; DeLine, Czerwinski, & Robertson, 2005). In the future, researchers could implement these techniques into Herbal's working set feature and confirm its effectiveness.

Future Work in Model Reuse

One of the exciting contributions made by this dissertation is the ability to build complex behavior on top of the Herbal high-level PSCM components using the Behavior Design Pattern Wizard. Modelers can name these behaviors and reuse them in models that run on different architectures.

Currently, Herbal supports procedural looping behaviors. However, researchers could add several other behavior patterns to the wizard. For example, support for the BDI framework would make it easier for BDI researchers to create and possibly reuse behavior between Herbal and JACK. In addition, support for the abstract constructs (e.g., activation tables) in HLSR would simplify behavior creation and reuse between Herbal and HLSR.

Future Work in Usability and Evaluation

Based on the summative usability study discussed in Chapter 6, Herbal could better support Error Proneness, and Premature Commitment. Observations of participants have given clues about why Error Proneness was a problem. For example, 13 errors resulted from participants having problems distinguishing between the different types of design rational. Twenty-one errors resulted from problems with the design and layout of the Working Set, New Project, and Design Pattern Wizard dialogs. There are opportunities for improving Herbal's support for Error Proneness by addressing these issues.

The concern with Premature Commitment is a bit more complicated. Because survey data supported the concern for Premature Commitment, but observations contradicted this concern, researchers should explore this further. One task where order did present problems was the model maintenance task. Connecting the debugger to the Vacuum Cleaner Environment requires a fixed order, and three participants were observed having problems with this rigid order. Future improvements to the integration between the debugger and the graphical environments would certainly help with Premature Commitment.

Another area for potential research would be to address the problems encountered with the open-ended questions in the cognitive dimensions survey. Students seemed to be in too much of a hurry to provide meaningful responses to the qualitative questions. This made it difficult to understand the reasons behind some of their responses. Future work exists for discovering how to improve a generalized cognitive dimensions survey to get useful responses to the open-ended questions.

Future Work in Graphical Agent Environments

This research has lead to the creation of two graphical agent environments: the baseball environment and the Vacuum Cleaner Environment. Both of these environments present opportunities for future work. Because the Herbal development environment automatically creates both Soar and Jess models, the opportunity exists for comparisons of a single Herbal high-level pitcher model running in two very different architectures. These types of comparisons have been shown to be important (Gluck & Pew, 2001a;

Gluck & Pew, 2001b; Morgan, Ritter, Cohen, Stevenson, & Schenck, 2005; Sun, Councill, Fan, Ritter, & Yen, 2004), and Herbal makes this easier to do.

In addition, future work could make improvements to the pitcher model by enhancing the reflective process so that benefit from negative experiences takes place without requiring previous positive experiences. In the absence of positive learned events, negative reflection should still lead to a decrease in the probability of repeating the action.

There are also opportunities to explore other parts of the baseball task. For example, researchers can expand the environment and its models to include other batting strategies, other batter sequences, batting tournaments, and learning batters.

Future work also exists in the Vacuum Cleaner Environment project. In the current version, vacuum cleaners never have mishaps. They always clean when told to, and they always move when commanded to. Of course, real world environments are much less predictable. The addition of random errors committed by the vacuum cleaner would allow for more interesting models.

Conclusion

This dissertation demonstrated the benefits of applying software engineering principles to cognitive model development, with the creation of a high-level language and development environment, and with evaluations of this language and environment, in use, by students and cognitive modelers.

The high-level language was designed to close the conceptual gap (Petre & Blackwell, 1997) between the mental model used by cognitive modelers and the low-level representations used to model behavior. In addition, this language uses Krueger's four dimensions to support reuse (Krueger, 1992). Finally, the compiler for this language supports multiple architectures, so modelers can compare, reuse, and integrate behavior across architectures.

Motivated by research confirming the importance of the maintenance phase of software development (Brooks, 1995; Ko, Myers, Coblenz, & Aung, 2006; Tassey, 2002), this dissertation leverages design patterns (Gamma, Helm, Johnson, & Vlissides, 1995), and working sets (Ko, Myers, Coblenz, & Aung, 2006), to bring modelers the type of maintenance support that is has been shown to benefit traditional software development.

This research concluded with two different evaluations to help validate the hypothesis that the theory embedded in this system simplifies the modeling task. In addition, these evaluations demonstrated that the system is usable. While these evaluations were positive, they have also suggested future work. The result of this dissertation is research that has made significant contributions to the modeling community and has set an important precedent of considering software engineering and usability to progress the field of cognitive modeling.

References

Agarwal, R., De, P., Sinha, A. P., & Tanniru, M. (2000). On the usability of OO representations. *Communications of the ACM, 43*(10), 83-89.

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Lawrence Erlbaum Associates.

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

Auerbach, J. S., Bacon, D. F., Goldberg, A. P., Goldszmidt, G. S., Kennedy, M. T., Lowry, A. R., Russell, J. R., Silverman, W., Strom, R. E., Yellin, D. M., & Yemini, S. A. (1991). *High-level language support for programming distributed systems* (No. RC 16441): IBM.

Bass, E. J., Baxter, G. D., & Ritter, F. E. (1995). Creating models to control simulations: A generic approach. *AI and Simulation Behaviour Quarterly, 93*, 18-25.

Beck, L. L., & Perkins, T. E. (1983). A survey of software engineering practice: Tools, method, and results. *IEEE Transactions on Software Engineering, 9*(5), 541-561.

Bigus, J. P., & Bigus, J. (2001). *Constructing Intelligent Agents Using Java: Professional Developer's Guide* (2nd ed.): Wiley.

Blackwell, A. F., & Green, T. (2003). Notational systems: The cognitive dimensions of notations frameworks. In J. M. Carroll (Ed.), *HCI Models, Theories, and Frameworks* (pp. 103-133). San Francisco, CA: Morgan Kaufmann.

Blackwell, A. F., & Green, T. R. G. (2000). A cognitive dimensions questionnaire optimised for users. *In Proceedings of the 12th Annual Meeting of the Psychology of Programming Interest Group,* 137-152.

Blank, D. S., Kumar, D., Meeden, L., & Yanco, H. (2006). The Pyro toolkit for AI and robotics. *AI Magazine, 27*.

Boehm, B. W. (1987). Improving software productivity. *IEEE Computer, 20*(9), 43-57.

Boehm, B. W. (1988a). A spiral model of software development and enhancement. *Computer, 21*(5), 61-72.

Boehm, B. W. (1988b). Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering, 15*(10), 1462-1477.

Boshernitsan, M. (2003). Program manipulation via interactive transformations. *In Proceedings of the OOPSLA,* 392-393. Anaheim CA.

Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer, 20,* 10-19.

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering* (20th Anniversary ed.): Addison Wesley.

Buchanan, B. G., Sutherland, G. L., & Feigenbaum, E. A. (1969). Heuristic DENDRAL: A program for generating explanatory hypothesis in organic chemistry. In B. Meltzer, D. Michie & M. Swann (Eds.), *Machine Intelligence 4* (pp. 209-254). Edinburgh Scotland: Edinburgh University Press.

Campbell, C. E., Eisenberg, A., & Melton, J. (2003). XML schema. *ACM SIGMOD Record, 32*(2), 96-101.

Chi, E., Pirolli, P., Chen, K., & Pitkow, J. (2001). Using information scent to model user information needs and actions on the web. *In Proceedings of the CHI 2001*: ACM Press.

Chidlovskii, B. (2003). A structural adviser for the XML document authoring. *In Proceedings of the ACM Symposium on Document Engineering,* 203-211. New York, NY: ACM.

Clancey, W. J. (1981). *The epistemology of a rule-based expert system: A framework for explanation* (No. STAN-CS-91-896). Stanford, CA: Stanford University.

Coad, P., & Yourdon, E. (1991). *Object-Oriented Analysis* (2nd ed.). Englewood Cliffs, NJ: Yourdon Press/Prentice Hall.

Coblenz, M. J., Ko, A. J., & Myers, B. A. (2006). JASPER: An Eclipse plug-in to facilitate software maintenance tasks. *In Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange,* 65-69. Portland, Oregon: ACM Press.

Cohen, M. A. (2005). Teaching agent programming using custom environments and Jess. *The Newsletter of the Society for the Study of Artificial Intelligence and the Simulation of Behavior, 120,* 4.

Cohen, M. A., Ritter, F. E., & Haynes, S. R. (2005). Herbal: A high-level language and development environment for developing cognitive models in Soar. *In Proceedings of the 14th Behavior Representation in Modeling and Simulation,* 133-140. Orlando, FL.: U. of Central Florida: 05-BRIMS-043.

Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., Durbin, J., Gossweiler, R., Kogi, S., Long, C., Mallory, B., Miale, S., Monkaitis, K., Patten, J., Pierce, J., Schochet, J., Staak, D., Stearns, B., Stoakley, R., Sturgill, C., Viega, J., White, J., Williams, G., & Pausch, R. (2000). Alice: Lessons learned from building a 3D system for novices. *In Proceedings of the CHI,* New York, NY: ACM.

Cooper, R. P., & Fox, J. (1998). COGENT: A visual design environment for cognitive modeling. *Behavior Research Methods, Instruments, & Computers, 30*(4), 553-564.

Cooper, R. P., & Yule, P. (2007). An introduction to the COGENT Modeling Environment. *In Proceedings of the International Conference on Cognitive Modeling,* Ann Arbor, MI.

Cox, M. T., & Ram, A. (1999). Introspective multistrategy learning: On the construction of learning strategies. *Artificial Intelligence, 112*, 1-55.

Cubranic, D., Murphy, G. C., Singer, J., & Booth, K. S. (2005). Hipikat: a project memory for software development. *IEEE Transactions on Software Engineering, 31*(6), 446-465.

Daly, E. B. (1977). Management of Software Development. *IEEE Transactions on Software Engineering, 3*(3), 229-242.

Dann, W. P., Cooper, S., & Pausch, R. (2008). *Learning to program with alice* (2nd ed.). Upper Sadde River, NJ: Pearson Education Inc.

Das, A., & Stuerzlinger, W. (2007). A cognitive simulation model for novice text entry on cell phone keypads. *In Proceedings of the 14th European Conference on Cognitive Ergonomics,* 141-147. New York, NY: ACM.

DeLine, R., Czerwinski, M., & Robertson, G. (2005). Easing program comprehension by sharing navigation data. *In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing,* 241-248. IEEE.

Duda, R., Gaschnig, J., & Hart, P. (1979). Model design in the PROSEPCTOR consultant system for mineral exploration. In D. Michie (Ed.), *Expert systems in the microelectronic age* (pp. 165-190). Edinburgh, Scotland: Edinburgh University Press.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol analysis: verbal reports as data*. Cambridge, MA: MIT Press.

Firby, R. J. (1989). *Adaptive Execution in Complex Dynamic Worlds.* Unpublished Doctoral Dissertation, Yale University.

Friedman-Hill, E. (2003). *Jess in action: Rule-based systems in Java*. Greenwich, CT: Manning Publications Company.

Friedrich, M. B. (2008). *Implementing diagrammatic reasoning strategies in a high level language: Extending and testing the existing model results by gathering additional data and creating additional strategies.*, University of Bamberg, Germany.

Friedrich, M. B., Cohen, A. M., & Ritter, F. E. (2007). *A gentle introduction to XML within Herbal*. State College, PA: Applied Cognitive Science Laboratory, College of Information Sciences and Technology, Penn State University.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Giarratano, J., & Riley, G. (1998). *Expert systems: principles and programming*. Boston: PWS Publishing.

Gluck, K. A., & Pew, R. W. (2001a). Lessons learned and future directions for the AMBR model comparison project. *In Proceedings of the 10th Computer Generated Forces and Behavioral Representation Conference,* 113-121. Orlando, FL.: Division of Continuing Education, University of Central Florida.

Gluck, K. A., & Pew, R. W. (2001b). Overview of the agent-based modeling and behavior representation (AMBR) model comparison project. *In Proceedings of the 10th Computer Generated Forces and Behavioral Representation Conference,* Orlando, FL.: Division of Continuing Education, University of Central Florida.

Goldman, A. I. (1993). The psychology of folk psychology. *Behavioral and Brain Sciences, 16*, 15-28.

Haugeland, J. (1987). Semantic engines: An introduction to mind design. In J. Haugeland (Ed.), *Mind Design* (pp. 368). Cambridge, MA: The MIT Press.

Haynes, S. R., Cohen, A. M., & Ritter, F. E. (2008). Design patterns for explaining intelligent agents. Manuscript submitted for publication (copy on file with author).

Heinath, M., Dzaack, J., Wiesner, A., & Urbas, L. (2007). Simplfying the development and the analysis of cognitive models. *In Proceedings of the EuroCogSci07,* Delphi, Greece.

Hirst, T. (1999). ViSoar - Towards an Agent Development Environment for the Soar Architecture. *In Proceedings of the 4th Online Workshop on Soft Computing (WSC4)*.

Hordijk, W., & Wieringa, R. (2005). Surveying the factors that influence maintainability. *In Proceedings of the ESEC-FSE,* 385-388. New York, NY: ACM Press.

Howden, N., Ronnquist, R., Hodgson, A., & Lucas, A. (2001). JACK intelligent agents - summary of an agent infrastructure. *In Proceedings of the 5th International Conference on Autonomous Agents,* Montreal, CA: ACM Press.

Howell, D. C. (1987). *Statistical methods for psychology* (2nd ed.). Boston, MA: PWS Publishers.

Ivory, M. Y., & Hearst, M. A. (2001). The state of the art in automating usability evaluation of user interfaces. *Computing Surveys, 3*(4), 470-516.

Jackson, D. (2002). Scalable vector graphics (SVG): the world wide web consortium's recommendation for high quality web graphics. *In Proceedings of the International Conference on Computer Graphics and Interactive Techniques* 319-319. New York, NY: ACM.

John, B. E. (2003). Information Processing and Skilled Behavior. In J. M. Carroll (Ed.), *HCI Models, Theories, and Frameworks* (pp. 55-101). San Francisco, CA: Morgan Kaufmann.

John, B. E., Prevas, K., Salvucci, D. D., & Koedinger, K. R. (2004). Predictive human performance modeling made easy. *In Proceedings of the 2004 Conference on Computer Human Interaction,* 455-462.

John, B. E., Remington, R. W., & Steier, D. M. (1991). *An analysis of space shuttle countdown activities: Preliminaries to a computational model of the NASA test director.* (No. CMU-CS-91-138). Pittsburgh, PA: Carnegie Mellon University School of Computer Science.

Jones, R. M., Crossman, J. A. L., Lebiere, C., & Best, B. J. (2006). An abstract language for cognitive modeling. *In Proceedings of the International Conference on Cognitive Modeling,* 160-165. Mahwah, NJ: Lawrence Erlbaum.

Jones, R. M., Laird, J. E., Nielson, P. E., Coulter, K. J., Kenny, P., & Koss, F. V. (1999). Automated Intelligent Pilots for Combat Flight Simulation. *AI Magazine, 20,* 27-41.

Jones, R. M., & Wray, B. (2003). Design principles for heavy intelligent agents. *In Proceedings of the International Conference on Autonomous Agents* 1022-1023. New York, NY: ACM.

Kadoda, G., Stone, R., & Diaper, D. (1999). Desirable features of educational theorem provers: A cognitive dimensions viewpoint. In T. R. G. Green, R. Abdullah & P.

Brna (Eds.), *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group* (pp. 18-23). Leeds: Leeds University Press.

Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling alice motivates middle school girls to learn computer programming. *In Proceedings of the Proceedings of the SIGCHI conference on Human factors in computing systems* 1455-1464. New York, NY: ACM.

Kieras, D., & Meyer, D. E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human Computer Interaction, 4*, 230-275.

Knudsen, K., Quist, M., Ray, D., & Wray, B. (2007). *Soar IDE*. Paper presented at the 2007 Soar Workshop.

Ko, A. J., Aung, H. H., & Myers, B. A. (2005). Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. *In Proceedings of the ICSE,* 126-135. New York, NY: ACM Press.

Ko, A. J., & Myers, B. A. (2003). Development and evaluation of a model of programming errors. *In Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments,* 7-14. Auckland, New Zealand: IEEE Computer Society.

Ko, A. J., & Myers, B. A. (2004). Designing the Whyline: A debugging interface for asking questions about program behavior. *In Proceedings of the SIGCHI conference on Human factors in computing systems* 151-158. Vienna, Austria: ACM Press.

Ko, A. J., Myers, B. A., Coblenz, M. J., & Aung, H. H. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering, 32*(12), 971-987.

Krueger, C. W. (1992). Software reuse. *ACM Computer Surveys, 24*(2), 131-183.

Laird, J. E. (1999). Visual Soar. *In Proceedings of the Soar Workshop 19,* 99-102. University of Michigan: Soar Group.

Laird, J. E. (2001a). It knows what you're going to do: Adding anticipation to a Quakebot. *In Proceedings of the Fifth International Conference on Autonomous Agents,* 385-392. New York, NY: ACM Press.

Laird, J. E. (2001b). Using a computer game to develop advanced AI. *IEEE Computer, 34*(7), 70-75.

Laird, J. E., & Congdon, C. B. (2005). *The Soar User's Manual Version 8.6*: The Soar Group: University of Michigan.

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining mental models: A study of developer work habits. *In Proceedings of the 28th International Conference on Software Engineering,* 492-501. Shanghai, China: ACM Press.

Lawrance, J., Bellamy, R., Burnett, M., & Rector, K. (2008). Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. *In Proceedings of the CHI 2008*: ACM Press.

Lehman, J. F., Laird, J. E., & Rosenbloom, P. S. (1996). A gentle introduction to Soar: An architecture for human cognition. In D. Scarborough & S. Sternberg (Eds.), *An invitation to cognitive science* (Vol. 4). New York: MIT Press.

Lewis, B. (2003). Debugging backwards in time. *In Proceedings of the 5th Workshop on Automated and Algorithmic Debugging,* 225-235. Ghent, Belgium.

Lewis, R. L., Newell, A., & Polk, T. A. (1989). Toward a Soar theory of taking instructions for immediate reasoning tasks. *In Proceedings of the Eleventh Annual Conference of the Cognitive Science Society,* 514-521.

Maxwell, K. D., Wassenhove, L. V., & Dutta, S. (1996). Software development productivity of European space, military, and industrial applications. *IEEE Transactions on Software Engineering, 22*(10), 706-718.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I *Communications of the ACM, 3*(4), 184-195.

McIlroy, M. D. (1968). Mass produced software components. *In Proceedings of the Software Engineering; Report on a conference by the NATO Science Committee,* 138-150. NATO Scientific Affairs Division.

Minsky, M. (1990). Logical vs. Analogical or Symbolic vs. Connectionist or Neat vs. Scruffy. In P. H. Winston (Ed.), *Artificial Intelligence at MIT, Expanding Frontiers* (Vol. 1): MIT Press.

Minsky, M., & Papert, S. (1987). *Perceptrons - expanded edition: An introduction to computational geometry* (Expanded ed.): MIT Press.

Morgan, G. P., Cohen, A. M., 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,* Charlottesville, VA: University of Virginia.

Morgan, G. P., Ritter, F. E., Cohen, M. A., Stevenson, W. E., & Schenck, I. N. (2005). dTank: An environment for architectural comparisons of competitive agents. *In Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation,* 133-140. Universal City, CA.

Morrison, J. E. (2003). *A review of computer-based human behavior representations and their relation to military simulations*. Alexandria, VA: Institute for Defense Analyses.

Musicant, D. R., & Exley, A. (2004). Easy Integration of LEGO Mindstorms into Vacuum World Simulations. *In Proceedings of the ACM SIGCSE,* Norfolk, VA.

Negnevitsky, M. (2004). *Artificial intelligence: A guide to intelligent systems* (2nd ed.): Addison Wesley.

Newell, A. (1990). *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

Newell, A., & Simon, H. A. (1972). *Human Problem Solving*. Englewood Cliffs, NJ: Prentice Hall.

Newell, A., Yost, G. R., Laird, J. E., Rosenbloom, P., & Altmann, E. (1991). Formulating the problem space computational model. In R. F. Rashid (Ed.), *Carnegie Mellon Computer Science: A 25-Year commemorative* (pp. 255-293). Reading, MA: ACM-Press (Addison-Wesley).

Norling, E. (2004). Folk psychology for human modeling: Extending the BDI paradigm. *In Proceedings of the International Conference on Autonomous Agents and Multiagent Systems,* 202-209. New York.

Norling, E., & Ritter, F. E. (2001). Embodying the JACK agent architecture. *In Proceedings of the 14th Austrailian Joint Conference on Artificial Intelligence,* 368-377. Berlin: Springer.

Norling, E., & Ritter, F. E. (2004). Towards supporting psychologically plausible variability in agent-based human modeling. *In Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems,* 758-765. New York: ACM Press.

Ormerod, T. C., & Shepherd, A. (2004). Using Task Analysis for Information Requirements Specification: The Sub-Goal Template (SGT) Method. In D. Diaper & N. A. Stanton (Eds.). Mahwah, NJ: LEA.

Petre, M., & Blackwell, A. F. (1997). A glimpse of expert programmers' mental imagery. *In Proceedings of the Seventh workshop on empirical studies of programmers,* 109-123. Alexandria, VA.

Pew, R. W., & Mavor, A. S. (Eds.). (1998). *Modeling human and organizational behavior: Application to military simulations*. Washington, DC: National Academy Press.

Pew, R. W., & Mavor, A. S. (Eds.). (2008). *Modeling human and organizational behavior: Application to military simulations*. Washington, DC: National Academy Press.

Phillips, E. M., & Pugh, D. S. (2005). *How to get a Ph.D.* (4th ed.). Berkshire, England: Open University Press.

Pirolli, P., & Card, S. (1999). Information foraging. *Psychology Review, 106*(4), 643-675.

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: teaching CS0 with Alice. *ACM SIGCSE Bulletin, 39*(1), 213-217.

Quint, V., & Vatton, I. (2004). Techniques for authoring complex XML documents. *In Proceedings of the ACM Symposium on Document Engineering* 115-123. New York, NY: ACM.

Reiss, S. P. (2006). Incremental maintenance of software artifacts. *IEEE Transactions on Software Engineering, 32*(9), 692-697.

Ripley, B. D. (1993). Statistical Aspects of Neural Networks. In B.-N. O. E., Jensen J. L. & K. W. S. (Eds.), *Networks and chaos - Statistical and probabilistic aspects*. London: Chapman and Hall.

Ritter, F. E. (1992). *TBPA: A methodology and software environment for testing process models' sequential predictions with protocols.* Carnegie Mellon University, Pittsburgh, PA.

Ritter, F. E., & Bibby, P. A. (2008). Modeling how, when, and what learning happens in a diagrammatic reasoning task. *Cognitive Science*.

Ritter, F. E., Haynes, S. R., Cohen, M. A., Howes, A., John, B. E., Best, B., Lebiere, C., Jones, R. M., Lewis, R. L., St Amant, R., McBride, S. P., Urbas, L., Leuchter, S., & Vera, A. (2006). High-level behavior representation languages revisited. *In Proceedings of the Seventh International Conference on Cognitive Modeling,* 404-407. Trieste, Italy: Edizioni Goliardiche.

Ritter, F. E., Kase, S. E., Bhandarkar, D., Lewis, B., & Cohen, A. M. (2007). dTank updated: Exploring moderator-influenced behavior in a light-weight synthetic environment. *In Proceedings of the the 16th Conference on Behavior Representation in Modeling and Simulation,* 51-60. Orlando, FL: U. of Central Florida.

Ritter, F. E., & Norling, E. (2006). Including human variability in a cognitive architecture to improve team simulation. In R. Sun (Ed.), *Cognition and multi-agent interaction* (pp. 417-427). New York, NY: Cambridge University Press.

Ritter, F. E., Shadbolt, N. R., Elliman, D., Young, R., Gobet, F., & Baxter, G. D. (2003). *Techniques for modeling human and organizational behavior in synthetic environments: A supplementary review*: Wright-Patterson Air Force Base, OH: Human Systems Information Analysis Center.

Ritter, F. E., & Wallach, D. P. (1998). Models of two-person games in ACT-R and Soar. *In Proceedings of the Second European Conference on Cognitive Modeling,* 202-203. Nottingham: Nottingham University Press.

Robillard, M. P., Coelho, W., & Murphy, G. C. (2004). How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering, 30*(12), 889-903.

Rosson, M. B., & Carroll, J. M. (2002). *Usability engineering: Scenario-based development of human-computer interaction*. San Francisco, CA: Morgan Kaufmann.

Royappa, A. V. (1999). Implementing catalog clearinghouses with XML and XSL. *In Proceedings of the ACM symposium on Applied computing* 616 - 621. New York, NY: ACM.

Rumelhart, D. E., & McClelland, J. L. (1987). *Parallel Distributed Processing. Explorations in the microstructure of cognition (2 Vol. Set)*. Cambridge, MA: The MIT Press.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A modern approach* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

Salomon, D. J. (1992). Four dimensions of programming-language independence. *ACM SIGPLAN Notices, 27*(3), 35-53.

Salvucci, D. D., & Lee, F. J. (2003). Simple cognitive modeling in a complex cognitive architecture. *In Proceedings of the SIGCHI conference on human factors in computing systems,* 265-272. Ft. Lauderdale, FL: ACM Press.

Scriven, M. (1967). The methodology of evaluation. In R. Tyler, R. Gagne & M. Scriven (Eds.), *Perspectives of curriculum evaluation* (pp. 39-83). Chicago: Rand McNally.

Shavor, S., D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., & McCarthy, P. (2003). *The Java developer's guide to Eclipse*: Addison-Wesley Professional.

Shepherd, G. M., & Koch, C. (1990). Introduction to synaptic circuits. In G. M. Shepherd (Ed.), *The synaptic organisation of the brain* (pp. 3-31). New York: Oxford University Press.

Shortliffe, E. H. (1976). *MYCIN: Computer-based medical consultations*. New York: Elsevier 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.

St. Amant, R., & Ritter, F. E. (2004). Model-based evaluation of cell phone menu interaction. *In Proceedings of the International Conference on Human Computer Interaction,* 343-350. Vienna, Austria.

Sun, R. (Ed.). (2006). *Cognition and multi-agent interaction*. Cambridge University Press: New York.

Sun, S., Councill, I. G., Fan, X., Ritter, F. E., & Yen, J. (2004). Comparing teamwork modeling in an empirical approach. *In Proceedings of the Sixth International Conference on Cognitive Modeling,* 388-389. Mahwah, NJ.: Erlbaum.

Tassey, G. (2002). *The economic impacts of inadequate infrastructure for software testing* (No. 7007.011): National Institute of Standards and Technology.

Vokac, M. (2004). Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering, 30*(12), 904-917.

Vullo, R. P., & Bogaard, D. S. (2004). Visualization with dynamically generated SVG. *In Proceedings of the 5th Conference on Information Technology Education,* 271-271. New York, NY: ACM.

W3C. (2003). Scalable Vector Graphics 1.1 Specification from www.w3.org/TR/SVG

W3C. (2004a). The Extensible Markup Language. from http://www.w3.org/XML/

W3C. (2004b). The Extensible Stylesheet Language Family. from www.w3.org/Style/XSL/

W3C. (2004c). XML Schema Part 0: Primer Second Edition. from http://www.w3.org/TR/xmlschema-0/

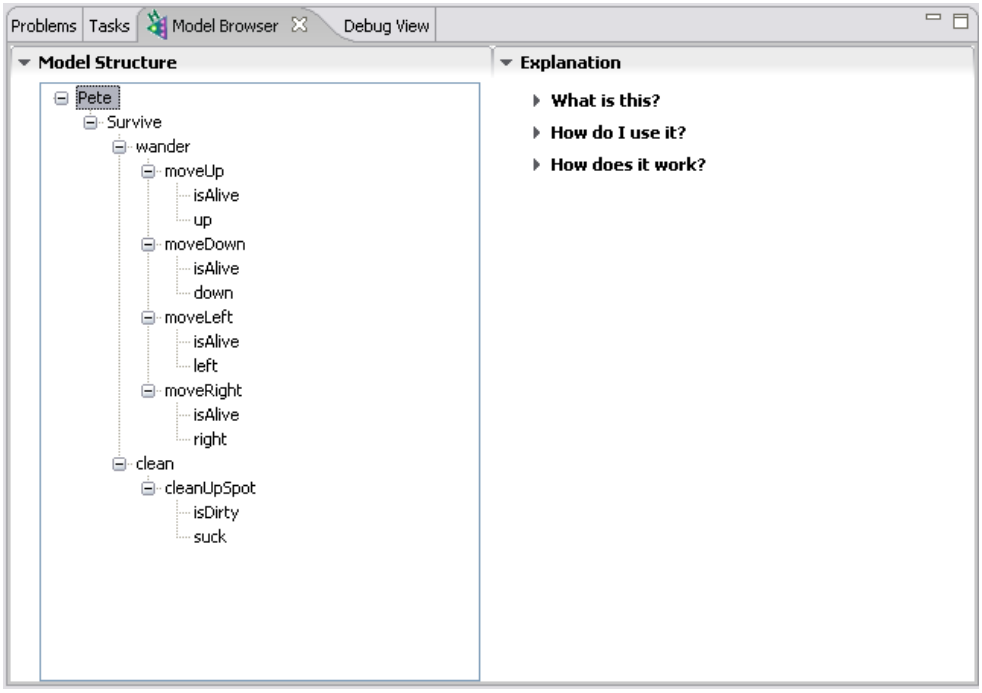
Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM, 25*(7), 446-452.

Yost, G. R. (1993). Acquiring knowledge in Soar. *IEEE Expert: Intelligent systems and their applications, 8*(3), 26-34.

Appendix A

A Comparison of Representations

Graphical PSCM Representation



XML PSCM Representation

```
<?xml version='1.0'?>
<models version='1.0'
  xmlns='http://acs.ist.psu.edu/herbal'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://acs.ist.psu.edu/herbal
    ../schema/models.xsd'>

  <model name='Pete'>
    <problemspaceref problemspace='Survive'>
      <problemspaceref problemspace='DesignPat.problemspaces.wander'>
      </problemspaceref>
      <problemspaceref problemspace='DesignPat.problemspaces.clean'>
      </problemspaceref>
      <impasse subspace='DesignPat.problemspaces.wander'>
        <conditionref condition='vacuum.conditions.isClean'/>
      </impasse>
      <impasse subspace='DesignPat.problemspaces.clean'>
        <conditionref condition='vacuum.conditions.isDirty'/>
      </impasse>
    </problemspaceref>
  </model>

</models>


<?xml version='1.0'?>
<problemspaces version='1.0'
  xmlns='http://acs.ist.psu.edu/herbal'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://acs.ist.psu.edu/herbal
    ../schema/problemspaces.xsd'>

  <problemspace name='Survive'>
    <init>
    </init>
  </problemspace>

</problemspaces>

<?xml version='1.0'?>
<problemspaces version='1.0'
  xmlns='http://acs.ist.psu.edu/herbal'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://acs.ist.psu.edu/herbal
    ../schema/problemspaces.xsd'>

  <problemspace name='wander'>
    <init>
    </init>
    <operatorref actionscope='top' conditionscope='top'
            operator='vacuum.operators.moveUp' elaboration='false'/>
    <operatorref actionscope='top' conditionscope='top'
            operator='vacuum.operators.moveDown' elaboration='false'/>
    <operatorref actionscope='top' conditionscope='top'
            operator='vacuum.operators.moveLeft' elaboration='false'/>
    <operatorref actionscope='top' conditionscope='top'
```

```
57              operator='vacuum.operators.moveRight'
    elaboration='false'/>
58    </problemspace>
59
60    <problemspace name='clean'>
61      <init>
62      </init>
63      <operatorref actionscope='top' conditionscope='top'
64              operator='vacuum.operators.cleanUpSpot'
65              elaboration='false'/>
66    </problemspace>
67
68  </problemspaces>
69
70  <?xml version='1.0'?>
71  <actions version='1.0'
72    xmlns='http://acs.ist.psu.edu/herbal'
73    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
74    xsi:schemaLocation='http://acs.ist.psu.edu/herbal
75      ../schema/actions.xsd'>
76
77    <action name='up'>
78      <add order='0' type='action'>
79        <set field='move'><value>up</value></set>
80      </add>
81    </action>
82
83    <action name='right'>
84      <add order='0' type='action'>
85        <set field='move'><value>right</value></set>
86      </add>
87    </action>
88
89    <action name='suck'>
90      <add order='0' type='action'>
91        <set field='move'><value>suck</value></set>
92      </add>
93    </action>
94
95    <action name='left'>
96      <add order='0' type='action'>
97        <set field='move'><value>left</value></set>
98      </add>
99    </action>
100
101   <action name='down'>
102     <add order='0' type='action'>
103       <set field='move'><value>down</value></set>
104     </add>
105   </action>
106
107 </actions>
108
109 <?xml version='1.0'?>
110 <conditions version='1.0'
111   xmlns='http://acs.ist.psu.edu/herbal'
```

```
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://acs.ist.psu.edu/herbal
      ../schema/conditions.xsd'>

    <condition name='isClean'>
      <match type='spot'>
        <restrict field='status'><eq>clean</eq></restrict>
      </match>
    </condition>

    <condition name='isAlive'>
      <match type='position'>
        <restrict field='y'></restrict>
        <restrict field='x'></restrict>
      </match>
    </condition>

    <condition name='isDirty'>
      <match type='spot'>
        <restrict field='status'><eq>dirty</eq></restrict>
      </match>
    </condition>

  </conditions>

  <?xml version='1.0'?>
  <operators version='1.0'
    xmlns='http://acs.ist.psu.edu/herbal'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://acs.ist.psu.edu/herbal
      ../schema/operators.xsd'>

    <operator name='moveDown'>
      <if>
        <conditionref condition='isAlive'/>
      </if>
      <then>
        <actionref action='down'>
        </actionref>
      </then>
    </operator>

    <operator name='moveRight'>
      <if>
        <conditionref condition='isAlive'/>
      </if>
      <then>
        <actionref action='right'>
        </actionref>
      </then>
    </operator>

    <operator name='moveLeft'>
      <if>
        <conditionref condition='isAlive'/>
      </if>
```

```
      <then>
        <actionref action='left'>
        </actionref>
      </then>
    </operator>

    <operator name='cleanUpSpot'>
      <if>
        <conditionref condition='isDirty'/>
      </if>
      <then>
        <actionref action='suck'>
        </actionref>
      </then>
    </operator>

    <operator name='moveUp'>
      <if>
        <conditionref condition='isAlive'/>
      </if>
      <then>
        <actionref action='up'>
        </actionref>
      </then>
    </operator>

  </operators>

  <?xml version='1.0'?>
  <types version='1.0'
    xmlns='http://acs.ist.psu.edu/herbal'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://acs.ist.psu.edu/herbal
      ../schema/types.xsd'>

    <type name='position' isIO='true'>
      <field name='x' type='number'/>
      <field name='y' type='number'/>
    </type>

    <type name='radar' isIO='true'>
      <field name='dir' type='string'/>
      <field name='reading' type='string'/>
    </type>

    <type name='action' isIO='true'>
      <field name='move' type='string'/>
    </type>

    <type name='spot' isIO='true'>
      <field name='status' type='string'/>
    </type>

  </types>

```

Jess Representation

```
1   ; -------------------------------------------------
2   ; Created automagically by Herbal 2.0.10
3   ; Date Compiled: Thu Jul 31 12:34:17 EDT 2008
4   ; Compiled By: mcohen
5   ; -------------------------------------------------
6
7   (bind ?rnd (new java.util.Random))
8   (deffunction randChoice ($?choices)
9     (bind ?i (call ?rnd nextInt (length$ ?choices)))
10    (return (nth$ (+ ?i 1) ?choices))
11  )
12  ;-------------------------------------------------------------------------
    ----------
13  ;define Survive
14  ;-------------------------------------------------------------------------
    ----------
15
16  ;define the module...
17  (defmodule Survive)
18
19  ;define the types...
20  (deftemplate vacuum.types.position
21    (slot x (default 0))
22    (slot y (default 0))
23  )
24  (deftemplate vacuum.types.radar
25    (slot dir (default ""))
26    (slot reading (default ""))
27  )
28  (deftemplate vacuum.types.action
29    (slot move (default ""))
30  )
31  (deftemplate vacuum.types.spot
32    (slot status (default ""))
33  )
34
35  ;define the impasses...
36  (defrule Pete.models.impasse1
37    ?f <- (Survive::initial-fact)
38    (Survive::vacuum.types.spot (status ?status1&:(eq* ?status1
    "clean")))
39    =>
40    (retract ?f)
41    (assert (Survive::initial-fact))
42    (initProblemspace-wander)
43  )
44  (defrule Pete.models.impasse2
45    ?f <- (Survive::initial-fact)
46    (Survive::vacuum.types.spot (status ?status2&:(eq* ?status2
    "dirty")))
47    =>
48    (retract ?f)
49    (assert (Survive::initial-fact))
50    (initProblemspace-clean)
51  )
52
```

```
53  ;define the inialize problemspace function...
54  (deffunction initProblemspace-Survive ()
55    (set-current-module Survive)
56    (assert (Survive::initial-fact))
57    (focus Survive)
58  )
59  ;-------------------------------------------------------------------------
    ----------
60  ;end Survive
61  ;-------------------------------------------------------------------------
    ----------
62  ;-------------------------------------------------------------------------
    ----------
63  ;define wander
64  ;-------------------------------------------------------------------------
    ----------
65
66  ;define the module...
67  (defmodule wander)
68
69  ;define the types...
70  (deftemplate vacuum.types.position
71    (slot x (default 0))
72    (slot y (default 0))
73  )
74  (defrule removevacuum.types.positions (declare (salience -100)) ?f
    <- (vacuum.types.position) => (retract ?f))
75  (deftemplate vacuum.types.radar
76    (slot dir (default ""))
77    (slot reading (default ""))
78  )
79  (defrule removevacuum.types.radars (declare (salience -100)) ?f <-
    (vacuum.types.radar) => (retract ?f))
80  (deftemplate vacuum.types.action
81    (slot move (default ""))
82  )
83  (defrule removevacuum.types.actions (declare (salience -100)) ?f <-
    (vacuum.types.action) => (retract ?f))
84  (deftemplate vacuum.types.spot
85    (slot status (default ""))
86  )
87  (defrule removevacuum.types.spots (declare (salience -100)) ?f <-
    (vacuum.types.spot) => (retract ?f))
88
89  ;define the rules...
90  (defrule vacuum.operators.moveUp
91    (Survive::vacuum.types.spot (status ?status3&:(eq* ?status3
    "clean")))
92    (Survive::vacuum.types.position (y ?y4)(x ?x5))
93    =>
94    (assert (Survive::vacuum.types.action (move "up") ))
95  )
96  (defrule vacuum.operators.moveDown
97    (Survive::vacuum.types.spot (status ?status6&:(eq* ?status6
    "clean")))
98    (Survive::vacuum.types.position (y ?y7)(x ?x8))
```

```
 99     =>
100     (assert (Survive::vacuum.types.action (move "down") ))
101   )
102   (defrule vacuum.operators.moveLeft
103     (Survive::vacuum.types.spot (status ?status9&:(eq* ?status9
      "clean")))
104     (Survive::vacuum.types.position (y ?y10)(x ?x11))
105     =>
106     (assert (Survive::vacuum.types.action (move "left") ))
107   )
108   (defrule vacuum.operators.moveRight
109     (Survive::vacuum.types.spot (status ?status12&:(eq* ?status12
      "clean")))
110     (Survive::vacuum.types.position (y ?y13)(x ?x14))
111     =>
112     (assert (Survive::vacuum.types.action (move "right") ))
113   )
114
115   ;define the exit problem space rule...
116   (defrule wander-exit (declare (salience -200)) ?f <-
      (wander::initial-fact) => (retract ?f) (focus Survive))
117
118   ;define the inialize problemspace function...
119   (deffunction initProblemspace-wander ()
120     (set-current-module wander)
121     (assert (wander::initial-fact))
122     (focus wander)
123   )
124   ;-------------------------------------------------------------------
      ----------
125   ;end wander
126   ;-------------------------------------------------------------------
      ----------
127   ;-------------------------------------------------------------------
      ----------
128   ;define clean
129   ;-------------------------------------------------------------------
      ----------
130
131   ;define the module...
132   (defmodule clean)
133
134   ;define the types...
135   (deftemplate vacuum.types.position
136     (slot x (default 0))
137     (slot y (default 0))
138   )
139   (defrule removevacuum.types.positions (declare (salience -100)) ?f
      <- (vacuum.types.position) => (retract ?f))
140   (deftemplate vacuum.types.radar
141     (slot dir (default ""))
142     (slot reading (default ""))
143   )
144   (defrule removevacuum.types.radars (declare (salience -100)) ?f <-
      (vacuum.types.radar) => (retract ?f))
145   (deftemplate vacuum.types.action
```

```
146     (slot move (default ""))
147   )
148   (defrule removevacuum.types.actions (declare (salience -100)) ?f <-
      (vacuum.types.action) => (retract ?f))
149   (deftemplate vacuum.types.spot
150     (slot status (default ""))
151   )
152   (defrule removevacuum.types.spots (declare (salience -100)) ?f <-
      (vacuum.types.spot) => (retract ?f))
153
154   ;define the rules...
155   (defrule vacuum.operators.cleanUpSpot
156     (Survive::vacuum.types.spot (status ?status15&:(eq* ?status15
      "dirty")))
157     (Survive::vacuum.types.spot (status ?status16&:(eq* ?status16
      "dirty")))
158     =>
159     (assert (Survive::vacuum.types.action (move "suck") ))
160   )
161
162   ;define the exit problem space rule...
163   (defrule clean-exit (declare (salience -200)) ?f <-
      (clean::initial-fact) => (retract ?f) (focus Survive))
164
165   ;define the inialize problemspace function...
166   (deffunction initProblemspace-clean ()
167     (set-current-module clean)
168     (assert (clean::initial-fact))
169     (focus clean)
170   )
171   ;-------------------------------------------------------------------
      ----------
172   ;end clean
173   ;-------------------------------------------------------------------
      ----------
174   ;-------------------------------------------------------------------
      ----------
175   ;initialize the top problem space so we are ready to run
176   ;-------------------------------------------------------------------
      ----------
177   (initProblemspace-Survive)
178
179
180
181
```

Soar Representation

```
# ----------------------------------------------------
# Created automagically by Herbal 2.0.10
# Date Compiled: Thu Jul 31 12:34:17 EDT 2008
# Compiled By: mcohen
# ----------------------------------------------------


#-------------------------------------------------------------------------
----------
#elaborations to retract completed output...
#-------------------------------------------------------------------------
----------
sp {apply*global*remove-vacuum-types-position
  (state <s> ^operator <o> ^io.output-link <out>)
  (<out> ^|vacuum.types.position| <x>)
  (<x> ^status complete)
  -->
  (<out> ^|vacuum.types.position| <x> -)
}

sp {apply*global*remove-vacuum-types-radar
  (state <s> ^operator <o> ^io.output-link <out>)
  (<out> ^|vacuum.types.radar| <x>)
  (<x> ^status complete)
  -->
  (<out> ^|vacuum.types.radar| <x> -)
}

sp {apply*global*remove-vacuum-types-action
  (state <s> ^operator <o> ^io.output-link <out>)
  (<out> ^|vacuum.types.action| <x>)
  (<x> ^status complete)
  -->
  (<out> ^|vacuum.types.action| <x> -)
}

sp {apply*global*remove-vacuum-types-spot
  (state <s> ^operator <o> ^io.output-link <out>)
  (<out> ^|vacuum.types.spot| <x>)
  (<x> ^status complete)
  -->
  (<out> ^|vacuum.types.spot| <x> -)
}

#-------------------------------------------------------------------------
----------
#define Pete-problemspaces-Survive
#-------------------------------------------------------------------------
----------

sp {propose*initialize-Pete-problemspaces-Survive
  (state <local> ^type state -^name)
  (<local> ^superstate nil)
  -->
  (<local> ^operator <o> +)
  (<o> ^name initialize-Pete-problemspaces-Survive)
```

```
53    (<o> ^top <local>)
54    (<o> ^parent <local>)
55  }
56
57  sp {apply*initialize-Pete-problemspaces-Survive
58    (state <local> ^operator <o>)
59    (<o> ^name initialize-Pete-problemspaces-Survive ^top <top>
    ^parent <parent>)
60    (<top> ^io <i1>)
61    (<i1> ^output-link <i2>)
62    -->
63    (<local> ^name Pete-problemspaces-Survive)
64    (<local> ^top <local>)
65    (<local> ^parent <local>)
66    (write |<hdb>Pete.models.Pete</hdb>| (crlf))
67    (write |<hdb>Pete.problemspaces.Survive</hdb>| (crlf))
68  }
69
70  #define the impasses...
71  sp
    {propose*Pete-problemspaces-Survive*impasse*DesignPat-problemspaces-wa
    nderps
72    (state <local> ^top <top> ^name Pete-problemspaces-Survive)
73    (<top> ^io <i1>)
74    (<i1> ^input-link <i2>)
75    (<i2> ^|vacuum.types.spot| <vacuum-types-spot1>)
76    (<vacuum-types-spot1> ^status <status1> |clean| )
77    -->
78    (<local> ^operator <o> + =)
79    (<o> ^name impasse*DesignPat-problemspaces-wanderps)
80    (write |<hdb>vacuum.conditions.isClean</hdb>| (crlf))
81  }
82
83  sp
    {propose*Pete-problemspaces-Survive*impasse*DesignPat-problemspaces-cl
    eanps
84    (state <local> ^top <top> ^name Pete-problemspaces-Survive)
85    (<top> ^io <i1>)
86    (<i1> ^input-link <i2>)
87    (<i2> ^|vacuum.types.spot| <vacuum-types-spot2>)
88    (<vacuum-types-spot2> ^status <status2> |dirty| )
89    -->
90    (<local> ^operator <o> + =)
91    (<o> ^name impasse*DesignPat-problemspaces-cleanps)
92    (write |<hdb>vacuum.conditions.isDirty</hdb>| (crlf))
93  }
94
95  #-------------------------------------------------------------------
    ----------
96  #define DesignPat-problemspaces-wander
97  #-------------------------------------------------------------------
    ----------
98
99  sp {propose*initialize-DesignPat-problemspaces-wander
100   (state <local> ^type state -^name)
101   (<local> ^impasse no-change ^attribute operator)
```

```
    (<local> ^superstate <parent>)
    (<parent> ^top <top>)
    (<parent> ^operator <imp>)
    (<imp> ^name impasse*DesignPat-problemspaces-wanderps)
    -->
    (<local> ^operator <o> +)
    (<o> ^name initialize-DesignPat-problemspaces-wander)
    (<o> ^top <top>)
    (<o> ^parent <parent>)
}

sp {apply*initialize-DesignPat-problemspaces-wander
    (state <local> ^operator <o>)
    (<o> ^name initialize-DesignPat-problemspaces-wander ^top <top>
^parent <parent>)
    (<top> ^io <i1>)
    (<i1> ^output-link <i2>)
    -->
    (<local> ^name DesignPat-problemspaces-wander)
    (<local> ^top <top>)
    (<local> ^parent <parent>)
    (write |<hdb>Pete.models.Pete</hdb>| (crlf))
    (write |<hdb>DesignPat.problemspaces.wander</hdb>| (crlf))
}

#define the rules...
sp {propose*DesignPat-problemspaces-wander*vacuum-operators-moveUp
    (state <local> ^top <top> ^parent <parent> ^name
DesignPat-problemspaces-wander)
    (<top> ^io <i1>)
    (<i1> ^input-link <i2>)
    (<i2> ^|vacuum.types.position| <vacuum-types-position3>)
    (<vacuum-types-position3> ^y <y3>)
    (<vacuum-types-position3> ^x <x4>)
    -->
    (write |<hdb>PROPOSAL.vacuum.operators.moveUp</hdb>| (crlf))
    (write |<hdb>vacuum.conditions.isAlive</hdb>| (crlf))
    (<local> ^operator <o> + =)
    (<o> ^name vacuum-operators-moveUp)
}
sp {apply*DesignPat-problemspaces-wander*moveUp
    (state <local> ^top <top> ^name DesignPat-problemspaces-wander
^operator <o>)
    (<o> ^name vacuum-operators-moveUp)
    (<top> ^io <i1>)
    (<i1> ^output-link <i2>)
    -->
    (write |<hdb>vacuum.operators.moveUp</hdb>| (crlf))
    (write |<hdb>vacuum.actions.up</hdb>| (crlf))
    (<i2> ^|vacuum.types.action| <vacuum-types-action5>)
    (<vacuum-types-action5> ^move |up| )
}
sp {propose*DesignPat-problemspaces-wander*vacuum-operators-moveDown
    (state <local> ^top <top> ^parent <parent> ^name
DesignPat-problemspaces-wander)
    (<top> ^io <i1>)
```

```
154     (<i1> ^input-link <i2>)
155     (<i2> ^|vacuum.types.position| <vacuum-types-position6>)
156     (<vacuum-types-position6> ^y <y6>)
157     (<vacuum-types-position6> ^x <x7>)
158     -->
159     (write |<hdb>PROPOSAL.vacuum.operators.moveDown</hdb>| (crlf))
160     (write |<hdb>vacuum.conditions.isAlive</hdb>| (crlf))
161     (<local> ^operator <o> + =)
162     (<o> ^name vacuum-operators-moveDown)
163   }
164   sp {apply*DesignPat-problemspaces-wander*moveDown
165     (state <local> ^top <top> ^name DesignPat-problemspaces-wander
      ^operator <o>)
166     (<o> ^name vacuum-operators-moveDown)
167     (<top> ^io <i1>)
168     (<i1> ^output-link <i2>)
169     -->
170     (write |<hdb>vacuum.operators.moveDown</hdb>| (crlf))
171     (write |<hdb>vacuum.actions.down</hdb>| (crlf))
172     (<i2> ^|vacuum.types.action| <vacuum-types-action8>)
173     (<vacuum-types-action8> ^move |down| )
174   }
175   sp {propose*DesignPat-problemspaces-wander*vacuum-operators-moveLeft
176     (state <local> ^top <top> ^parent <parent> ^name
      DesignPat-problemspaces-wander)
177     (<top> ^io <i1>)
178     (<i1> ^input-link <i2>)
179     (<i2> ^|vacuum.types.position| <vacuum-types-position9>)
180     (<vacuum-types-position9> ^y <y9>)
181     (<vacuum-types-position9> ^x <x10>)
182     -->
183     (write |<hdb>PROPOSAL.vacuum.operators.moveLeft</hdb>| (crlf))
184     (write |<hdb>vacuum.conditions.isAlive</hdb>| (crlf))
185     (<local> ^operator <o> + =)
186     (<o> ^name vacuum-operators-moveLeft)
187   }
188   sp {apply*DesignPat-problemspaces-wander*moveLeft
189     (state <local> ^top <top> ^name DesignPat-problemspaces-wander
      ^operator <o>)
190     (<o> ^name vacuum-operators-moveLeft)
191     (<top> ^io <i1>)
192     (<i1> ^output-link <i2>)
193     -->
194     (write |<hdb>vacuum.operators.moveLeft</hdb>| (crlf))
195     (write |<hdb>vacuum.actions.left</hdb>| (crlf))
196     (<i2> ^|vacuum.types.action| <vacuum-types-action11>)
197     (<vacuum-types-action11> ^move |left| )
198   }
199   sp {propose*DesignPat-problemspaces-wander*vacuum-operators-moveRight
200     (state <local> ^top <top> ^parent <parent> ^name
      DesignPat-problemspaces-wander)
201     (<top> ^io <i1>)
202     (<i1> ^input-link <i2>)
203     (<i2> ^|vacuum.types.position| <vacuum-types-position12>)
204     (<vacuum-types-position12> ^y <y12>)
205     (<vacuum-types-position12> ^x <x13>)
```

```
206     -->
207     (write |<hdb>PROPOSAL.vacuum.operators.moveRight</hdb>| (crlf))
208     (write |<hdb>vacuum.conditions.isAlive</hdb>| (crlf))
209     (<local> ^operator <o> + =)
210     (<o> ^name vacuum-operators-moveRight)
211   }
212   sp {apply*DesignPat-problemspaces-wander*moveRight
213     (state <local> ^top <top> ^name DesignPat-problemspaces-wander
      ^operator <o>)
214     (<o> ^name vacuum-operators-moveRight)
215     (<top> ^io <i1>)
216     (<i1> ^output-link <i2>)
217     -->
218     (write |<hdb>vacuum.operators.moveRight</hdb>| (crlf))
219     (write |<hdb>vacuum.actions.right</hdb>| (crlf))
220     (<i2> ^|vacuum.types.action| <vacuum-types-action14>)
221     (<vacuum-types-action14> ^move |right| )
222   }
223   #-----------------------------------------------------------------------
      ----------
224   #define DesignPat-problemspaces-clean
225   #-----------------------------------------------------------------------
      ----------
226
227   sp {propose*initialize-DesignPat-problemspaces-clean
228     (state <local> ^type state -^name)
229     (<local> ^impasse no-change ^attribute operator)
230     (<local> ^superstate <parent>)
231     (<parent> ^top <top>)
232     (<parent> ^operator <imp>)
233     (<imp> ^name impasse*DesignPat-problemspaces-cleanps)
234     -->
235     (<local> ^operator <o> +)
236     (<o> ^name initialize-DesignPat-problemspaces-clean)
237     (<o> ^top <top>)
238     (<o> ^parent <parent>)
239   }
240
241   sp {apply*initialize-DesignPat-problemspaces-clean
242     (state <local> ^operator <o>)
243     (<o> ^name initialize-DesignPat-problemspaces-clean ^top <top>
      ^parent <parent>)
244     (<top> ^io <i1>)
245     (<i1> ^output-link <i2>)
246     -->
247     (<local> ^name DesignPat-problemspaces-clean)
248     (<local> ^top <top>)
249     (<local> ^parent <parent>)
250     (write |<hdb>Pete.models.Pete</hdb>| (crlf))
251     (write |<hdb>DesignPat.problemspaces.clean</hdb>| (crlf))
252   }
253
254   #define the rules...
255   sp {propose*DesignPat-problemspaces-clean*vacuum-operators-cleanUpSpot
256     (state <local> ^top <top> ^parent <parent> ^name
      DesignPat-problemspaces-clean)
```

```
257     (<top> ^io <i1>)
258     (<i1> ^input-link <i2>)
259     (<i2> ^|vacuum.types.spot| <vacuum-types-spot15>)
260     (<vacuum-types-spot15> ^status <status15> |dirty| )
261     -->
262     (write |<hdb>PROPOSAL.vacuum.operators.cleanUpSpot</hdb>| (crlf))
263     (write |<hdb>vacuum.conditions.isDirty</hdb>| (crlf))
264     (<local> ^operator <o> + =)
265     (<o> ^name vacuum-operators-cleanUpSpot)
266   }
267   sp {apply*DesignPat-problemspaces-clean*cleanUpSpot
268     (state <local> ^top <top> ^name DesignPat-problemspaces-clean
      ^operator <o>)
269     (<o> ^name vacuum-operators-cleanUpSpot)
270     (<top> ^io <i1>)
271     (<i1> ^output-link <i2>)
272     -->
273     (write |<hdb>vacuum.operators.cleanUpSpot</hdb>| (crlf))
274     (write |<hdb>vacuum.actions.suck</hdb>| (crlf))
275     (<i2> ^|vacuum.types.action| <vacuum-types-action16>)
276     (<vacuum-types-action16> ^move |suck| )
277   }
278
279
280
281
```

Appendix B

Summative Evaluation Materials

User Background Survey

Participant ID: _____ **Date: _____**

Major: Computer Science / Computer Information Science / BS Psychology

Age: _____ **Gender:** M / F

Number of undergraduate credits completed: _____

On average, how many hours a day do you use a computer: _____

1. **Please describe what tasks you perform on your computer during a typical week:**

2. **Please list any courses that you have taken that covered computer programming:**

3. **Please list any courses that you have taken that covered cognitive modeling or intelligent agent development:**

4. **How stressful do you expect the upcoming task to be?**

1	*2*	*3*	*4*	*5*
Not at all		*Moderately*		*Very*

5. **How well do you think you will be able to cope with the upcoming task?**

1	*2*	*3*	*4*	*5*
Not at all		*Moderately*		*Very*

6. **How demanding do you expect the upcoming task to be?**

1	*2*	*3*	*4*	*5*
Not at all		*Moderately*		*Very*

Questions or comments about this study should be directed to Mark Cohen (mcohen@lhup.edu)

User Background Survey

Participant ID: _____ **Date: _____**

7. How well do you think you will perform in the upcoming task?

1	*2*	*3*	*4*	*5*
Not at all		*Moderately*		*Very*

8. Is there anything else you would like to tell us about your interests or background that you think we should know? If yes, briefly describe:

General Task Instructions for the Herbal Study

In the next 40 minutes or so, you will be performing a task using the Herbal Development Environment. You will be responsible for one of the following three tasks: to create a reusable library for creating vacuum cleaner agents; to create a specific vacuum cleaner agent using a library; or to debug and fix an existing vacuum cleaner agent.

It is important that you take your time during this task. The task instructions you will be using are, at times, intentionally vague in order to measure how intuitive the interface is. Please "think out loud" (narrate your actions) as you work so I can get a better idea about what you are doing and why. Also, you should feel free at any point during the task to ask questions. In addition, if I see that you are in need of help I will intervene. Enjoy!

Questions or comments about this study should be directed to Mark Cohen (mcohen@lhup.edu)

Specific Task Instructions for the Library Creation Task

Participant ID: _____ **Date:** _____

Library Creation

Background: You have been asked to create a general library that will make it easier to create agents that operate in the Vacuum Cleaner Environment. The library you create here will be reused by other developers so they can quickly develop vacuum cleaner agents.

Steps

1. Execute Herbal by double-clicking the icon labeled Herbal.
2. Using the File->New->Project menu item create a new, empty Herbal project named **vacuum**. Select the type of project first and then click Next to give your project a name. Be sure to use all lower-case letters in the project name.
3. Using the Herbal menu, open the Herbal GUI Editor and add the following types to the library (if you see certain items in the wizard that you are unsure about, feel free to accept the default values):
    a. **action** which contains a single string field called **move**. This type will be used by the agent to perform actions like moving or cleaning a square. This type should be marked as "used for I/O" and should be placed in the vacuum.types library.
    b. **position** which contains two number fields, named **x** and **y**. This type will be used to specify the location of the vacuum cleaner agent. This type should be marked as "used for I/O" and should be placed in the vacuum.types library.
    c. **radar** which contains two string fields named **dir** and **reading**. This type will contain information about the clean or dirty status of the squares around the vacuum cleaner. This type should be marked as "used for I/O" and should be placed in the vacuum.types library.
    d. **spot** which contains a single string field named **status**. This type will be used to specify the clean or dirty status of the square currently occupied by the agent. This type should be marked as "used for I/O" and should be placed in the vacuum.types library.

BREAK

4. Using the Herbal GUI Editor add the following actions to the library (if you see certain items in the wizard that you are unsure about, feel free to accept the default values):

Participant ID: _____ Date: _____

a. **up** which contains a single action clause that adds a new fact of type vacuum.types.action and sets the move field to the literal value **up**. This action should be placed in the vacuum.actions library and can be used to make the vacuum cleaner move up one square.

b. **down** which contains a single action clause that adds a new fact of type vacuum.types.action and sets the move field to the literal value **down**. This action should be placed in the vacuum.actions library and can be used to make the vacuum cleaner move down one square.

c. **left** which contains a single action clause that adds a new fact of type vacuum.types.action and sets the move field to the literal value **left**. This action should be placed in the vacuum.actions library and be used to make the vacuum cleaner move left one square.

d. **right** which contains a single action clause that adds a new fact of type vacuum.types.action and sets the move field to the literal value **right**. This action should be placed in the vacuum.actions library and can be used to make the vacuum cleaner move right one square.

e. **suck** which contains a single action clause that adds a new fact of type vacuum.types.action and sets the move field to the literal value **suck**. This action should be placed in the vacuum.actions library and can be used to make the vacuum cleaner clean the square that the vacuum cleaner is on.

BREAK

5. Using the Herbal GUI Editor add the following conditions to the Vacuum namespace. You can ignore any fields related to output or input variables (if you see certain items in the wizard that you are unsure about, feel free to accept the default values):

   a. **clean** which tests to see if there is a vacuum.types.spot item with a status value restricted to the literal value equal to **clean**. This condition should be placed in the vacuum.conditions library and will be true if the current square occupied by the vacuum cleaner is clean.

   b. **dirty** which tests to see if there is a vacuum.types.spot item with a status value restricted to the literal value equal to **dirty**. This condition should be placed in the vacuum.conditions library and will be true if the current square occupied by the vacuum cleaner is dirty.

   c. **alive** which tests to see if there is a vacuum.types.position item with no restrictions on its fields. This condition should be placed in the vacuum.conditions library and will be true as long as the vacuum cleaner is still alive and well.

6. Your boss has decided that it is better to begin the names of condition with the prefix "is". Using the Herbal GUI Editor rename all of the conditions you just created so they contain the prefix "is". For example, clean should be renamed to **isClean**.

BREAK

7. Using the Herbal GUI Editor add the following operators to the vacuum.operators library:
   a. **moveLeft**: if the isAlive condition is true then perform the left action.
   b. **moveRight**: if the isAlive condition is true then perform the right action.
   c. **moveUp**: if the isAlive condition is true then perform the up action.
   d. **moveDown**: if the isAlive condition is true then perform the down action.
   e. **cleanUpSpot**: if the isDirty condition is true then perform the suck action.
8. Choose either the conditions, actions, **or** operators that you created in the previous steps and add design rationale to them. You should only enter information in the "What is this element?" field in the design rationale. When entering information, keep in mind that someone else will be using this library, so add information that will helpful to other people.
9. Use the Herbal GUI Editor to browse the elements in your library and ensure that you have created them properly.
10. Using the Herbal menu, export the vacuum.operators library to a file in the HerbalEvaluation folder on the desktop called **P[your participant id].hlib** (for example P12.hlib).

Specific Task Instructions for the Model Creation Task

Participant ID: _____ **Date:** _____

Model Creation

Background: In this task you will create a vacuum cleaner agent called Pete that operates in the Vacuum Cleaner Environment. Pete will wander randomly until it finds a dirty square. When a dirty square is encountered Pete will clean it. Your task will be simplified by reusing a library of model elements created previously.

Steps

1. Execute Herbal by double-clicking the icon labeled Herbal.
2. Using the File->New->Project menu item create a new, empty Herbal project named **Pete**.
3. Using the Herbal menu, open the Herbal GUI Editor.
4. Using the Herbal menu, import the library named **P?.hlib** that is located on the desktop.
5. Using the Herbal GUI Editor, browse all of the library elements that were imported. Specifically, examine the actions, conditions, and operators that were imported. Feel free to use the Rationale button to view details about each element.
6. Perform the next two steps in any order:
    a. Using the Herbal GUI Editor (use the agent tab in the editor) to create a new agent in the Pete.models library named **Pete**.
    b. Using the Herbal GUI Editor (use the problem space tab in the editor) create a new problem space named **survive**. This problem space will be used as the top level problem space for Pete. All of Pete's behavior will take place within this problem space, or a problem space below it.
7. Using the Herbal GUI Editor (use the agent tab in the editor), add the survive problem space to the agent named Pete.

BREAK

8. Using the Behavior Design Pattern Wizard you will create a new behavior called wander. The purpose of this behavior is to randomly move left, right, up, and down while the vacuum cleaner is on a clean square. This can be accomplished by using the while loop design pattern.
    a. Specifically, use the Herbal->Behavior Design Patterns menu item to create an unordered while loop behavior called **wander**. Specify that Pete should exhibit

this behavior, and be sure to trigger this behavior while Pete is in the survive problem space. The condition that should trigger this behavior should be the isClean condition because we want Pete to wander when it is on a clean square. Finally, the operators that should be executed while the square is clean are moveUp, moveDown, moveLeft, and moveRight. These operators will happen in any order causing Pete to wander randomly!

9. Using the Behavior Design Pattern Wizard you will create a new behavior called clean. The purpose of this behavior is to clean the current square when Pete is on a dirty square. This can be accomplished by using the while loop design pattern.
    a. Specifically, use the Herbal->Behavior Design Patterns menu item to create an unordered while loop behavior called **clean**. Specify that Pete should exhibit this behavior, and be sure to trigger this behavior while Pete is in the survive problem space. The condition that should trigger this behavior should be the isDirty condition because we want Pete to clean only when it is on a dirty square. Finally, the operator that should be executed while the square is dirty is the cleanUpSpot operator.

BREAK

10. Browse your agent using the Model Browser View located at the bottom of the Herbal window. Make sure that the agent shown in this view matches the agent you intended to build. Check closely for any errors.
11. Go back to each problem space and agent (use the agent and problem space tabs in the Herbal GUI Editor) that you created in the previous steps and add design rationale to each of them. You don't have to fill in all of the fields. However, keep in mind that over time you may forget what these model components do. The design rationale you enter here will help you recall how your agent works and is also helpful for anyone else who tries to understand how agent Pete works.

BREAK

12. It is now time to test your agent. Double-click on the file named vacuum_2.0.jar in the My Computer Window currently showing in the task bar at the bottom of the screen. This will execute the Vacuum Cleaner Environment. Using the File->Open Soar Agent menu item, brows to the HerbalEvaluation\Workspace\Pete\output\soar directory and open the Pete.soar file.
13. Click on the Run button and watch Pete go to work. Is Pete executing as you expected?

14. Go back to the Herbal window and make sure that the Navigator view is visible on the left-hand side of the Herbal window. Open the Pete project node and then open the output\soar folder. Double click on the Pete.soar file and the contents of the file will be displayed. This is the computer program that was created automatically by Herbal!

Specific Task Instructions for the Model Maintenance Task

Participant ID: _____ **Date:** _____

Model Maintenance

Background: A vacuum cleaner agent has been created named Pete that operates in the Vacuum Cleaner Agent Environment. Your job is to execute Pete and determine if it is operating correctly. Pete is supposed to wander the environment randomly. When Pete encounters a dirty square it should clean it and the resume wandering. If you observe a problem with Pete's behavior you will debug the agent to find and correct the problem.

Steps

1. Double-click on the file named vacuum_2.0.jar. This will execute the Vacuum Cleaner Environment. Using the File->Open Soar Agent menu item, browse to the HerbalEvaluation\Workspace\BrokenPete\output\soar directory and open the Pete2.soar file.
2. Click on the Run button and watch Pete go to work. Is it executing as you expected? Are squares getting cleaned?
3. Stop the vacuum cleaner agent.
4. You will need to debug the vacuum cleaner agent so you can understand the problem that you discovered in step 2. Execute Herbal by double-clicking the icon labeled Herbal, and open the Herbal GUI Editor. Next, click on the Debug View tab on the bottom of the Herbal window. Expand the Debug View so that it takes up a larger portion of the Herbal window.
5. Go back to the Vacuum Cleaner Environment, reset the board, and run the agent. Quickly switch to Herbal and click on the Connect button in the Debug View. Next, select the agent from the drop down list box. Finally, click on Listen button.
6. Allow the vacuum cleaner agent to run for a while and wait while Herbal generates a trace of the running agent. After you get at least 15 events, click the Stop button in the Vacuum Cleaner Agent Environment and then click on Disconnect in the Herbal Debug View.
7. Examine the trace to see if you can find the problem with the agent.
8. You will now try and fix the problem. Using the Working Set View on the left-hand side of the Herbal window, create a new working set and then use the Add Elements button to search the model for elements that might help you find the problem. Select search

criteria that will give you model elements that are likely to be related to the problem (what keywords should you search for?. Click Finish when you are done searching and the elements that were found will be added to your working set.

9. Double-click on each element in your working set to get a closer look. Study these elements in detail until you find the problem that is causing Pete to malfunction. Try to fix the problem.
10. If you think you have fixed the problem, go back to the Vacuum Cleaner Environment and run the agent again to see you were successful.

User Reaction Survey for the Herbal Study

Participant ID: _____ **Date:** _____

Task (circle one): library creation / model creation / library maintenance

Visibility and Juxtaposability

1) **How easy was it to see or find the various parts (e.g., problem spaces, operators, conditions) of your agent or library while it was being created, changed, or debugged.**

very easy easy neutral difficult very difficult

2) **If you needed to compare different parts (e.g., problem spaces, operators, conditions) of your agent or library, you could easily see these parts at the same time.**

strongly agree agree neutral disagree strongly disagree

Viscosity

3) **How easy was it to make changes to your agent or library?**

very easy easy neutral difficult very difficult

4) **Were there changes that were especially difficult to make?**

Diffuseness

5) **The elements (e.g., problem spaces, operators, and conditions) you used to build your agent or library allowed you to say what you wanted to say reasonably briefly.**

strongly agree agree neutral disagree strongly disagree

User Reaction Survey for the Herbal Study

Participant ID: _____ **Date:** _____

Task (circle one): library creation / model creation / library maintenance

6) **Where there any things (e.g., problem spaces, operators, and conditions) in your agent or library that took too much space to describe?**

Hard Mental Operations

7) **In general, the task you performed did not seem especially complex or difficult to work out in your head.**

strongly agree agree neutral disagree strongly disagree

8) **During this task, what kinds of things required a lot of mental effort?**

Error Proneness

9) **During this task, you often found yourself making small mistakes that irritated you or made you feel stupid.**

strongly agree agree neutral disagree strongly disagree

User Reaction Survey for the Herbal Study

Participant ID: _____ **Date:** _____

Task (circle one): library creation / model creation / library maintenance

10) What mistakes did you encounter during the task that seemed particularly common or easy to make?

Closeness of Mapping

11) The notation (e.g., problem spaces, operators, and conditions) you used to describe your agent or library was closely related to how you might describe the agent or library naturally.

strongly agree agree neutral disagree strongly disagree

12) Which parts of the notation (e.g., problem spaces, operators, conditions) used to describe your agent or library seemed to be a particularly strange way to describe something?

Role Expressiveness

13) During the task, you often did not know what many of the agent or library pieces meant (e.g., problem spaces, operators, conditions) but you put them in anyway.

strongly agree agree neutral disagree strongly disagree

User Reaction Survey for the Herbal Study

Participant ID: _____ **Date:** _____

Task (circle one): library creation / model creation / library maintenance

14) For which parts (e.g., problem spaces, operators, conditions) of the agent or library did you not understand the meaning?

Progressive Evaluation

15) It was easy to stop in the middle of creating the agent or library, and check your work so far.

strongly agree agree neutral disagree strongly disagree

16) During this task, it was easy to find out how much progress you made, or check what stage in your work you were in.

strongly agree agree neutral disagree disagree

Premature Commitment

17) When working on this task, there were times when you felt like you could have changed the order you performed the steps without breaking the agent or library.

strongly agree agree neutral disagree strongly disagree

Appraisal

16) How stressful did you find task to be?

1	*2*	*3*	*4*	*5*
Not at all		*Moderately*		*Very*

Questions or comments about this study should be directed to Mark Cohen (mcohen@lhup.edu)

User Reaction Survey for the Herbal Study

Participant ID: _____ **Date:** _____

Task (circle one): library creation / model creation / library maintenance

17) How well do you think you coped with the upcoming task?

1	*2*	*3*	*4*	*5*
Not at all		*Moderately*		*Very*

18) How demanding did you find the task to be?

1	*2*	*3*	*4*	*5*
Not at all		*Moderately*		*Very*

19) How well do you think you performed in the task?

1	*2*	*3*	*4*	*5*
Not at all		*Moderately*		*Very*

Wrap Up

20) After completing this questionnaire, can you think of obvious ways that the design of the system could be improved? What are they?

Data Collection Form for Herbal Study

Participant ID: _____ Date: _____

Task (circle one): library creation / model creation / library maintenance

Start Time: _____ End Time: _____

Comments made by participant:

Errors observed including any assistance offered:

Other:

VITA

Mark A. Cohen, Ph.D.

Residence: MARK A. COHEN
91 Ravine Drive
Jersey Shore, Pennsylvania 17740
(570) 753-8178
mark.cohen@acm.org

Office: 226 Akeley Hall
Department of Bus., CS & IT
Lock Haven University
Lock Haven, PA 17745
(570) 484-2493
mcohen@lhup.edu

EDUCATION

PhD, College of Information Sciences and Technology, 2008, The Pennsylvania State University

MS, Computer Science**,** 1996, Drexel University

BS, Electrical Engineering, 1991, Lafayette College

TEACHING EXPERIENCE

Assistant Professor of Computer Information Science — Since September 2002
Department of Business Administration, Computer Science, and Information Technology
Lock Haven University of Pennsylvania, Lock Haven, PA

Adjunct Instructor — January 1997 to January 2004
University of Massachusetts, Lowell, MA

INDUSTRY EXPERIENCE

Senior Java Consultant — August 1999 to August 2002
GlaxoSmithKline, King Of Prussia, PA

SELECTED PEER REVIEWED PUBLICATIONS

Cohen, A. M., Ritter, F. E., & Haynes, S. R. (2007). Using Reflective Learning to Master Opponent Strategy in a Competitive Environment. In Proceedings of International Conference on Cognitive Modeling, 219-228 Ann Arbor, MI.

Cohen, M. A. (2005). The Development of a Game Playing Framework Using Interface-based Programming. *Best of Crossroads: The ACM Student Magazine, Fall 2005*.

Cohen, M. A., Ritter, F. E., & Haynes, S. R. (2005). Herbal: A high-level language and development environment for developing cognitive models in Soar. In proceedings of the *14th Behavior Representation in Modeling and Simulation,* 133-140. University City, CA.

Cohen, M. A. (2005). Teaching agent programming using custom environments and Jess. *The Newsletter of the Society for the Study of Artificial Intelligence and Simulation Behavior, 120,* 4.