Published Summer 2010. AI Magazine, 31(2), 25-44.

Applying Software Engineering to Agent Development

Mark A. Cohen
mcohen@lhup.edu
Lock Haven University
401 N. Fairview St.
Lock Haven, PA 17745
USA

Frank E. Ritter
frank.ritter@psu.edu
Penn State University
College of Information Sciences & Technology
The Pennsylvania State University
University Park, PA 16802
USA

Steven R. Haynes
shaynes@ist.psu.edu
Penn State University
College of Information Sciences & Technology
The Pennsylvania State University
University Park, PA 16802
USA

Wednesday, March 03, 2010

Abstract

Developing intelligent agents and cognitive models is a complex software engineering activity. This article shows how tools to create intelligent agents can be improved by taking advantage of established software engineering principles such as high-level languages, maintenance-oriented development environments, and software reuse. We describe how these principles have been realized in the *Herbal* integrated development environment, a collection of tools that allows agent developers to exploit modern software engineering principles.

Keywords

Cognitive Modeling, Intelligent Agents, Problem Space Computation Model, Soar, Jess, High-level Languages, Software Engineering

Introduction

This paper is about how to make it easier to create intelligent agents by applying established software engineering principles. We present an example integrated agent development environment that realizes these principles, and provides lessons for other agent architectures, both existing and in development.

Creating complex software is not a new problem, and the software engineering community has developed principles about how to solve complex problems with software. Developing intelligent agents is a complex software engineering activity but the benefits of applying software engineering principles such as high-level languages, maintenance-oriented development environments, and software reuse to intelligent agent development have not yet fully migrated to the agent development community. We demonstrate how these principles have been realized in the *Herbal Toolset*, a collection of tools that embody these three core principles. In this paper, we introduce the Herbal Toolset and the principles that informed its design.

High-Level Languages for AI and Cognitive Modeling

Intelligent agents and cognitive models are useful, but for different reasons. The processing speed and accuracy of an intelligent agent can help humans dial cell phones quickly and accurately when driving a car, while a cognitive model can help predict common 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. There are many interesting and important applications of cognitive models and intelligent agents. Agents can be used for education and training, especially 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).

Unfortunately, development of intelligent agents and cognitive models is challenging. In more conventional software engineering domains, high-level languages are often used by developers because they simplify the creation of complex systems. However, many of the intelligent agent and cognitive modeling architectures in use today rely on low-level, rule-based programming languages (e.g., Soar, ACT-R). While programming in these languages is not as primitive as using assembler, these low-level production systems do exhibit a similar problem: The concepts that are embodied in the low-level rules sometimes bear little resemblance to the concepts that are used by the programmer to solve the problem. Higher-level representations are needed to close the gap between the theory and the behavior representation languages (e.g., Cooper & Fox, 1998; Jones, Crossman, Lebiere, & Best, 2006; Salvucci & Lee, 2003).

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.

For example, 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 (John, 2003). 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).

ACT-Simple is another example of a high-level language designed to simplify cognitive modeling. 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 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. Unfortunately, G2A and ACT-Simple support only one underlying architecture (ACT-R). In addition, the high-level language supported by G2A and ACT-Simple is based on GOMS, which is limited to modeling expert behavior in simple tasks. Providing support for multiple architectures and a high-level language based on a richer psychological theory would be beneficial.

The High Level Symbolic Representation (HLSR) project is another example of a high-level cognitive modeling language. The HLSR project aims at creating a formal language that encompasses a wide variety of modeling tasks using a variety of cognitive architectures (Jones, Crossman, Lebiere, & Best, 2006). Currently, HLSR creates Soar and ACT-R productions.

HLSR supports multiple architectures using *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.

However, the architectural neutrality of HLSR results in the lack of explicit support for widely used unified theories of cognition (e.g., PSCM and ACT-R). Instead, the 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.

The lessons described above are important for builders of agent and cognitive modeling development environments and these lessons informed the high-level agent construction language adopted by the Herbal Toolset introduced in this article.

Maintenance-Oriented Development Environments

For complex systems, the process of maintenance is the most expensive phase of a system's development life cycle (Boehm, 1987; Brooks, 1995). 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. For programmers of

intelligent agents and cognitive modelers this problem may be even more acute, given the complexity of the software they typically develop.

Fortunately, the use of high-level languages can help with maintenance (Brooks, 1995). A 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, a survey done by Hordijk and Wieringa (2005) also identified development environments as a factor that influences maintainability. This implies that creating *maintenance-oriented environments* (environments that explicitly support software maintenance) can help reduce the cost of software development.

Researchers have taken notice. For example, Ko, Aung, and Myers (2005) looked at how Java programmers perform software maintenance. Their study suggested that programmers form a *working set* of task relevant code fragments that is typically built by using a find and replace dialog or by visually searching the source code. Ko, Aung, and Myers believe that support for working sets can help simplify maintenance.

Another way to streamline the maintenance process is to make it easier for programmers to access the design rationale underlying a system (LaToza, Venolia, & DeLine, 2006). Easy access to this design rationale is crucial, but developers currently spend much of their time reconstructing design rationale that is implicitly embedded within a program's source code (LaToza, Venolia, & DeLine, 2006). It has long been recognized that one of the key barriers to comprehending software is the invisibility of its structure relative to the functions it performs (Brooks, 1987). For intelligent agents and cognitive models, this problem is especially acute as these systems are often intended to carry out complex and consequential operations similar to and sometimes in place of people. Because of this, intelligent agents are expected to be accountable for the actions they perform, to explain how and why their reasoning led to a particular action.

An early implementation of this idea was the XPLIAN architecture for creating intelligent systems (Swartout, 1983). XPLAIN was designed expressly to provide explanations of an expert system using the design rationale underlying its structure and behaviors. The XPLAIN knowledge base includes justifications for system structure, behavior, and general problem solving strategies, as well as a mapping between terms and definitions used in the design to those in its domain of intended use. Other examples of how design rationale has been used to support explanation in intelligent and other complex systems are described in (Haynes, 2006).

More recently, Haynes, Cohen, and Ritter (2008) have proposed a novel approach to supporting explanation in agent development environments. Their work presents a set of guidelines for developing agents that explain themselves based on a study of the questions that users asked while working with an intelligent system. Three general design patterns for creating explainable agents emerged from the analysis of this study: the ontological explanation pattern, the mechanistic ontological explanation pattern, and the operational explanation pattern. These three explanation types make explicit the rationale underlying development of an intelligent agent or cognitive model.

Ontological explanations are designed to provide answers to questions about the static structure of an agent's design. *Mechanistic explanations* provide insight into how

the components within an agent interact to produce behavior. Finally, the *operational explanations* describe how a modeler can access and utilize an agent's functionality. All three of these explanation types help provide the rationale underlying the design of the agent. The advantage of access to these explanations is that developers and users can understand the intent of the program designers. Building support for these types of explanations into a high-level language and agent development environment can simplify the creation of intelligent software.

Software Component Reuse

Reuse of source code, even within a single program, can reduce development and maintenance costs (Boehm, 1987; Brooks, 1995; Krueger, 1992). One of the earliest discussions of the importance of software reuse was presented by McIlroy (1968). He described a view of the future of software reuse that included the availability of safe, well-tested software components tailored to each user's needs.

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

Abstraction allows programmers to consider a programming task separate from the concrete realities of the modeling language, and is the reason why high-level languages provide more effective 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, allows programmers to locate and select appropriate reusable components. Good maintenance-oriented environments can make it easy for clients to search for reusable components based on criteria that are tied directly to the design rationale.

S*pecialization* allows programmers to tailor reusable components to their specific needs. Without support for specialization, developers are unable to configure a component for their specific use.

The fourth of Krueger's dimensions is *integration*, which allows developers to combine reusable components into a working program. A maintenance-oriented environment should play a major role in simplifying the integration piece of reusable software.

Design patterns can provide an effective way to implement Krueger's dimensions. *Design patterns* are reusable templates that provide solutions to recurring problems (Gamma, Helm, Johnson, & Vlissides, 1995). A design pattern consists of four central 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.

The Herbal Toolset

The software engineering community has developed strong theories and principles about how to solve complex problems with software solutions. High-level

languages have been shown to simplify software development (Beck & Perkins, 1983; Daly, 1977; Maxwell, Wassenhove, & Dutta, 1996), yet many popular agent development environments do not yet support high-level programming languages (Jones, Crossman, Lebiere, & Best, 2006). The advantages of software maintenance and accessible design rationale are also clear (Boehm, 1987; Brooks, 1987; LaToza, Venolia, & DeLine, 2006; Tassey, 2002), yet many popular agent development architectures (e.g., Soar and ACT-R) are lacking development environments that incorporate these features (Cooper & Fox, 1998; Pew & Mavor, 1998; Ritter et al., 2003). Finally, code reuse has been shown to reduce development and maintenance costs (Boehm, 1987; Brooks, 1995; Krueger, 1992), yet reuse is difficult to achieve in most intelligent agent development environments, which are built around lower-level languages.

The Herbal Toolset is an attempt to improve agent development by providing a high-level language and maintenance-oriented agent development environment that offers first-class support for design rationale and software reuse. Software engineering research suggests that this toolset, or any toolset designed with these principles in mind, will lead to more productive agent developers and useful agents. The next section explains how the Herbal Toolset realizes these three principals.

Herbal: A High-Level Behavior Representation Language

To simplify agent programming and cognitive modeling, a high-level behavior representation language, and associated parser and compiler were designed and implemented as the Herbal Toolset. This high-level language is based on the *Problem Space Computational Model* (PSCM) and is represented using the *Extensible Markup Language* (XML) (www.w3.org/XML/). This language is currently compiled 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 as a High-Level Language

The high-level language supported by the Herbal Toolset is based on the PSCM (Lehman, Laird, & Rosenbloom, 1996; Newell, Yost, Laird, Rosenbloom, & Altmann, 1991). The PSCM defines behavior as movement through a problem space (see Figure 1), which is a high-level organizational representation to explain high-level behavior. The PSCM provides a proven conceptual model for implementing a high-level language for agent development.

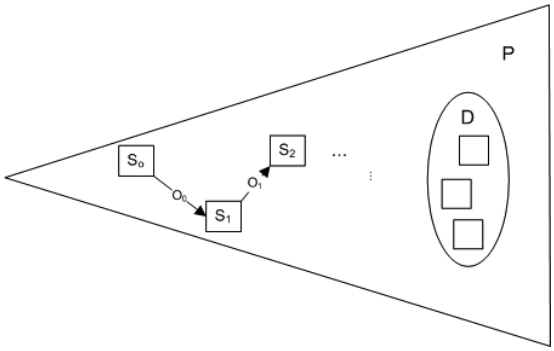

Figure 1. The PSCM defines behavior as movement through a problem space. Based on Exhibit 11.7 in Newell, Yost, Laird, Rosenbloom, & Altmann (1991).

A *problem space* is defined by a set of *states* (e.g., 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).

The PSCM was first proposed by Allen Newell as a unified theory of cognition, and this theory was implemented in the Soar Cognitive Architecture (1990). The successful use of the PSCM by Soar for the creation of cognitively plausible agents provides evidence of the utility of the PSCM as a unified theory of cognition (Jones et al., 1999; Tambe et al., 1995). This prior success, along with our own experience using Soar, has provided motivation for our use of the PSCM as the foundation for the high-level language used by the Herbal Toolset.

The PSCM can also serve as a general organizational structure for AI-oriented intelligent agents. Clancey (1981) argued that rule-based agent development is complex because the problem solving strategy is often implicitly hidden within the rules. A language explicitly providing PSCM constructs can alleviate this problem by partitioning behavior into a hierarchy of problem spaces, operators, states, and desired goals.

XML and XML Schema

The PSCM-based high-level language supported by the Herbal Toolset takes the form of an XML application, which implements the PSCM and is translated into a low-level rule-based representation for execution within an agent environment (see Figure 2).

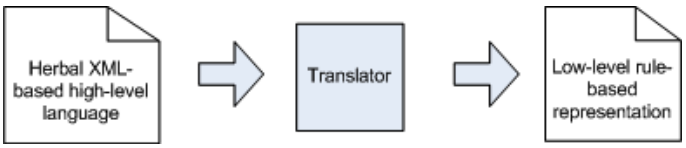

Figure 2. High-level XML representation is translated into low-level rule-based representations.

Using an XML as an intermediate representation provides many benefits. For example, XML allows for the creation of structured documents that can directly represent the hierarchical structure of the PSCM as a higher-level agent development language. In

addition, 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). XML can also be transformed into other formats using the *Extensible Stylesheet Language* (XSL) (www.w3.org/Style/XSL/). This makes it possible to transform Herbal agent code into other formats such as HTML documentation, or *Scalable Vector Graphics* (SVG) (www.w3.org/Graphics/SVG/). Finally, the popularity of XML reduces the developer's learning curve that might otherwise form a barrier to its adoption.

The Herbal high-level language specification is defined using *XML Schema* (www.w3.org/XML/Schema). The use of XML Schema for our language was advantageous for many reasons. XML Schema allowed us to provide clear documentation of the structure and content of the Herbal high-level language. In addition, the Herbal XML Schema is used directly by XML parsers to validate the content of an Herbal program. Lastly, most commercial and open source XML editors utilize XML Schema to provide features such as syntax highlighting and auto completion.

An Herbal program is made up of six different types of XML documents, each defining a set of reusable components: *types*, *conditions*, *actions*, *operators*, *problem spaces*, and *agents*. These documents are referred to as *libraries* in the Herbal language. In many cases, the components in these libraries mirror the elements of the PSCM. However, there are components in the Herbal high-level language (such as types, conditions, and actions) that supplement the PSCM by providing additional levels of abstraction, and support reuse in ways the rules cannot because the rules are too coarse.

The left-hand side of Table 1 below shows a section of an XML Schema file that defines an operator element. An operator is a major component of the PSCM and is a first-class object in the Herbal high-level language. According to the specification shown in Table 1, an operator element 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 are defined in a separate XML document (library) and whose syntax is specified in a separate XML Schema.

The right-hand side of Table 1 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 1. 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 given in additional libraries.

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>
```

Table 1. The operator XML Schema specification and an operator instance of this specification.

As discussed earlier, the XML Schema and associated XML code shown in Table 1 can be edited graphically using any of the commercial or open source XML editors. For example, Figure 3 shows an XML document, containing instantiations of several operators, being edited in XML Notepad. In Figure 3, XML Notepad is indicating that an operator is missing the required name attribute, and can help the programmer add this attribute. For additional information about the Herbal high-level language specification, see the Herbal XML Programmer's Guide available from the Herbal website (acs.ist.psu.edu/herbal).

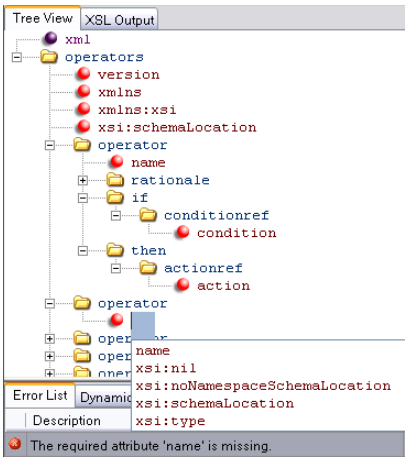

Figure 3. Herbal programming using XML Notepad.

The Herbal Parser and Compiler

Code written in the Herbal high-level language can be transformed into executable productions for either the Soar or Jess agent architectures. The first phase in this transformation (shown in Figure 4) consists of parsing the XML code and creating a *Document Object Model* (DOM) of the PSCM. The Herbal parser is written in Java, and the DOM consists of a hierarchical collection of Java objects.

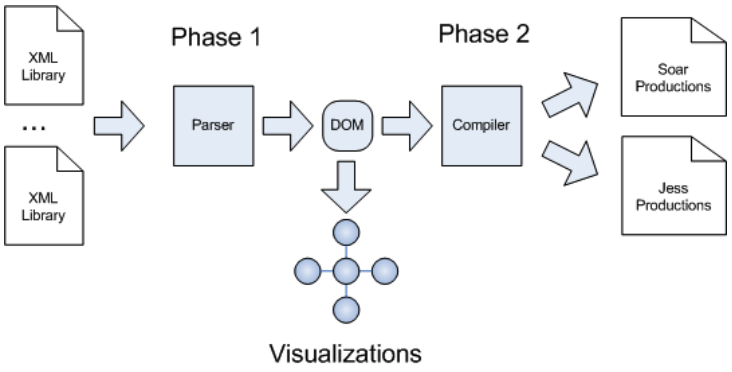

Figure 4. Parsing and compiling Herbal XML source code.

A standard XML parser is used to parse the XML libraries. This parser validates the XML based on the associated XML Schema. In addition, the parser is extended with custom logic that checks for semantic errors.

The DOM is used for the creation of useful visualizations or the creation of executable productions. The Herbal compiler is responsible for the transformation of the DOM into executable code.

The *Herbal Application Programming Interface* consists of a set of Java interfaces and abstract classes that make it possible to create compilers for multiple architectures. The compiler API can be implemented and extended to transform the DOM into different types of executable code. There are currently two concrete compilers in the API: one that produces Soar productions and another that produces Jess productions and facts.

The main challenge in creating an agent 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 is related to how explicitly the underlying language supports the PSCM. For example, the Soar architecture is a direct instantiation of the PSCM, while Jess provides no explicit support for the PSCM.

A few examples are provided to illustrate how the Soar and Jess compilers transform the PSCM DOM into appropriate productions. Consider the Herbal XML code shown in Table 2. This code defines a condition called *dirty,* that tests if a vacuum cleaner agent is on a dirty square (Cohen, 2005). Table 2 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. However, this example shows one of the main benefits of the high-level language: Unlike the resulting Soar and Jess code, the Herbal XML Language makes conditions explicit. In other words, in Herbal conditions are named and unambiguous, making it easy for developers to recognize them, and importantly, reuse them.

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")))`

Table 2. A translation from an Herbal condition to Soar and Jess source code.

A second example, given in Table 3, 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 with the environment. Again, this example shows the advantage of the high-level language. The Herbal actions are easy to identify and reuse.

Soar defines explicit structures to support an agent's communication with its environment. These structures take the form of an *input link* and an *output link*. As a result, the Herbal compiler adds the *Clean* working memory element directly to the output link (labeled <*i2*> in Table 3). Jess, on the other hand, has no special language constructs that deal with agent/environment interaction so the *Clean* command is treated like any other fact in working memory.

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") ))`

Table 3. A translation from an Herbal action to Soar and Jess source code.

The third example, shown in Table 4, demonstrates how the Herbal compiler transforms an Herbal operator. Recall that the operator is an important component of the PSCM. Surprisingly, Soar does not have a single explicit syntax for declaring operators. Instead, operators arise implicitly with productions. In addition, unlike Soar, the Jess language has no concept of operators at all. As a result, the operator concept must be simulated in Jess using a basic production.

Table 4 shows the differences between how the Herbal compiler produces operators in Soar and Jess. This example clearly illustrates the advantages of using a high-level language. In Table 4, the Herbal high-level language declaration of an operator is far more obvious, and in the case of Soar, far more concise. The first-class status of conditions and actions in the Herbal high-language made this possible because the detailed specification of conditions and actions are referenced rather than duplicated. This is a clear example of how high-level languages can reduce errors and development time by eliminating code duplication and making code easier to read and understand.

For the Herbal to Jess rendering, the Herbal operator is translated directly into a single 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 also necessary to support learning in Soar. The Soar productions shown in Table 4 show how the Herbal compiler produces both the proposal and the application rules for the Herbal operator, reducing the time and chance for error inherent in coding these operators by hand.

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. A translation from an Herbal operator to Soar and Jess source code.

The examples given in Table 2, Table 3, and Table 4 illustrate how the Herbal high-level language represents and implements the PSCM. In some cases (e.g., the addition of conditions and actions as first-class objects), these modifications have added greater granularity, which allows for easier 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 creating models in architectures that do not provide direct support the PSCM. For example, interruptability, which was described previously as an important part of the psychological plausibility of Soar agents, is not accounted for in agents compiled to Jess. While minimized, these trade-offs are common throughout the design of the Herbal Toolset. In all cases, however, the high-level code is more explicit, making it easier for the developer to recognize and reuse the key components of the PSCM.

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 by leveraging the popular

Eclipse extensible platform (eclipse.org), and by providing integral support for design rationale and working sets.

The Herbal IDE

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

The Herbal IDE allows for the creation of Herbal agents graphically and by programming directly in the Herbal High-Level Language. Agent programmers can freely switch between these two modes.

Graphical editing in Herbal is accomplished with the *Herbal GUI Editor* (shown in Figure 5). Like the Herbal high-level language, the editor is library centric. Using the editor, programmers can create or modify existing library components (i.e., types, conditions, actions, operators, problem spaces, and agents) without having to write code in the Herbal high-level language—the Herbal XML code is created as the developer interacts with the GUI Editor.

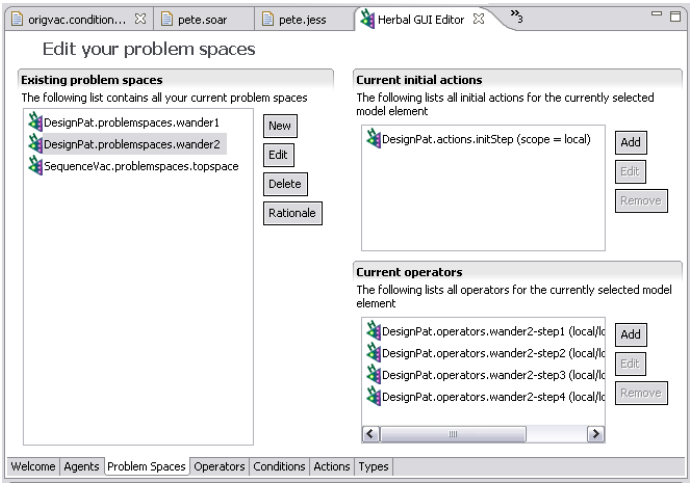

Figure 5. The Herbal GUI Editor.

In addition to saving time and reducing programming errors, the GUI Editor can also be used as a means for teaching the Herbal XML language. Developers can create a PSCM component using the GUI Editor and then inspect the generated XML code. Switching between the editor and the generated code, programmers can 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. At any time during

development, programmers can edit the Herbal XML code directly, and these changes are immediately reflected in the GUI Editor (see Figure 6).

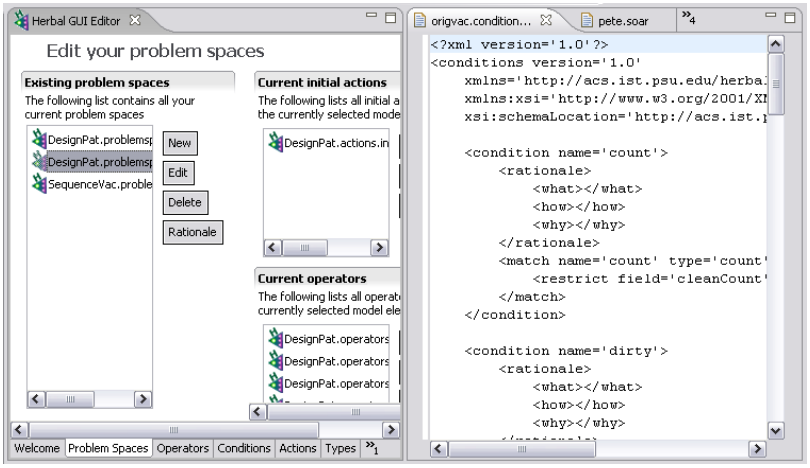

Figure 6. Developing agents using both the GUI Editor and by editing the Herbal XML by hand.

Typical of most Eclipse plug-ins, the Herbal compiler is automatically invoked as the programmer is working. 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 important mechanism to support novice developers 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 and then inspect the generated Soar and Jess code to learn how these constructs can be implemented in the underlying architectures. In classroom evaluations of Herbal, this strategy proved to be very useful, especially in computer science classes in which learning how to program the underlying architecture (in this case Jess) was a course objective.

Figure 7 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 the Herbal high-level XML code. The bottom two views in Figure 7 show the generated Jess and Soar code. Finally, along the very bottom of Figure 7 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. By implementing Herbal within Eclipse, developers automatically gain the advantage of its modern code view and debugging capabilities.

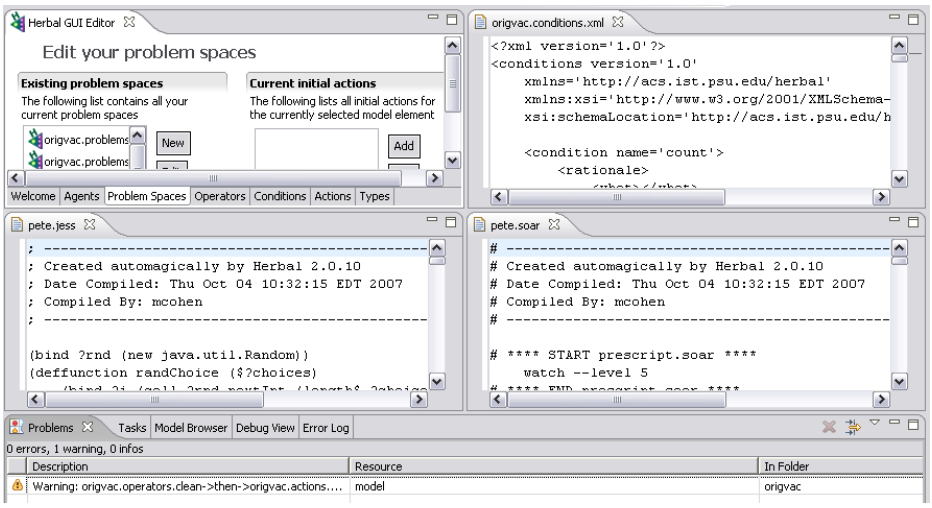

Figure 7. The Herbal IDE showing multiple views of an Herbal library.

Design Rationale

The Herbal IDE incorporates the three general explanation design patterns introduced by Haynes, Cohen, and Ritter (2007). These are the ontological explanation pattern, the mechanistic ontological explanation pattern, and the operational explanation pattern. These design patterns help with the creation of explainable agents, and explainable agents are easier to understand, debug, and modify.

Ontological explanations are designed to provide answers to questions about the static structure of an agent's design. In support of ontological explanations, the Herbal IDE provides the Model Browser View shown in Figure 8. The Model Browser View makes it easy to browse the static PSCM structure of an Herbal agent and therefore simplifies the maintenance of these structures.

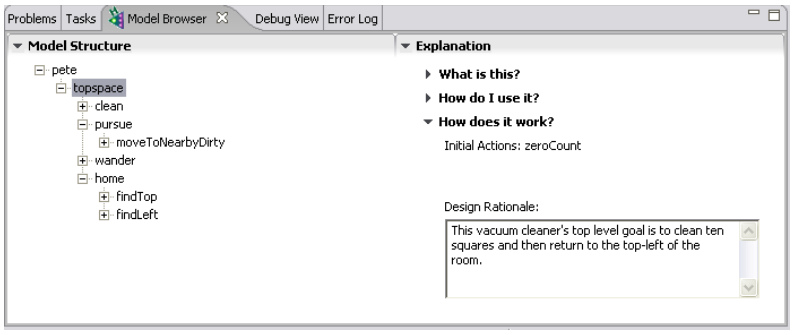

Figure 8. Ontological explanations supported using the Model Browser View.

Mechanistic explanations provide insight into how the components within an agent interact to produce behavior. Mechanistic explanations are typically generated while an agent is executing. As a result, the Herbal IDE provides these types of explanations in the form of a runtime debugger and trace tool. The Herbal Debug View (see Figure 9) shows a trace of the agent interacting with its environment. For each event, the agent and its current problem space is shown, including the currently satisfied

conditions and operators and any actions that have been executed. Finally, relevant working memory is also given.

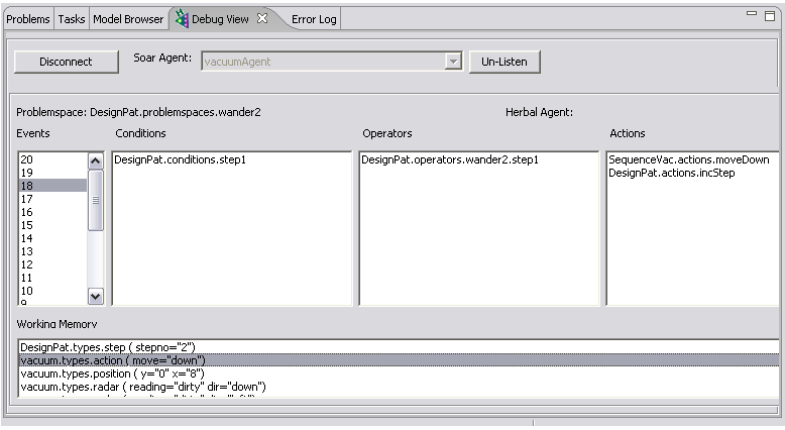

Figure 9. Mechanistic explanations supported using the Debugger View.

Finally, the operational explanations describe how a modeler can access and utilize an agent's functionality. When making changes to an agent's behavior, operational explanations help the programmer decide what components are available and how these components can be used. The operational explanations are available in the Model Browser View as answers to questions such as "How do I use It" and "How does it work" (see Figure 8). This information must be provided by the developer as they build library components.

These three explanation patterns and the design rationale they use as explanatory content support software maintenance and reuse by providing developers access to the intent and design decision making carried out by a component's original developers. This information may make it easier to identify, select, and specialize components developed for analogous applications, and to fully comprehend why a piece of software has the structure and behavior that it does. This understanding contributes to maintenance by avoiding code changes that conflict with designers' intent and consideration of implementation constraints. In addition, because the explanations are given in the context of the PSCM, they can be more psychologically plausible.

Working Sets

As discussed earlier, studies done by Ko, Aung, and Myers (2005) suggest that better support for working sets can help simplify the maintenance task. As a result, support for working sets was added to the Herbal IDE.

As shown in Figure 10, the Herbal IDE makes it possible for developers to build a working set of task relevant code fragments. Working sets can be built manually by the developer, or by searching the libraries using keywords related to the current maintenance task. The collection of code fragments can then be saved as a named working set and shared between developers. Finally, double-clicking on items in the working set will open the code fragment in the Herbal GUI Editor for inspection.

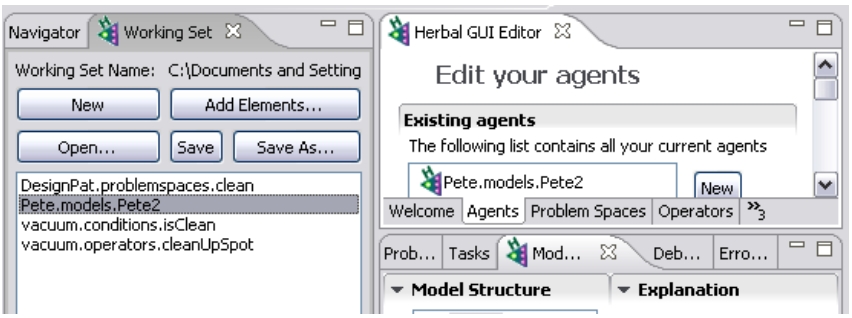

Figure 10. Support for working sets in the Herbal IDE.

Herbal: A Tool for Supporting Reuse

The Herbal Toolset was designed to support several different forms of reuse from the creation of libraries, instantiation of behaviors design, and the support for the reuse of low-level PSCM components. The Herbal high-level language allows for the creation of libraries of reusable components that are uniquely defined using namespaces. In addition, the language allows for the rationale behind the design of each component to be captured as part of the components definition. This rationale makes it easier for developers to understand how to reuse existing components.

Libraries

The Herbal high-level language is library centric, in that Herbal projects 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.

The dependencies between the contents of these libraries are shown in Figure 11. The foundation of all the Herbal libraries is the *types* library. This library contains the set of data types available to the agent programmer. 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. Operators are then built from these conditions and actions, and problems spaces are built from a set of conditions and operators that activate the problem space. Finally, agent behavior can be defined by a hierarchy of problem spaces. This layered approach allows developers to specify behavior at the most appropriate level of abstraction for a given problem.

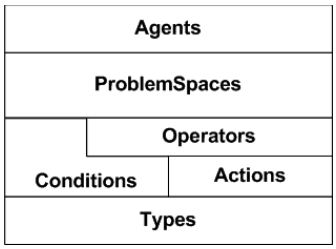

Figure 11. The dependencies between the six different types of libraries in Herbal.

Herbal libraries are uniquely qualified using a namespace. This allows developers to create any number of libraries and share them across models, a fundamental benefit of reusable code implemented in most modern, higher-level programming languages. The Herbal IDE supports library sharing graphically using wizards for the importing and exporting of libraries across projects. This feature automatically detects library dependencies, thus ensuring that the required library components are included in the export.

As an example of how library reuse is supported in Herbal, libraries of reusable components for the vacuum cleaner agent environment were created and prefixed with the namespace vacuum (Cohen, 2005). These libraries were then combined with other libraries and reused to build new vacuum cleaner agents. For example, Figure 12 shows how more aggressive vacuum cleaner agents can be created by reusing the existing vacuum cleaner libraries. In Figure 12, more aggressive operators (aggressive operators could be operators that cause vacuums to clean more aggressively and do less wandering) might are created based on conditions and actions contained in the stock vacuum cleaner libraries. These operators are then used to build new problem spaces and aggressive agents based on these problem spaces. This type of reuse has been used in the classroom environment so that students can assemble agents from reusable components, thus allowing them to spend more time focusing on modeling more complex behavior and less time developing components to implement fundamental behaviors.

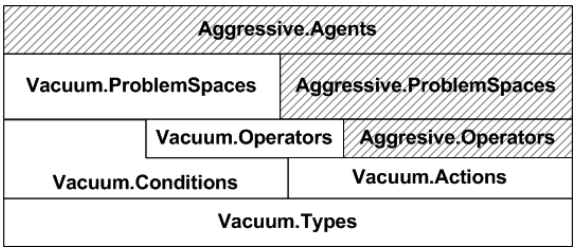

Figure 12. Building custom agents by reusing libraries.

Design Rationale

Operational explanations help with Krueger's selection and integration principals and therefore are an important key to supporting reuse. The Herbal IDE supports Krueger's concept of selection and integration by allowing developers to filter components based on the component's design rationale.

For example, suppose that a developer wishes to find the set of model components that are responsible for a vacuum cleaner agent cleaning dirty squares. Using the Herbal

IDE, model components can be found that are related to the keyword "clean", and these components can be placed into a working set that can be saved and reused. Herbal's selection mechanism takes full advantage of design rationale, which makes it possible for developers to browse library components based on their operation explanations.

Behavior Design Patterns

In evaluations of Herbal in classroom settings, it became clear that there are certain common meta-behaviors. For example, many of the agents created by students for the vacuum cleaner environment and the dTank environment (Ritter, Kase, Bhandarkar, Lewis, & Cohen, 2007) implemented 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*" were common. For the dTank agents, behaviors like "*while no enemy tank is spotted search for an enemy using this search strategy*" were repeatedly implemented.

Structured programming paradigms like looping constructs are useful in agent programming, but can be a challenge to program in a typical rule-based language. This challenge presented a barrier to the students that limited what they could accomplish in their projects. In addition, similar looping constructs are often repeated throughout an agent program. High-level support for these constructs can allow modelers to reuse the behavior they generate, as opposed to duplicating it or creating it from scratch.

To address this problem and to promote the reuse of meta-behaviors such as looping, the Behavior Design Pattern Wizard (see Figure 13) was incorporated into the Herbal development environment. This wizard makes it possible for the agent developer to generate instantiations of useful meta-behaviors using existing PSCM components.

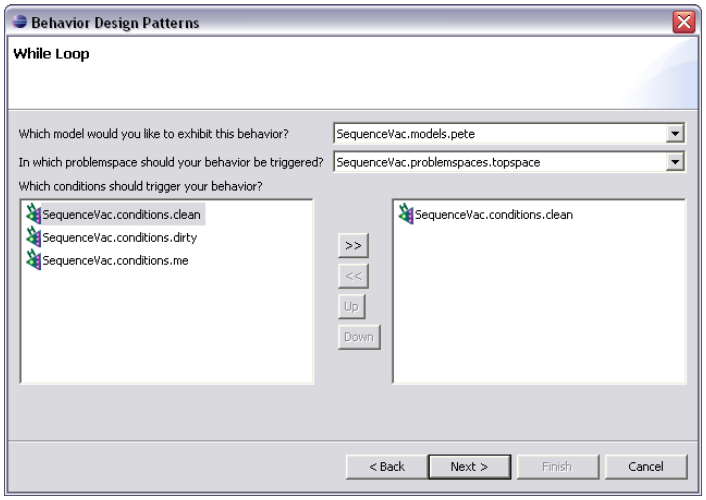

Figure 13. The Behavior Design Pattern Wizard.

Looping Patterns Common in Agent Behavior

Currently, the Behavior Design Pattern Wizard allows developers to instantiate three different patterns of looping: the fixed order loop, the implied order loop, and the unordered loop. These loops are currently novel to Herbal, but would be a useful design pattern for most rule-based modeling languages.

The fixed order loop mimics the classic structured while loop. This loop creates behavior in which a collection of actions is performed in a specific order, but only as long as a set of entry conditions evaluates to true. While a simple construct, this type of loop can be a challenge to create using a rule-based language. Table 5 shows pseudo code for a fixed loop implemented by a vacuum cleaner agent.

The implied order loop is similar to the fixed order loop except that some of the items within the loop might not execute during loop iteration. Table 5 shows pseudo code for an implied order loop implemented for a vacuum cleaner agent.

The unordered loop has no counterpart in the classical structured programming paradigm because the order of the contents within the loop is not specified. As long as the loop conditions are true, the operators inside the loop become candidates for application. Pseudo code for an unordered loop is also given in Table 5.

Pattern Type	Pseudo code

Fixed Order Loop	`while (current square is clean)`
	`{ move one square to the left`
	`move one square up`
	`move one square to the right`
	`move one square down`
	`move one square in a random direction }`

Implied Order Loop	`while (current square is clean)`
	`{ if (certain conditions are true)`
	`move one square to the left`
	`if (certain conditions are true)`
	`move one square up`
	`if (certain conditions are true)`
	`move one square to the right`
	`if (certain conditions are true)`
	`move one square down`
	`if (certain conditions are true)`
	`move one square in a random direction }`

Unordered Loop	`while (current square is clean)`
	`{ Choose only one of the following:`
	`{ if (certain conditions are true)`
	`move one square to the left`
	`if (certain conditions are true)`
	`move one square up`
	`if (certain conditions are true)`
	`move one square to the right`
	`if (certain conditions are true)`
	`move one square down`
	`if (certain conditions are true)`
	`move one square in a random direction } }`

Table 5. Pseudo code for three different types of looping behavior patterns.

Additional meta-behaviors can be included in the Herbal Behavior Design Pattern Wizard by editing a configuration file and providing a custom Java class. This allows developers to increase the library of meta-behaviors as more behaviors are identified.

Discussion and Conclusion

The Herbal Toolset is an example of applying modern software engineering principles to agent programming environments. Specifically, Herbal leverages the software engineering principles of high-level languages, maintenance-oriented development environments, and software reuse to simplify the agent development task.

Early evaluations of the Herbal Toolset have been promising. In an early study using six undergraduate computer science students (Cohen, Ritter, & Haynes, 2009), participants were asked to create an intelligent agent in Jess that piloted two vacuum cleaner agents through a simulated environment: the first agent was created without the use of the PSCM, and the second took advantage of the PSCM as a hierarchical behavior organizational tool. Because these agents were not designed to be cognitively plausible, the PSCM was used primarily for organizing the rules and for making the problem solving strategy explicit. Jess's support for the concept of *modules* and *focus* (see Friedman-Hill, 2003 for more details) made it relatively easy for the participants to implement the PSCM in Jess.

Surveys given to the students at the end of the study showed that they favored the use of the PSCM in their programs. Student responses showed that they agreed that the use of the PSCM made it possible to break up complicated behavior into smaller, less complicated parts. In addition, the survey showed that the students strongly disagreed with the statement that it would be easier to create complicated agents without the use of the PSCM.

A second evaluation, using an early version of Herbal, also showed promise. 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 for an undergraduate to create productions as the library of reusable components (e.g., conditions and actions) expanded (25 minutes for the first few production pairs decreased to five minutes or less beyond the 10^{th} production pair). This reduction in time is believed to be the benefit of increased reuse. 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).

A third, summative evaluation of Herbal was also conducted and the results were positive (Cohen, 2008). Using a cognitive dimensions questionnaire (Blackwell & Green, 2000), 24 undergraduate students majoring in Computer Science, Computer Information Science, Management Information Science, and Psychology were asked to create an agent using Herbal. The task was broken into three subtasks: creating a reusable library, creating an agent using the library, and finding and fixing a bug in the resulting agent. Data were collected using participant observation and a user reaction survey based on cognitive dimensions (Blackwell & Green, 2000). Herbal scored high in five of nine dimensions: the ability to easily make changes to a model (Viscosity); the conciseness of the language (Diffuseness); the ability to evaluate and obtain feedback an incomplete agent (Progressive Evaluation); the closeness of the language to the way the agent behavior is described naturally (Closeness of Mapping), and the lack of hard mental processing required at the notational level (Hard-Mental Operations). In addition, there was no statistical evidence that there is a correlation between the number and type of observed events during task completion, and the participants' major. This is especially

interesting because it suggests that Herbal may help make cognitive modeling more accessible to students majoring in areas other than computer science.

In addition to the positive preliminary empirical results, the implementation of the Herbal Toolset also provided some important lessons. For example, the trade-off between the power of programming close to the architecture, and the simplicity of programming at a higher-level, was continually reinforced. 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 the underlying architectural support for the PSCM in Jess created a need to limit or simulate portions of the PSCM.

Interestingly, our high-level language also generated opportunities for improving the PSCM. As illustrated in Table 2, Table 3, and Table 4, the addition of conditions and actions as first-class objects of the PSCM added another level of granularity, which allowed for better reuse. In other words, operators that utilize similar conditions and actions no longer need to duplicate the whole operator (previously the smallest unit in the PSCM). This feature was not easy to implement because of the dependencies between actions and conditions (some actions are designed to work with specific conditions). These dependencies were reduced by providing language support for *wiring* conditions to actions at the time they are used. This suggests that the preconditions and actions of operators, previously on the level below the PSCM, can be promoted from the Symbol level to the PSCM level. Without the higher-level constructs provided by Herbal, this would not be possible in Soar.

Another interesting lesson was the need to support structured programming techniques in what is traditionally an unstructured rule-based environment. For example, many of the agents created by our students in various class projects implemented different types of looping constructs. However, creating these constructs in a rule-based language was quite challenging. By including high-level support for structured programming paradigms, such as looping constructs, the agent-programming task was significantly simplified, and yet the rule-based paradigm was extended, not broken.

Some unexpected instructional benefits of the Herbal Toolset were also discovered during this project. 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, programmers can learn the Herbal language by making changes graphically and then viewing the generated XML code. In addition, by editing the herbal XML code directly, and then viewing the generated low-level productions, programmers can learn native Jess and Soar programming.

Developing intelligent agents is a complex software engineering activity. 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. The Herbal Toolset is one example of how these lessons from Software Engineering can be applied to help simplify the task of agent development. Some of the benefits of applying software engineering principles are being realized with the creation of the Herbal Toolset. The complete Herbal Toolset is currently available and can be downloaded for the Herbal website at acs.ist.psu.edu/herbal.

Acknowledgements

The development of this software was supported by ONR (contract N00014-06-1-0164). Comments from Mary Beth Rosson, Richard Carlson, Thomas George, Sue Kase, Maik Friedrich, Olivier Georgeon, and Jonathan Singel have influenced this work. Recognition is also given to the undergraduate and graduate students that have used Herbal and provided important feedback about its design.

Biographical Sketches

Mark Cohen is an associate professor in the Business Administration, CS and IT Department at Lock Haven University. His current research efforts include developing software that simplifies the creation and maintenance of cognitive models. His email address is mcohen@lhup.edu.

Frank Ritter is one of the founding faculty of the College of IST, an interdisciplinary academic unit at Penn State to study how people process information using technology. He works on the development, application, and methodology of cognitive models, particularly as applied to interfaces and emotions. His email address is frank.ritter@psu.edu.

Steven Haynes is a professor of practice in the College of IST. He researches system design, modeling, and development; human-computer interaction; design rationale; system explanation; and the philosophy of technology. Prior to entering academia he worked at Apple Computer, Adobe Systems, and several smaller technology companies in the US and in Europe. His email address is shaynes@ist.psu.edu.

References

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.

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.

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.

Boehm, B. W. (1987). Improving software productivity. *IEEE Computer, 20*(9), 43-57.

Boehm, B. W. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering, 15*(10), 1462-1477.

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.

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.

Cohen, M. A., Ritter F. E., & Haynes S. R. (2009), Evaluating design: A formative evaluation of agent development environments used for teaching rule-based programming. *In proceedings of the Information Systems Education Conference 2009*, 1542-7382, Washington DC

Cohen, M. A. (2008). A theory-based environment for creating reusable cognitive models. (Doctoral dissertation, The Pennsylvania State University, 2008). Retrieved January 6, 2010 from The CAT database.

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.

Cooper, R. P., & Fox, J. (1998). COGENT: A visual design environment for cognitive modeling. *Behavior Research Methods, Instruments, & Computers, 30*(4), 553-564.

Daly, E. B. (1977). Management of Software Development. *IEEE Transactions on Software Engineering, 3*(3), 229-242.

Friedman-Hill, E. (2003). *Jess in action: Rule-based systems in Java*. Greenwich, CT: Manning Publications Company.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Haynes, S. R., Cohen, M. A., Ritter, F. E. (in press, September 2008). Design patterns for explaining intelligent systems. International Journal of Human-Computer Studies.

Haynes, S. R. (2006) Three Studies of Design Rationale as Explanation, in Dutoit, A. H., McCall, R.., Mistrik, I., and Paech, B. (Eds.) *Rationale Management in Software Engineering*. Springer-Verlag, pp.53-71.

Hordijk, W., & Wieringa, R. (2005). Surveying the factors that influence maintainability. *In Proceedings of ESEC-FSE,* 385-388. New York, NY: ACM Press.

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.

Jones, R. M., Crossman, J. A. L., Lebiere, C., & Best, B. J. (2006). An abstract language for cognitive modeling. *In Proceedings of 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.

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 ICSE,* 126-135. New York, NY: ACM Press.

Krueger, C. W. (1992). Software reuse. *ACM Computer Surveys, 24*(2), 131-183.

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.

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining mental models: A study of developer work habits. *In Proceedings of 28th International Conference on Software Engineering,* 492-501. Shanghai, China: 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.

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.

McIlroy, M. D. (1968). Mass produced software components. *In Proceedings of Software Engineering; Report on a conference by the NATO Science Committee,* 138-150. NATO Scientific Affairs Division.

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

Newell, A. (1990). *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

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).

Pew, R. W., & Mavor, A. S. (Eds.). (1998). *Modeling human and organizational behavior: Application to military simulations*. Washington, DC: National Academy Press.

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 16th Conference on Behavior Representation in Modeling and Simulation,* 51-60. Orlando, FL: U. of Central Florida.

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., 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.

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.

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.

Swartout, W.R. XPLAIN: A system for creating and explaining expert consulting programs. *Artificial Intelligence*, 21, (1983), 285-325.

Tambe, M., Johnson, W. L., Jones, R. M., Koss, F. V., J., L., Rosenbloom, P., & Schwamb, K. (1995). Intelligent agents for interactive simluation environments. *AI Magazine*, 15-39.

Tassey, G. (2002). *The economic impacts of inadequate infrastructure for software testing* (No. 7007.011): National Institute of Standards and Technology.

Yost, G. R. (1993). Acquiring knowledge in Soar. *IEEE Expert: Intelligent systems and their applications, 8*(3), 26-34.