26 Case Studies: JESS and other Expert Systems Shells in Java

Chapter Objectives This chapter examines Java expert system shells available on the world wide web

Chapter Contents
26.1 Introduction
26.2 JESS
26.3 Other Expert System Shells
26.4 Using Open Source Tools

26.1 Introduction

In the last three chapters we demonstrated the creation of a simple expert system shell in Java. Chapter 22 presented a representational formalism for describing predicate calculus expressions, the representation of choice for expert rule systems and many other AI problem solvers. Chapter 24 created a procedure for unification. We demonstrated this algorithm with a set of predicate calculus expressions, and then built a simple Prolog in Java interpreter. Chapter 25 added full backtracking to our unification algorithm so that it could check all possible unifications in the processes of finding sets of consistent substitutions across sets of predicate calculus specifications. In Chapter 25 we also created procedures for answering why and how queries, as well as for setting up a certainty factor algebra.

In this chapter we present a number of expert system shell libraries written in Java. As mentioned throughout our presentation of Java, the presence of extensive code libraries is one of the major reasons for the broad acceptance of Java as a problem-solving tool. We have explored these expert shells at the time of writing this chapter. We realize that many of these libraries will change over time and may well differ (or not even exist!) when our readers considers them. So we present their urls, current as of January 2008, with minimal further comment.

26.2 JESS

The first library we present is JESS, the Java Expert System Shell, built and maintained by programmers at Sandia National Laboratories in Albuquerque New Mexico. JESS is a rule engine for the Java platform. Unlike the unification system presented in Chapters 23 and 24, JESS is driven by a lisp-style scripting language built in Java itself. There are advantages and disadvantages to this approach. One main advantage of an independent scripting language is that it is easier to work with for the code builder. For example, Prolog has its own language that is suitable for rule

languages, which makes it easy and clear to write static rule systems.

On the other hand, Prolog is not intended to be embedded in other applications. In the case of Java, rules may be generated and controlled by some external mechanism, and in order to use JESS's approach, the data needs to be converted into text that this interpreter can handle.

A disadvantage of an independent scripting language is the disconnect between Java and the rule engine. Once external files and strings are used to specify rules, standard Java syntax cannot be used to verify and check syntax. While this is not an issue for stand-alone rule solving systems, once the user wants to embed the solver into existing Java environments, she must learn a new language and decide how to interface and adapt the library to her project.

In an attempt to address standardization of rule systems in Java, the Java Community Process defined an API for rule engines in Java. The Java Specification Request #94 defines the javax.rules package and a number of classes for dealing with rule engines. Our impression of this system is that it is very vague and seemingly tailored for JESS. It abstracts the rule system as general objects with general methods for getting/setting properties on rules.

RuleML, although not Java specific, provides a standardized XML format for defining rules. This format can theoretically be used for any rule interpreter, as the information can be converted into the rule interpreter's native representations.

JESS has its own JessML format for defining rules in XML, which can be converted to RuleML and back using XSLT (eXtensible Stylesheet Language Transformations). These formats, unfortunately, are rather verbose and not necessarily intended for being read and written by people.

Web links for using JESS include:

http://www.jessrules.com/- The JESS web site,

http://jcp.org/en/jsr/detail?id=94 - JSR 94: Java™ Rule Engine API,

http://www.jessrules.com/jess/docs/70/api/javax/rules/package-summary.html - javadocs about javax.rules (from JSR 94), and

http://www.ruleml.org/ - RuleML.

26.3 Other Expert System Shells

We have done some research into other Java expert rule systems, and found dozens of them. The following url introduces a number of these (not all in Java):

http://www.kbsc.com/rulebase.html

The general trend of these libraries is to use some form of scripting-language based rule engine. There is even a Prolog implementation in Java! There are many real implementations and issues that these things introduce, including RDF, OWL, SPARQL, Semantic Web, Rete, and more.

Thie following url discusses a high level look at rule engines in Java (albeit

from a couple years ago):

http://today.java.net/pub/a/today/2004/08/19/rulingout.html

Finally, we conclude with a set of links to the seemingly more interesting rule engines. We only picked the engines listed as free, some are open-source, some are not:

http://www.drools.org/

http://www.agfa.com/w3c/euler/

http://jlogic.sourceforge.net/ - a prolog interpreter in Java

http://jlisa.sourceforge.net/ - A Clips-like (NASA rule based shell in C) Rule engine accessible from Java with the power of Common Lisp.

http://mandarax.sourceforge.net/ - this one has some simple straightforward examples on the site, but the javadocs themselves are daunting.

http://tyruba.sourceforge.net/

Related to rule interpreters designed to search knowledge-based specifications, are interpreters intended to transfer knowledge, rules, or general specifications between code modules. These general module translation and integration programs are often described under the topic of the Semantic Web:

http://www.w3.org/2001/SW/

26.4 Using Open Source Tools

The primary advantage these tools have over our simple expert system shell is their range of features. Jess, for example, provides a rule language that frees the programmer from having to declare each rule as a set of nested class instances as in our simple set of tools. An interesting thought experiment would be to consider what it would take to write a parser for a rule language that would construct these class instantiations from a user-friendly rule language. A more ambitious effort, possibly suitable for an advanced undergraduate or masters level thesis project would be to implement such a front end.

In using these tools, the reader should not forget the lessons in extending java classes from the earlier chapter. Inheritance allows the programmer to extend open source code to include additional functionality if necessary. More often, we may simply use these tools as a module in a larger program simply by including the jar files.

In particular, the authors have seen the Jess tool used in a number of large applications at Sandia Laboratories and the University of New Mexico. Typical application architecture uses Jess as an inference engine in a larger system with databases, html front ends using Java Server Faces or similar technologies, and various tools to assist in file I/O, session archiving, etc.

For example, a development team at Sandia Laboratories led by Kevin Stamber, Richard Detry, and Shirley Starks has developed a system called FAIT (Fast Analysis Infrastructure Tool) for use in the National Infrastructure Simulation and Analysis Center (NISAC). FAIT addresses

the problem of charting and analyzing the interdependencies between infrastructure elements to help the Department of Homeland Security respond to hurricanes and other natural disasters. Although there are databases that show the location of generating plants, sub-stations, power lines, gas lines, telecommunication facilities, roads and other infrastructure elements, there are two problems with this data:

1. Interdependencies between elements are not shown explicitly in the databases. For example, databases of electrical power generation elements do not explicitly state which substations service which generating plants, relying on human experts to infer this from factors like co-location, ownership by the same utility, etc.
2. Interactions between different types of utilities, such as the effect of an electrical power outage on telecommunications hubs or gas pumping stations must be inferred from multiple data sources.

FAIT uses Jess to apply rules obtained from human experts to solve these problems. What is especially interesting about the FAIT architecture is its integration of Jess with multiple sources of infrastructure data, its use of a geographic information system to display interdependencies on maps, and its presentation of all this through an html front end.

The success of FAIT is intimately tied to its use of both open-source and commercially purchased tools. If the development team had faced the challenge of building all these components from scratch, the system would have cost an order of magnitude more than it did – if it could have been built at all. This approach of building extremely large systems from collections of independently designed components using the techniques discussed in this section has become an essential part of modern software development.