College of Information Sciences and Technology

The Pennsylvania State University

A Gentle Introduction to Herbal

(Version 3.1)

Mark A. Cohen	Frank E. Ritter	Damodar Bhandarkar	Olivier Georgeon
mcohen@lhup.edu	ritter@ist.psu.edu	dnb133@psu.edu	olg1@psu.edu

Technical Report No. ACS 2007 - 1

November 7, 2008

Revised 3 oct 2011

acs.ist.psu.edu

Phone +1 (814) 865-4455

Applied Cognitive Science Lab.

The College of Information Science and Technology.

The Pennsylvania State University, University Park, PA 16802

Table of Contents

1   Introduction

Herbal is a high level behavior representation language that is realized through an integrated development environment consisting of a high-level language, a compiler, and a graphical editor that acts as a first step towards creating development tools that support the wide range of users of intelligent agents and cognitive models.

1.1   Why Use Herbal

The main objective of Herbal is to allow developers to focus on the architectural aspects of the cognitive agent while the detailed aspects of the programming nuances are managed by the Herbal compiler.

Additionally, the distinguishing characteristic of Herbal is to create models that explain themselves (Haynes, Cohen, & Ritter, 2009). To achieve this, Herbal formalizes the programming process through the use of an explicit ontology of classes (Mark A. Cohen, Ritter, & Haynes, 2005; M. A. Cohen, Ritter, & Haynes, 2010) that represent concepts of the Problem Space Computational Model (Newell, 1990) including models, states, operators, elaborations, conditions, actions and working memory, all as first class model objects. Programming in Herbal involves instantiating objects using these ontological classes. Thus, the programming process is reduced to simply instantiating objects, rather than coding the classes and structure implicitly in a large set of heterogeneous Soar productions. As a result of using Herbal, the power of cognitive architectures like Soar can be taken advantage of more directly and easily.

Current development of Herbal is focused on supporting ACT-R models better, using the information contained in the Herbal ontology to generate explanations while the model is running, and supporting visual models.

1.2   Structure of Herbal

Herbal is built on the Problem Space Computational Model. For a detailed introduction to the model refer to "A Gentle Introduction to Soar", by Lehman, Laird and Rosenbloom (1998), and earlier works (Newell, 1990; Newell, Yost, Laird, Rosenbloom, & Altmann, 1991).

Herbal is designed based on the concept of self-explanation, where the model is built with the capability of explaining itself (Haynes, Cohen, & Ritter, 2009). In Herbal, the topmost entity, the

agent, operates within a problem space which is driven by a global goal. The goal defines the reason for the agent's existence. Each problem space is a collection of several sub problem spaces which are also goal driven and, in turn, define smaller and more local goals in the service of the topmost problem space.

Problem spaces also form a collection of procedural knowledge realized as operators. An operator associates conditions to actions. Conditions are, in essence, patterns that match facts in working memory. Some of these facts may be the result of sensor readings. Actions result in the creation of new facts in working memory that may result in the agent performing tasks. When an operator is created, it defines the conditions in the environment that trigger necessary actions.

At the most basic level, working memory is defined by instantiations of working memory elements based on types. Types are similar to data types in traditional programming languages and provide structures for storing facts from the environment. A schematic view of the Herbal structure is given in Figure 1.
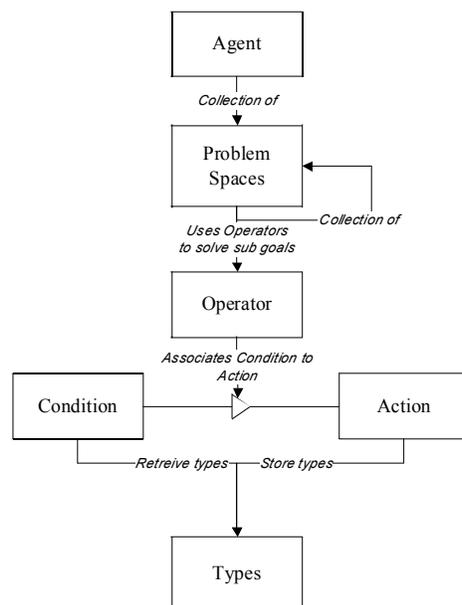
Figure 1 - Structure of Herbal.

2   Installing Herbal

Herbal is developed in the Eclipse Integrated Development Platform, and requires that Eclipse (version 3.1.2 or version 3.2) be installed on your computer. The Eclipse Platform is written in the

Java language and supports extensive plug-in based toolkit construction. Full documentation and downloads of Eclipse are available at *http://www.eclipse.org*.

Before starting the Herbal installation, we encourage you to spend some time reviewing the Eclipse tutorial to reduce your cognitive load while learning Herbal. Specifically, you should read the Getting Started section of the Eclipse Workbench User Guide. This can be found by clicking on the *Help>Help Contents* menu item in Eclipse.

To install Herbal, follow the instructions below. The installation process is remarkably similar for Macintosh and Microsoft Windows machines.

1. To begin with, you need the ability to install software, and have write permission to the applications you install.

2. Go to http://acs.ist.psu.edu/Herbal/index.html

3. Click *Download* on the left frame.

4. On the *Download Page*, right-click on *Download Herbal (Version xx Release)*.

5. Download it to a local folder.

6. Make sure the extension of the file is ".jar" and not "zip" (Internet Explorer 7 may convert the .jar file to a .zip file). If the extension is ".zip" rename it to ".jar"

7. Go to the Eclipse application's plugin folder. For example, go to *C:\Program Files\eclipse\eclipse\plugins* . On the Mac, go to *Macintosh HD/ My Applications/ Eclipse/ Eclipse/ Plugins/*

8. Copy and paste the downloaded ".jar" file to this folder.

9. Close the Eclipse application (if it was open) and reopen it.

10. If an Herbal menu has been added to the main menu in Eclipse, the installation was successful.

You are now ready to begin writing agents in Herbal.

3   Lesson 1: The Basics of the Herbal Interface

The goal of this lesson is to demonstrate how to execute Herbal and provide an overview of the Herbal Interface.

3.1   Starting Herbal and Initializing the Interface

Figure 2 provides a brief overview of the major components of the Herbal interface within the Eclipse environment. All development within the Herbal environment should be done starting with the window arrangement shown in Figure 2. To configure Eclipse so that it contains this arrangement, perform the following steps:

1.   Start Eclipse.

2.   After the Eclipse application finishes loading, you may be presented with the *Eclipse Welcome Screen*. If the welcome screen is displayed, close it.

3.   If the *Outline* view is open on the right hand side of the Eclipse window, close it.

4.   Click on the *Open Perspective Button* located near the top right corner of the Eclipse window. Choose *Other...* and then select the *Herbal* perspective.

5.   Select the *Herbal>Show GUI Editor...* menu Item
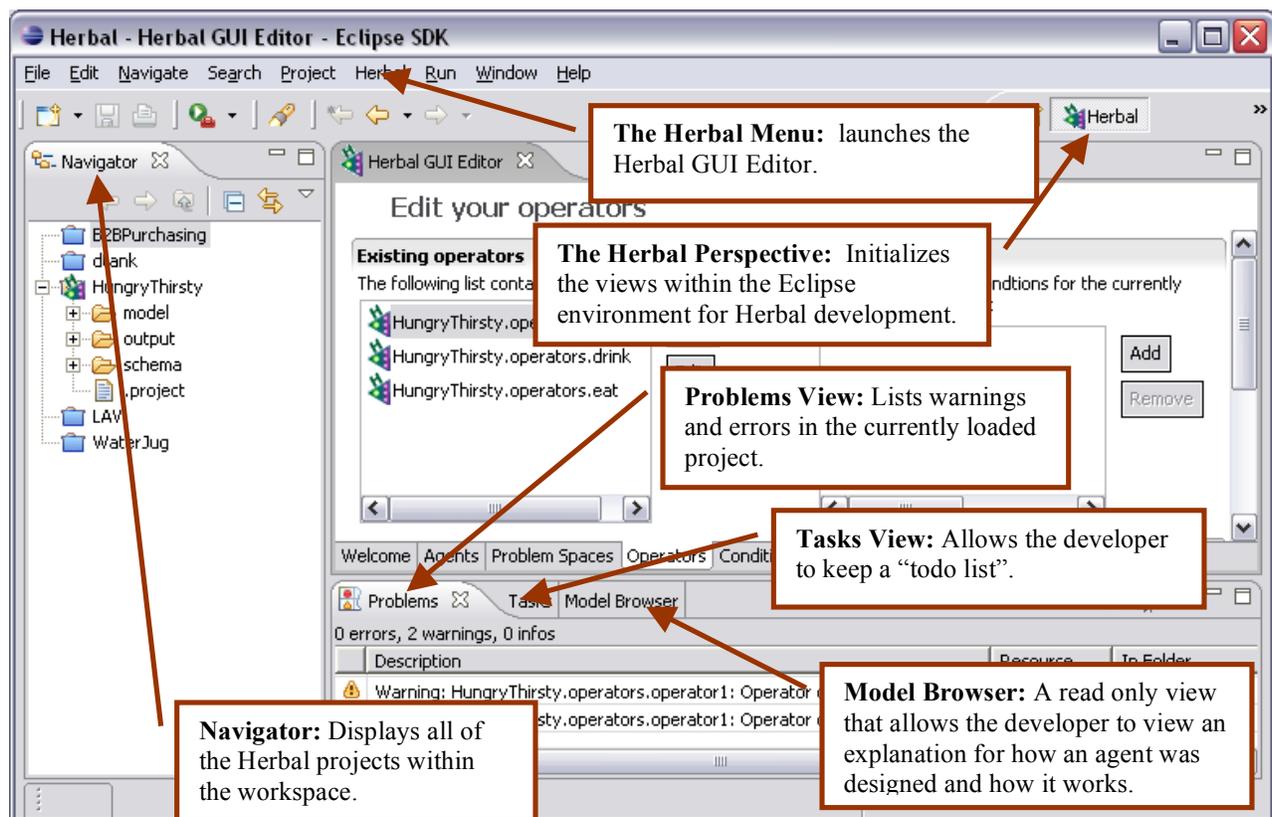
Figure 2 - The Herbal Window.

3.2   Additional Exercises

TBD

4   Lesson 2: Creating a Very Hungry and Thirsty Agent Named Sally

To get you started with Herbal, this section will guide you through the creation of a very hungry and thirsty agent named Sally.

Sally has a simple life: at any given time she may be hungry, thirsty, or both. If Sally is hungry, she will eat; thus causing her to no longer be hungry. Unfortunately, eating causes Sally to become thirsty. If Sally is thirsty, she will drink; thus causing her to no longer be thirsty. Naturally, drinking causes Sally to become hungry again. This cycle continues until Sally's life is ended by way of halting her execution.

4.1   Creating a New Herbal Project

To create a new Herbal project, follow the sequence below.

1. Select *File>New>Project*

2. On the *New Project* window, select *Herbal Project* under the Herbal folder and click *Next*.

3. For the project name enter *HungryThirsty* and choose the default file location to store the project.

4. Click *Next*.

5. The next screen allows you to select from a set of predefined models. In this example, please select *Empty* because we will be building a model from scratch.

6. Click *Finish*.

7. Make sure that your workspace in Eclipse resembles the one shown in Figure 2. If it doesn't, review the steps given in Section 0.

After your project is created, you should see it listed in the *Navigator* view located in the left pane of the Eclipse window.

4.2   Building Agent Sally

In the following sections, Sally will be constructed from a collection of problem spaces, operators, conditions, actions, and types. The order in which Sally is constructed is arbitrary and only loosely constrained by the Herbal environment. Keep this in mind when you start developing your own agents in Herbal. In addition, you should find Figure 2 useful, as the upcoming sections frequently reference the components shown in Figure 2.

4.3   Creating an Agent

From this point on you will be working primarily within the *Herbal GUI Editor*. Taking a top-down approach, we begin by creating the agent itself. To create an agent in Herbal, perform the following steps:

1. Select the *Agents* tab inside the *Herbal GUI Editor* (if you don't see the *Herbal GUI Editor*, select the *Herbal>Show GUI Editor…* menu item).

2. Click on the *New* button located next to the list of existing agents.

3. Accept the default value given in the *Library* field. Libraries are an advanced feature and will be explained later.

4. Enter *Sally* in the *Element Name* field and click *Finish*.

Agents must have at least one goal and operate within at least one problem space to achieve this goal. At this point, Sally is not associated with any problem spaces and therefore will not exhibit any behavior. Herbal recognizes this and places a warning in the Problems View pane to remind the developer that at some point Sally should be associated with a problem space. Take the time now to view this reminder in the Problems View pane (if you can't find this pane, go back, now, to the Eclipse tutorial and don't get lost!).

It is good practice to consult the Problems View often; as it will keep you informed of the various components of the model that remain incomplete. As we build more model components, and create relationships between these components, these warnings will disappear and the model will be ready for testing.

Note: Eclipse defaults set the Problems View to show problems for all projects. To change this so that it only shows problems for the current project, perform the following steps:

1. Select the *Problems* tab on the bottom of the Eclipse IDE.

2. Click the *View Menu* button (it is a downward facing triangle).

3. Select *Configure Contents*

4. Select the box *All Error* in the left pane and *On any element in the same project* in the Scope pane.

5. Click *Okay*.

4.4   Creating a Problem Space

To keep this initial example simple, Sally's world will consist of a single goal—survival—realized using a single problem space. To create a problem space for Sally to operate within, perform the following steps:

1. Select the *Problem Spaces* tab inside the *Herbal GUI Editor* (if you don't see the *Herbal GUI Editor*, select the *Herbal>Show GUI Editor*... menu item).

2. Click on the *New* button located next to the list of existing problem spaces.

3. Accept the default value given in the *Library* field.

4. Enter *Survive* in the *Element Name* field and click *Finish*.

4.5   Creating Operators

To stay alive, Sally must eat when she is hungry and drink when she is thirsty. Actions such as these are performed using operators. To create an *eat* and *drink* operator, perform the following steps:

1. Select the *Operators* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing operators.

3. Accept the default value given in the *Library* field.

4. Enter *eat* in the *Element Name* field and click *Finish*.

5. Repeat steps 1-4 to create an additional operator called *drink*.

Operators are similar to "if-then" rules (but are more complex, as noted in the Psychological Soar Tutorial, Laird's tutorial, Newell's book, etc.) and contain conditions and actions. When the conditions in the operator are true, the operator may be chosen by the architecture, resulting in the

execution of actions. Because the operators just created (*eat* and *drink*) do not contain conditions or actions yet, additional warnings will appear in the Problems view. In the upcoming sections, these warnings will be resolved as conditions and actions are created and eventually associated with these operators.

4.6   Creating Types

For Sally to operate in her very simplified world, a representation of Sally's world must be created. In Herbal, the world is represented by a set of data types. In this simplified world, only a single type is needed to represent Sally's current state of hunger and/or thirst. To create this data type, follow these steps:

1. Select the *Types* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing data types.

3. Accept the default value given in the *Library* field.

4. Enter *Person* in the *Element Name* field

5. Because this agent does not operate within a simulated environment, Sally will not be interacting with sensors and motors. As a result, this type will not be used for input or output. Please do not check the *Will this type be used for I/O?* checkbox.

6. Click *Finish*.

Each data type should contain fields that further qualify the data type. For example, the *Person* data type needs to contain two boolean fields: one that is true if Sally is hungry and one that is true if Sally is thirsty. To add these fields to the *Person* data type, follow these steps:

1. Select the *Person* data type in the list of existing data types.

2. Click on the *Add* button located next to the list of current fields.

3. Select *boolean* and click *Next*.

4. Enter *hungry* for the name of the field and click *Finish*.

5. Repeat steps 1-4 to create an additional boolean field named *thirsty*.

4.7   Creating Conditions

For Sally to achieve her goal of survival, there are two specific conditions that she needs to look out for: hunger and thirst. These conditions are indicated by the presence of a fact of data type *Person* with a hungry field of true and/or a thirsty field of true. To create these conditions, follow these steps:

1. Select the *Conditions* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing conditions.

3. Accept the default value given in the *Library* field.

4. Enter *isHungry* for the name of the condition and click *Finish*.

5. Repeat steps 1-4 to create an additional condition name *isThirsty*.

6. Each condition should contain one or more clauses that must all be true for the condition to be true. To add clauses to the *isHungry* and *isThirsty* conditions, follow these steps:

7. Select the *isHungry* condition in the list of existing conditions.

8. Click on the *Add* button located next to the list of current clauses.

9. Choose the data type that you want to use in the clause. In this case, you want to see if an instance of the *Person* data type exists in working memory with its hungry field equal to *true*. As a result, select the *Person* data type, but do not click *Next* yet.

10. If the clause being created happens to successfully match an instance of the *Person* data type, it is helpful to give that instance a name so that it can be used later on in an action. For example, if we find an instance of the *Person* data type with a *true* hungry field, we need to set that field to *false* after Sally eats. To give the matching instance a name, enter *hungryPerson* into the *Output Variable Name* field in the dialog, and click *Next*.

11. Select the *hungry* field and then click on the *Restrict* button.

12. The next window allows you to specify the conditions upon which you will restrict the hungry field. In this case, you want to restrict the value to be equal to *true*. To accomplish this, select = from the list of *Relational Operators* and type *true* in the *Literal* field.

13. Click *Finish* and click *Finish* again.

14. Repeat steps 7-13 for the *isThirsty* condition, naming the matching instance thirstyPerson.

4.8   Creating Actions

Creating actions for Sally is perhaps the most involved step in the model creation process. The actions Sally will need are shown below in Table 1.

Table 1. Actions used by Sally.

Action Name	Description
init	Initialize Sally's hungry and thirsty status by creating an instance of the *Person* data type and setting its hungry field to *true* and its thirsty field to *false*.
markHungry	Sets the hungry field of a named instance of the *Person* data type to *true*.
markNotHungry	Sets the hungry field of a named instance of the *Person* data type to *false*.
markThirsty	Sets the thirsty field of a named instance of the *Person* data type to *true*.
markNotThirsty	Sets the thirsty field of a named instance of the *Person* data type to *false*.
printSlurp	Prints "Slurp" to the console.
printChomp	Prints "Chomp" to the console.

To create the actions shown in Table 1, follow these steps:

1. Select the *Actions* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing actions.

3. Accept the default value given in the *Library* field.

4. Enter *init* for the name of the action and click *Finish*.

5. Repeat steps 1-4 to create the remaining actions shown in Table 1.

The steps actually performed by an action are determined by the action clauses. To give our actions some substance, we need to add clauses to all seven of the actions shown in Table 1. Starting with the *init* action, follow these steps:

1. Select the *init* action in the list of existing actions.

2. Click on the *Add* button located next to the list of current clauses.

3. The purpose of the *init* action is to create an initial fact in working memory representing Sally's initial state. As a result, select *Add a new fact* from the list of choices for the action and then click *Finish*.

4. In the next window, select the type of fact you want to create. In this case, select the *Person* data type and click *Next*.

5. We want Sally to start life hungry and not thirsty, so we need to set the two fields of the *Person* data type appropriately. Select the hungry field and click on the Literal button. This allows you to specify a literal value for the hungry field. Enter *true* and click *Finish*.

6. Repeat step 5 for the thirsty field, setting it to *false*.

7. Click on *Finish*.

The actions in Table 1 starting with "mark" are responsible for altering the field values of a named *Person* fact. In other words, these actions edit existing, named facts. To complete the actions, follow these steps:

1. Select the *markHungry* action in the list of existing actions.

2. Click on the *Add* button located next to the list of current clauses.

3. Select "edit an existing fact" from the list of choices for the action and then click *Finish*.

4. In the next window, select the type of fact you want to edit. In this case, select the *Person* data type (but don't click *Next* yet).

5. This action relies on having a named instance available to it for editing. The name you provide will be used later when the action is assigned to an operator. To provide a name for this instance to be edited, type "person" in the Input Variable Name field and click *Next*.

6. Select the hungry field and click on the Literal button. This allows you to specify a literal value for the hungry field. Enter *true* and click *Finish* and then click *Finish* again.

7. Repeat steps 1-6 for the remaining actions in Table 1 that start with "mark".

The final two actions that need to be created are the *printChomp* and *printSlurp* actions. These actions exist only to provide visual feedback about Sally's behavior. To create the actions, follow these steps:

1. Select the *printChomp* action in the list of existing actions.

2. Click on the *Add* button located next to the list of current clauses.

3. Select *print* from the list of choices for the action and then click *Finish*.

4. In the next window, click on the Literal button and enter *"Chomp, Chomp, …"* (be sure to include quotes around the text) and click *Finish* and then click *Finish* again.

5. Repeat steps 1-4 for *printSlurp* action.

4.9   Associating Conditions and Actions with Operators

As mentioned earlier, operators can be seen to be similar to "if-then" rules and contain conditions and actions. If all of an operator's conditions are *true*, it may be chosen by the architecture resulting in its actions being performed. To associate the appropriate conditions to the *eat* and *drink* operators, follow these steps:

1. Select the *Operators* tab inside the *Herbal GUI Editor*.

2. Select the *eat* operator located in the existing operators list.

3. Click the *Add* button located next to the list of current conditions.

4. Select the *isHungry* condition and click *Finish*.

5. Repeat steps 1-4 to add the *isThirsty* condition to the *drink* operator.

The final step for completing our operators is to associate the appropriate actions to the *eat* and *drink* operators. The act of eating in our simple world involves the execution of the following actions:

- *printChomp* (when Sally eats she makes noise)

- *markNotHungry* (when Sally eats she is no longer hungry)

- *markThirsty* (when Sally eats she becomes thirsty)

To add these actions to the *eat* operator, follow these steps:

1. Select the *eat* operator located in the existing operators list.

2. Click the *Add* button located next to the list of current actions.

3. Select the *printChomp* action and click *Finish*.

4. Click the *Add* button located next to the list of current actions.

5. Select the *markNotHungry* action. Because this action is expecting input variables, additional information must be provided. As a result, you need to click *Next*.

6. The next window allows you to assign (wire) the output variables captured by the conditions with input variables required by the actions. In this case, the *markNotHungry* action is expecting an instance of the *Person* data type named person. Meanwhile, the *isHungry* condition names the instance of the *Person* data type that matches its clause to *hungryPerson*. As a result, we need to assign the *hungryPerson* output variable to the person input variable. To do this, select the *hungryPerson* variable in the Condition Output Variables list, and the *person* variable in the *Action Input* variable list, and then click on the *Assign* button. This creates a *wire* between the input and output variables so that working memory elements matched in the conditions are "passed" to the input variables in the actions.

7. Click on *Finish*.

8. Repeat steps 4-7 to add the *markThirsty* action. However, in step 6, because we have already wired the output variable *hungryPerson* to the input variable *person*, you do not need to make the assignment again (so you just click *Finish*).

The *drink* operator functions much like the *eat* operator. At this point you should feel comfortable with adding the appropriate actions (*printSlurp*, *markNotThirsty*, and *markHungry*) to the *drink* operator.

4.10   Wrapping Up the Sally Agent

We are almost done creating the Sally agent. To complete the agent, an initial action, along with the *eat* and *drink* operators, must be added to the *Survive* problem space so that this problem space can use the operators to achieve its goal of survival. Lastly, the *Survive* problem space must be assigned to Sally so that she is aware of this goal.

To add the initial action and operators to the *Survive* problem space, follow these steps:

1. Select the *Problem Spaces* tab inside the *Herbal GUI Editor*.

2. Select the *Survive* problem space in the list of existing problem spaces.

3. Click on the *Add* button located next to the list of current initial actions.

4. Select the *init* action and click on *Finish*.

5. Click on the *Add* button located next to the list of initial current operators.

6. Select the *eat* operator and click on *Finish*.

7. Repeat steps 5-6 to add the *drink* operator.

Finally, to add the *Survive* problem space to the agent named Sally, follow these steps:

1. Select the *Agents* tab inside the *Herbal GUI Editor*.

2. Select Sally in the list of existing agents.

3. Click on the *Add* button located next to the list of current problem spaces.

4. Select the *Survive* problem space and click on *Finish*.

Congratulations! A quick glance at the Problems View should reveal that all warnings have been resolved. In other words, your agent is complete!

4.11 Adding Design Rationale

To make it easier to understand and explain the design of an agent, Herbal supports the addition of design rationale to model elements. For each element in the model, rationale about what the element is, how it is used, and how it works, can be provided. For the purposes of this tutorial, you will add design rationale for Sally. Specifying rationale for the rest of the model elements is left as an additional exercise.

To add design rationale to Sally, follow these steps:

1. Select the *Agents* tab inside the *Herbal GUI Editor*.

2. Select the Sally agent located in the existing agents list.

3. Click on the *Rationale* button.

4. In the *What is this element?* field, enter the following: *This agent is designed specifically to show beginner users how to program a simple agent in Herbal.*

5. In the *How do I use this element?* field type: *This agent does not need special agent environment and can be executed directly in both the Soar and Jess debugging environments.*

6. In the *How does this element work?* field, enter the following: *This agent operates in a single problem space called Survive that has the goal of survival. Survival is accomplished by eating when hungry and drinking when thirsty.*

7. Click *Finish*.

4.12   Browsing the Agent in the Model Browser View

Herbal provides a *Model Browser* view (see Figure 2) that allows you to view the structure of an agent, along with design rationale and other explanatory information about an agent's components. The *Model Browser* is included by default in the Herbal perspective (if you don't see the *Model Browser*, make sure your Eclipse workspace matches Figure 2 by reviewing the steps given in Section 0). However, due to the amount of information in this view, it is best viewed when maximized within the Eclipse environment. To browse your agent's structure, follow these steps:

1. Maximize the *Model Browser* view by double clicking on the *Model Browser* tab.

2. On the left side of the view click Sally.

3. On the right side of the view, expand the "What is this?", "How do I use it?", and "How does it work?" sections. Within each of these sections you will find the design rationale you entered about Sally.

4. On the left side of the view, expand Sally and then select the *Survive* problem space. Information about the *Survive* problem space, if you had entered it, would be shown on the right side of the view.

5. Continue to expand and select nodes in the model structure section of the view to see the operators and their conditions and actions. Naturally, the more diligent you are about entering design rationale for your model elements, the more useful the *Model Browser* will be.

6. When you are done viewing your model, you can restore the view to its original size and position by double clicking on the *Model Browser* tab.

4.13   Testing Sally

To test Sally out, you need to load the agent source code into an environment and execute it. Herbal currently supports two different environments: Soar and Jess. Before you execute the code, it is a good idea to view the code that was generated by Herbal. In fact, alternating between the *Herbal GUI Editor* and the generated code is a good way to learn Soar and Jess syntax.

To view the source code generated by Herbal for the Sally agent, follow these steps:

1. In the *Navigator* view, locate and expand the *HungryThirsty* project.

2. Expand the output folder.

3. Expand the jess and soar folders.

4. The jess and soar folders will each contain a file containing the source code for Sally. These files will be named *Sally.soar* and *Sally.jess*.

5. Double click on either of these two files to view the source code.

After examining the source code, you are ready to execute Sally.

4.13.1   Executing Sally in Soar

To run Sally in Soar, follow these steps:

1. Launch the Soar Debugger (consult the Soar documentation if you are unsure how to launch the debugger).

2. Select the *File>Load Source File…* menu item and browse to the *Sally.soar* file. You will find this file located in the output subfolder of the *HungryThirsty* project. If you are not sure where your project is located on your disk, right-click (ctrl-click for the Mac) on the *HungryThirsty* project listed in the *Eclipse Navigator* view and choose the properties menu item. This will show you the project's properties, which includes the complete path to the project folder.

3. Select *Sally.soar* and click on the *Open* button.

4. Repeatedly click on the *Step* button and watch Sally survive by eating and drinking. The output you see in the Soar Debugger should resemble Table 2. Don't worry if you are getting different output because in the next section you will learn how to debug your model in Soar.

Table 2. Watching Sally Eat and Drink in the Soar Debugger.

Line	Soar Trace

```
1          1: O: O1 (|initialize-Survive|)
2          2: O: O2 (eat)
3       Chomp, chomp....
4          3: O: O3 (drink)
5       Slurp, slurp....
6          4: O: O4 (eat)
7       Chomp, chomp....
8          5: O: O5 (drink)
9       Slurp, slurp....
```

4.13.2   Executing Sally in Jess

To run Sally in Jess, follow these steps:

1. Open a console window and navigate to the output subfolder of the *HungryThirsty* project. If you are not sure where your project is located on your disk, right-click (ctrl-click for the Mac) on the *HungryThirsty* project listed in the *Eclipse Navigator View* and choose the properties menu item.

2. Launch the Jess Shell from this console window (consult the Jess documentation if you unsure how to start the Jess Shell).

3. Type *(reset)*

4. Type *(watch focus)*

5. Type *(watch rules)*

6. Type *(batch Sally.jess)*

7. Type *(run 10)* and watch Sally survive by eating and drinking. The output you see in the Jess Shell should resemble Table 3. Don't worry if you are getting different output because the next section you will learn how to debug your model in Jess.

Table 3. Watching Sally Eat and Drink in the Jess Shell.

Line	Jess Trace
1	FIRE 1 Survive::HungryThirsty.operators.eat f-2
2	Chomp, chomp....
3	FIRE 2 Survive::HungryThirsty.operators.drink f-2
4	Slurp, slurp....
5	FIRE 3 Survive::HungryThirsty.operators.eat f-2
6	Chomp, chomp....
7	FIRE 4 Survive::HungryThirsty.operators.drink f-2
8	Slurp, slurp....

4.14 Additional Exercises

TBD

5   Lesson 3: Debugging Sally

Both Soar and Jess provide features that make it possible to observe Sally in action at a much more detailed level. By debugging Sally at this level, you can obtain a deeper understanding of how the model and the architectures work and fix any problems with the model. If the output of your running model did not match Table 2 (for Soar) or Table 3 (for Jess), you may find this lesson especially useful.

5.1   Debugging Sally in Soar

In this section you will re-run Sally, using the Soar debugger to closely follow the execution of the model. Follow these steps to learn how to debug Sally:

1. Launch the Soar Debugger if it is not already open (consult the Soar documentation if you are unsure how to launch the debugger).

2. Click on the *Excise all* button to remove all the rules that are currently in Soar (if you just opened the debugger this is not necessary).

3. Select the *File>Load Source File…*menu item and browse the *Sally.soar* file. You will find this file located in the output subfolder of the *HungryThirsty* project. If you are not sure where your project is located on your disk, right-click (ctrl-click for the Mac) on the *HungryThirsty* project listed in the *Eclipse Navigator* view and choose the properties menu item.

4. Select *Sally.soar* and click on the *Open* button.

5. Click on the *Step* button.

6. Type *print s1* (or the ID of the top state, e.g., S3) into the command text box near the
   bottom of the Soar debugger. This will print the contents of the *Survive* problem space.
   This should produce the output shown in Table 4. The names of the attributes for the
   problem space are identified using the ^ symbol. Each attribute name is followed by its
   value. You can tell what problem space you are looking at by examining the name attribute
   (in this case you are looking at the *Survive* problem space).

Table 4. Printing the Survive Problem Space.

Survive Problem Space

```
(S1
    ^io I1
    ^name Survive
    ^operator O2 +
    ^parent S1
    ^superstate nil
    ^top S1
    ^type state
    ^|HungryThirsty.types.Person| H1)
```

7. If you recall, the Sally model uses a working memory element called *Person* that keeps
   track of Sally's current hungry and thirsty state. This working memory element is located
   in the ^|*HungryThirsty.types.Person*| attribute of the *Survive* problem space (see Table 4).
   If you look at the output shown in Table 4 you will notice that this working memory
   element is designated as *H1*. To view the contents of this working memory element, type
   print H1 into the command text box. This should produce the output shown in Table 5.
   Notice that the hungry attribute of the *Person* working memory element is currently equal
   to *true* (Sally starts out hungry) and that the thirsty element is currently *false* (Sally is not
   yet thirsty). Also note, that the IDs may change across Soar versions and runs.

Table 5. Printing the Person Working Memory Element.

Person Working Memory

```
(H1 ^hungry true ^thirsty false)
```

8. Click on the *Step* button and watch the output generated in the debugger window. As
   expected, the *eat* operator is applied and the text "Chomp, Chomp, …" is printed in the

window (the *printChomp* action was responsible for this). This operator will cause Sally to no longer be hungry and to become thirsty. To verify this, type *print H1* again. This time you should see that the hungry attribute has been changed to *false* by the *eat* operator (specifically the *markNotHungry* action changed the attribute) and that the thirsty attribute has been changed to *true* (this was done by the *markThirsty* action).

9. If you continue to hit the *Step* button and type *print H1* you should be able to watch the *Person* working memory element change as the *eat* and *drink* operators are applied.

10. If you wish to restart the model at any time, click on the *init-soar* button. This button initializes Soar by emptying working memory and re-initializing any run-time statistics. However, *init-soar* does not delete the rules in your model. As a general rule, it is a good idea to click on the *Excise all* button every time before (not after) you load the source code for a new model, and it is a good idea to click on the *init-soar* button every time you restart a model that is already loaded. Or, put these statements into your model.

The Soar debugger provides a lot more functionality than was described here. You will learn a few more features in the next lesson. In addition, you can learn more sophisticated techniques by referring to Chapter 5 of the Soar User's Manual and also the "Intro to the Soar Debugger" manual. Both of these documents are included with the Soar distribution.

5.2   Debugging Sally in Jess

TBD

5.3   Additional Exercises

TBD

6   Lesson 4: Adding Hierarchy to Sally

Herbal is based on the Problem Space Computational Model, which allows for behavior based on a hierarchy of problem spaces. Recall that a problem space is designed to accomplish a specific goal. To further divide a problem solving strategy into smaller pieces, the goal of a problem space can be broken down into sub goals; each sub goal is represented by a child problem space. In the upcoming lesson, we will change Sally so that her behavior is driven by a hierarchy of problem spaces.

For the purposes of this lesson, Sally will accomplish her goal of surviving by continually achieving two sub goals: *ResolveHunger* and *ResolveThirst*. These sub goals will take the form of problems spaces that are children of the top level *Survive* problem space.

As the design of Sally becomes more complex, it is useful to draw upon design patterns to improve the model. A *design pattern* is a general solution that can be implemented to solve recurring design problems (Gamma, Helm, Johnson, & Vlissides, 1995). In this particular case we will use a design pattern that is commonly used to coordinate the interaction between a parent problem space and its many children. The design pattern works as follows:

1. The parent problem space (in this case the *Survive* problem space) does not contain any operators. Instead, this problem space acts only as a controller for its child problem spaces (in this case *ResolveHunger* and *ResolveThirst*).

2. When a child problem space is assigned to a parent, it is given a set of conditions. During model execution, if the set of conditions is met, the child problem space is created and remains active until the conditions are no longer true. At that point, the child space is destroyed and the parent space regains control.

3. If the set of conditions for more than one child problem space is satisfied, the underlying architecture (Soar or Jess) decides which child problem space should be created and activated.

6.1   Preparing the Problem Spaces

To implement the design described above, the *Survive* problem space must be cleared of its operators. In addition, two new problem spaces must be created and operators must be assigned to them.

To clear the operators from the *Survive* problem space follow these steps:

1. Select the *Problem Spaces* tab inside the *Herbal GUI Editor*.

2. Select the *Survive* problem space located in the existing problem spaces list.

3. Select the *eat* operator located in the current operators list.

4. Click on the Remove button located to the right of the current operators list.

5. Repeat steps 1-4 for the *drink* operator.

To create two new problem spaces, follow these steps:

1. Select the *Problem Spaces* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing problem spaces.

3. Accept the default value given in the *Library* field.

4. Enter *ResolveHunger* in the *Element Name* field and click *Finish*.

5. Repeat steps 1-4 for the *ResolveThirst* problem space.

Recall that in our new design, the *Survive* problem space acts as a controller and its two child problem spaces actually perform all of the interesting work. In Herbal, all of the actual work is done by operators. As a result, the two new problem spaces need to have operators assigned to them that *eat* and *drink*. Fortunately, these operators were created in the previous lesson and can be reused here. All that has to be done is to add these operators to the new problem spaces.

To assign operators to the *ResolveHunger* and *ResolveThirst* problem spaces, follow these steps:

1. Select the *Problem Spaces* tab inside the *Herbal GUI Editor*.

2. Select the *ResolveHunger* problem space in the list of current problem spaces.

3. Click the *Add* button to the left of the current operators list.

4. Select the *eat* operator and click *Finish*.

5. Repeat steps 1-4 to add the *drink* operator to the *ResolveThirst* problem space.

6.2   *Building the Hierarchy*

Now that the problem spaces are configured correctly, they can be assigned to an agent and arranged into a hierarchy. The *Survive* problem space is already assigned to Sally as the top most problem space. All that is left to do is assign *ResolveHunger* and *ResolveThirst* as children of the *Survive* problem space. In addition, the conditions that will be used to activate these child problem spaces need to be specified.

To add the child problem spaces and specify the appropriate conditions, follow these steps:

1. Click on the *Agents* tab in the *Herbal GUI Editor*.

2. Select Sally in the list of current agents.

3. Select *Survive* in the list of current problem spaces.

4. Click on the *Add* button located to the right of the current problem spaces list. This allows you to added children to the *Survive* problem space.

5. Select *ResolveHunger* and click *Next*.

6. Select *isHungry* in the list of available conditions and click on the button labeled >>. This will move the *isHungry* condition into the list of conditions that will be used to determine when the *ResolveHunger* problem space will be created and activated. In other words, when Sally is hungry, *RevolveHunger* will become active. Notice how the *isHungry* condition can be reused in different contexts. All of the components in Herbal can be reused freely within a single model, or across multiple models.

7. Click *Finish*.

8. Repeat steps 1-7 to add the *ResolveThirst* problem space as a child of the *Survive* problem space. Use the *isThirsty* condition when you specify the conditions in step 6.

Congratulations! You just created your first hierarchical model using Herbal!

6.3   Viewing the Hierarchical Model in the Model Browser

Now that the Sally model has more structure in its behavior, it is a good idea to go back to the *Model* Browser view (see Figure 2) to view the model (if you don't see the *Model Browser*, make sure your Eclipse workspace matches Figure 2 by reviewing the steps given in Section 0).

1. Maximize the *Model Browser* view by double clicking on the *Model Browser* tab.

2. On the left side of the view click the Sally node and expand it. You should see the *Survive* problem space located as a child of Sally.

3. Expand the *Survive* problem space. This should display the child problem spaces *ResolveHunger* and *ResolveThirst*.

4. Expand *ResovleHunger* and *ResolveThirst* to see the operators used by these problem spaces. You can also expand the operators to view the conditions and actions that make up the operators.

5. Click on the *ResolveHunger* problem space and, on the right side of the view, expand the "How does it work?" section. Within this section you can see the entry conditions for *ResolveHunger*. Notice that *ResolveHunger* is created and activated when the *isHungry* condition is true.

6. Continue to expand and select nodes in the model structure section of the view to see the structure of the model. Once again, the more design rationale that you enter for your model (see Section 0), the more useful the *Model Browser* will be.

7. When you are done viewing your model, you can restore the view to its original size and position by double clicking on the *Model Browser* tab.

6.4 Running the Hierarchical Model

Please run your model using either Soar or Jess by following the steps in Section 0. The output you see should match Table 6 (if you ran your model in Soar) or Table 7 (if you ran your model in Jess). If it does not match, you will need to debug your model (see Section 0) but you should finish reading the rest of this section first.

Table 6. Watching the Hierarchical Model Execute in Soar.

Line	Soar Trace
1	1: O: O1 (|initialize-Survive|)
2	2: O: O2 (|impasse*ResolveHungerps|)
3	3: ==>S: S2 (operator no-change)
4	4: O: O3 (|initialize-ResolveHunger|)
5	5: O: O4 (eat)
6	Chomp, chomp....
7	6: O: O5 (|impasse*ResolveThirstps|)
8	7: ==>S: S3 (operator no-change)
9	8: O: O6 (|initialize-ResolveThirst|)
10	9: O: O7 (drink)
11	Slurp, slurp....

Table 7. Watching the Hierarchical Model Execute in Jess.

Line	Jess Trace
1	FIRE 1 Survive::HungryThirsty.models.impasse1 f-1, f-2
2	<== Focus Survive
3	==> Focus ResolveHunger
4	FIRE 2 ResolveHunger::HungryThirsty.operators.eat f-2, f-2
5	Chomp, chomp....
6	FIRE 3 ResolveHunger::ResolveHunger-exit f-4
7	<== Focus ResolveHunger
8	==> Focus Survive
9	FIRE 4 Survive::HungryThirsty.models.impasse2 f-3, f-2
10	<== Focus Survive
11	==> Focus ResolveThirst
12	FIRE 5 ResolveThirst::HungryThirsty.operators.drink f-2, f-2
13	Slurp, slurp....

The hierarchical nature of the model can be seen in the output traces shown in Table 6 and Table 7. Understanding this output is an important part of understanding the new behavior of Sally. The next two sections will explain the Soar and Jess output respectively.

6.4.1 Interpreting the Soar output trace

The first line of output in the Soar trace illustrates the initialization of the *Survive* problem space. The second line is new: the model has encountered an impasse. In other words, the *Survive* problem space has recognized the fact that Sally is hungry, but it does not know how to solve this dilemma and thus has reached an impasse. This impasse causes Sally to enter the *ResolveHunger* problem space in an attempt to resolve this impasse. The impasse is considered resolved as soon as Sally is no longer hungry. Lines 2-4 illustrate the impasse and the creation of the *ResolveHunger* problem space.

Fortunately for Sally, the *ResolveHunger* problem space is well equipped to handle the situation because it is armed with the *eat* operator. Lines 5 and 6 show the operator in action: the *eat* operator changes the *Person* working memory element so that it is no longer hungry, and at the same time it marks the thirsty field in working memory *true*. Because Sally is not hungry anymore, the conditions that caused the *ResolveHunger* problem space to be created are no longer satisfied and the problem space is destroyed. This brings the model back to the Surivive problem space.

At this point in time Sally is thirsty and this causes a new impasse, as shown on line 7-8. The impasse results in the model entering the *ResolveThirst* problem space (line 9) and eventually drinking (line 10-11).

6.4.2 Interpreting the Jess output trace

Although not shown in the trace, the Jess model begins in an initialized *Survive* problem space. Because Sally starts life hungry, the model immediately encounters an impasse: the *Survive* problem space has recognized the fact that Sally is hungry, but it does not know how to solve this dilemma (line 1). This impasse causes Sally to leave the *Survive* problem space and enter the *ResolveHunger* problem space, all in an attempt to resolve this impasse (line 2 and 3). The impasse is considered resolved as soon as Sally is no longer hungry.

Fortunately for Sally, the *ResolveHunger* problem space is well equipped to handle the situation because it is armed with the *eat* operator. Lines 4 and 5 show the operator in action: the *eat* operator changes the *Person* working memory element so that it is no longer hungry, and at the same time it marks the thirsty field in working memory *true*. Because Sally is not hungry anymore, the conditions that caused the *ResolveHunger* problem space to be created are no longer satisfied and the problem space is destroyed (line 6 and 7). This brings the model back to the *Survive* problem space (line 8).

At this point in time Sally is thirsty and this causes a new impasse, as shown on line 9. The impasse results in the model entering the *ResolveThirst* problem space (line 11) and eventually drinking (line 12 and 13).

6.5   Additional Exercises

TBD

7   Lesson 5: Debugging a Hierarchical Model

As models become more complicated, it becomes more important to be able to debug them effectively. In this lesson you will learn how to debug a more complex, hierarchical model.

7.1   Debugging the Hierarchical Model in Soar

In this section you will run the new hierarchical Sally model, using the Soar debugger to closely follow the execution of the model. It is assumed that you are already familiar with the basics of the Soar Debugger covered earlier.

Table 8 contains the output from the interactive debugging session you are about to perform. You will find it useful to refer to Table 8 while you perform the following steps:

1. Launch the Soar Debugger if it is not already open (consult the Soar documentation if you are unsure how to launch the debugger).

2. Click on the *Excise all* button to remove all the rules that are currently in Soar (if you just opened the debugger this is not necessary).

3. Select the *File>Load Source File…*menu item and browse the *Sally.soar* file. If you are not sure where your project is located on your disk, right-click on the *HungryThirsty* project listed in the *Eclipse Navigator* view and choose the properties menu item. This will show you the project's properties, which includes the complete path to the project folder.

4. Select *Sally.soar* and click on the *Open* button.

5. Click on the *Step* button. You should see the output shown on line 2 in Table 8. At this point the model has entered the top most problem space: *Survive*.

6. Type *print s1* into the command text box near the bottom of the debugger to see the attributes for the *Survive* problem space (lines 4-5 in Table 8).

7. Type *print H1* to see the state of the *Person* working memory element (line 7 in Table 8).

8. Type *step* and you should see an impasse encountered in the *Survive* problem space (line 9). Type *step* two more times and you should see the creation of the *ResolveHunger* problem space to resolve the impasse (lines 10-13 in Table 8).

9. Type *print s2* to view the attributes for the new *ResolveHunger* problem space (lines 15-17 in Table 8).

10. Type *print -s* to see the current problem space hierarchy (lines 19-21 in Table 8). Notice that at the top of the hierarchy is *S1* (*Survive*). Also notice that *S1* has a child problem space *S2* (*ResolveHunger*) that was created by an impasse.

11. Type *step* again to see the *eat* operator get applied (lines 23-24 in Table 8).

12. Type *print H1* to see the state of the *Person* working memory element (line 26 in Table 8). Notice that Sally is no longer hungry and has become thirsty.

13. Type *print -s* to view the problem space hierarchy again (line 27-28 in Table 8). Notice that the *ResolveHunger* problem space has been destroyed and that the hierarchy only contains the top state (*Survive*). This happened because the conditions that caused the impasse have been resolved (Sally is no longer hungry!).

If you continue to debug the model you should see Sally enter the *ResolveThirst* problem space and drink.

Table 8. Output From the Interactive Debugging Session of a Hierarchical Model.

Line	Output From the Interactive Debugging Session

```
1     Step
2         1: O: O1 (|initialize-Survive|)
3     print s1
4     (S1 ^io I1 ^name |Survive| ^operator O2 + ^parent S1 ^superstate nil ^top S1
5            ^type state ^|HungryThirsty.types.Person| H1)
6     print H1
7     (H1 ^hungry true ^thirsty false)
8     step
9         2: O: O2 (|impasse*ResolveHungerps|)
10    step
11        3: ==>S: S2 (operator no-change)
12    step
13        4:    O: O3 (|initialize-ResolveHunger|)
14    print s2
15    (S2 ^attribute operator ^choices none ^impasse no-change ^name
16    |ResolveHunger|
17           ^operator O4 + ^parent S1 ^quiescence t ^superstate S1 ^top S1
18           ^type state)
19    print -s
20         : ==>S: S1
21         :    O: O2 (|impasse*ResolveHungerps|)
22         :    ==>S: S2 (operator no-change)
23    step
24        5:    O: O4 (eat)
25    Chomp, chomp....
26    print H1
27    (H1 ^hungry false ^thirsty true)
28    print -s
             : ==>S: S1
```

7.2   Debugging the Hierarchical Model in Jess

TBD

7.3   Debugging the Hierarchical Model in Herbal

Herbal provides its own debugging support for Soar models for version Soar 8.6 (only). This support requires the Soar/Java bridge provided with the Soar 8.6 distribution.

To use the Herbal debugger, follow these steps:

1. Copy the following Soar distribution files (located in the Soar installation folder) into the Eclipse program directory (the same directory that contains eclipse.exe):

For Windows:

> soar-library\ElementXML.dll
>
> soar-library\Java_sml_ClientInterface.dll
>
> soar-library\SoarKernelSML.dll
>
> soar-library\sml.jar

For Mac OS:

> lib/libElementXML.0.0.0.dylib
>
> lib/libElementXML.0.dylib
>
> lib/libElementXML.dylib
>
> lib/libJava_sml_ClientInterface.0.0.0.jnilib
>
> lib/libJava_sml_ClientInterface.0.jnilib
>
> lib/libJava_sml_ClientInterface.jnilib
>
> lib/libSoarKernelSML.0.0.0.dylib
>
> lib/libSoarKernelSML.0.dylib
>
> lib/libSoarKernelSML.dylib
>
> soar-library/sml.jar

2. Be sure that your Herbal perspective is up to date by selecting *Window->Reset Perspective* in Eclipse.

3. Activate the *Herbal Debug* view by clicking on the *Debug View* tab near the bottom of the Eclipse window.

4. Run the Soar Debugger and load the Sally model as described in section 4.

5. In Herbal, click on the *Connect* button.

6. Next, in Herbal, select the Soar agent from the drop down list (it will probably be called *soar1*).

7. In Herbal, click on the *Listen* button.

8. In the Soar Debugger click on the *Step* button several times and watch as the Herbal debugger builds a trace of your model in the debug view. The Herbal debugger keeps track of debug events on the left hand side of the window. Clicking on an event displays the conditions that were true during that event, the operators that were proposed, the operator

that was applied (shown in all capital letters), and the actions that were performed. Finally, the state of working memory, *after* the actions were executed, is shown at the bottom of the debug window.

7.4   Additional Exercises

TBD

8   Lesson 6: Learning

As described earlier, Herbal is capable of producing models in both Soar and Jess and each of these architectures has advantages and disadvantages. One of the advantages of using Soar is that it is capable of modeling learning. This lesson will demonstrate how to execute the Sally model so that it learns.

8.1   Running Sally With Learning Enabled

Soar models must operate within a hierarchy of problem spaces for them to be able to learn. Fortunately, the model created in Section 0 is hierarchical and as a result can learn when it is run within Soar.

Soar simulates learning by creating a *chunk* when a model successfully uses problem solving to resolve an impasse. If the model encounters the same impasse later on, it can immediately resolve it using the learned *chunk,* instead of repeating the problem solving tasks it used to resolve the impasse the first time. This allows the model to accomplish tasks more quickly.

Getting the latest Sally model to learn is as easy as turning on learning inside the Soar debugger. Output generated by the Soar Debugger during the execution of a learning Sally model is listed in Table 9. You will find it useful to refer to Table 9 throughout this lesson.

To watch Sally learn, follow these steps:

1. Launch the Soar Debugger (consult the Soar documentation if you are unsure how to launch the debugger).

2. Select the *File>Load Source File…*menu item and browse to the *Sally.soar* file. You will find this file located in the output subfolder of the *HungryThirsty* project. If you are not sure where your project is located on your disk, right-click (ctrl-click for the Mac) on the

HungryThirsty project listed in the *Eclipse Navigator* view and choose the properties menu item. This will show you the project's properties, which includes the complete path to the project folder.

3. Select *Sally.soar* and click on the *Open* button.

4. Turn learning on by typing *learn -e* in the command text box near the bottom of the Soar debugger (see line 1 in Table 9).

5. Instruct the Soar debugger to print *chunks* as they are created by typing *watch -L print* in the command text box near the bottom of the Soar debugger (see line 2 in Table 9).

6. Click on the *Step* button until Sally successfully resolves the first impasse that is encountered (ResolveHunger). This impasse is resolved by applying the *eat* operator. Immediately after the *eat* operator is applied, and "Chomp, chomp…" is printed, the debugger will notify you of the creation of a *chunk* (see line 14 in Table 9).

7. Click on the *Step* button until Sally successfully resolves the second impasse that is encountered (ResolveThirst). This impasse is resolved by applying the *drink* operator. Immediately after the *drink* operator is applied, and "Slurp, slurp…" is printed, the debugger will notify you of the creation of a *chunk* (see line 24 in Table 9)

8. Continue to click on the *Step* button. From this point on, Sally will know what to do to resolve her hunger and her thirst without having to create the *ResolveHunger* or *ResolveThirst* problem spaces (see lines 25-28 in Table 9).

You may be wondering why Sally stopped printing "Chomp, chomp…" and "Slurp, slurp…" after learning. Recall that the *eat* and *drink* operators perform three actions. Two of these actions modify working memory to record the fact that Sally has just eaten or drank and one of these actions prints a message. The learning process in Soar only includes actions that alter working memory. As a result, when the *chunks* are learned only two of the three actions are included in the chunk: the actions that print are included in the learned knowledge (rules), and are thus not printed.

Table 9. Debug Output for a Learning Sally.

Line	Output From the Interactive Debugging Session

```
1       learn -e
2       watch -L print
3       step
4            1: O: O1 (|initialize-Survive|)
5       step
6            2: O: O2 (|impasse*ResolveHungerps|)
7       step
8            3: ==>S: S2 (operator no-change)
9       step
10           4:   O: O3 (|initialize-ResolveHunger|)
11      step
12           5:   O: O4 (eat)
13      Chomp, chomp....
14      Building chunk-1*d5*opnochange*1
15      step
16           6: O: O5 (|impasse*ResolveThirstps|)
17      step
18           7: ==>S: S3 (operator no-change)
19      step
20           8:   O: O6 (|initialize-ResolveThirst|)
21      step
22           9:   O: O7 (drink)
23      Slurp, slurp....
24      Building chunk-2*d9*opnochange*1
25      step
26          10: O: O8 (|impasse*ResolveHungerps|)
27      step
28          11: O: O9 (|impasse*ResolveThirstps|)
```

8.2   Prescript and Postscript

In the previous section, you learned how to turn on learning and instruct the Soar debugger to notify you when a *chunk* is learned. If you are using learning often, you may find it tedious to enter these commands each time you load and run a model. Fortunately, Herbal supports the automatic insertion of Soar or Jess commands at the beginning and end of the generated source code.

To have Herbal automatically add the required learning commands to the top the generated Soar code for Sally, follow these steps:

1. Look in the *Navigator* view for a file called *prescript.soar*. This file should be located in the *model* folder of the *HungryThirsty* project. If you do not find the *prescript.soar* file, you can create it by following these steps:

    a. Right-clicking on the *model* folder and selecting *New>Other…*

    b. In the *New Dialog Box*, open the *Simple Folder*, select *File*, and then click *Next*.

c.   Type *prescript.soar* in the File name text field and click Finish.

2. Double click on the *prescript.soar* file. This will open the file in the Eclipse editor. The file should be empty.

3. Everything you type in this file will automatically be included at the top of the generated Soar code. Type *learn -e* on the first line of the file and *watch -L print* on the second line of the file. (In Soar 9.3.1, type *watch –L 1*).

4. When you save the file, the model will automatically be regenerated. To confirm that the commands were added to the model, double-click on the *Sally.soar* file located in the *output/soar* folder of the *HungryThirsty* project.

Herbal supports four different script files: *prescript.soar, prescript.jess, postscript.soar*, and *postscript.jess*. The contents of these files are included at the top or bottom of the Soar or Jess model output files. Feel free to create/edit these files as needed to insert custom code into the source files generated by Herbal.

8.3   Additional Exercises

TBD

9   Lesson 7: Creating a Simple Vacuum Cleaner Model in Herbal

To make the process of learning Herbal more interesting, an animated vacuum cleaner agent environment was created, and is available on the Herbal website as a separate download. This environment is based on the vacuum cleaner world introduced in the popular Artificial Intelligence textbook written by Russell and Norvig (2003). The goal of this lesson is to demonstrate how to create an Herbal model that executes in this environment.

9.1   Interacting With an External Environment

The model created in this lesson differs from the previous Sally models because it interacts with and executes in an external environment. Herbal assumes that agents interact with their environment by way of sensors and effectors. Information that is detected by an agent's sensor is

placed in working memory so the agent can react to it. In addition, agents can perform actions by placing commands in working memory that activate its effectors.

Each environment must provide documentation on the working memory elements represent sensor readings and the working memory elements that represent agent actions. The input and output working memory elements supported by the vacuum cleaner environment are listed in Table 10.

Table 10. Vacuum Cleaner Environment Input and Output Working Memory Elements.

Type	Field	Description
vacuum.types.action	move (string)	Allows the vacuum cleaner to perform an operation within the environment. Supported operations are: *left, right, up, down*, and *suck*.
vacuum.types.position	x (number)	Represents the vacuum cleaner's current horizontal position
	y (number)	Represents the vacuum cleaner's current vertical position.
vacuum.types.spot	status (string)	Represents the status of the vacuum cleaners current location. Possible values are *clean* or *dirty*.
vacuum.types.radar	dir (string)	Represents the location of the radar reading. Possible values are *left, right, up,* and *down*.
	reading (string)	Represents the status of the location specified by *dir*. Possible values are *clean, dirty* or *wall.*

9.2   Managing Multiple Projects

The first step towards creating a vacuum cleaner model is to create a new Herbal project. Please take the time now to create a new project called *VacuumCleaner* (if you do not remember how to create an Herbal project, please revisit Section 0).

You should now have two projects listed in the *Navigation* view: *HungryThirsty* and *VacuumCleaner*. Herbal allows for multiple projects to be open at the same time and switching

between projects can be done from the *Welcome* tab in the *Herbal GUI Editor*. To make sure that the VacuumCleaner project is the current project, follow these steps:

1. Select the *Welcome* tab inside the *Herbal GUI Editor* (if you don't see the *Herbal GUI Editor*, select the *Herbal>Show GUI Editor* menu item).

2. In the middle of the *Welcome* tab should be a link that contains the name of the current project. If the *VacuumCleaner* project is not the current project, click on this link to switch to the *VacuumCleaner* project.

You can use the *Welcome* tab at any time to determine what the current project is and to switch between Herbal projects.

9.3   Creating Types that Interact With an Environment

The next step towards creating a vacuum cleaner model is to build the data types that will allow the model to interact in the environment. The required types are listed in Table 10 and can be created from within the *Types* tab in the *Herbal GUI Editor*. These new types will be created a little differently than those created in Section 0 because they must be placed in a different library, and because they interact with the environment.

Please complete the following steps to create these types:

1. Select the *Types* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing data types.

3. The I/O types names are defined by the vacuum environment, as given in Table 10. To conform to these names, we have to create them in a different library. To do that, change the default value given in the *Library* field to *vacuum.types*. If *vacuum.types* is not a choice in the drop-down list, you will have to type it directly into the *Library* field (be careful not to capitalize the v of vacuum).

4. Enter *action* in the *Element Name* field.

5. Because this type is used to interact within the vacuum cleaner environment, please be sure to check the *Will this type be used for I/O?* checkbox.

6. Click *Finish*.

7. Select the *action* data type in the list of existing data types.

8. Click on the *Add* button located next to the list of current fields.

9. Select *string* and click *Next*.

10. Enter *move* for the name of the field and click *Finish*.

11. Repeat steps 1-11 for the remaining types listed in Table 10.

9.4   Viewing, Editing, and Sharing Library Code

Each library in Herbal is represented in its own XML file (W3C, 2004). While it is not required that Herbal modelers know XML, it can be useful for advanced users. As a result, this section will demonstrate how to view and edit a library's XML code.

In the previous section you created a new type library called *vacuum.types*. Consequently, Herbal has created a new XML file called *vacuum.types.xml* located in the *model* folder of the *VacuumCleaner* project. To view the contents of this file, just double-click on it in the *Navigation* view.

Even if you are not familiar with XML, it is still relatively easy to figure out how the *vacuum.types.xml* file is organized. Take the time to browse the contents of this file looking for the *spot* type that was created in Section 0. You should be able to find a section of text that resembles Table 11.

Table 11. The Spot Type Represented in XML.

Line	XML
1	`<type name='spot' isIO='true'>`
2	`    <rationale>`
3	`        <what></what>`
4	`        <how></how>`
5	`        <why></why>`
6	`    </rationale>`
7	`    <field name='status' type='string'/>`
8	`</type>`

The code shown in Table 11 is used internally by Herbal to represent the vacuum.types.spot type. In fact, users comfortable with XML can create entire models by editing the XML directly instead of using the Herbal GUI Editor. To demonstrate how models can be edited using XML, complete the following steps:

1. Locate the *what* section of the XML that represents the *vacuum.types.spot* type (see line 3 in Table 11.

2. Add design rationale by changing line 3 (in Table 11) as follows:

```
<what>
This type represents the condition of the current square (clean or dirty).
</what>
```

3. Save the file.

4. Select the *Types* tab inside the *Herbal GUI Editor*.

5. Select the *spot* data type in the list of existing data types.

6. Click on the *Rationale* button located next to the list of current data types. You should see the design rationale that you entered directly into the XML file in step 2 above.

7. Click *Finish*.

In Herbal, all libraries (agents, problem spaces, operators, conditions, actions, and types) are represented in XML and can be found in a project's *model* folder (in the *Navigation* view). As you learn more about Herbal, and become more adventurous, you may choose to slowly move away from the *Herbal GUI Editor*, and do more of your model development by directly editing the XML.

Another advantage of storing libraries in separate XML files is that it makes them easy to share. For example, the next time you, or someone you know, create an Herbal project that uses vacuum cleaner agents, you can copy the *vacuum.types.xml* file into the *model* folder of your new project and the types will automatically become available.

Herbal can be used to parse XML and compile it into Soar, Jess, and ACT-R output files as well. Please refer to Figure 3 for a block diagram of how this is accomplished. Models are specified in Herbal using XML libraries that contain either PSCM components (for Soar or Jess models) or declarative memory elements (for ACT-R models). These XML libraries are parsed and represented in memory using a Document Object Model (DOM). The in-memory DOM is displayed in the Herbal development environment and can be edited visually using the Herbal GUI Editor. Changes made to the DOM via the visual editor are reflected back into the XML libraries. The compiler traverses the DOM and transforms it into models written for both Soar and Jess, or

for ACT-R. Custom native Soar, Jess, or ACT-R code specified in pre-script and post-scripts are put on the front and back of the compiled code.
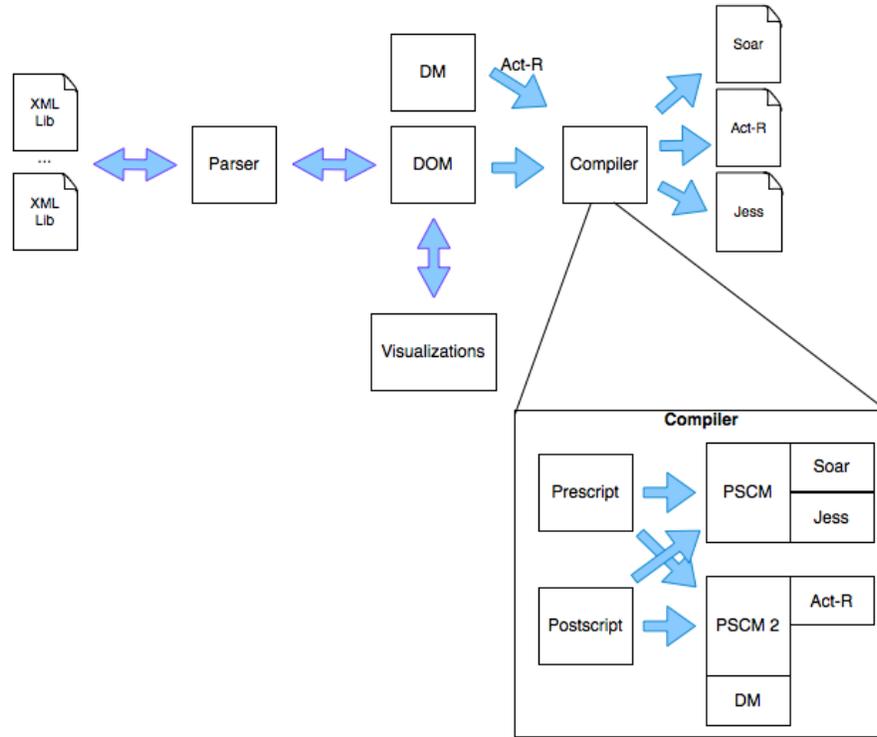

Figure 3 – Parsing and Compiling XML to Several Output Types in Herbal.

9.5   Creating Vacuum Cleaner Conditions

Each vacuum cleaner is equipped with radar that informs it of the status of the nearby squares. Radar is represented in working memory using the *vacuum.types.radar* type (see Table 10). For example, Table 12 shows the status of working memory when the vacuum cleaner has a clean square above it, a wall below it, and a dirty square to its right and to its left.

Table 12. Example of Radar in Working Memory.

Working Memory

```
vacuum.types.radar
    dir = up
    reading = clean
vacuum.types.radar
    dir = down
    reading = wall
vacuum.types.radar
    dir = right
    reading = dirty
vacuum.types.radar
    dir = left
    reading = dirty
```

In this section, two conditions will be created: one to detect when there is at least one dirty square near the vacuum cleaner and another that detects when there are no dirty square nearby. These conditions will be called *nearbyDirty* and *nearbyAllClean* respectively.

Complete these steps to create the *nearbyDirty* condition:

1. Select the *Conditions* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing conditions.

3. Accept the default value given in the *Library* field.

4. Enter *nearbyDirty* for the name of the condition and click *Finish*.

5. Select the *nearbyDirty* condition in the list of existing conditions.

6. Click on the *Add* button located next to the list of current clauses.

7. Because you want to see if an instance of the radar data type exists in working memory with its *reading* field equal to *dirty*, select the *radar* data type, but do not click *Next* yet.

8. If this clause happens to successfully match an instance of the *radar* data type, it is necessary to give that instance a name so that it can be used later on in an action. For example, if we find an instance of the *radar* data type that is reporting a dirty square, we may want to move the vacuum cleaner in the direction of that dirty square. To give the matching instance a name, enter *dirtySquare* into the *Output Variable Name* field and click *Next*.

9. Select the *reading* field and then click on the *Restrict* button.

10. Select = from the list of *Relational Operators* and type *dirty* in the *Literal* field.

11. Click *Finish* and click *Finish* again.

Complete these steps to create the *nearbyAllClean* condition:

1. Select the *Conditions* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing conditions.

3. Accept the default value given in the *Library* field.

4. Enter *nearbyAllClean* for the name of the condition and click *Finish*.

5. Select the *nearbyAllClean* condition in the list of existing conditions.

6. Click on the *Add* button located next to the list of current clauses.

7. Select the *radar* data type and click *Next*.

8. A radar reading indicates that a square is not dirty if its *reading* field is equal to either *clean* or *wall*. To specify this condition, click on the *reading* field and then click on the *Restrict* button.

9. Select = from the list of *Relational Operators* and type *clean* in the *Literal* field.

10. Click on the *Or* button.

11. Select = from the list of *Relational Operators* and type *wall* in the *Literal* field.

12. Click *Finish*.

13. Click on the *dir* field and then click on the *Restrict* button.

14. Select = from the list of *Relational Operators* and type *up* in the *Literal* field.

15. Click *Finish* and click *Finish* again.

16. You have just created a clause for the current action that is *true* if the square above the vacuum cleaner is either a wall or is clean. However, we want this condition to be *true* if this is the case for all squares, not just the one above the vacuum. As a result, you need to create three more clauses for the remaining directions. Please repeat steps 6-15 three times, specifying *down*, *left*, and *right* for the value of the *dir* field. When you are done, your condition clauses should resemble Figure 4.
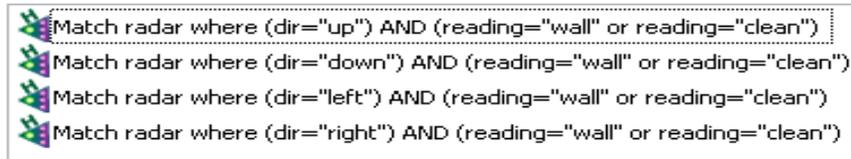
```
Match radar where (dir="up") AND (reading="wall" or reading="clean")
Match radar where (dir="down") AND (reading="wall" or reading="clean")
Match radar where (dir="left") AND (reading="wall" or reading="clean")
Match radar where (dir="right") AND (reading="wall" or reading="clean")
```

Figure 4 - Condition Clauses for the nearbyAllClean Condition.

9.6   Creating Vacuum Cleaner Actions

In this section, two actions will be created in Herbal. The first action, called *moveToDirt*, will cause the vacuum cleaner to move to a nearby dirty square. The second action, called moveRandom, will cause the vacuum cleaner to move in a random direction.

Follow these steps to create the *moveToDirt* action:

1. Select the *Actions* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing actions.

3. Accept the default value given in the *Library* field.

4. Enter *moveToDirt* for the name of the action and click *Finish*.

5. Select the *moveToDirt* action in the list of existing actions.

6. Click on the *Add* button located next to the list of current clauses.

7. Select "add a new fact" from the list of choices for the action and then click *Finish*.

8. In the next window, select *action* data type and click *Next*.

9. For this particular action we want to move in the direction of a nearby dirty square. The actual direction that the vacuum cleaner will move will depend on its current situation. As a result, we must depend on a condition to provide us with the correct direction. Specifically, we will use the output variable of the *nearbyDirty* condition to determine what direction to move to. Select the *move* field and click on the *Get* button.

10. Because we will rely on a matched a *radar* fact to tell us what direction to move in, select on radar and click *Next*, and then select *dir* and click *Next*.

11. This action relies on having a named instance available to it so it knows what direction to move in. The name you provide will be used later when the action is assigned to an

operator. To provide a name for this instance, type *dirtySquare* in the *Input Variable Name* field and click *Finish*.

12. Click *Finish* again.

Follow these steps to create the *moveRandom* action:

1. Select the *Actions* tab inside the *Herbal GUI Editor*.

2. Click on the *New* button located next to the list of existing actions.

3. Accept the default value given in the *Library* field.

4. Enter *moveRandom* for the name of the action and click *Finish*.

5. Select the *moveRandom* action in the list of existing actions.

6. Click on the *Add* button located next to the list of current clauses.

7. Select "add a new fact" from the list of choices for the action and then click *Finish*.

8. In the next window, select *action* data type and click *Next*.

9. For this particular action we want to move in a random direction. This can be accomplished using the random function. Select the *move* field and click on the *Function* button.

10. Select the *rand* function and click on the *Literal* button.

11. Type *"left"* (be sure to include the quotation marks) in the *Literal* field and click on *Finish*.

12. Repeat steps 10 and 11 entering *"right", "up"*, and *"down"* for the literal values.

13. Click *Finish* and *Finish* again.

9.7   Creating Vacuum Cleaner Operators

Operators can be created to give our vacuum cleaner some tools in which to exhibit behavior. In this section you will create two operators: *randomMove* and *smartMove*. The *randomMove* operator should cause the vacuum cleaner to move in a random direction when there is no nearby dirty square. The *smartMove* operator should cause the vacuum cleaner to move to a nearby dirty square when such a square exists.

Figure 5 illustrates these operators. Using what you learned in Section 0, please create the operators shown in Figure 5. Leave the scope of conditions and actions at their default value: "Top".
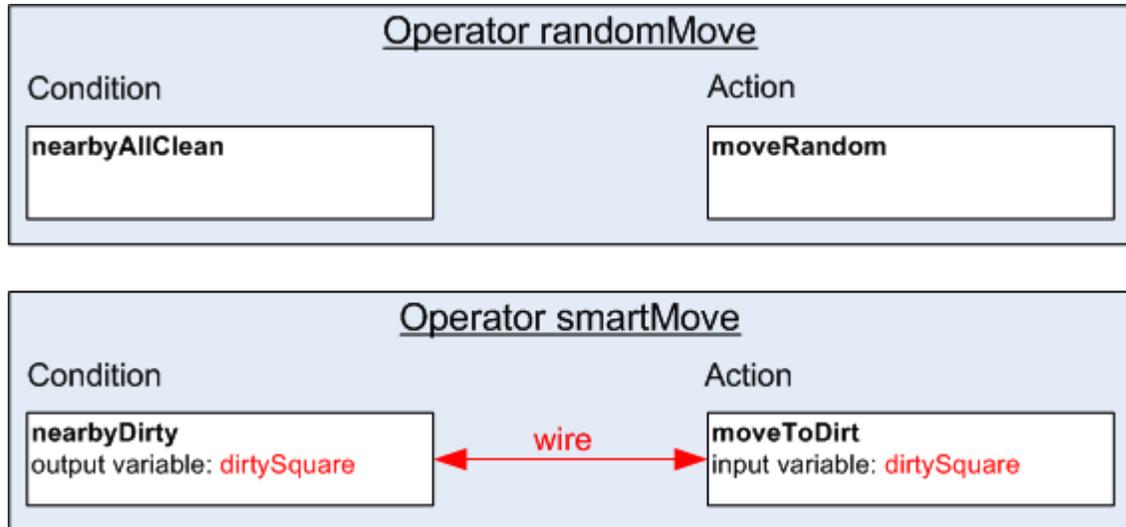

Figure 5 - The smartMove and randomMove Operators.

9.8 Creating Vacuum Cleaner Problem Spaces

To facilitate learning, and to provide some organization to the behavior of the vacuum cleaner, it will be useful to create some problem spaces for our agent to operate within. These problem spaces will produce the higher level behaviors of wandering around the board (*Wander*) and pursuing dirty squares (*Pursue*).

For now, each of these higher level behaviors will be implemented with a single operator and will be controlled by a top-level space called *Top*. However, if the model were made more intelligent in the future, these behaviors would become more complex; utilizing many operators and residing in a deeper problem space hierarchy.

Using what you learned in Section 0 and Section 0, please create the problem spaces listed in Table 13.

Table 13. Vacuum Cleaner Problem Spaces.

Problem Space	Initial Actions	Operators
Top	None	None

Wander	None	*randomMove*
Pursue	None	*smartMove*

9.9   Wrapping Up the Vacuum Cleaner Agent

The last step needed to complete our vacuum cleaner agent is to instantiate an Agent object and assign a problem space hierarchy to it. Using what you learned in Section 0, create an agent named *Tom*. Finally, using what you learned in Section 0 assign a hierarchy of problems to *Tom* based on.
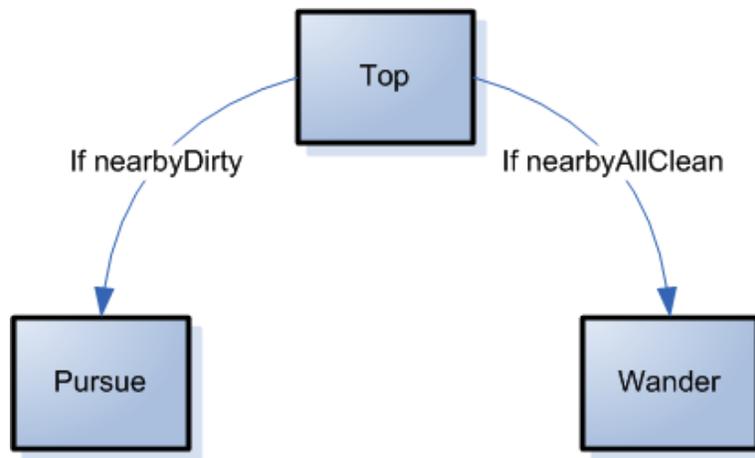

Figure 6 - Vacuum Cleaner Problem Space Hierarchy.

9.10   Running Tom in the Vacuum Cleaner Environment

To run Tom (the new vacuum cleaner agent), you will need to install the vacuum cleaner environment. Instructions for downloading and installing the environment can be found from Mark Cohen's webpage at http://www.marklisa.us/markacohen.

Once you get the environment running, you can load your model using the *File* menu. Because the environment supports both Jess and Soar, you can run either version of Tom. While Tom is running, trace statements will appear in the console that will help you view the details of the model while it is executing. The exact output depends on if you chose to run the Soar or Jess model.

9.11   Additional Exercises

Once you have Tom up and running there is a whole host of things you can try. Here are just two you may find interesting:

1. Use the *Options>Configure Board…* menu item in Vacuum to create a board that is 2 x 2 and has 2 dirty squares. Try running your model four times, each time with a different instance of this board configuration (hitting reset will generate a new random instance of the board). What does the model do? Why?

2. Create a second agent (within the same project) that operates in the problem space hierarchy shown in Figure . You will find the syntax of the action and perception you need in Table 10. Here are a few hints:

    a. You will need to create a condition *squareDirty* that is *true* when the agent is on a dirty square, and another *squareClean* that is *true* when it is on a clean square.

    b. You will need to create an action *cleanUp* that causes the agent to clean up the current square.

    c. You will need to create an operator *CleanUp* that will clean up the current square if it is dirty. This operator will use the condition *squareDirty* and the action *cleanUp*, created it step (a) and (b).

    d. You will need a new problem space called *Clean* that contains the operator *CleanUp* created in step (c).

    e. You will need to create a new agent called *CleaningTom* (within the same *VacuumCleaner* project), and add the problem space hierarchy shown in Figure to this agent.

Figure 7 – The problem space hierarchy used by CleaningTom the vacuum cleaner.

10   Lesson 8: Multi problem space levels and scope of facts

In this lesson, we will see that each fact "belongs" to a specific problem space. This problem space constitutes the "scope" of this fact. That is, when a fact is asserted, its scope is defined and cannot be changed.

In Herbal 3, if the scope is not explicitly overridden, when an Operator is triggered within a problem space, its scope is set to this current problem space. That means that its conditions will refer to facts whose scope is this problem space, and its actions will modify, remove or assert new facts in the scope of this problem space.

However, this default scope can be overridden at two levels:

- At the level of the association of the operator to the problem space: when an operator is associated to a problems pace, the scope of its conditions and the scope of its actions can be overridden.
- At the level of the association of conditions and actions to an operator. When a condition or an action is associated to an operator, its scope can be overridden.

This second overriding takes prevalence over the first one.

This overriding can take three values: *Top*, *Parent*, or *Local*. That means that each problem space has the possibility to access facts either in the Top problem space, or in its direct parent, or in itself.

Note that in Herbal 3, when an operator is associated to a problem space, the first overriding is set to *Top* by default.

In addition, we should notice that in Herbal 3, the scope of the conditions to enter a sub problem space cannot be overridden. Their scope is always the current problem space, which is the parent of the new sub problem space.

That will be illustrated by the example of the multi level vacuum cleaner presented below.

10.1 Creating a multi level Vacuum model

In this new lesson, we want to improve our vacuum cleaner model. When observing the behavior of our previous model CleaningTom, we can notice that it could be improved in several ways. For example, it can be improved if, when wandering, it would at least never go back to the square where it just comes from.

Let us call such a model CleaningBull. To implement its behavior, we will need to provide CleaningBull with a memory of where it was just coming from, so that we can prevent it to go back there.

We propose a solution consisting of decomposing the movement into two steps:

- First step: choose a direction acceptable with regard to the previous memorized move.

- Second step: memorize this new direction and move to it.

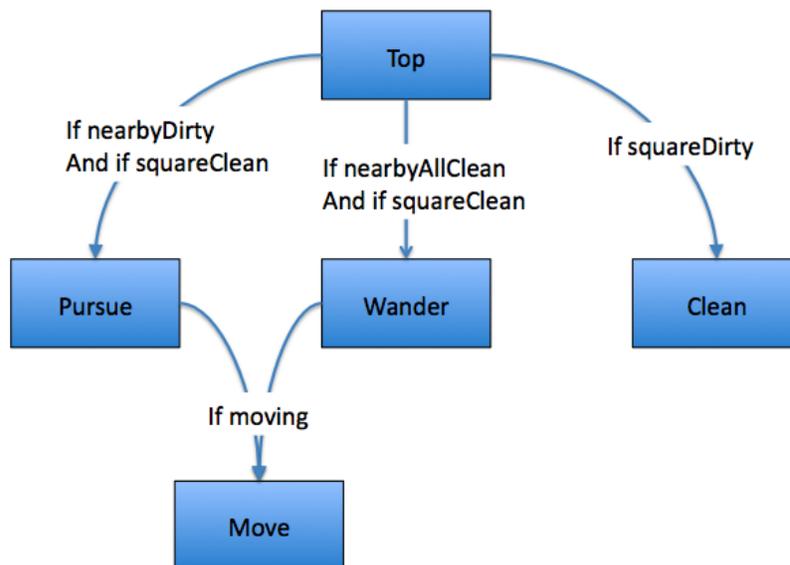We can implement this model with the problem space hierarchy shown in Figure 8.

Figure 8 – The problem space hierarchy used by CleaningBull the vacuum cleaner.

In this new hierarchy, the choice of the direction will be done in the Pursue and Wander problem spaces, and the memorization and actual move will be performed in the Move sub problem space.

10.2   Creating types

This new model will have to hold the previous move, and the choice for the next move, in working memory. We can store them in a single fact of type *direction*, which will have two fields:

1. The field *from*, of type string, which will store the direction from where the vacuum cleaner is just coming.

2. The field *to*, of type string, which will store the next chosen move, that is, the direction to where the vacuum cleaner attends to go.

Notice that only one fact of *direction* type needs to be instantiated in the model at any time. Because this memory of direction has to be available in both the *Wander* and the *Pursue* problem spaces, then its scope has to be the *Top* problem space. Because the next chosen move will have to be compared to the previous one, we have to give it also the *Top* scope, otherwise we could not compare them in a single condition.

We will also need another fact to indicate that a move direction has been chosen and that we are ready to enter the Move sub problem space. This fact has to have either the scope of the *Wander* or the *Pursue* problem space.

We can call the type of this new fact: *moving*. It needs no field.

Using what you have learned in Section 0, please create the *direction* type with its two fields *from* and *to*; and the *moving* type, with no field.

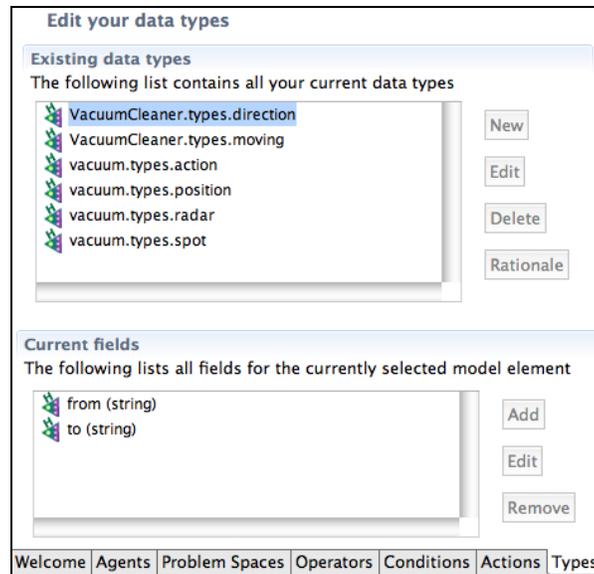When this is done, your *Types* tab should look like Figure .

Figure 9 - CleaningBull's data types

10.3 Creating conditions

We will need several new conditions for our new model:

1. An *isMoving* condition, which is *true* if it exists a fact of type *moving*. This condition will trigger the sub problem space Move. It will be also used to remove the fact *moving*.

2. An *isDir* condition, which is *true* if the *to* field of the *moving* fact is not equal to "no". It will be used to trigger the operator that will perform the actual move.

3. A *noDir* condition, which is *true* if the *to* field of the *moving* fact is equal to "no". It will be used to trigger the operator that will propose a new moving direction.

4. A condition *goDown*, which is true if the field *to* of the *direction* fact is equal to "down". It will be used to memorize that the vacuum cleaner was going to down, and thus was coming from up.

5. Three other conditions *goLeft*, *goRight*, *goUp*, similar to the *goDown* condition, but which are respectively *true* when the field *to* of the *direction* fact is equal to "Left", "Right", and "up".

6. A condition *goBack*, which is *true* if the field *to* of the fact of type *direction* is equal to its field *from*. This condition will be used to reject a chosen direction that would go back to the square where the vacuum cleaner was just before.

7. A condition *goToWall* which is *true* if the radar detects a wall in the chosen direction. This condition will be used to reject the chosen direction in this case.

Please use what you have learned in paragraph 0 to create the conditions *isMoving, isDir, noDir, goDown, goLeft, goRight, goUp.*

To create the clause of the condition *goBack*, you will have to use a literal value to bind the *to* field to the *from* field in a comparison clause. Please follow the steps bellow:

1. Make sure you have the *goBack* condition selected in the list of existing conditions, in the edit condition tab of the Herbal GUI Editor.

2. Click on the *Add* button located next to the list of current clauses.

3. Select the *direction* data type in the first page of the wizard. Type "direction" as an output variable name. Click *Next*.

4. In the "restrict field values" page (second page of the wizard), select *from* and click Restrict.

5. In the "Enter Literal Value" page, select "=" as a Rational operator and type *?to* as a Literal. Then click finish.

6. Back in the "restrict field values" page, select *to.* Type *?to* as a "Local variable name". Click Finish.

To create the clause of the condition *goToWall*, you will have to use a literal value to bind the *to* field of the *direction* type to the *dir* field of the *radar* type. Please follow the steps bellow:

1. Make sure you have the *goToWall* condition selected in the list of existing conditions, in the edit condition tab of the Herbal GUI Editor.

2. Click on the *Add* button located next to the list of current clauses.

3. Select the *direction* data type in the first page of the wizard. Type "direction" as an output variable name. Click *Next*.

4. In the "restrict field values" page, select *to.* Type *?to* as a "Local variable name". Click Finish.

5. Click Finish again to terminate the clause definition.

6. Click again on the *Add* button located next to the list of current clauses.

7. Select the *radar* data type in the first page of the wizard. Click *Next*.

8. In the "restrict field values" page, select *dir* and click Restrict.

9. In the "Enter Literal Value" page, select "=" as a Rational operator and type *?to* as a Literal. Then click finish.

10. Back in the "restrict field values" page, select *reading* and click Restrict.

11. In the "Enter Literal Value" page, select "=" as a Rational operator and type *wall* as a Literal. Then click finish.

12. Click Finish again to terminate the clause definition

Note that the two conditions *goBack* and *goToWall* could have been merged into a single one, but they are kept separated for more clarity.

10.4   Creating actions

We will also need several new actions:

1. An *init* action, which will assert the *direction* fact with both its *from* and *to* field initialized to "no". Note that this action will have to be performed in the *Top* scope.

2. A *moveToDir* action, which will add a new output fact of type *action* with its *move* field binded to a *direction* input variable. It will perform the actual move to the chosen direction. This action will have to be performed in the *Top* scope.

3. A *startMoving* action, which will assert a *moving* fact. It will trigger the entrance into the *Move* sub problem space. This action will have to be performed in either the *Wander* or the *Pursue* scope.

4. A *stopMoving* action, which will delete the *moving* fact, when the move will have been performed. This action will be performed in the *Move* problem space by referring to its parent scope which will be either *Wander* or *Pursue*. It will use a *moving* input variable

5. An *abortMove* action, which will set the *to* field of the *direction* fact to the value of "no". It will be used to reject a direction if the conditions *goToWall* or *goBack* are *true*. This action will have to be performed in the *Top* scope.

6. A *memoFromDown* actions, which will change the *from* field of the *direction* fact to "down". It will be called when the *goToUp* condition is *true*, to memorize this move. This action will have to be performed in the *Top* scope.

7. Three other actions *memoFromLeft*, *memoFromRight*, *memoFromUp*, which will respectively change the *from* field of the fact of *direction* type to "Left", "Right", "Up", when the conditions *goToRight*, *goToLeft*, *goToDown* will be *true*.

In addition, compared to the previous model of cleaningTom, the *moveRandon* and *moveToDirt* actions have to be changed, to change the *direction* fact instead of performing the actual move. So:

1. The *moveRandon* action must edit the *direction* fact and set its *to* field to the value rendered by the randon function. This action will have to be performed in the *Top* scope.

2. The *moveToDirt* action must edit the *direction* fact and set its *to* field to the value given by the *dir* field of the fact of *radar* type, which will be identified by the input variable called *dirtySquare*. This action will have to be performed in the *Top* scope. Please refer to paragraph 0 for the creation of this action.

10.5   Creating operators

Now that we have created conditions and actions, we have to link them within operators. We will need the following additional operators:

1. The *abortGoBak* operator. It will be triggered if the *goBack* condition is *true* and will fire the *abortMove* action. It will wire the *direction* variables of this condition and this action. We can keep blank the "overriding scope" property of its conditions and actions.

2. The *abortGoToWall* operator. It will be triggered if the *goToWall* condition is *true* and will fire the *abortMove* action. It will wire the *direction* variables of this condition and this action. We can keep blank the "overriding scope" property of its conditions and actions.

3. The *memoFromDown* operator. It will be triggered if the *goUp* condition is *true* and will fire the *memoFromDown* action. It will wire the *direction* variables of this condition and this action. We can keep blank the "overriding scope" property of its conditions and actions.

4. The *memoFromLeft*, *memoFromRight*, *memoFromUp* operators similar to the *memoFromDown*.

5. The *moveToDir* operator. It will be triggered if the *isDir* condition is *true* and will fire the *moveToDir* action. It will wire the *direction* variables of this condition and this action. We can keep blank the "overriding scope" property of its conditions and actions.

6. The *stopMoving* operator. It will be triggered if the *isMoving* condition is *true* and will fire the *stopMoving* action. It will wire the *moving* variable of this condition to the *moving* variable of this action. This operator will be trigger by the Move problem space and should be executed in its *parent* scope, as it applies to the moving fact, which is either asserted in the *Wander* or in the *Pursue* problem space.

The *randomMove* and *smartMove* operators will remain unchanged from the cleaningTom model. Only their actions have been changed.

10.6   Creating the problem spaces

Now we can create the problem spaces of our model.

Pursue:

> Initial action : *startMoving*
>
> Operators: *smartMove*

Wander

> Initial action : startMoving
>
> Operators : abortGoBack , abortGoToWall, randomMove (best)

Move

> Operators : memoFromDown (best), memoFromLeft(best), memorFromUp(best),

moveToDir (10), stopMoving (worst).

10.7   Creating the cleaningBull agent

We can now organize our problem space hierarchy according to the Figure 8.

The Move problem space can be added as a sub problem space of both the Wander and the Pursue problem spaces. In each case the condition to enter it is the isMove condition. Note that it will not be entered as long as there is still an operator to fire in its parent problem space. Therefore, it is ok if the moving fact is assessed as an initial action of the Wander and Pursue problem spaces.

11   Lesson 9: Creating a Simple dTank Model in Herbal

While the vacuum cleaner environment is interesting, the challenges it presents are quickly overcome. A more challenging and rich environment is needed to teach more advanced agent programming. dTank (acs.ist.psu.edu/dtank) is such an environment, and in this lesson you will learn how to create a dTank agent.

11.1 dTank I/O

dTank provides a competitive battleground where battalions of tanks fight it out for supremacy. Tanks can be driven by human interaction or by intelligent agents written in a variety of languages.

Much like the vacuum cleaner environment, dTank defines a set of working memory that allows the agent to interact with its environment. All of the nuances of the dTank environment are not specified in this tutorial. Instead, just enough information is provided to get you started building simple models. For a more complete description of dTank, see the dTank manual located on the dTank website (acs.ist.psu.edu/dtank). And, you will find some help by reading the Soar models.

The input and output working memory elements supported by the dTank environment are listed in Table 14.

Table 14. dTank I/O

Type	Field	Description
dtank.types.turretHeading	value (number)	Turns the tank's turret in the specified direction. Direction is specified as degrees clockwise from 0 to 360. What the tank operator sees, and the direction of its fire, is based on the direction that the turret is pointing.
dtank.types.tankHeading	value (number)	Turns the tank's body in the specified direction. Direction is specified as degrees clockwise from 0 to 360.
dtank.types.throttle	value (number)	Starts the tank moving either forward or backward. The throttle can be set to a number between -1.0 and 1.0. Zero stops the tank; negative numbers move the tank backward; and positive numbers move it forward.
dtank.types.action	value (string)	Can have any of the following values: *fire, faster, slower, rotateTurret, rotateTank*. Fire sends a missile in the direction that the turret is facing. Faster and slower either speed the tank up or slow the tank down by 0.1. Finally, rotate Turret and rotateTank move the turret or tank body an addition 30 degrees clockwise from its current orientation.
dtank.types.friend	x (number)	Represents the x location of a friendly tank. Note, you only see the tank if your turret is pointing in its direction.

Type	Field	Description
	y (number)	Represents the y location of a friendly tank. Note, you only see the tank if your turret is pointing in its direction.
	nationality (string)	Represents the nationality of the friendly tank.
	distance (number)	Represents the distance the friendly tank is from your tank.
	heading (number)	Represents the heading (in degrees) of the friendly tank.
dtank.types.enemySpotted	flag (boolean)	This flag is set to true when an enemy tank is spotted.
dtank.types.inTrouble	flag (boolean)	This flag is set to true when your tank is severely damaged.
dtank.types.enemy	x (number)	Represents the x location of an enemy tank. Note, you only see the tank if your turret is pointing in its direction.
	y (number)	Represents the y location of an enemy tank. Note, you only see the tank if your turret is pointing in its direction.
	nationality (string)	Represents the nationality of the enemy tank.
	distance (number)	Represents the distance the enemy tank is from your tank.
	heading (number)	Represents the heading (in degrees) of the enemy tank.
dtank.types.me	x (number)	Represents the x location of your tank.
	y (number)	Represents the y location of your tank.

Type	Field	Description
	nationality (string)	Represents the nationality of your tank.
	ammunition (number)	Represents the amount of ammunition your tank has left.
	fuel (number)	Represents the amount of fuel your tank has left.
	tankHeading (number)	Represents the direction (in degrees) your tank is facing.
	speedMPS (number)	The speed of your tank in miles per hour.
	speedKPS (number)	The speed of your tank in km per hour.
	throttle (number)	The current setting of your tank's throttle (-1.0 to 1.0).
	armor (number)	The state of your tank's armor.
dtank.types.terrain	right (string)	The type of terrain located to the right of your tank. Possible values include *OFF_MAP, STONE, WOODS, LOW_HILL, HIGH_HILL, GRASS, ROAD.*
	left (string)	The type of terrain located to the left of your tank. Possible values include *OFF_MAP, STONE, WOODS, LOW_HILL, HIGH_HILL, GRASS, ROAD.*
	up (string)	The type of terrain located above your tank. Possible values include *OFF_MAP, STONE, WOODS, LOW_HILL, HIGH_HILL, GRASS, ROAD.*

Type	Field	Description
	down (string)	The type of terrain located to below your tank. Possible values include *OFF_MAP, STONE, WOODS, LOW_HILL, HIGH_HILL, GRASS, ROAD*.

11.2 Instantiating the dTank Predefined Model

At this point in the tutorial you should be fairly familiar with the Herbal environment. As a result, in this lesson you will be instructed how to instantiate a predefined dTank model that you can later modify on your own. This predefined model serves as a starting point for more complicated models.

To instantiate the predefined dTank model, follow the sequence below.

8. Select *File>New>Project*

9. On the *New Project Window*, select *Herbal Project* and click *Next*.

10. For the project name enter *tank* and choose the default file location to store the project.

11. Click *Next*.

12. The next screen allows you to select from a set of predefined models. Select *dTank* and Click *Finish*.

11.3 Understanding the Predefined dTank Model

The best way to get to know the predefined dTank model is to browse it using the *Model Browser* view.

Double-click on the *Model Browser* tab to maximize this view (remember, you can double-click again to restore it to its original size). On the left hand side of the view, expand all of the model elements. Your model browser should look like Figure 10.
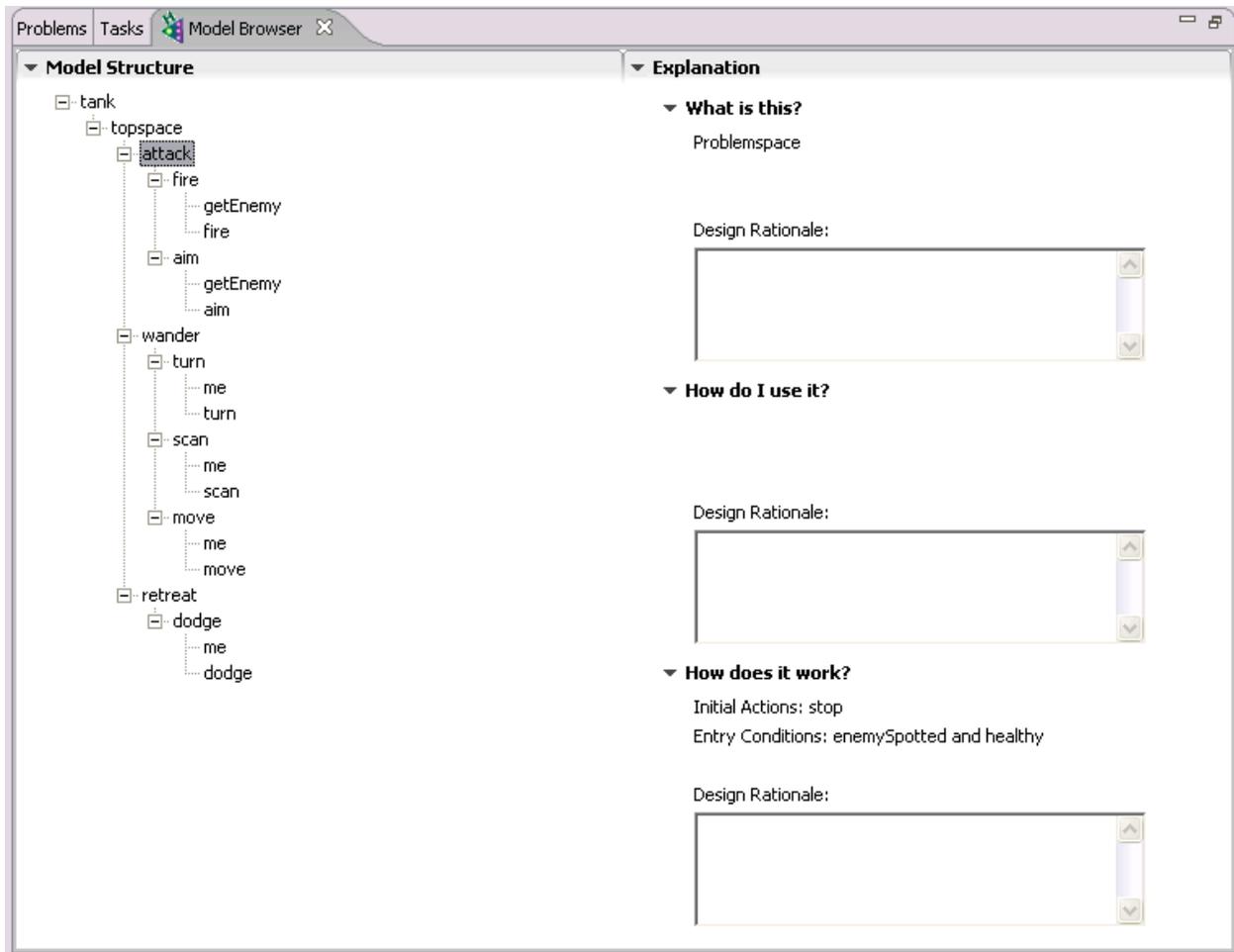
Figure 10 - Browsing the dTank Model

From the *Model Browser* view you can see that the model's behavior is represented using three problem spaces: *attack*, *wander*, and *retreat*. The *attack* problem space contains two operators: *aim* and *fire,* and its entry conditions occur when an enemy is spotted and your tank is healthy (Figure 10). You can also tell from the *Model Browser* that upon entering the attack problem space your tank will stop moving. Please continue to browse the model using the *Model Browser* so that you can determine the complete behavior of the tank.

The *Model Browser* will provide you with a good understanding of the high-level structure of the model. However, without any design rationale, it is hard to grasp the details without using the *Herbal GUI Editor*. If you plan on modifying the model to suit your needs, you will want to explore the details.

Take the time now to open the *Herbal GUI Editor* and explore the inner workings of the dTank model. As you discover how the model works, be sure to enter design rational for the model elements so that you can capture this information for future reference.

11.4 Executing the Herbal Tank in the dTank Environment

Before you make any modifications to the dTank model, you should run it in the dTank environment to see how well it does against other tanks. The dTank environment is available from the dTank website (acs.ist.psu.edu/dtank). The website contains instructions on how to install and execute dTank. Please take the time now to download and install dTank.

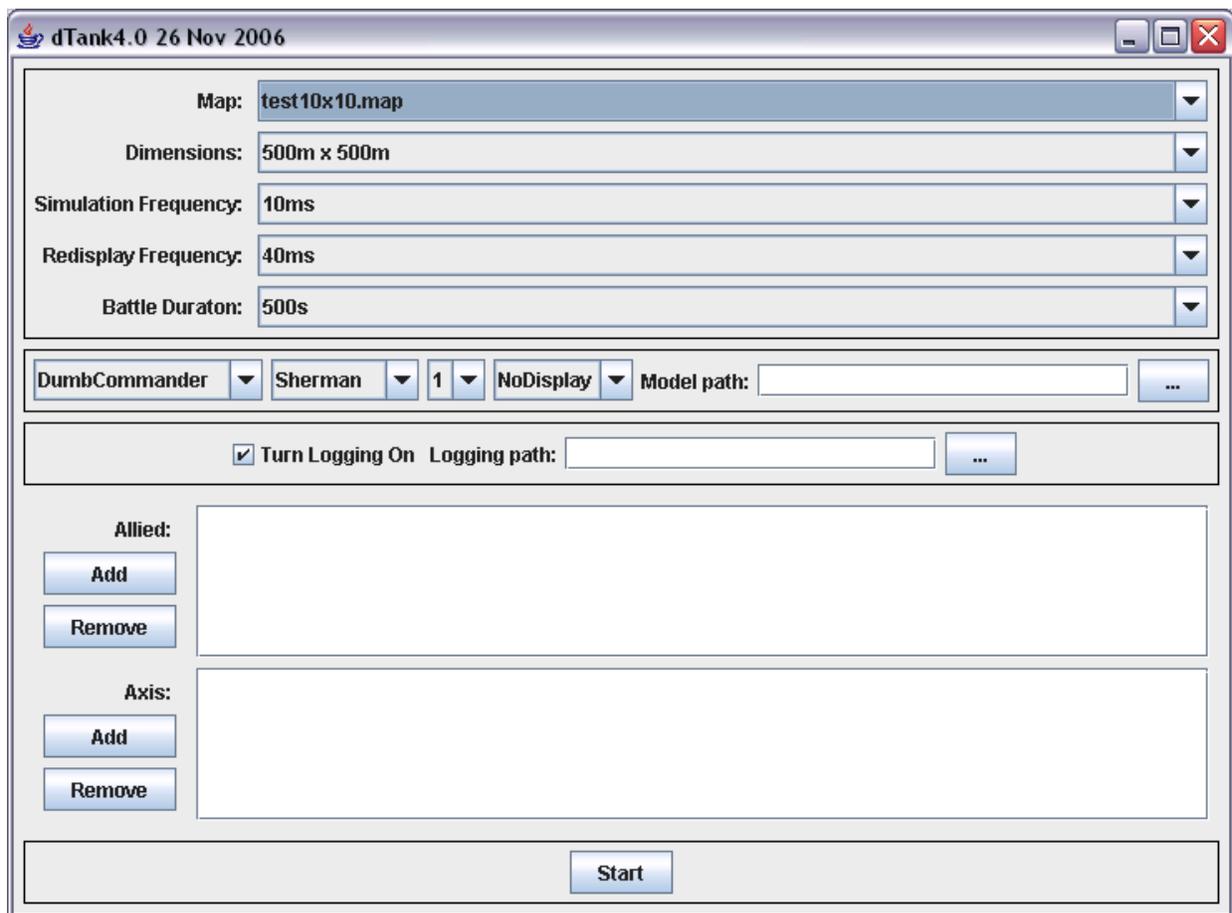When you execute dTank you will see the dTank *Startup Control Panel* shown in Figure 11.

Figure 11 - dTank Startup Control Panel

Using the *Startup Control Panel*, you can configure a battle. Each battle consists of an allied battalion of tanks and an axis battalion of tanks. You can add tanks to a battalion by selecting the type of commander to drive your tank, and the tank type. If you choose a *SoarCommander*, you

must also select a file that contains the Soar rules (generated by Herbal) that will be used by your commander. To add a Tiger tank driven by a *SoarCommander* to the allied battalion, follow these steps:

1. Select *SoarCommander* from the drop down list of commanders.

2. Select *Tiger* from the drop down list of tank types.

3. Click on the button labeled "..." and browse to the *tank.soar* file generated by Herbal for the predefined dTank model. You will find this file located in the output subfolder of the *tank* project. If you are not sure where your project is located on your disk, right-click (ctrl-click for the Mac) on the *tank* project listed in the *Eclipse Navigator* view and choose the properties menu item. This will show you the project's properties, which includes the complete path to the project folder.

4. Click on the *Add* button next to the list of allied tanks.

To add an opponent to the axis battalion, follow these steps:

1. Select *SmartCommander* from the drop down list of commanders.

2. Select *Sherman* from the drop down list of tank types.

3. Click on the *Add* button next to the list of axis tanks.

To watch the battle, click on the *Start* button. Your Soar tank will show up on the right hand side of the map.

11.5 Debugging dTank Models

The details of a dTank battle are meticulously recorded in a log file during execution. This log file is stored in the dTank installation folder and is called *dTankResults.log*. By default, this file records only the output produced by print statements in your model. For example,

Table 15 contains the top of a log file generated by a *SoarCommander*.

Table 15. Default dTank/Soar Log File Output

Log File Contents

```
========== Starting dTank4.0 version: 26 Nov 2006 ==========


========== Starting Battle 'null'    Combattants: Axis: AlliedCommander vs.
Allied: AlliedCommander ===========
SoarCommander Allied Tiger NoDisplay 1 (C:\Documents and Settings\mcohen\My
Documents\Development\runtime-EclipseApplication\tank\output\soar\tank.soar )
SmartCommander Axis Sherman NoDisplay 1 (C:\Documents and Settings\mcohen\My
Documents\Development\runtime-EclipseApplication\tank\output\soar\tank.soar )
Soar agent file: C:\Documents and Settings\mcohen\My
Documents\Development\runtime-EclipseApplication\tank\output\soar\tank.soar
SoarAgent0: turning
SoarAgent0: scanning
SoarAgent0: moving
SoarAgent0: scanning
SoarAgent0: turning
```

The default output shown in

Table 15 is helpful, but there are times when more information is needed. Additional debug information can be generated by adding Soar *watch* commands to the *prescript.soar* file. To add more debug information to the log file, follow these steps:

1. In Herbal, look in the *Navigator* view for a file called *prescript.soar*. This file should be located in the *model* folder of the *tank* project. If you do not find the *prescript.soar* file, you can create it by following these steps:

    a. Right-clicking on the model folder and selecting *New>Other…*

    b. In the *New Dialog Box*, open the *Simple Folder*, select *File*, and then click *Next*.

    c. Type *prescript.soar* in the File name text field and click Finish.

2. Add the following soar command to the *prescript.soar* file:

    watch --productions

    This command will tell Soar to trace all production firings. For a list of all of the available watch commands see Chapter 5 of the Soar User's Manual (Laird & B., 2005).

3. Rerun the tank battle. The output in your log file should contain the kind of detail shown in Table 16. The trace shown in Table 16 illustrates the tank's transition from the wander problem space into the attack problem space. This transition took place because an enemy was spotted during a scanning operation.

Table 16. dTank Log File With Production Watch Enabled

Log File Contents

```
SoarAgent3>: Firing |apply*global*remove-dtank-types-turretHeading|
Firing propose*wander*turn
Firing propose*wander*scan
Firing propose*wander*move
Retracting propose*wander*move
Retracting propose*wander*scan
Retracting propose*wander*turn
SoarAgent3>: Firing apply*wander*scan
scanning
SoarAgent3>: Firing propose*topspace*impasse*attackps
Retracting propose*topspace*impasse*wanderps
SoarAgent3>: Firing |apply*global*remove-dtank-types-turretHeading|
SoarAgent3>: Firing propose*initialize-attack
SoarAgent3>: Firing apply*initialize-attack
stopping
Firing propose*attack*fire
Firing propose*attack*aim
Retracting propose*initialize-attack
SoarAgent3>: Firing apply*attack*fire
firing
```

11.6 Additional Exercises

TBD

12   Advanced Features

Herbal contains a set of advanced features that you will find useful as you create more complicated models. These features often behave differently depending on the architecture you run your model in. As a result, you should avoid many of these features unless you are planning on running your model in only a single architecture (i.e. only Soar or only Jess).

In addition, it is strongly recommended that the Soar or Jess manual is consulted to get a better understanding of how these features work, before using them in Herbal.

The following sections describe these features in detail. Be sure to pay close attention to the areas in which the features vary by architecture.

12.1 Preferences

As you learned earlier in the tutorial, a problem space consists of a set of operators that are proposed and applied when the conditions for those operators are true. There will be times when the conditions for several operators, within a single problem space, will be true. As a result, more

than one operator will be proposed. This raises the question of which operator will actually be applied.

The underlying architecture you are using (Jess or Soar) will determine how this conflict will be resolved. If you want more control over which operator will be chosen in the event of a conflict, you can use preferences. Preferences make it possible to specify an ordering between proposed operators within a problem space.

Preferences are specified when you add an operator to a problem space. Follow these steps to specify a preference for a specific operator:

1. Select the *Problem Spaces* tab inside the *Herbal GUI Editor*.

2. Select the operator you would like to add a preference to, and then click on the *Edit* button located next to the list of existing operators.

3. Click *Next*.

4. Select either *best, worst,* or enter a positive integer in the drop down list entitled *Preference*. The meaning of this preference value is described in detail in Table 17.

5. Click *Finish*.

Table 17. Description of Allowed Preference Values

Preference Value	Behavior in Soar	Behavior in Jess
best	The operator will be favored over all other operators not given *best* preference. If two operators have *best* preference, Soar will choose randomly between them.	The operator will be favored over all other operators not given *best* preference. If two operators have *best* preference, Jess will choose using a depth first strategy.
worst	The operator will rank below all other operators that were not given *worst* preference. If all operators have *worst* preference, Soar will choose randomly between them.	The operator will rank below other all operators that were not given *worst* preference. If all operators have *worst* preference, Jess will choose using a depth-first strategy.

positive integer	The operator will be chosen based on a probability calculated using the numeric preferences of all other operators. For example, if operator 1 has a numeric preference of 25 and operator 2 has a numeric preference of 100, operator 2 will be 4 times more likely to be chosen. See the Soar manual for more details.	Operators given an integer preference will be ranked numerically from highest to lowest, and the operator with the highest numeric preference will be chosen first. See the Jess manual for more details. *NOTE: this behavior is different than what happens in the Soar compilation (see column to the left)*.
negative integer	Negative preference values are not allowed in Herbal.	Negative preference values are not allowed in Herbal.
all other values	Value will be inserted in place of the default preferences used by Herbal (indifferent and acceptable). This allows the Herbal programmer to override the default preferences used by Herbal. This feature is for advanced users only.	Value will be ignored by Herbal. *NOTE: this behavior is different than what happens in the Soar compilation (see column to the left).*

12.2 Elaborations

In Soar, a special type of rule called a state elaboration can be created to alter working memory before operators are proposed and applied. You can think of elaborations as background processes that wait for a specific event, and then alter working memory when that event takes place.

Elaborations are similar to operators in that they have an "if-part" and a "then-part". The "if-part" contains a series of conditions and the "then-part" contains a series of actions. When the conditions are true, the actions are executed.

What makes elaborations different than operators is that the working memory elements they create are retracted as soon as the conditions that supported them change. In addition, elaborations take place before the operators are proposed. This allows you to create elaborations that configure working memory in a way that supports your operators. It is recommended that you consult the Soar Manual to learn more about elaborations before using them in Herbal.

Because elaborations are made using the same if/then structure that operators use, you create elaborations by marking an operator as an elaboration. This allows for an added dimension of reuse, in which you can use a rule as both an operator and an elaboration within, and across problem spaces. To specify that an operator should be used as an elaboration, follow these steps:

1. Select the *Problem Spaces* tab inside the *Herbal GUI Editor*.

2. Select the operator you would like turn into an elaboration and then Click on the *Edit* button located next to the list of existing operators.

3. Click *Next*.

4. Check the *"Is this an elaboration?"* checkbox.

5. Click *Finish*.

6. Notice that the operator will be displayed with an asterisk "*" at the end of its name. This indicates that the operator will be used as an elaboration.

Currently, the elaboration flag is ignored in the Jess compilation. As a result, in the Jess compilation operators marked as elaborations will function as if they were "plain-old" operators. Due to the special properties of elaborations in Soar, models that use elaborations will likely demonstrate different behavior between the Soar and Jess compilations. *For this reason, elaborations should be avoided unless you only plan to run your model in Soar.*

References

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 *14th Behavior Representation in Modeling and Simulation*, 133-140. University City, CA.

Cohen, M. A., Ritter, F. E., & Haynes, S. R. (2010). Applying software engineering to agent development. *AI Magazine, 31*(2), 25-44.

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. (2009). Designs for explaining intelligent agents. *International Journal of Human-Computer Studies, 67*(1), 99-110.

Intro to the Soar Debugger in Java. (2005).): ThreePenny Software LLC.

Laird, J. E., & B., C. C. (2005). *The Soar user's manual version 8.6*: University of Michigan.

Lehman, J. F., Laird, J. E., & Rosenbloom, P. S. (1998). 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.

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

Newell, A., Yost, G. R., Laird, J. E., Rosenbloom, P. S., & 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).

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

W3C. (2004). The Extensible Markup Language.